climlab-0.7.2 Documentation

Release 0.7.2.dev0

Moritz Kreuzer and Brian E. J. Rose

Feb 08, 2019

Contents

1 Introduction **1**
- 1.1 climlab . . . 1
  - 1.1.1 Python package for process-oriented climate modeling . . . 1
    - 1.1.1.1 Author . . . 1
    - 1.1.1.2 About climlab . . . 1
    - 1.1.1.3 Installation . . . 2
    - 1.1.1.4 Links . . . 3
    - 1.1.1.5 Dependencies . . . 3
    - 1.1.1.6 Documentation and Examples . . . 3
    - 1.1.1.7 Release history . . . 4
    - 1.1.1.8 Contact and Bug Reports . . . 5
    - 1.1.1.9 License . . . 5
    - 1.1.1.10 Support . . . 6

2 Quickstart Guide **7**
- 2.1 Installation . . . 7
- 2.2 Single-column Radiative-Convective model . . . 7

3 Installation **9**
- 3.1 Installing pre-built binaries with conda (Mac OSX, Linux, and Windows) . . . 9
- 3.2 Installing from source . . . 9
- 3.3 Installing from source without a Fortran compiler . . . 10

4 Source Code **11**

5 Dependencies **13**
- 5.1 Required . . . 13
- 5.2 Recommended for full functionality . . . 13

6 Architecture **15**
- 6.1 Process . . . 16
  - 6.1.1 Basic Dictionaries . . . 16
  - 6.1.2 Subprocesses . . . 16
  - 6.1.3 Process Integration over time . . . 17
    - 6.1.3.1 Time Dependency of a State Variable . . . 17
    - 6.1.3.2 Time Dependency of an Energy Budget . . . 17
    - 6.1.3.3 Classification of Subprocess Types . . . 19

6.2 Domain . . . 19
6.3 Axis . . . 19
6.4 Accessibility . . . 20

7 Models 21
7.1 Energy Balance Model . . . 21
7.1.1 EBM Subprocesses . . . 21
7.1.1.1 Insolation . . . 21
7.1.1.2 Albedo . . . 22
7.1.1.3 Outgoing Longwave Radiation . . . 22
7.1.1.4 Energy Transport . . . 22
7.1.2 EBM templates . . . 23
7.1.2.1 EBM . . . 23
7.1.2.2 EBM_seasonal . . . 23
7.1.2.3 EBM_annual . . . 23
7.2 Column Models . . . 23

8 Tutorials 25

9 Integration with `xarray` 27

10 CLIMLAB API reference 31
10.1 climlab.convection . . . 31
10.1.1 ConvectiveAdjustment . . . 31
10.1.2 EmanuelConvection . . . 33
10.2 climlab.domain . . . 37
10.2.1 axis . . . 37
10.2.2 domain . . . 40
10.2.3 field . . . 47
10.2.4 initial . . . 52
10.3 climlab.dynamics . . . 53
10.3.1 BudykoTransport . . . 53
10.3.2 Diffusion . . . 56
10.3.3 MeridionalDiffusion . . . 62
10.3.4 MeridionalHeatDiffusion . . . 64
10.3.5 MeridionalMoistDiffusion . . . 67
10.3.5.1 Derivation of the moist diffusion equation . . . 67
10.4 climlab.model . . . 70
10.4.1 ebm . . . 71
10.4.1.1 Building the Moist EBM . . . 71
10.4.2 column . . . 81
10.5 climlab.process . . . 85
10.5.1 diagnostic . . . 85
10.5.2 energy_budget . . . 86
10.5.3 external_forcing . . . 90
10.5.4 implicit . . . 92
10.5.5 process . . . 93
10.5.6 time_dependent_process . . . 103
10.6 climlab.radiation . . . 108
10.6.1 AplusBT . . . 108
10.6.2 Boltzmann . . . 114
10.6.3 CAM3 . . . 118
10.6.4 insolation . . . 122
10.6.5 nband . . . 132
10.6.6 Radiation . . . 138

10.6.7 RRTMG . . . 144
10.6.7.1 RRTMG . . . 144
10.6.7.2 RRTMG_LW . . . 145
10.6.7.3 RRTMG_SW . . . 147
10.6.8 SimpleAbsorbedShortwave . . . 149
10.6.9 transmissivity . . . 150
10.6.10 water_vapor . . . 152
10.7 climlab.solar . . . 154
10.7.1 insolation . . . 155
10.7.2 orbital . . . 157
10.7.3 orbital_cycles . . . 158
10.8 climlab.surface . . . 160
10.8.1 albedo . . . 160
10.8.2 climlab.surface.turbulent . . . 169
10.9 climlab.utils . . . 172
10.9.1 constants . . . 172
10.9.2 heat_capacity . . . 173
10.9.3 legendre . . . 174
10.9.4 thermo . . . 175
10.9.5 walk . . . 177
10.10 Inheritance Diagram . . . 179

11 Contributing to CLIMLAB **181**
11.1 Usage in publications, teaching, etc. . . . 181
11.2 Reporting bugs, issues, new feature requests, and documentation problems . . . 181
11.3 Seeking help and support . . . 182
11.4 Contributing bug fixes and new features . . . 182
11.5 Building and Testing CLIMLAB . . . 183
11.6 Contributing improved documentation . . . 184

12 References **185**

13 License **187**

14 Acknowledgement **189**

15 Contact **191**

CHAPTER 1

Introduction

1.1 climlab

1.1.1 Python package for process-oriented climate modeling

1.1.1.1 Author

Brian E. J. Rose
Department of Atmospheric and Environmental Sciences
University at Albany
brose@albany.edu

1.1.1.2 About climlab

`climlab` is a flexible engine for process-oriented climate modeling. It is based on a very general concept of a model as a collection of individual, interacting processes. `climlab` defines a base class called `Process`, which can contain an arbitrarily complex tree of sub-processes (each also some sub-class of `Process`). Every climate process (radiative, dynamical, physical, turbulent, convective, chemical, etc.) can be simulated as a stand-alone process model given appropriate input, or as a sub-process of a more complex model. New classes of model can easily be defined and run interactively by putting together an appropriate collection of sub-processes.

Currently, `climlab` has out-of-the-box support and documented examples for

- **Radiative and radiative-convective column models, with various radiation schemes:**
  - RRTMG (a widely used radiative transfer code)
  - CAM3 (from the NCAR GCM)
  - Grey Gas

    - Simplified band-averaged models (4 bands each in longwave and shortwave)

- **Convection schemes:**
    - Emanuel moist convection scheme
    - Hard convective adjustment (to constant lapse rate or to moist adiabat)
- Diffusion solvers for moist and dry Energy Balance Models
- Flexible insolation including: - Seasonal and annual-mean models - Arbitrary orbital parameters
- Boundary layer scheme including sensible and latent heat fluxes
- **Arbitrary combinations of the above, for example:**
    - 2D latitude-pressure models with radiation, horizontally-varying diffusion, and fixed relative humidity

1.1.1.3 Installation

Installing pre-built binaries with conda (Mac OSX, Linux, and Windows)

By far the simplest and recommended way to install `climlab` is using conda (which is the wonderful package manager that comes with Anaconda Python).

You can install `climlab` and all its dependencies with:

```
conda install -c conda-forge climlab
```

Or (recommended) add `conda-forge` to your conda channels with:

```
conda config --add channels conda-forge
```

and then simply do:

```
conda install climlab
```

Binaries are available for OSX, Linux, and Windows. You may need to update your `numpy` if you are using are using a version prior to 1.11

Installing from source

If you do not use conda, you can install `climlab` from source with:

```
pip install climlab
```

(which will download the latest stable release from the pypi repository and trigger the build process.)

Alternatively, clone the source code repository with:

```
git clone https://github.com/brian-rose/climlab.git
```

and, from the `climlab` directory, do:

```
python setup.py install
```

You will need a Fortran compiler on your system. The build has been tested with both gcc/gfortran and ifort (Linux)

Installing from source without a Fortran compiler

Many parts of `climlab` are written in pure Python and should work on any system. Fortran builds are necessary for the RRTMG and CAM3 radiation schemes and for the Emanuel convection scheme. If you follow the instructions for installing from source (above) without a valid Fortran compiler, you should find that you can still:

```
import climlab
```

and use most of the package. You will see warning messages about the missing components.

1.1.1.4 Links

- HTML documentation: http://climlab.readthedocs.io/en/latest/intro.html
- Issue tracker: http://github.com/brian-rose/climlab/issues
- Source code: http://github.com/brian-rose/climlab
- JOSS meta-paper: https://doi.org/10.21105/joss.00659

1.1.1.5 Dependencies

These are handled automatically if you install with conda.

Required

- Python 2.7, 3.5, 3.6, 3.7 (as of version 0.7.1)
- `numpy`
- `scipy`
- `xarray`
- `attrdict`

Recommended for full functionality

- `numba` (used for acceleration of some components)
- `pytest` (to run the automated tests, important if you are developing new code)

Anaconda Python is highly recommended and will provide everything you need. See "Installing pre-built binaries with conda" above.

1.1.1.6 Documentation and Examples

Full user manual is available here.

The directory `climlab/courseware/` also contains a collection of Jupyter notebooks (`*.ipynb`) used for teaching some basics of climate science, and documenting use of the `climlab` package.

These are self-describing, and should all run out-of-the-box once the package is installed, e.g:

```
jupyter notebook Insolation.ipynb
```

1.1.1.7 Release history

Version 0.7.1 (released January 2019) Deeper xarray integration, include one breaking change to `climlab.solar.orbital.OrbitalTable`, Python 3.7 compatibility, and minor enhancements.

Details:

- Removed `climlab.utils.attr_dict.AttrDict` and replaced with AttrDict package (a new dependency)
- Added `xarray` input and output capabilities for `climlab.solar.insolation.daily_insolation()`
- `climlab.solar.orbital.OrbitalTable` and `climlab.solar.orbital.long.OrbitalTable` now return `xarray.Dataset` objects containing the orbital data.
- The `lookup_parameter()` method was removed in favor of using built-in xarray interpolation.
- New class `climlab.process.ExternalForcing()` for arbitrary externally defined tendencies for state variables.
- New input option `ozone_file=None` for radiation components, sets ozone to zero.
- Tested on Python 3.7. Builds will be available through conda-forge.

Version 0.7.0 (released July 2018) New functionality, improved documentation, and a few breaking changes to the API.

Major new functionality includes convective adjustment to the moist adiabat and moist EBMs with diffusion on moist static energy gradients.

Details:

- **`climlab.convection.ConvectiveAdjustement` now allows non-constant critical lapse rates, stored in input**
  - New switches to implement automatic adjustment to **dry** and **moist** adiabats (pseudoadiabat)
- **`climlab.EBM()` and its daughter classes are significantly reorganized to better respect CLIMLAB principles:**
  - Essentially all the computations are done by subprocesses
  - SW radiation is now handled by `climlab.radiation.SimpleAbsorbedShortwave` class
  - Diffusion and its diagnostics now handled by `climlab.dynamics.MeridionalHeatDiffusion` class.
  - Diffusivity can be altered at any time by the user, e.g. during timestepping
  - Diffusivity input value `K` in class `climlab.dynamics.MeridionalDiffusion` is now specified in physical units of m2/s instead of (1/s). This is consistent with its parent class `climlab.dynamics.Diffusion`.
- A new class `climlab.dynamics.MeridionalMoistDiffusion` for the moist EBM (diffusion down moist static energy gradient)
- Tests that require compiled code are now marked with `pytest.mark.compiled` for easy exclusion during local development

Under-the-hood changes include

- Internal changes to the timestepping; the `compute()` method of every subprocess is now called explicitly.
- `compute()` now always returns tendency dictionaries

Version 0.6.5 (released April 2018) Some improved documentation, associated with publication of a meta-description paper in JOSS.

Version 0.6.4 (released February 2018) Some bug fixes and a new `climlab.couple()` method to simplify creating complete models from components.

Version 0.6.3 (released February 2018) Under-the-hood improvements to the Fortran builds which enable successful builds on a wider variety of platforms (incluing Windows/Python3).

Version 0.6.2 (released February 2018) Introduces the Emanuel moist convection scheme, support for asynchonous coupling, and internal optimzations.

Version 0.6.1 (released January 2018) Provides basic integration with xarray (convenience methods for converting climlab objects into `xarray.DataArray` and `xarray.Dataset` objects)

Version 0.6.0 (released December 2017) Provides full Python 3 compatibility, updated documentation, and minor enhancements and bug fixes.

Version 0.5.5 (released early April 2017) Finally provides easy binary distrbution with conda

Version 0.5.2 (released late March 2017) Many under-the-hood improvements to the build procedure, which should make it much easier to get *climlab* installed on user machines. Binary distribution with conda is coming soon!

Version 0.5 (released March 2017) Bug fixes and full functionality for the RRTMG radiation module, an improved common API for all radiation modules, and better documentation.

Version 0.4.2 (released January 2017) Introduces the RRTMG radiation scheme, a much-improved build process for the Fortran extension, and numerous enhancements and simplifications to the API.

Version 0.4 (released October 2016) Includes comprehensive documentation, an automated test suite, support for latitude-longitude grids, and numerous small enhancements and bug fixes.

Version 0.3 (released February 2016) Includes many internal changes and some backwards-incompatible changes (hopefully simplifications) to the public API. It also includes the CAM3 radiation module.

Version 0.2 (released January 2015) The package and its API was completely redesigned around a truly object-oriented modeling framework in January 2015.

It was used extensively for a graduate-level climate modeling course in Spring 2015: http://www.atmos.albany.edu/facstaff/brose/classes/ATM623_Spring2015/

Many more examples are found in the online lecture notes for that course: http://nbviewer.jupyter.org/github/brian-rose/ClimateModeling_courseware/blob/master/index.ipynb

Version 0.1 The first versions of the code and notebooks were originally developed in winter / spring 2014 in support of an undergraduate course at the University at Albany.

See the original course webpage at http://www.atmos.albany.edu/facstaff/brose/classes/ENV480_Spring2014/

The documentation was first created by Moritz Kreuzer (Potsdam Institut for Climate Impact Research) as part of a thesis project in Spring 2016.

1.1.1.8 Contact and Bug Reports

Users are strongly encouraged to submit bug reports and feature requests on github at https://github.com/brian-rose/climlab

1.1.1.9 License

This code is freely available under the MIT license. See the accompanying LICENSE file.

1.1.1.10 Support

Development of climlab is partially supported by the National Science Foundation under award AGS-1455071 to Brian Rose.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 2

Quickstart Guide

2.1 Installation

By far the simplest and recommended way to install `climlab` is using conda (which is the wonderful package manager that comes with Anaconda Python).

You can install `climlab` and all its dependencies with:

```
conda install -c conda-forge climlab
```

Or (recommended) add `conda-forge` to your conda channels with:

```
conda config --add channels conda-forge
```

and then simply do:

```
conda install climlab
```

Binaries are available for OSX, Linux, and Windows. You may need to update your `numpy` if you are using are using a version prior to 1.11

2.2 Single-column Radiative-Convective model

Here is a quick example of setting up a single-column Radiative-Convective model with fixed relative humidity, using the RRTMG radiation scheme:

```
import climlab
alb = 0.25
#  State variables (Air and surface temperature)
state = climlab.column_state(num_lev=30)
#  Parent model process
rcm = climlab.TimeDependentProcess(state=state)
```

(continues on next page)

(continued from previous page)

```
#  Fixed relative humidity
h2o = climlab.radiation.ManabeWaterVapor(state=state)
#  Couple water vapor to radiation
rad = climlab.radiation.RRTMG(state=state, specific_humidity=h2o.q, albedo=alb)
#  Convective adjustment
conv = climlab.convection.ConvectiveAdjustment(state=state, adj_lapse_rate=6.5)
#  Couple everything together
rcm.add_subprocess('Radiation', rad)
rcm.add_subprocess('WaterVapor', h2o)
rcm.add_subprocess('Convection', conv)
#  Run the model
rcm.integrate_years(1)
#  Check for energy balance
print rcm.ASR - rcm.OLR
```

CHAPTER 3

Installation

3.1 Installing pre-built binaries with conda (Mac OSX, Linux, and Windows)

By far the simplest and recommended way to install `climlab` is using conda (which is the wonderful package manager that comes with Anaconda Python).

You can install `climlab` and all its dependencies with:

```
conda install -c conda-forge climlab
```

Or (recommended) add `conda-forge` to your conda channels with:

```
conda config --add channels conda-forge
```

and then simply do:

```
conda install climlab
```

Binaries are available for OSX, Linux, and Windows. You may need to update your `numpy` if you are using are using a version prior to 1.11.

For Windows, builds are available for 64-bit versions of Python 3.5 and Python 3.6, and will require numpy 1.14 or later.

3.2 Installing from source

If you do not use conda, you can install `climlab` from source with:

```
pip install climlab
```

(which will download the latest stable release from the pypi repository and trigger the build process.)

Alternatively, clone the source code repository with:

```
git clone https://github.com/brian-rose/climlab.git
```

and, from the `climlab` directory, do:

```
python setup.py install
```

You will need a Fortran compiler on your system. The build has been tested with both gcc/gfortran and ifort (Linux)

3.3 Installing from source without a Fortran compiler

Many parts of `climlab` are written in pure Python and should work on any system. Fortran builds are necessary for the RRTMG and CAM3 radiation schemes. If you follow the instructions for installing from source (above) without a valid Fortran compiler, you should find that you can still:

```
import climlab
```

and use most of the package. You will see warning messages about the missing radiation components.

CHAPTER 4

Source Code

Stables releases as well as the current development version can be found on github:

- Stable Releases
- Development Version

CHAPTER 5

Dependencies

These are handled automatically if you install with conda.

5.1 Required

- Python 2.7, 3.5, 3.6, 3.7 (as of version 0.7.1)
- `numpy`
- `scipy`
- `xarray` (for data i/o)
- `attrdict`

5.2 Recommended for full functionality

- `numba` (used for acceleration of some components)
- `pytest` (to run the automated tests, important if you are developing new code)

Anaconda Python is highly recommended and will provide everything you need. See "Installing pre-built binaries with conda" above.

CHAPTER 6

Architecture

The backbone of the climlab architecture are the *Process* (page 93) and *TimeDependentProcess* (page 103) classes. All model components in *climlab* are instances of *Process* (page 93). Conceptually, a *Process* (page 93) object represents any physical mechanism that can be described in terms of one or more **state variables** and processes that modify those variables.

As all relevant procedures and events that can be modelled with *climlab* are expressed in *Processes*, they build the basic structure of the package.

For example, if you want to model the incoming solar radiation on Earth, *climlab* implements it as a *Process*, namely in the *Diagnostic Process* `_Insolation` (or one of its specific daughter classes).

Another example: the emitted energy of a surface can be computed through the `Boltzmann` class which is also a climlab *Process* and implements the Stefan Boltzmann Law for a grey body. Like that, all events and procedures that *climlab* can model are organized in *Processes*.

Note: The implementation of a whole model, for example an Energy Balance Model (*EBM* (page 72)), is also an instance of the *Process* (page 93) class in *climlab*.

For more information about models, see the climlab *Models* (page 21) chapter.

A *Process* (page 93) object contains a *subprocess* dictionary, which itself can contain an arbitraily complex collection of other *Process* (page 93) objects.

A *Process* that represents a whole model will typically have some *subprocesses* which represent specific physical components of the model, for example the albedo or the insolation component. More details about subprocesses can be found below.

The *state* variables of a *Process* are always defined on a **Domain** which itself is based on **Axes** or a single **Axis**. The following section will give a basic introduction about their role in the package, their dependencies and their implementation.

6.1 Process

A Process is an instance of the class `Process` (page 93). Most processes are time-dependent and therefore an instance of the daughter class `TimeDependentProcess` (page 103).

6.1.1 Basic Dictionaries

A *climlab.Process* object has several iterable dictionaries (`dict`) of named, gridded variables[1]:

- **`process.state`** contains the *process*' state variables, which are usually time-dependent and which are major quantities that identify the condition and status of the *process*. This can be the (surface) temperature of a model for instance.
- **`process.input`** contains boundary conditions and other gridded quantities independent of the *process*. This dictionary is often set by a parent *process*.
- **`process.param`** contains parameter of the *Process* or model. Basically, this is the same as `process.input` but with scalar entries.
- **`process.tendencies`** is an iterable dictionary of time tendencies (d/dt) for each state variable defined in `process.state`.

  Note: A non TimeDependentProcess (but instance of `Process` (page 93)) does not have this dictionary.

- **`process.diagnostics`** contains any quantity derived from the current state. In an Energy Balance Model this dictionary can have entries like `'ASR'`, `'OLR'`, `'icelat'`, `'net_radiation'`, `'albedo'` or `'insolation'`.
- **`process.subprocess`** holds subprocesses of the *process*. More about subprocesses is described below.

The *process* is fully described by contents of *state*, *input* and *param* dictionaries. *tendencies* and *diagnostics* are always computable from the current state.

6.1.2 Subprocesses

Subprocesses are representing and modeling certain components of the parent process. A model consists of many subprocesses which are usually defined on the same state variables, domains and axes as the parent process, at least partially.

Example The subprocess tree of an EBM may look like this:

```
model_EBM                #<head process>
   diffusion             #<subprocess>
   LW                    #<subprocess>
   albedo                #<subprocess>
      iceline            #<sub-subprocess>
      cold_albedo        #<sub-subprocess>
      warm_albedo        #<sub-subprocess>
   insolation            #<subprocess>
```

It can be seen that subprocesses can have subprocesses themselves, like `albedo` in this case.

A `subprocess` is similar to its `parent process` an instance of the `Process` (page 93) class. That means a `subprocess` has dictionaries and attributes with the same names as its `parent process`. Not necessary all will

[1] In the following the small written *process* refers to an instance of the `Process` (page 93) class.

be the same or have the same entries, but a `subprocess` has at least the basic dictionaries and attributes created during initialization of the *Process* (page 93) instance.

Every *subprocess* should work independently of its *parent process* given appropriate *input*.

Example Investigating an individual *process* (possibly with its own *subprocesses*) isolated from its parent can be done through:

```
newproc = climlab.process_like(procname.subprocess['subprocname'])
newproc.compute()
```

Thereby anything in the *input* dictionary of `'subprocname'` will remain fixed.

6.1.3 Process Integration over time

A *TimeDependentProcess* (page 103) can be integrated over time to see how the state variables and other diagnostic variables vary in time.

6.1.3.1 Time Dependency of a State Variable

For a state variable S which is dependendet on processes P_A, P_B, ... the time dependency can be written as

$$\frac{dS}{dt} = \underbrace{P_A(S)}_{S \text{ tendency by } P_A} + \underbrace{P_B(S)}_{S \text{ tendency by } P_B} + ...$$

When the state variable S is discretized over time like

$$\frac{dS}{dt} = \frac{\Delta S}{\Delta t} = \frac{S(t_1) - S(t_0)}{t_1 - t_0} = \frac{S_1 - S_0}{\Delta t},$$

the state tendency can be calculated through

$$\Delta S = \big[P_A(S) + P_B(S) + \ ...\big]\Delta t$$

and the new state of S after one timestep Δt is then:

$$S_1 = S_0 + \Big[\underbrace{P_A(S)}_{S \text{ tendency by } P_A} + \underbrace{P_B(S)}_{S \text{ tendency by } P_B} + ... \Big] \Delta t \, .$$

Therefore, the new state of S is calculated by multiplying the process tendencies of S with the timestep and adding them up to the previous state of S.

6.1.3.2 Time Dependency of an Energy Budget

The time dependency of an EBM energy budget is very similar to the above noted equations, just differing in a heat capacity factor C. The state variable is temperature T in this case, which is altered by subprocesses SP_A, SP_B, ...

$$\frac{dE}{dt} = C\frac{dT}{dt} = \underbrace{SP_A(T)}_{\text{heating-rate of } SP_A} + \underbrace{SP_B(T)}_{\text{heating-rate of } SP_B} + ...$$

$$\Leftrightarrow \frac{dT}{dt} = \underbrace{\frac{SP_A(T)}{C}}_{T \text{ tendency by } SP_A} + \underbrace{\frac{SP_B(T)}{C}}_{T \text{ tendency by } SP_B} + ...$$

Therefore, the new state of T after one timestep Δt can be written as:

$$T_1 = \underbrace{T_0 + \underbrace{\left[\frac{SP_A(T)}{C} + \frac{SP_B(T)}{C} + \ ... \right]}_{\text{compute()}} \Delta t}_{\text{step_forward()}}$$

The integration procedure is implemented in multiple nested function calls. The top functions for model integration are explained here, for details about computation of subprocess tendencies see *Classification of Subprocess Types* (page 19) below.

- **`compute()` (page 105) is a method that computes tendencies d/dt for all state variables**
  - it returns a dictionary of tendencies for all state variables

    Temperature tendencies are $\frac{SP_A(T)}{C}$, $\frac{SP_B(T)}{C}$, ... in this case, which are summed up like:

    $$\text{tendencies}(T) = \frac{SP_A(T)}{C} + \frac{SP_B(T)}{C} + ...$$

  - the keys for this dictionary are the same as keys of state dictionary

    As temperature T is the only state variable in this energy budget, the tendencies dictionary also just has the one key, representing the state variable T.

  - the tendency dictionary holds the total tendencies for each state including all subprocesses

    In case subprocess SP_A itself has subprocesses, their T tendencies get included in tendency computation by `compute()` (page 105).

  - the method only computes d/dt but **does not apply changes** (which is done by `step_forward()` (page 107))
  - therefore, the method is relatively independent of the numerical scheme
  - method **will update** variables in `proc.diagnostic` dictionary. Therefore, it will also **gather all diagnostics** from the *subprocesses*
- **`step_forward()` (page 107) updates the state variables**
  - it calls `compute()` (page 105) to get current tendencies
  - the method multiplies state tendencies with the timestep and adds them up to the state variables
- `integrate_years()` (page 106) etc will automate time-stepping by calling the `step_forward` (page 107) method multiple times. It also does the computation of time-average diagnostics.
- `integrate_converge()` (page 105) calls `integrate_years()` (page 106) as long as the state variables keep changing over time.

  Example Integration of a *climlab* EBM model over time can look like this:

```
import climlab
model = climlab.EBM()

# integrate the model for one year
model.integrate_years(1)
```

6.1.3.3 Classification of Subprocess Types

Processes can be classified in types: *explicit*, *implicit*, *diagnostic* and *adjustment*. This makes sense as subprocesses may have different impact on state variable tendencies (*diagnostic* processes don't have a direct influence for instance) or the way their tendencies are computed differ (*explixit* and *implicit*).

Therefore, the `compute()` (page 105) method handles them seperately as well as in specific order. It calls private `_compute()` methods that are specified in daugther classes of `Process` (page 93) namely `DiagnosticProcess` (page 85), `EnergyBudget` (page 86) (which are explicit processes) or `ImplicitProcess` (page 92).

The description of `compute()` (page 105) reveals the details how the different process types are handeled:

> The function first computes all diagnostic processes. They don't produce any tendencies directly but they may effect the other processes (such as change in solar distribution). Subsequently, all tendencies and diagnostics for all explicit processes are computed.
>
> Tendencies due to implicit and adjustment processes need to be calculated from a state that is already adjusted after explicit alteration. For that reason the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated by matrix inversions and similar to the explicit tendencies, the implicit ones are applied to the states temporarily. Subsequently, all instantaneous adjustments are computed.
>
> Then the changes that were made to the states from explicit and implicit processes are removed again as this `compute()` (page 105) function is supposed to calculate only tendencies and not apply them to the states.
>
> Finally, all calculated tendencies from all processes are collected for each state, summed up and stored in the dictionary `self.tendencies`, which is an attribute of the time-dependent-process object, for which the `compute()` (page 105) method has been called.

6.2 Domain

A *Domain* defines an area or spatial base for a climlab `Process` (page 93) object. It consists of axes which are `Axis` (page 37) objects that define the dimensions of the *Domain*.

In a *Domain* the heat capacity of grid points, bounds or cells/boxes is specified.

There are daughter classes `Atmosphere` (page 40) and `Ocean` (page 41) of the private `_Domain` (page 42) class implemented which themselves have daughter classes `SlabAtmosphere` (page 41) and `SlabOcean` (page 42).

Every `Process` (page 93) needs to be defined on a *Domain*. If none is given during initialization but latitude `lat` is specified, a default *Domain* is created.

Several methods are implemented that create *Domains* with special specifications. These are

- `single_column()` (page 45)
- `zonal_mean_column()` (page 46)
- `box_model_domain()` (page 44)

6.3 Axis

An `Axis` (page 37) is an object where information of a `_Domain` (page 42)'s spacial dimension are specified.

These include the *type* of the axis, the *number of points*, location of *points* and *bounds* on the spatial dimension, magnitude of bounds differences *delta* as well as their *unit*.

The *axes* of a `_Domain` (page 42) are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of `_Domain` (page 42).

6.4 Accessibility

For convenience with interactive work, each subprocess `'name'` should be accessible as `proc.subprocess.name` as well as the regular way through the subprocess dictionary `proc.subprocess['name']`. Note that `proc` is an instance of the `Process` (page 93) class here.

Example

```
import climlab
model = climlab.EBM()

# quick access
longwave_subp = model.subprocess.LW

# regular path
longwave_subp = model.subprocess['LW']
```

climlab will remain (as much as possible) agnostic about the data formats. Variables within the dictionaries will behave as `numpy.ndarray` objects.

Grid information and other domain details are accessible as attributes of each process. These attributes are `lat`, `lat_bounds`, `lon`, `lon_bounds`, `lev`, `lev_bounds`, `depth` and `depth_bounds`.

Example the latitude points of a *process* object that is describing an EBM model

```
import climlab
model = climlab.EBM()

# quick access
lat_points = model.lat

# regular path
lat_points = model.domains['Ts'].axes['lat'].points
```

Shortcuts like `proc.lat` will work where these are unambiguous, which means there is only a single axis of that type in the process.

Many variables will be accessible as process attributes `proc.name`. This restricts to unique field names in the above dictionaries.

Warning: There may be other dictionaries that do have name conflicts: e.g. dictionary of tendencies `proc.tendencies`, with same keys as `proc.state`.

These will **not be accessible** as `proc.name`, but **will be accessible** as `proc.dict_name.name` (as well as regular dictionary interface `proc.dict_name['name']`).

CHAPTER 7

Models

As indicated in the *Introduction* (page 1), *climlab* can implement different types of models out of the box. Here, we focus on Energy Balance Models which are refered to as EBMs.

7.1 Energy Balance Model

Currently, there are three "standard" Energy Balance Models implemented in the *climlab* code. These are `EBM` (page 72), `EBM_seasonal` (page 78) and `EBM_annual` (page 76), which are explained below.

Let's first give an overview about different (sub)processes that are implemented:

7.1.1 EBM Subprocesses

7.1.1.1 Insolation

- **`FixedInsolation` (page 128)** defines a constant solar value for all spatial points of the domain:

$$S(\varphi) = S_{\text{input}}$$

- **`P2Insolation` (page 130)** characterizes a parabolic solar distribution over the domain's latitude on the basis of the second order Legendre Polynomial P_2:

$$S(\varphi) = \frac{S_0}{4}\Big[1 + s_2 P_2\big(\sin(\varphi)\big)\Big]$$

  Variable φ represents the latitude.

- **`DailyInsolation` (page 126)** computes the daily solar insolation for each latitude of the domain on the basis of orbital parameters and astronomical formulas.
- **`AnnualMeanInsolation` (page 122)** computes a latitudewise yearly mean for solar insolation on the basis of orbital parameters and astronomical formulas.

7.1.1.2 Albedo

- *ConstantAlbedo* **(page 160)** defines constant albedo values at all spatial points of the domain:

$$\alpha(\varphi) = a_0$$

- *P2Albedo* **(page 164)** initializes parabolic distributed albedo values across the domain on basis of the second order Legendre Polynomial P_2:

$$\alpha(\varphi) = a_0 + a_2 P_2\big(\sin(\varphi)\big)$$

- *Iceline* **(page 162)** determines which part of the domain is covered with ice according to a given freezing temperature.
- *StepFunctionAlbedo* **(page 166)** implements an albedo step function in dependence of the surface temperature by using instances of the above described albedo classes as subprocesses.

7.1.1.3 Outgoing Longwave Radiation

- **AplusBT** calculates the Outgoing Longwave Radiation (OLR) in form of a linear dependence of surface temperature T:

$$\text{OLR} = A + B \cdot T$$

- **AplusBT_CO2** calculates OLR in the same way as `AplusBT` but uses parameters A and B dependent of the atmospheric CO_2 concentration c.

$$\text{OLR} = A(c) + B(c) \cdot T$$

- **Boltzmann** calculates OLR according to the Stefan-Boltzmann law for a grey body:

$$\text{OLR} = \sigma\varepsilon T^4$$

7.1.1.4 Energy Transport

These classes calculate the transport of energy $H(\varphi)$ across the latitude φ in an energy budget noted as:

$$C(\varphi)\frac{dT(\varphi)}{dt} = R\downarrow(\varphi) - R\uparrow(\varphi) + H(\varphi)$$

- **MeridionalDiffusion** calculates the energy transport in a diffusion like process along the temperature gradient:

$$H(\varphi) = \frac{D}{\cos\varphi}\frac{\partial}{\partial\varphi}\left(\cos\varphi\frac{\partial T(\varphi)}{\partial\varphi}\right)$$

- *BudykoTransport* **(page 53)** calculates the energy transport for each latitude φ depending on the global mean temperature $\bar{T}$:

$$H(\varphi) = -b[T(\varphi) - \bar{T}]$$

7.1.2 EBM templates

The preconfigured Energy Balance Models *EBM* (page 23), *EBM_seasonal* (page 23) and *EBM_annual* (page 23) use the described suprocesses above:

7.1.2.1 EBM

The `EBM` (page 72) class sets up a typical Energy Balance Model with following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via `AplusBT`
- solar insolation paramterization via `P2Insolation` (page 130)
- albedo parametrization in dependence of temperature via `StepFunctionAlbedo` (page 166)
- energy diffusion via `MeridionalDiffusion`

7.1.2.2 EBM_seasonal

The `EBM_seasonal` (page 78) class implements Energy Balance Models with realistic daily insolation. It uses following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via `AplusBT`
- solar insolation paramterization via `DailyInsolation` (page 126)
- albedo parametrization in dependence of temperature via `StepFunctionAlbedo` (page 166)
- energy diffusion via `MeridionalDiffusion`

7.1.2.3 EBM_annual

The `EBM_annual` (page 76) class that implements Energy Balance Models with annual mean insolation. It uses following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via `AplusBT`
- solar insolation paramterization via `AnnualMeanInsolation` (page 122)
- albedo parametrization in dependence of temperature via `StepFunctionAlbedo` (page 166)
- energy diffusion via `MeridionalDiffusion`

7.2 Column Models

Information on column models located in `column` (page 81) in the `Climlab Reference`.

Note: For information how to set up individual models or modify instances of the classes above, see the *Tutorials* (page 25) chapter.

CHAPTER 8

Tutorials

Below is a collection of Jupyter notebooks giving example usage of the `climlab` package. These notebooks are based on classroom material developed by Brian Rose at the University at Albany. For a more comprehensive set of examples and graduate-level lecture notes, click here.

The notebooks can be viewed on nbviewer through the links below. If you want to run the code yourself, these notes are all available as Jupyter *.ipynb files in the `courseware` directory of the climlab source.

- Basic EBM
- EBM with Boltzmann radiation
- EBM with Budkyko transport
- Snowball Earth in the EBM
- Insolation
- The seasonal cycle of surface temperature
- Vertical structure of air temperature
- The spectral band model
- Grey Radiation models
- Radiative-Convective Equilibrium with the CAM3 radiation model
- Models of polar amplification

CHAPTER 9

Integration with xarray

xarray is a powerful Python package for geospatial data analysis. It provides `DataArray` and `Dataset` structures for self-describing gridded data.

For the convenience of xarray users, climlab provides tools for automatic translation of the native `Field` object to xarray format.

Additionally, as of climlab v0.7.1, the insolation and orbital functions have been updated with an xarray-compatible interface:

- `climlab.solar.orbital.OrbitalTable` returns an `xarray.Dataset` object with orbital data.
- `climlab.solar.insolation.daily_insolation` accepts input in labeled `xarray.DataArray` format and return the same.

Example 1 Create a single column radiation model and view air temperature as `xarray.DataArray`:

```
>>> import climlab
>>> state = climlab.column_state(num_lev=20)
>>> model = climlab.radiation.RRTMG(state=state)

>>> # display a single variable as xarray.DataArray
>>> model.Tatm.to_xarray()
<xarray.DataArray (lev: 20)>
array([ 200.      ,  204.105263,  208.210526,  212.315789,  216.421053,
        220.526316,  224.631579,  228.736842,  232.842105,  236.947368,
        241.052632,  245.157895,  249.263158,  253.368421,  257.473684,
        261.578947,  265.684211,  269.789474,  273.894737,  278.      ])
Coordinates:
  * lev      (lev) float64 25.0 75.0 125.0 175.0 225.0 275.0 325.0 375.0␣
↪...
```

Example 2 Display the entire model state dictionary as `xarray.Dataset`:

```
>>> model.to_xarray()
<xarray.Dataset>
Dimensions:       (depth: 1, depth_bounds: 2, lev: 20, lev_bounds: 21)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * depth          (depth) float64 0.5
  * depth_bounds   (depth_bounds) float64 0.0 1.0
  * lev            (lev) float64 25.0 75.0 125.0 175.0 225.0 275.0 325.0 .
↪..
  * lev_bounds     (lev_bounds) float64 0.0 50.0 100.0 150.0 200.0 250.0 .
↪..
Data variables:
    Ts             (depth) float64 288.0
    Tatm           (lev) float64 200.0 204.1 208.2 212.3 216.4 220.5 224.
↪6 ...
```

Example 3 Combine model state and diagnostics into a single `xarray.Dataset`:

```
>>> # take a single timestep to populate the diagnostic variables
>>> model.step_forward()

>>> # Now look at the full output in xarray format
>>> model.to_xarray(diagnostics=True)
<xarray.Dataset>
Dimensions:            (depth: 1, depth_bounds: 2, lev: 20, lev_bounds:␣
↪21)
Coordinates:
  * depth              (depth) float64 0.5
  * depth_bounds       (depth_bounds) float64 0.0 1.0
  * lev                (lev) float64 25.0 75.0 125.0 175.0 225.0 275.0␣
↪325.0 ...
  * lev_bounds         (lev_bounds) float64 0.0 50.0 100.0 150.0 200.0␣
↪250.0 ...
Data variables:
    Ts                 (depth) float64 288.7
    Tatm               (lev) float64 201.3 204.0 208.0 212.0 216.1 220.2 .
↪..
    ASR                (depth) float64 240.0
    ASRcld             (depth) float64 0.0
    ASRclr             (depth) float64 240.0
    LW_flux_down       (lev_bounds) float64 0.0 12.63 19.47 26.07 32.92␣
↪40.1 ...
    LW_flux_down_clr   (lev_bounds) float64 0.0 12.63 19.47 26.07 32.92␣
↪40.1 ...
    LW_flux_net        (lev_bounds) float64 240.1 231.2 227.6 224.1 220.5␣
↪...
    LW_flux_net_clr    (lev_bounds) float64 240.1 231.2 227.6 224.1 220.5␣
↪...
    LW_flux_up         (lev_bounds) float64 240.1 243.9 247.1 250.2 253.4␣
↪...
    LW_flux_up_clr     (lev_bounds) float64 240.1 243.9 247.1 250.2 253.4␣
↪...
    LW_sfc             (depth) float64 128.9
    LW_sfc_clr         (depth) float64 128.9
    OLR                (depth) float64 240.1
    OLRcld             (depth) float64 0.0
    OLRclr             (depth) float64 240.1
    SW_flux_down       (lev_bounds) float64 341.3 323.1 318.0 313.5 309.5␣
↪...
    SW_flux_down_clr   (lev_bounds) float64 341.3 323.1 318.0 313.5 309.5␣
↪...
```

(continues on next page)

(continued from previous page)

```
    SW_flux_net        (lev_bounds) float64 240.0 223.3 220.2 217.9 215.9␣
↪...
    SW_flux_net_clr    (lev_bounds) float64 240.0 223.3 220.2 217.9 215.9␣
↪...
    SW_flux_up         (lev_bounds) float64 101.3 99.88 97.77 95.64 93.57␣
↪...
    SW_flux_up_clr     (lev_bounds) float64 101.3 99.88 97.77 95.64 93.57␣
↪...
    SW_sfc             (depth) float64 163.8
    SW_sfc_clr         (depth) float64 163.8
    TdotLW             (lev) float64 -1.502 -0.6148 -0.5813 -0.6173 -0.
↪6426 ...
    TdotLW_clr         (lev) float64 -1.502 -0.6148 -0.5813 -0.6173 -0.
↪6426 ...
    TdotSW             (lev) float64 2.821 0.5123 0.3936 0.3368 0.3174 0.
↪3299 ...
    TdotSW_clr         (lev) float64 2.821 0.5123 0.3936 0.3368 0.3174 0.
↪3299 ...
```

Example 4 Use the `climlab.to_xarray()` method to convert the `timeave` dictionary to `xarray.Dataset`:

```
>>> # integrate forward one year and automatically store time averages
>>> model.integrate_years(1)
Integrating for 365 steps, 365.2422 days, or 1 years.
Total elapsed time is 0.9993368783782377 years.

>>> # Now look at model.timeave dictionary in xarray format
>>> climlab.to_xarray(model.timeave)
<xarray.Dataset>
Dimensions:           (depth: 1, depth_bounds: 2, lev: 20, lev_bounds:␣
↪21)
Coordinates:
  * depth             (depth) float64 0.5
  * depth_bounds      (depth_bounds) float64 0.0 1.0
  * lev               (lev) float64 25.0 75.0 125.0 175.0 225.0 275.0␣
↪325.0 ...
  * lev_bounds        (lev_bounds) float64 0.0 50.0 100.0 150.0 200.0␣
↪250.0 ...
Data variables:
    Ts                (depth) float64 296.9
    Tatm              (lev) float64 217.1 203.1 200.8 200.4 201.7 204.2 .
↪..
    ASR               (depth) float64 240.1
    ASRcld            (depth) float64 0.0
    ASRclr            (depth) float64 240.1
    LW_flux_down      (lev_bounds) float64 0.0 16.55 20.24 24.12 28.15␣
↪32.57 ...
    LW_flux_down_clr  (lev_bounds) float64 0.0 16.55 20.24 24.12 28.15␣
↪32.57 ...
    LW_flux_net       (lev_bounds) float64 243.0 226.5 223.4 221.0 218.8␣
↪...
    LW_flux_net_clr   (lev_bounds) float64 243.0 226.5 223.4 221.0 218.8␣
↪...
    LW_flux_up        (lev_bounds) float64 243.0 243.0 243.7 245.1 246.9␣
↪...
```

(continues on next page)

(continued from previous page)

```
    LW_flux_up_clr     (lev_bounds) float64 243.0 243.0 243.7 245.1 246.9␣
↪...
    LW_sfc             (depth) float64 162.5
    LW_sfc_clr         (depth) float64 162.5
    OLR                (depth) float64 243.0
    OLRcld             (depth) float64 0.0
    OLRclr             (depth) float64 243.0
    SW_flux_down       (lev_bounds) float64 341.3 323.1 317.9 313.5 309.5␣
↪...
    SW_flux_down_clr   (lev_bounds) float64 341.3 323.1 317.9 313.5 309.5␣
↪...
    SW_flux_net        (lev_bounds) float64 240.1 223.3 220.3 217.9 216.0␣
↪...
    SW_flux_net_clr    (lev_bounds) float64 240.1 223.3 220.3 217.9 216.0␣
↪...
    SW_flux_up         (lev_bounds) float64 101.2 99.81 97.69 95.56 93.5 .
↪..
    SW_flux_up_clr     (lev_bounds) float64 101.2 99.81 97.69 95.56 93.5 .
↪..
    SW_sfc             (depth) float64 163.7
    SW_sfc_clr         (depth) float64 163.7
    TdotLW             (lev) float64 -2.789 -0.5133 -0.4154 -0.3732 -0.
↪3626 ...
    TdotLW_clr         (lev) float64 -2.789 -0.5133 -0.4154 -0.3732 -0.
↪3626 ...
    TdotSW             (lev) float64 2.836 0.5078 0.3898 0.3332 0.3138 0.
↪3267 ...
    TdotSW_clr         (lev) float64 2.836 0.5078 0.3898 0.3332 0.3138 0.
↪3267 ...
```

CHAPTER 10

CLIMLAB API reference

This chapter documents the source code of the `climlab` package. The focus is on the methods and functions that the user invokes while using the package.

Nevertheless also the underlying code of the `climlab` architecture has been documented for a comprehensive understanding and traceability.

10.1 climlab.convection

Modules for atmospheric convection.

For simple adjustment of temperature to a prescribed lapse rate, use `ConvectiveAdjustment`

For a full convection scheme including interactive water vapor, use `EmanuelConvection`

10.1.1 ConvectiveAdjustment

class climlab.convection.convadj.**ConvectiveAdjustment**(*adj_lapse_rate=None*, ***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

Hard Convective Adjustment to a prescribed lapse rate.

This process computes the instantaneous adjustment to conservatively remove any instabilities in each column.

Instability is defined as a temperature decrease with height that exceeds the prescribed critical lapse rate. This critical rate is set by input argument `adj_lapse_rate`, which can be either a numerical or string value.

Numerical values for `adj_lapse_rate` are given in units of K / km. Both array and scalar values are valid. For scalar values, the assumption is that the critical lapse rate is the same at every level.

If an array is given, it is assumed to represent the in-situ critical lapse rate (in K/km) at every grid point.

Alternatively, string arguments can be given as follows:

- `'DALR'` or `'dry adiabat'`: critical lapse rate is set to g/cp = 9.8 K / km
- `'MALR'` or `'moist adiabat'` or `'pseudoadiabat'`: critical lapse rate follows the in-situ moist pseudoadiabat at every level

Adjustment includes the surface if `'Ts'` is included in the `state` dictionary. This implicitly accounts for turbulent surface fluxes. Otherwise only the atmospheric temperature is adjusted.

If `adj_lapse_rate` is an array, its size must match the number of vertical levels of the adjustment. This is number of pressure levels if the surface is not adjusted, or number of pressure levels + 1 if the surface is adjusted.

This process implements the conservative adjustment algorithm described in Akmaev (1991) Monthly Weather Review.

Attributes

Tcol

adj_lapse_rate

ccol

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

pcol

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.

Continued on next page

Table 1 – continued from previous page

`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

Tcol

adj_lapse_rate

ccol

pcol

10.1.2 EmanuelConvection

A climlab process for the Emanuel convection scheme

class climlab.convection.emanuel_convection.**EmanuelConvection**(*MINORIG=0*, *ELCRIT=0.0011*, *TLCRIT=-55.0*, *ENTP=1.5*, *SIGD=0.05*, *SIGS=0.12*, *OMTRAIN=50.0*, *OMTSNOW=5.5*, *COEFFR=1.0*, *COEFFS=0.8*, *CU=0.7*, *BETA=10.0*, *DTMAX=0.9*, *ALPHA=0.2*, *DAMP=0.1*, *IPBL=0*, ***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

The climlab wrapper for Kerry Emanuel's moist convection scheme <https://emanuel.mit.edu/FORTRAN-subroutine-convect>

From the documentation distributed with the Fortran 77 code CONVECT:

> The subroutine is designed to be used in time-marching models of mesoscale to global-scale dimensions. It is meant to represent the effects of all moist convection, including shallow, non-precipitating cumulus. It also contains a dry adiabatic adjustment scheme.
>
> Since the method of calculating the convective fluxes involves a relaxation toward quasi-equilibrium, subroutine CONVECT must be run for at least several time steps to give meaningful results. At the first time step, the tendencies and convective precipitation will be zero. If the initial sounding is unstable, these will rapidly increase over successive time steps, depending on the values of the constants ALPHA and DAMP. Thus the user interested in convective fluxes and precipitation associated with a single initial sounding (i.e., without large-scale forcing) should still march CONVECT forward enough time steps that the fluxes have returned back to zero; the net tendencies and precipitation integrated over this time interval are then the desired results. But it should be cautioned that these quantities will not necessarily be independent of other model parameters such as the time step. CONVECT is very much built on the philosophy that convection, to the extent it can be represented in terms of large-scale variables, is never very far away from statistical equilibrium with the large-scale flow. To achieve a smooth evolution of the convective forcing, CONVECT should be called at least every 20 minutes during the time integration. CONVECT will work at longer time intervals, but the convective tendencies may become noisy.

Basic characteristics:

State:

- Ts (surface radiative temperature – optional, and ignored)
- Tatm (air temperature in K)
- q (specific humidity in kg/kg)
- U (zonal velocity in m/s – optional)
- V (meridional velocity in m/s – optional)

Input arguments and default values (taken from convect43.f fortran source):

- MINORIG = 0, index of lowest level from which convection may originate (zero means lowest)

- ELCRIT = 0.0011, autoconversion threshold water content (g/g)
- TLCRIT = -55.0, critical temperature below which the auto-conversion threshold is assumed to be zero (the autoconversion threshold varies linearly between 0 C and TLCRIT)
- ENTP = 1.5, coefficient of mixing in the entrainment formulation
- SIGD = 0.05, fractional area covered by unsaturated downdraft
- SIGS = 0.12, fraction of precipitation falling outside of cloud
- OMTRAIN = 50.0, assumed fall speed (Pa/s) of rain
- OMTSNOW = 5.5, assumed fall speed (Pa/s) of snow
- COEFFR = 1.0, coefficient governing the rate of evaporation of rain
- COEFFS = 0.8, coefficient governing the rate of evaporation of snow
- CU = 0.7, coefficient governing convective momentum transport
- BETA = 10.0, coefficient used in downdraft velocity scale calculation
- DTMAX = 0.9, maximum negative temperature perturbation a lifted parcel is allowed to have below its LFC
- ALPHA = 0.2, first parameter that controls the rate of approach to quasi-equilibrium
- DAMP = 0.1, second parameter that controls the rate of approach to quasi-equilibrium (DAMP must be less than 1)
- IPBL = 0, switch to bypass the dry convective adjustment (bypass if IPBL==0)

Tendencies computed:

- air temperature (K/s)
- specific humidity (kg/kg/s)
- **optional:**
  - U and V wind components (m/s/s), if U and V are included in state dictionary

Diagnostics computed:

- CBMF (cloud base mass flux in kg/m2/s) – this is actually stored internally and used as input for subsequent timesteps
- PRECIP (convective precipitation rate in mm/day)

Example Here is an example of setting up a single-column Radiative-Convective model with interactive water vapor.

This example also demonstrates *asynchronous coupling*: the radiation uses a longer timestep than the other model components:

```
import climlab
from climlab import constants as const
# Temperatures in a single column
full_state = climlab.column_state(num_lev=30, water_depth=2.5)
temperature_state = {'Tatm':full_state.Tatm,'Ts':full_state.Ts}
#  Initialize a nearly dry column (small background␣
↪stratospheric humidity)
q = np.ones_like(full_state.Tatm) * 5.E-6
#  Add specific_humidity to the state dictionary
full_state['q'] = q
```

(continues on next page)

(continued from previous page)

```
#  ASYNCHRONOUS COUPLING -- the radiation uses a much longer␣
↪timestep
#  The top-level model
model = climlab.TimeDependentProcess(state=full_state,
                              timestep=const.seconds_per_hour)
#  Radiation coupled to water vapor
rad = climlab.radiation.RRTMG(state=temperature_state,
                              specific_humidity=full_state.q,
                              albedo=0.3,
                              timestep=const.seconds_per_day
                              )
#  Convection scheme -- water vapor is a state variable
conv = climlab.convection.EmanuelConvection(state=full_state,
                              timestep=const.seconds_per_hour)
#  Surface heat flux processes
shf = climlab.surface.SensibleHeatFlux(state=temperature_state,␣
↪Cd=0.5E-3,
                              timestep=const.seconds_per_hour)
lhf = climlab.surface.LatentHeatFlux(state=full_state, Cd=0.5E-
↪3,
                              timestep=const.seconds_per_hour)
#  Couple all the submodels together
model.add_subprocess('Radiation', rad)
model.add_subprocess('Convection', conv)
model.add_subprocess('SHF', shf)
model.add_subprocess('LHF', lhf)
print(model)

#  Run the model
model.integrate_years(1)
#  Check for energy balance
print(model.ASR - model.OLR)
```

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.2 climlab.domain

Modules for self-describing gridded fields in climlab.

10.2.1 axis

Axis

class `climlab.domain.axis.`**`Axis`**(*axis_type='abstract'*, *num_points=10*, *points=None*, *bounds=None*)

Bases: `object`

Creates a new climlab Axis object.

An *Axis* (page 37) is an object where information of a spacial dimension of a *_Domain* (page 42) are specified.

These include the *type* of the axis, the *number of points*, location of *points* and *bounds* on the spatial dimension, magnitude of bounds differences *delta* as well as their *unit*.

The *axes* of a *_Domain* (page 42) are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of *_Domain* (page 42).

Initialization parameters

An instance of `Axis` is initialized with the following arguments *(for detailed information see Object attributes below)*:

Parameters

- **axis_type** (*str*) – information about the type of axis [default: 'abstract']
- **num_points** (*int*) – number of points on axis [default: 10]
- **points** (*array*) – array with specific points (optional)
- **bounds** (*array*) – array with specific bounds between points (optional)

Raises `ValueError` if `axis_type` is not one of the valid types or their euqivalents (see below).

Raises `ValueError` if `points` are given and not array-like.

Raises `ValueError` if `bounds` are given and not array-like.

Object attributes

Following object attributes are generated during initialization:

Variables

- **axis_type** (*str*) – Information about the type of axis. Valid axis types are:
  - `'lev'`
  - `'lat'`
  - `'lon'`
  - `'depth'`
  - `'abstract'` (default)
- **num_points** (*int*) – number of points on axis
- **units** (*str*) – Unit of the axis. During intialization the unit is chosen from the `defaultUnits` dictionary (see below).
- **points** (*array*) – array with all points of the axis (grid)
- **bounds** (*array*) – array with all bounds between points (staggered grid)
- **delta** (*array*) – array with spatial differences between bounds

Axis Types

A couple of differing axis type strings are rendered to valid axis types. Alternate forms are listed here:

- **'lev'**
  - `'p'`
  - `'press'`
  - `'pressure'`

  - 'P'
  - 'Pressure'
  - 'Press'
- **'lat'**
  - 'Latitude'
  - 'latitude'
- **'lon'**
  - 'Longitude'
  - 'longitude'
- **'depth'**
  - 'Depth'
  - 'waterDepth'
  - 'water_depth'
  - 'slab'

The **default units** are:

```
defaultUnits = {'lev': 'mb',
                'lat': 'degrees',
                'lon': 'degrees',
                'depth': 'meters',
                'abstract': 'none'}
```

If bounds are not given during initialization, **default end points** are used:

```
defaultEndPoints = {'lev': (0., climlab.constants.ps),
                    'lat': (-90., 90.),
                    'lon': (0., 360.),
                    'depth': (0., 10.),
                    'abstract': (0, num_points)}
```

Example Creation of a standalone Axis:

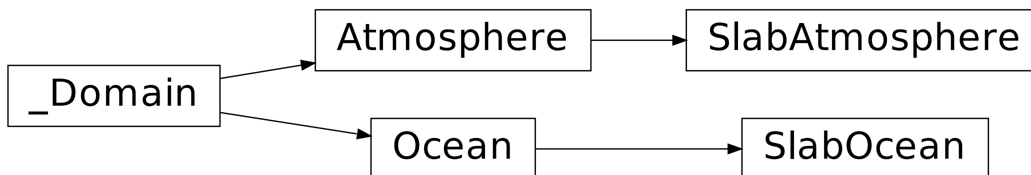
```
>>> import climlab
>>> ax = climlab.domain.Axis(axis_type='Latitude', num_points=36)

>>> print ax
Axis of type lat with 36 points.

>>> ax.points
array([-87.5, -82.5, -77.5, -72.5, -67.5, -62.5, -57.5, -52.5, -47.5,
       -42.5, -37.5, -32.5, -27.5, -22.5, -17.5, -12.5,  -7.5,  -2.5,
         2.5,   7.5,  12.5,  17.5,  22.5,  27.5,  32.5,  37.5,  42.5,
        47.5,  52.5,  57.5,  62.5,  67.5,  72.5,  77.5,  82.5,  87.5])

>>> ax.bounds
array([-90., -85., -80., -75., -70., -65., -60., -55., -50., -45., -
↪40.,
       -35., -30., -25., -20., -15., -10.,  -5.,   0.,   5.,  10., ␣
↪15.,
```

(continues on next page)

(continued from previous page)

```
        20.,   25.,   30.,   35.,   40.,   45.,   50.,   55.,   60.,   65.,  ␣
↪70.,
        75.,   80.,   85.,   90.])

>>> ax.delta
array([ 5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.
↪,
        5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.
↪,
        5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.,   5.])
```

10.2.2 domain



class climlab.domain.domain.**Atmosphere**(***kwargs*)

Bases: *climlab.domain.domain._Domain* (page 42)

Class for the implementation of an Atmosphere Domain.

Object attributes

Additional to the parent class *_Domain* (page 42) the following object attribute is modified during initialization:

Variables domain_type (*str*) – is set to 'atm'

Example Setting up an Atmosphere Domain:

```
>>> import climlab
>>> atm_ax = climlab.domain.Axis(axis_type='pressure', num_points=10)
>>> atm_domain = climlab.domain.Atmosphere(axes=atm_ax)

>>> print atm_domain
climlab Domain object with domain_type=atm and shape=(10,)

>>> atm_domain.axes
{'lev': <climlab.domain.axis.Axis object at 0x7fe5b8ef8e10>}

>>> atm_domain.heat_capacity
array([ 1024489.79591837,  1024489.79591837,  1024489.79591837,
        1024489.79591837,  1024489.79591837,  1024489.79591837,
        1024489.79591837,  1024489.79591837,  1024489.79591837,
        1024489.79591837])
```

Methods

set_heat_capacity (page 41)()	Sets the heat capacity of the Atmosphere Domain.

set_heat_capacity()
Sets the heat capacity of the Atmosphere Domain.

Calls the utils heat capacity function *atmosphere()* (page 173) and gives the delta array of grid points of it's level axis `self.axes['lev'].delta` as input.

Object attributes

During method execution following object attribute is modified:

Variables ***heat_capacity*** (page 173) (*array*) – the ocean domain's heat capacity over the `'lev'` Axis.

class climlab.domain.domain.**Ocean**(***kwargs*)
Bases: *climlab.domain.domain._Domain* (page 42)

Class for the implementation of an Ocean Domain.

Object attributes

Additional to the parent class *_Domain* (page 42) the following object attribute is modified during initialization:

Variables domain_type (*str*) – is set to `'ocean'`

Example Setting up an Ocean Domain:

```
>>> import climlab
>>> ocean_ax = climlab.domain.Axis(axis_type='depth', num_points=5)
>>> ocean_domain = climlab.domain.Ocean(axes=ocean_ax)

>>> print ocean_domain
climlab Domain object with domain_type=ocean and shape=(5,)

>>> ocean_domain.axes
{'depth': <climlab.domain.axis.Axis object at 0x7fe5b8f102d0>}

>>> ocean_domain.heat_capacity
array([ 8362600.,  8362600.,  8362600.,  8362600.,  8362600.])
```

Methods

set_heat_capacity (page 41)()	Sets the heat capacity of the Ocean Domain.

set_heat_capacity()
Sets the heat capacity of the Ocean Domain.

Calls the utils heat capacity function *ocean()* (page 174) and gives the delta array of grid points of it's depth axis `self.axes['depth'].delta` as input.

Object attributes

During method execution following object attribute is modified:

Variables ***heat_capacity*** (page 173) (*array*) – the ocean domain's heat capacity over the `'depth'` Axis.

class climlab.domain.domain.**SlabAtmosphere**(*axes=<climlab.domain.axis.Axis object>*, ***kwargs*)

Bases: `climlab.domain.domain.Atmosphere` (page 40)

A class to create a SlabAtmosphere Domain by default.

Initializes the parent `Atmosphere` (page 40) class with a simple axis for a Slab Atmopshere created by `make_slabatm_axis()` (page 44) which has just 1 cell in height by default.

Example Creating a SlabAtmosphere Domain:

```
>>> import climlab
>>> slab_atm_domain = climlab.domain.SlabAtmosphere()

>>> print slab_atm_domain
climlab Domain object with domain_type=atm and shape=(1,)

>>> slab_atm_domain.axes
{'lev': <climlab.domain.axis.Axis object at 0x7fe5c4281610>}

>>> slab_atm_domain.heat_capacity
array([ 10244897.95918367])
```

Methods

set_heat_capacity()	Sets the heat capacity of the Atmosphere Domain.

class climlab.domain.domain.**SlabOcean**(*axes=<climlab.domain.axis.Axis object>*, ***kwargs*)

Bases: `climlab.domain.domain.Ocean` (page 41)

A class to create a SlabOcean Domain by default.

Initializes the parent `Ocean` (page 41) class with a simple axis for a Slab Ocean created by `make_slabocean_axis()` (page 44) which has just 1 cell in depth by default.

Example Creating a SlabOcean Domain:

```
>>> import climlab
>>> slab_ocean_domain = climlab.domain.SlabOcean()

>>> print slab_ocean_domain
climlab Domain object with domain_type=ocean and shape=(1,)

>>> slab_ocean_domain.axes
{'depth': <climlab.domain.axis.Axis object at 0x7fe5c42814d0>}

>>> slab_ocean_domain.heat_capacity
array([ 41813000.])
```

Methods

set_heat_capacity()	Sets the heat capacity of the Ocean Domain.

class climlab.domain.domain.**_Domain**(*axes=None*, ***kwargs*)

Bases: `object`

Private parent class for *Domains*.

A *Domain* defines an area or spatial base for a climlab `Process` (page 93) object. It consists of axes which are `Axis` (page 37) objects that define the dimensions of the *Domain*.

In a *Domain* the heat capacity of grid points, bounds or cells/boxes is specified.

There are daughter classes `Atmosphere` (page 40) and `Ocean` (page 41) of the private `_Domain` (page 42) class implemented which themselves have daughter classes `SlabAtmosphere` (page 41) and `SlabOcean` (page 42).

Several methods are implemented that create *Domains* with special specifications. These are

- `single_column()` (page 45)
- `zonal_mean_column()` (page 46)
- `box_model_domain()` (page 44)

Initialization parameters

An instance of `_Domain` is initialized with the following arguments:

Parameters **axes** (dict or `Axis` (page 37)) – Axis object or dictionary of Axis object where domain will be defined on.

Object attributes

Following object attributes are generated during initialization:

Variables

- **domain_type** (*str*) – Set to `'undefined'`.
- **axes** (*dict*) – A dictionary of the domains axes. Created by `_make_axes_dict()` (page 43) called with input argument `axes`
- **numdims** (*int*) – Number of `Axis` (page 37) objects in `self.axes` dictionary.
- **ax_index** (*dict*) – A dictionary of domain axes and their corresponding index in an ordered list of the axes with:
  - `'lev'` or `'depth'` is last
  - `'lat'` is second last
- **shape** (*tuple*) – Number of points of all domain axes. Order in tuple given by `self.ax_index`.
- **heat_capacity** (page 173) (*array*) – the domain's heat capacity over axis specified in function call of `set_heat_capacity()` (page 44)

Methods

`set_heat_capacity` (page 44)()	A dummy function to set the heat capacity of a domain.

_make_axes_dict(*axes*)

Makes an axes dictionary.

Note: In case the input is `None`, the dictionary `{'empty': None}` is returned.

Function-call argument

Parameters **axes** (dict or single instance of *Axis* (page 37) object or None) – axes input

Raises ValueError if input is not an instance of Axis class or a dictionary of Axis objetcs

Returns dictionary of input axes

Return type dict

set_heat_capacity()
A dummy function to set the heat capacity of a domain.

Should be overridden by daugter classes.

climlab.domain.domain.**box_model_domain**(*num_points=2*, ***kwargs*)
Creates a box model domain (a single abstract axis).

Parameters **num_points** (*int*) – number of boxes [default: 2]

Returns Domain with single axis of type 'abstract' and self.domain_type = 'box'

Return type *_Domain* (page 42)

Example

```
>>> from climlab import domain
>>> box = domain.box_model_domain(num_points=2)

>>> print box
climlab Domain object with domain_type=box and shape=(2,)
```

climlab.domain.domain.**make_slabatm_axis**(*num_points=1*)
Convenience method to create a simple axis for a slab atmosphere.

Function-call argument

Parameters **num_points** (*int*) – number of points for the slabatmosphere Axis [default: 1]

Returns an Axis with axis_type='lev' and num_points=num_points

Return type *Axis* (page 37)

Example

```
>>> import climlab
>>> slab_atm_axis = climlab.domain.make_slabatm_axis()

>>> print slab_atm_axis
Axis of type lev with 1 points.

>>> slab_atm_axis.axis_type
'lev'

>>> slab_atm_axis.bounds
array([    0.,   1000.])

>>> slab_atm_axis.units
'mb'
```

climlab.domain.domain.**make_slabocean_axis**(*num_points=1*)
Convenience method to create a simple axis for a slab ocean.

Function-call argument

Parameters **num_points** (*int*) – number of points for the slabocean Axis [default: 1]

Returns an Axis with `axis_type='depth'` and `num_points=num_points`

Return type *Axis* (page 37)

Example

```
>>> import climlab
>>> slab_ocean_axis = climlab.domain.make_slabocean_axis()

>>> print slab_ocean_axis
Axis of type depth with 1 points.

>>> slab_ocean_axis.axis_type
'depth'

>>> slab_ocean_axis.bounds
array([  0.,   10.])

>>> slab_ocean_axis.units
'meters'
```

climlab.domain.domain.**single_column**(*num_lev=30*, *water_depth=1.0*, *lev=None*, ***kwargs*)

Creates domains for a single column of atmosphere overlying a slab of water.

Can also pass a pressure array or pressure level axis object specified in `lev`.

If argument `lev` is not `None` then function tries to build a level axis and `num_lev` is ignored.

Function-call argument

Parameters

- **num_lev** (*int*) – number of pressure levels (evenly spaced from surface to TOA) [default: 30]
- **water_depth** (*float*) – depth of the ocean slab [default: 1.]
- **lev** (*Axis* (page 37) or pressure array) – specification for height axis (optional)

Raises `ValueError` if *lev* is given but neither Axis nor pressure array.

Returns a list of 2 Domain objects (slab ocean, atmosphere)

Return type `list` of *SlabOcean* (page 42), *SlabAtmosphere* (page 41)

Example

```
>>> from climlab import domain

>>> sfc, atm = domain.single_column(num_lev=2, water_depth=10.)

>>> print sfc
climlab Domain object with domain_type=ocean and shape=(1,)

>>> print atm
climlab Domain object with domain_type=atm and shape=(2,)
```

climlab.domain.domain.**surface_2D**(*num_lat=90*, *num_lon=180*, *water_depth=10.0*, *lon=None*, *lat=None*, ***kwargs*)

Creates a 2D slab ocean Domain in latitude and longitude with uniform water depth.

Domain has a single heat capacity according to the specified water depth.

Function-call argument

Parameters

- **num_lat** (*int*) – number of latitude points [default: 90]
- **num_lon** (*int*) – number of longitude points [default: 180]
- **water_depth** (*float*) – depth of the slab ocean in meters [default: 10.]
- **lat** (*Axis* (page 37) or latitude array) – specification for latitude axis (optional)
- **lon** (*Axis* (page 37) or longitude array) – specification for longitude axis (optional)

Raises ValueError if *lat* is given but neither Axis nor latitude array.

Raises ValueError if *lon* is given but neither Axis nor longitude array.

Returns surface domain

Return type *SlabOcean* (page 42)

Example

```
>>> from climlab import domain
>>> sfc = domain.surface_2D(num_lat=36, num_lat=72)

>>> print sfc
climlab Domain object with domain_type=ocean and shape=(36, 72, 1)
```

climlab.domain.domain.**zonal_mean_column**(*num_lat=90, num_lev=30, water_depth=10.0, lat=None, lev=None, **kwargs*)

Creates two Domains with one water cell, a latitude axis and a level/height axis.

- SlabOcean: one water cell and a latitude axis above (similar to *zonal_mean_surface()* (page 47))
- Atmosphere: a latitude axis and a level/height axis (two dimensional)

Function-call argument

Parameters

- **num_lat** (*int*) – number of latitude points on the axis [default: 90]
- **num_lev** (*int*) – number of pressure levels (evenly spaced from surface to TOA) [default: 30]
- **water_depth** (*float*) – depth of the water cell (slab ocean) [default: 10.]
- **lat** (*Axis* (page 37) or latitude array) – specification for latitude axis (optional)
- **lev** (*Axis* (page 37) or pressure array) – specification for height axis (optional)

Raises ValueError if *lat* is given but neither Axis nor latitude array.

Raises ValueError if *lev* is given but neither Axis nor pressure array.

Returns a list of 2 Domain objects (slab ocean, atmosphere)

Return type list of *SlabOcean* (page 42), *Atmosphere* (page 40)

Example

```
>>> from climlab import domain
>>> sfc, atm = domain.zonal_mean_column(num_lat=36,num_lev=10)

>>> print sfc
```

(continues on next page)

(continued from previous page)

```
climlab Domain object with domain_type=ocean and shape=(36, 1)

>>> print atm
climlab Domain object with domain_type=atm and shape=(36, 10)
```

climlab.domain.domain.**zonal_mean_surface**(*num_lat=90*, *water_depth=10.0*, *lat=None*, ***kwargs*)

Creates a 1D slab ocean Domain in latitude with uniform water depth.

Domain has a single heat capacity according to the specified water depth.

Function-call argument

Parameters

- **num_lat** (*int*) – number of latitude points [default: 90]
- **water_depth** (*float*) – depth of the slab ocean in meters [default: 10.]
- **lat** (*Axis* (page 37) or latitude array) – specification for latitude axis (optional)

Raises ValueError if *lat* is given but neither Axis nor latitude array.

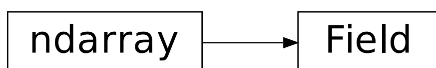
Returns surface domain

Return type *SlabOcean* (page 42)

Example

```
>>> from climlab import domain
>>> sfc = domain.zonal_mean_surface(num_lat=36)

>>> print sfc
climlab Domain object with domain_type=ocean and shape=(36, 1)
```

10.2.3 field



class climlab.domain.field.**Field**

Bases: numpy.ndarray

Custom class for climlab gridded quantities, called Field.

This class behaves exactly like numpy.ndarray but every object has an attribute called self.domain which is the domain associated with that field (e.g. state variables).

Initialization parameters

An instance of Field is initialized with the following arguments:

Parameters

- **input_array** (*array*) – the array which the Field object should be initialized with
- **domain** (*_Domain* (page 42)) – the domain associated with that field (e.g. state variables)

Object attributes

Following object attribute is generated during initialization:

Variables ***domain*** (page 37) (*_Domain* (page 42)) – the domain associated with that field (e.g. state variables)

Example

```
>>> import climlab
>>> import numpy as np
>>> from climlab import domain
>>> from climlab.domain import field

>>> # distribution of state
>>> distr = np.linspace(0., 10., 30)
>>> # domain creation
>>> sfc, atm = domain.single_column()
>>> # build state of type Field
>>> s = field.Field(distr, domain=atm)

>>> print s
[  0.           0.34482759   0.68965517   1.03448276   1.37931034
   1.72413793   2.06896552   2.4137931    2.75862069   3.10344828
   3.44827586   3.79310345   4.13793103   4.48275862   4.82758621
   5.17241379   5.51724138   5.86206897   6.20689655   6.55172414
   6.89655172   7.24137931   7.5862069    7.93103448   8.27586207
   8.62068966   8.96551724   9.31034483   9.65517241  10.        ]

>>> print s.domain
climlab Domain object with domain_type=atm and shape=(30,)

>>> # can slice this and it preserves the domain
>>> #  a more full-featured implementation would have intelligent
>>> #  slicing like in iris
>>> s.shape == s.domain.shape
True
>>> s[:1].shape == s[:1].domain.shape
False

>>> #  But some things work very well. E.g. new field creation:
>>> s2 = np.zeros_like(s)

>>> print s2
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  ␣
↪0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

>>> print s2.domain
climlab Domain object with domain_type=atm and shape=(30,)
```

Attributes

T Same as self.transpose(), except that self is returned if self.ndim < 2.

base Base object if memory is from some other object.

ctypes An object to simplify the interaction of the array with the ctypes module.

data Python buffer object pointing to the start of the array's data.

dtype Data-type of the array's elements.

flags Information about the memory layout of the array.

flat A 1-D iterator over the array.

imag The imaginary part of the array.

itemsize Length of one array element in bytes.

nbytes Total bytes consumed by the elements of the array.

ndim Number of array dimensions.

real The real part of the array.

shape Tuple of array dimensions.

size Number of elements in the array.

strides Tuple of bytes to step in each dimension when traversing an array.

Methods

`all`([axis, out, keepdims])	Returns True if all elements evaluate to True.
`any`([axis, out, keepdims])	Returns True if any of the elements of *a* evaluate to True.
`argmax`([axis, out])	Return indices of the maximum values along the given axis.
`argmin`([axis, out])	Return indices of the minimum values along the given axis of *a*.
`argpartition`(kth[, axis, kind, order])	Returns the indices that would partition this array.
`argsort`([axis, kind, order])	Returns the indices that would sort this array.
`astype`(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
`byteswap`([inplace])	Swap the bytes of the array elements
`choose`(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
`clip`([min, max, out])	Return an array whose values are limited to `[min, max]`.
`compress`(condition[, axis, out])	Return selected slices of this array along given axis.
`conj`()	Complex-conjugate all elements.
`conjugate`()	Return the complex conjugate, element-wise.
`copy`([order])	Return a copy of the array.
`cumprod`([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
`cumsum`([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
`diagonal`([offset, axis1, axis2])	Return specified diagonals.
`dot`(b[, out])	Dot product of two arrays.
`dump`(file)	Dump a pickle of the array to the specified file.
`dumps`()	Returns the pickle of the array as a string.
`fill`(value)	Fill the array with a scalar value.
`flatten`([order])	Return a copy of the array collapsed into one dimension.

Continued on next page

Table 8 – continued from previous page

getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out, keepdims])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims])	Returns the average of the array elements along given axis.
min([axis, out, keepdims])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod([axis, dtype, out, keepdims])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort([axis, kind, order])	Sort an array, in-place.
squeeze([axis])	Remove single-dimensional entries from the shape of *a*.
std([axis, dtype, out, ddof, keepdims])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
to_xarray (page 51)()	Convert Field object to xarray.DataArray
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as a (possibly nested) list.

Continued on next page

Table 8 – continued from previous page

tostring([order])	Construct Python bytes containing the raw data bytes in the array.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims])	Returns the variance of the array elements, along given axis.
view([dtype, type])	New view of array with the same data.

to_xarray()

Convert Field object to xarray.DataArray

climlab.domain.field.**global_mean**(*field*)

Calculates the latitude weighted global mean of a field with latitude dependence.

Parameters **field** (Field (page 47)) – input field

Raises ValueError if input field has no latitude axis

Returns latitude weighted global mean of the field

Return type float

Example initial global mean temperature of EBM model:

```
>>> import climlab
>>> model = climlab.EBM()
>>> climlab.global_mean(model.Ts)
Field(11.997968598413685)
```

climlab.domain.field.**to_latlon**(*array*, *domain*, *axis='lon'*)

Broadcasts a 1D axis dependent array across another axis.

Parameters

- **input_array** (*array*) – the 1D array used for broadcasting
- **domain** – the domain associated with that array
- **axis** – the axis that the input array will be broadcasted across [default: 'lon']

Returns Field with the same shape as the domain

Example

```
>>> import climlab
>>> from climlab.domain.field import to_latlon
>>> import numpy as np

>>> state = climlab.surface_state(num_lat=3, num_lon=4)
>>> m = climlab.EBM_annual(state=state)
>>> insolation = np.array([237., 417., 237.])
>>> insolation = to_latlon(insolation, domain = m.domains['Ts'])
>>> insolation.shape
(3, 4, 1)
>>> insolation
Field([[[ 237.],    [[ 417.],    [[ 237.],
        [ 237.],     [ 417.],     [ 237.],
        [ 237.],     [ 417.],     [ 237.],
        [ 237.]],    [ 417.]],    [ 237.]]])
```

10.2.4 initial

Convenience routines for setting up initial conditions.

`climlab.domain.initial.`**`column_state`**(*num_lev=30*, *num_lat=1*, *lev=None*, *lat=None*, *water_depth=1.0*)

Sets up a state variable dictionary consisting of temperatures for atmospheric column (`Tatm`) and surface mixed layer (`Ts`).

Surface temperature is always 288 K. Atmospheric temperature is initialized between 278 K at lowest altitude and 200 at top of atmosphere according to the number of levels given.

Function-call arguments

Parameters

- **num_lev** (*int*) – number of pressure levels (evenly spaced from surface to top of atmosphere) [default: 30]
- **num_lat** (*int*) – number of latitude points on the axis [default: 1]
- **lev** (*Axis* (page 37) or pressure array) – specification for height axis (optional)
- **lat** (*array*) – size of array determines dimension of latitude (optional)
- **water_depth** (*float*) – *irrelevant*

Returns dictionary with two temperature *Field* (page 47) for atmospheric column `Tatm` and surface mixed layer `Ts`

Return type dict

Example

```
>>> from climlab.domain import initial
>>> T_dict = initial.column_state()

>>> print T_dict
{'Tatm': Field([ 200.        ,  202.68965517,  205.37931034,  208.
↪06896552,
        210.75862069,  213.44827586,  216.13793103,  218.82758621,
        221.51724138,  224.20689655,  226.89655172,  229.5862069 ,
        232.27586207,  234.96551724,  237.65517241,  240.34482759,
        243.03448276,  245.72413793,  248.4137931 ,  251.10344828,
        253.79310345,  256.48275862,  259.17241379,  261.86206897,
        264.55172414,  267.24137931,  269.93103448,  272.62068966,
        275.31034483,  278.        ]), 'Ts': Field([ 288.])}
```

`climlab.domain.initial.`**`surface_state`**(*num_lat=90*, *num_lon=None*, *water_depth=10.0*, *T0=12.0*, *T2=-40.0*)

Sets up a state variable dictionary for a surface model (e.g. *EBM* (page 72)) with a uniform slab ocean depth.

The domain is either 1D (latitude) or 2D (latitude, longitude) depending on whether the input argument num_lon is supplied.

Returns a single state variable *Ts*, the temperature of the surface mixed layer (slab ocean).

The temperature is initialized to a smooth equator-to-pole shape given by

$$T(\phi) = T_0 + T_2 P_2(\sin\phi)$$

where ϕ is latitude, and P_2 is the second Legendre polynomial *P2* (page 175).

Function-call arguments

Parameters

- **num_lat** (*int*) – number of latitude points [default: 90]
- **num_lat** – (optional) number of longitude points [default: None]
- **water_depth** (*float*) – depth of the slab ocean in meters [default: 10.]
- **T0** (*float*) – global-mean initial temperature in °C [default: 12.]
- **T2** (*float*) – 2nd Legendre coefficient for equator-to-pole gradient in initial temperature, in °C [default: -40.]

Returns dictionary with temperature *Field* (page 47) for surface mixed layer Ts

Return type dict

Example

```
>>> from climlab.domain import initial
>>> import numpy as np

>>> T_dict = initial.surface_state(num_lat=36)

>>> print np.squeeze(T_dict['Ts'])
[-27.88584094 -26.97777479 -25.18923361 -22.57456133 -19.21320344
 -15.20729309 -10.67854785  -5.76457135  -0.61467228   4.61467228
   9.76457135  14.67854785  19.20729309  23.21320344  26.57456133
  29.18923361  30.97777479  31.88584094  31.88584094  30.97777479
  29.18923361  26.57456133  23.21320344  19.20729309  14.67854785
   9.76457135   4.61467228  -0.61467228  -5.76457135 -10.67854785
 -15.20729309 -19.21320344 -22.57456133 -25.18923361 -26.97777479
 -27.88584094]
```

10.3 climlab.dynamics

Modules for simple dynamics, mostly for use in Energy Balance Models.

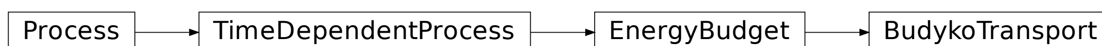
`BudykoTransport` is a relaxation to global mean.

Other modules are 1D diffusion solvers (implemented using implicit timestepping).

`MeridionalHeatDiffusion` is the appropriate class for the traditional diffusive EBM, in which transport is parameterized as a meridional diffusion process down the zonal-mean surface temperature gradient.

`MeridionalMoistDiffusion` implements the moist EBM, with transport down an approximate gradient in near-surface moist static energy.

10.3.1 BudykoTransport



class climlab.dynamics.budyko_transport.**BudykoTransport**(*b=3.81*, ***kwargs*)
Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

calculates the 1 dimensional heat transport as the difference between the local temperature and the global mean temperature.

Parameters b (*float*) – budyko transport parameter n - unit:
$textrmW/$
$left($
$textrmm^2$
$\circ$
$textrmC$
$right)$ n - default value: 3.81

As BudykoTransport is a *Process* (page 93) it needs a state do be defined on. See example for details.

Computation Details: n

In a global Energy Balance Model

$$C$$
$$fracdTdt = R \downarrow - R \uparrow - H$$

with model state T, the energy transport term H can be described as

$$H = b[T -$$
$$barT]$$

where T is a vector of the model temperature and
$barT$ describes the mean value of T.

For further information see [Budyko_1969].

Example Budyko Transport as a standalone process:

```
import climlab
from climlab.dynamics.budyko_transport import BudykoTransport
from climlab import domain
from climlab.domain import field
from climlab.utils.legendre import P2
import numpy as np
import matplotlib.pyplot as plt

# create domain
sfc = domain.zonal_mean_surface(num_lat = 36)

lat = sfc.lat.points
lat_rad = np.deg2rad(lat)

# define initial temperature distribution
T0 = 15.
T2 = -20.
Ts = field.Field(T0 + T2 * P2(np.sin(lat_rad)), domain=sfc)

# create BudykoTransport process
budyko_transp = BudykoTransport(state=Ts)

### Integrate & Plot ###

fig = plt.figure( figsize=(6,4))
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(111)

for i in np.arange(0,3,1):
    ax.plot(lat, budyko_transp.default, label='day %s' % (i*40))
    budyko_transp.integrate_days(40.)

ax.set_title('Standalone Budyko Transport')
ax.set_xlabel('latitude')
ax.set_xticks([-90,-60,-30,0,30,60,90])
ax.set_ylabel('temperature ($^{\circ}$C)')
ax.legend(loc='best')
plt.show()
```

Attributes

b (page 56) the budyko transport parameter in unit

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.

Continued on next page

Table 9 – continued from previous page

`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

b

the budyko transport parameter in unit $fractextrmWtextrmm^2textrmK$

Getter returns the budyko transport parameter

Setter sets the budyko transport parameter

Type float

10.3.2 Diffusion

General solver of the 1D diffusion equation:

$$\frac{\partial}{\partial t} \Psi(x,t) = -\frac{1}{w(x)} \frac{\partial}{\partial x} \left[w(x) \ F(x,t) \right]$$
$$F = -K \frac{\partial \Psi}{\partial x}$$

for a state variable $\Psi(x,t)$ and arbitrary diffusivity $K(x,t)$ in units of $x^2 \ t^{-1}$.

$w(x)$ is an optional weighting function for the divergence operator on curvilinear grids.

The diffusivity K can be a single scalar, or optionally a vector *specified at grid cell boundaries* (so its length must be exactly 1 greater than the length of x).

K can be modified by the user at any time (e.g., after each timestep, if it depends on other state variables).

A fully implicit timestep is used for computational efficiency. Thus the computed tendency $\frac{\partial \Psi}{\partial t}$ will depend on the timestep.

In addition to the tendency over the implicit timestep, the solver also calculates two diagnostics from the updated state:

- `diffusive_flux` given by $F(x)$ in units of $[\Psi]$ $[x]$/s
- `diffusive_flux_convergence` given by the right hand side of the first equation above, in units of $[\Psi]$/s

This base class can be used without modification for diffusion in Cartesian coordinates ($w = 1$) on a regularly spaced grid.

The state variable Ψ may be multi-dimensional, but the diffusion will operate along a single dimension only.

Other classes implement the weighting for spherical geometry.

class `climlab.dynamics.diffusion.`**`Diffusion`**(*K=None*, *diffusion_axis=None*, *use_banded_solver=False*, ***kwargs*)

Bases: *climlab.process.implicit.ImplicitProcess* (page 92)

A parent class for one dimensional implicit diffusion modules.

Initialization parameters

Parameters

- **K** (*float*) – the diffusivity parameter in units of $\frac{[\text{length}]^2}{\text{time}}$ where length is the unit of the spatial axis on which the diffusion is occuring.
- **diffusion_axis** (*str*) – dictionary key for axis on which the diffusion is occuring in process's domain axes dictionary
- **use_banded_solver** (*bool*) – input flag, whether to use `scipy.linalg.solve_banded()` instead of `numpy.linalg.solve()` [default: False]

Note: The banded solver `scipy.linalg.solve_banded()` is faster than `numpy.linalg.solve()` but only works for one dimensional diffusion.

Object attributes

Additional to the parent class *ImplicitProcess* (page 92) following object attributes are generated or modified during initialization:

Variables

- **param** (*dict*) – parameter dictionary is extended by diffusivity parameter K (unit: $\frac{[\text{length}]^2}{\text{time}}$)
- **use_banded_solver** (*bool*) – input flag specifying numerical solving method (given during initialization)
- **diffusion_axis** (*str*) – dictionary key for axis where diffusion is occuring: specified during initialization or output of method *_guess_diffusion_axis()* (page 59)
- **_K_dimensionless** (*array*) – diffusion parameter K multiplied by the timestep and divided by mean of diffusion axis delta in the power of two. Array has the size of diffusion axis bounds. $K_{\text{dimensionless}}[i] = K\frac{\Delta t}{(\Delta \text{bounds})^2}$
- **_diffTriDiag** (*array*) – tridiagonal diffusion matrix made by *_make_diffusion_matrix()* (page 60) with input `self._K_dimensionless`

Example Here is an example showing implementation of a vertical diffusion. It shows that a subprocess can work on just a subset of the parent process state variables.

```
import climlab
from climlab.dynamics.diffusion import Diffusion
import matplotlib.pyplot as plt

c = climlab.GreyRadiationModel()
K = 0.5
d = Diffusion(K=K, state = {'Tatm':c.state['Tatm']}, **c.param)
```

(continues on next page)

(continued from previous page)

```
c.add_subprocess('diffusion',d)

### Integrate & Plot ###

fig = plt.figure( figsize=(6,4))
ax = fig.add_subplot(111)

ax.plot(c.lev, c.state['Tatm'], label='step 0')
c.step_forward()
ax.plot(c.lev, c.state['Tatm'], label='step 1')

ax.invert_xaxis()
ax.set_title('Diffusion subprocess')
ax.set_xlabel('level (mb)')
#ax.set_xticks([])
ax.set_ylabel('temperature (K)')
ax.legend(loc='best')
plt.show()
```

Attributes

K

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.

Continued on next page

Table 10 – continued from previous page

declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

K

_implicit_solver()

Invertes and solves the matrix problem for diffusion matrix and temperature T.

The method is called by the *_compute()* (page 93) function of the *ImplicitProcess* (page 92) class and solves the matrix problem

$$A \cdot T_{\text{new}} = T_{\text{old}}$$

for diffusion matrix A and corresponding temperatures. T_{old} is in this case the current state variable which already has been adjusted by the explicit processes. T_{new} is the new state of the variable. To derive the temperature tendency of the diffusion process the adjustment has to be calculated and muliplied with the timestep which is done by the *_compute()* (page 93) function of the *ImplicitProcess* (page 92) class.

This method calculates the matrix inversion for every state variable and calling either solve_implicit_banded() or numpy.linalg.solve() dependent on the flag self.use_banded_solver.

Variables

- **state** (*dict*) – method uses current state variables but does not modify them
- **use_banded_solver** (*bool*) – input flag whether to use *_solve_implicit_banded()* (page 61) or numpy.linalg.solve() to do the matrix inversion
- **_diffTriDiag** (*array*) – the diffusion matrix which is given with the current state variable to the method solving the matrix problem

_update_diagnostics(*newstate*)

This method is called each timestep after the new state is computed with the implicit solver. Daughter classes can implement this method to compute any diagnostic quantities using the new state.

climlab.dynamics.diffusion.**_guess_diffusion_axis**(*process_or_domain*)

Scans given process, domain or dictionary of domains for a diffusion axis and returns appropriate name.

In case only one axis with length > 1 in the process or set of domains exists, the name of that axis is returned. Otherwise an error is raised.

Parameters **process_or_domain** (*Process* (page 93), *_Domain* (page 42) or dict of domains) – input from where diffusion axis should be guessed

Raises ValueError if more than one diffusion axis is possible.

Returns name of the diffusion axis

Return type str

climlab.dynamics.diffusion.**_make_diffusion_matrix**(*K*, *weight1=None*, *weight2=None*)

Builds the general diffusion matrix with dimension nxn.

Note: n = number of points of diffusion axis $n+1$ = number of bounts of diffusion axis

Function-all argument

Parameters

- **K** (*array*) – dimensionless diffusivities at cell boundaries *(size: 1xn+1)*
- **weight1** (*array*) – weight_1 *(size: 1xn+1)*
- **weight2** (*array*) – weight_2 *(size: 1xn)*

Returns completely listed tridiagonal diffusion matrix *(size: nxn)*

Return type array

Note: The elements of array K are acutally dimensionless:

$$K[i] = K_{\text{physical}} \frac{\Delta t}{(\Delta y)^2}$$

where K_{physical} is in unit $\frac{\text{length}^2}{\text{time}}$

The diffusion matrix is build like the following

$$\text{diffTriDiag} = \left[\begin{array}{cccccc} 1+\frac{s_1}{w_{2,0}} & -\frac{s_1}{w_{2,0}} & 0 & & ... & 0 \\ -\frac{s_1}{w_{2,1}} & 1+\frac{s_1+s_2}{w_{2,1}} & -\frac{s_2}{w_{2,1}} & 0 & ... & 0 \\ 0 & -\frac{s_2}{w_{2,2}} & 1+\frac{s_2+s_3}{w_{2,2}} & -\frac{s_3}{w_{2,2}} & ... & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & 0 & ... & -\frac{s_{n-2}}{w_{2,n-2}} & 1+\frac{s_{n-2}+s_{n-1}}{w_{2,n-2}} & -\frac{s_{n-1}}{w_{2,n-2}} \\ 0 & 0 & ... & 0 & -\frac{s_{n-1}}{w_{2,n-1}} & 1+\frac{s_{n-1}}{w_{2,n-1}} \end{array} \right]$$

where

$$\begin{array}{lllllll} K & = [K_0, & K_1, & K_2, & ..., & K_{n-1}, & K_n] \\ w_1 & = [w_{1,0}, & w_{1,1}, & w_{1,2}, & ..., & w_{1,n-1}, & w_{1,n}] \\ w_2 & = [w_{2,0}, & w_{2,1}, & w_{2,2}, & ..., & w_{2,n-1}] & \end{array}$$

and following subsitute:

$$s_i = w_{1,i} K_i$$

`climlab.dynamics.diffusion._make_meridional_diffusion_matrix(K, lataxis)`

Calls `_make_diffusion_matrix()` (page 60) with appropriate weights for the meridional diffusion case.

Parameters

- **K** (*array*) – dimensionless diffusivities at cell boundaries of diffusion axis `lataxis`
- **lataxis** (*axis*) – latitude axis where diffusion is occuring

Weights are computed as the following:

$$\begin{aligned} w_1 &= \cos(\text{bounds}) \\ &= [\cos(b_0), \cos(b_1), \cos(b_2), \ \dots, \cos(b_{n-1}), \cos(b_n)] \\ w_2 &= \cos(\text{points}) \\ &= [\cos(p_0), \cos(p_1), \cos(p_2), \ \dots, \cos(p_{n-1})] \end{aligned}$$

when bounds and points from `lataxis` are written as

$$\begin{aligned} \text{bounds} &= [b_0, b_1, b_2, \ \dots, b_{n-1}, b_n] \\ \text{points} &= [p_0, p_1, p_2, \ \dots, p_{n-1}] \end{aligned}$$

Giving this input to `_make_diffusion_matrix()` (page 60) results in a matrix like:

$$\text{diffTriDiag} = \begin{bmatrix} 1 + \frac{u_1}{\cos(p_0)} & -\frac{u_1}{\cos(p_0)} & 0 & & \dots & 0 \\ -\frac{u_1}{\cos(p_1)} & 1 + \frac{u_1 + u_2}{\cos(p_1)} & -\frac{u_2}{\cos(b_1)} & 0 & \dots & 0 \\ 0 & -\frac{u_2}{\cos(p_2)} & 1 + \frac{u_2 + u_3}{\cos(p_2)} & -\frac{u_3}{\cos(p_2)} & \dots & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & 0 & \dots & -\frac{u_{n-2}}{\cos(p_{n-2})} & 1 + \frac{u_{n-2} + u_{n-1}}{\cos(p_{n-2})} & -\frac{u_{n-1}}{\cos(p_{n-2})} \\ 0 & 0 & \dots & 0 & -\frac{u_{n-1}}{\cos(p_{n-1})} & 1 + \frac{u_{n-1}}{\cos(p_{n-1})} \end{bmatrix}$$

with the substitue of:

$$u_i = \cos(b_i) K_i$$

`climlab.dynamics.diffusion._solve_implicit_banded(current, banded_matrix)`

Uses a banded solver for matrix inversion of a tridiagonal matrix.

Converts the complete listed tridiagonal matrix *(nxn)* into a three row matrix *(3xn)* and calls `scipy.linalg.solve_banded()`.

Parameters

- **current** (*array*) – the current state of the variable for which matrix inversion should be computed
- **banded_matrix** (*array*) – complete diffusion matrix (*dimension: nxn*)

Returns output of `scipy.linalg.solve_banded()`

Return type array

10.3.3 MeridionalDiffusion

General solver of the 1D meridional diffusion equation on the sphere:

$$\frac{\partial}{\partial t}\Psi(\phi,t) = -\frac{1}{a\cos\phi}\frac{\partial}{\partial \phi}\left[\cos\phi \, F(\phi,t)\right]$$
$$F = -\frac{K}{a}\frac{\partial \Psi}{\partial \phi}$$

for a state variable $\Psi(\phi,t)$ and arbitrary diffusivity $K(\phi,t)$ in units of $x^2 \, t^{-1}$. ϕ is latitude and a is the Earth's radius (in meters).

The diffusivity K can be a single scalar, or optionally a vector *specified at grid cell boundaries* (so its length must be exactly 1 greater than the length of ϕ).

K can be modified by the user at any time (e.g., after each timestep, if it depends on other state variables).

A fully implicit timestep is used for computational efficiency. Thus the computed tendency $\frac{\partial \Psi}{\partial t}$ will depend on the timestep.

In addition to the tendency over the implicit timestep, the solver also calculates two diagnostics from the updated state:

- `diffusive_flux` given by $F(\phi)$ in units of $[\Psi]$ m/s
- `diffusive_flux_convergence` given by $-\frac{1}{a\cos\phi}\frac{\partial}{\partial \phi}\left[\cos\phi \, F(\phi,t)\right]$ in units of $[\Psi]$/s

The grid must be *evenly spaced in latitude*.

The state variable Ψ may be multi-dimensional, but the diffusion will operate along the latitude dimension only.

class `climlab.dynamics.meridional_diffusion.`**`MeridionalDiffusion`**(*K=None*, ***kwargs*)

Bases: *`climlab.dynamics.diffusion.Diffusion`* (page 57)

A parent class for Meridional diffusion processes.

Calculates the energy transport in a diffusion like process along the temperature gradient:

$$H(\varphi) = \frac{D}{\cos\varphi}\frac{\partial}{\partial\varphi}\left(\cos\varphi\frac{\partial T(\varphi)}{\partial\varphi}\right)$$

for an Energy Balance Model whose Energy Budget can be noted as:

$$C(\varphi)\frac{dT(\varphi)}{dt} = R\downarrow(\varphi) - R\uparrow(\varphi) + H(\varphi)$$

Initialization parameters

An instance of `MeridionalDiffusion` is initialized with the following arguments:

Parameters **K** (*`float`*) – diffusion parameter in units of m^2/s

Object attributes

Additional to the parent class *`Diffusion`* (page 57) which is initialized with `diffusion_axis='lat'`, following object attributes are modified during initialization:

Variables

- **_K_dimensionless** (*array*) – As _K_dimensionless has been computed like $K_{\text{dimensionless}} = K \frac{\Delta t}{(\Delta \text{bounds})^2}$ with K in units $1/s$, the $\Delta(\text{bounds})$ have to be converted from `deg` to `rad` to make the array actually dimensionless. This is done during initialiation.
- **_diffTriDiag** (*array*) – the diffusion matrix is recomputed with appropriate weights for the meridional case by `_make_meridional_diffusion_matrix()`

Example Meridional Diffusion of temperature as a stand-alone process:

```
import numpy as np
import climlab
from climlab.dynamics.diffusion import MeridionalDiffusion
from climlab.utils import legendre

sfc = climlab.domain.zonal_mean_surface(num_lat=90, water_depth=10.)
lat = sfc.lat.points
initial = 12. - 40. * legendre.P2(np.sin(np.deg2rad(lat)))

# make a copy of initial so that it remains unmodified
Ts = climlab.Field(np.array(initial), domain=sfc)

# thermal diffusivity in W/m**2/degC
D = 0.55

# meridional diffusivity in 1/s
K = D / sfc.heat_capacity
d = MeridionalDiffusion(state=Ts, K=K)

d.integrate_years(1.)

import matplotlib.pyplot as plt

fig = plt.figure( figsize=(6,4))
ax = fig.add_subplot(111)
ax.set_title('Example for Meridional Diffusion')
ax.set_xlabel('latitude')
ax.set_xticks([-90,-60,-30,0,30,60,90])
ax.set_ylabel('temperature ($^{\circ}$C)')
ax.plot(lat, initial,         label='initial')
ax.plot(lat, Ts, label='Ts (1yr)')
ax.legend(loc='best')
plt.show()
```

Attributes

K

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

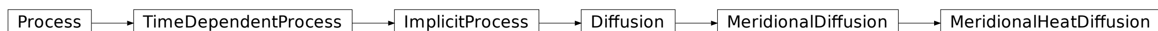
lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.3.4 MeridionalHeatDiffusion



Solver for the 1D meridional heat diffusion equation on the sphere:

$$C \frac{\partial}{\partial t} T(\phi,t) = \frac{1}{\cos\phi} \frac{\partial}{\partial \phi} \left[\cos\phi ~ D ~ \frac{\partial T}{\partial \phi} \right]$$

for a temperature state variable $T(\phi,t)$, a vertically-integrated heat capacity C, and arbitrary thermal diffusivity $D(\phi,t)$ in units of W/m2/K.

The diffusivity D can be a single scalar, or optionally a vector *specified at grid cell boundaries* (so its length must be exactly 1 greater than the length of ϕ).

D can be modified by the user at any time (e.g., after each timestep, if it depends on other state variables).

The heat capacity C is normally handled automatically by CLIMLAB as part of the grid specification.

A fully implicit timestep is used for computational efficiency. Thus the computed tendency $\frac{\partial T}{\partial t}$ will depend on the timestep.

The diagnostics `diffusive_flux` and `diffusive_flux_convergence` are computed as described in the parent class `MeridionalDiffusion`. Two additional diagnostics are computed here, which are meaningful if T represents a *zonally averaged temperature*:

- `heat_transport` given by $\mathcal{H}(\phi) = -2\pi \; a^2 \; \cos\phi \; D \; \frac{\partial T}{\partial \phi}$ in units of PW (petawatts).
- `heat_transport_convergence` given by $-\frac{1}{2\pi \; a^2 \cos\phi} \frac{\partial \mathcal{H}}{\partial \phi}$ in units of W/m2

The grid must be *evenly spaced in latitude*.

The state variable T may be multi-dimensional, but the diffusion will operate along the latitude dimension only.

class climlab.dynamics.meridional_heat_diffusion.**MeridionalHeatDiffusion**(*D=0.555, use_banded_solver=True, **kwargs*)

Bases: *climlab.dynamics.meridional_diffusion.MeridionalDiffusion* (page 62)

A 1D diffusion solver for Energy Balance Models.

Solves the meridional heat diffusion equation

$$ C

rac{partial T}{partial t} = - rac{1}{cosphi} rac{partial}{partial phi} left[-D cosphi rac{partial T}{partial phi} ight]$$

on an evenly-spaced latitude grid, with a state variable T, a heat capacity C and diffusivity D.

Assuming T is a temperature in K or ^circC, then the units are:

- D in W m^{-2} K^{-1}
- C in J m^{-2} K^{-1}

If the state variable has other units, then D and C should be expressed per state variabe unit.

D is provided as input, and can be either scalar or vector defined at latitude boundaries (length).

C is normally handled automatically for temperature state variables in CLIMLAB.

Attributes

D

K

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

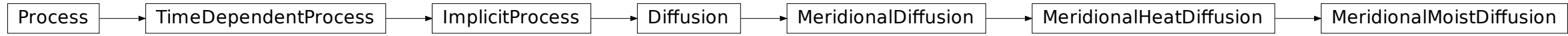
`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

D

_update_diagnostics(*newstate*)
This method is called each timestep after the new state is computed with the implicit solver. Daughter classes can implement this method to compute any diagnostic quantities using the new state.

_update_diffusivity()

10.3.5 MeridionalMoistDiffusion



Solver for the 1D meridional moist static energy diffusion equation on the sphere:

$$C\frac{\partial}{\partial t}T(\phi,t) = \frac{1}{\cos\phi}\frac{\partial}{\partial\phi}\left[\cos\phi\, D\,(1+f(T))\,\frac{\partial T}{\partial\phi}\right]$$

where $f(T)$ is a temperature-dependent moisture amplification factor given by

$$f(T) = \frac{L^2 r q^*(T)}{c_p R_v T^2}$$

which expresses the effect of latent heat on the near-surface moist static energy, where $q^*(T)$ is the saturation specific humidity at temperature T and r is a relative humidity.

This class operates identically to `MeridionalHeatDiffusion` but calculates f automatically at each timestep and applies it to the diffusivity.

The magnitude of the moisture amplification is controlled by the input parameter *relative_humidity* (i.e. r in the equation above).

It can be used to implement a modified Energy Balance Model accounting for the effects of moisture on the heat transport efficiency.

10.3.5.1 Derivation of the moist diffusion equation

Assume that heat transport is down the gradient of **moist static energy** $m = c_p T + Lq + gZ$

For an EBM we want to parameterize everything in terms of a surface temperature T_s. So we write $m_s = c_p T_s + Lrq^*(T_s)$, where m_s is the moist static energy of near-surface air parcels, r is a near-surface relative humidity, and q^* is the **saturation specific humidity** at a reference surface pressure.

Now express this quantity in temperature units by defining a *moist temperature*

$$T_m = \frac{m_s}{c_p} = T_s + \frac{Lr}{c_p}q^*(T_s)$$

T_m is the temperature a dry air parcel would have that has the same total enthalpy as a moist air parcel at temperature T_s

The down-gradient heat transport parameterization can then be written

$$\mathcal{H} = -2\pi a^2 D_m \frac{\partial T_m}{\partial\phi}$$

where D_m is the thermal diffusion coefficient for this moist model, in units of W/m2/K.

The equation we are trying to solve is thus

$$C\frac{\partial T_s}{\partial t} = \frac{1}{\cos\phi}\frac{\partial}{\partial\phi}\left(\cos\phi D_m \frac{\partial T_m}{\partial\phi}\right)$$

which we can write in terms of T_s only by substituting in for T_m:

$$C\frac{\partial T_s}{\partial t} = \frac{1}{\cos\phi}\frac{\partial}{\partial\phi}\left(\cos\phi D_m\left(\frac{\partial T_s}{\partial\phi} + \frac{\partial}{\partial\phi}\left(\frac{Lr}{c_p}q^*(T_s)\right)\right)\right)$$

If we make the simplifying assumption that the **relative humidity :math:‘r‘ is constant** (not a function of latitude), then

$$C \frac{\partial T_s}{\partial t} = \frac{1}{\cos\phi} \frac{\partial}{\partial \phi} \left(\cos\phi D_m \left(\frac{\partial T_s}{\partial \phi} + \frac{L r}{c_p} \frac{\partial q^*}{\partial \phi} \right) \right)$$

To a good approximation (see Hartmann’s book and others), the Clausius-Clapeyron relation for saturation specific humidity gives

$$\frac{\partial q^*}{dT} = \frac{L}{R_v T^2} q^*(T)$$

Then using a chain rule we have

$$\frac{\partial q^*}{\partial \phi} = \frac{\partial q^*}{\partial T_s} \frac{\partial T_s}{\partial \phi} = \frac{L q^*(T_s)}{R_v T_s^2} \frac{\partial T_s}{\partial \phi}$$

Plugging this into our model equation we get

$$C \frac{\partial T_s}{\partial t} = \frac{1}{\cos\phi} \frac{\partial}{\partial \phi} \left(\cos\phi D_m \frac{\partial T_s}{\partial \phi} \left(1 + \frac{L^2 r q^*(T_s)}{c_p R_v T_s^2} \right) \right)$$

This is now in a form that is compatible with our diffusion solver.

Just let

$$D = D_m \left(1 + f(T_s) \right)$$

where

$$f(T_s) = \frac{L^2 r q^*(T_s)}{c_p R_v T_s^2}$$

or, equivalently,

$$f(T_s) = \frac{L r}{c_p} \frac{\partial q^*}{dT} \bigg|_{T_s}$$

Given a temperature distribution $T_s(\phi)$ at any given time, we can calculate the diffusion coefficient $D(\phi)$ from this formula.

This calculation is implemented in the `MeridionalMoistDiffusion` class.

class `climlab.dynamics.meridional_moist_diffusion.`**`MeridionalMoistDiffusion`**(*D=0.24*, *relative_humidity=0.8*, ***kwargs*)

Bases: *climlab.dynamics.meridional_heat_diffusion.MeridionalHeatDiffusion* (page 65)

Attributes

D

K

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

_implicit_solver()

Invertes and solves the matrix problem for diffusion matrix and temperature T.

The method is called by the *_compute()* (page 93) function of the *ImplicitProcess* (page 92) class and solves the matrix problem

$$A \cdot T_{\text{new}} = T_{\text{old}}$$

for diffusion matrix A and corresponding temperatures. T_{old} is in this case the current state variable which already has been adjusted by the explicit processes. T_{new} is the new state of the variable. To derive the temperature tendency of the diffusion process the adjustment has to be calculated and muliplied with the timestep which is done by the `_compute()` (page 93) function of the `ImplicitProcess` (page 92) class.

This method calculates the matrix inversion for every state variable and calling either `solve_implicit_banded()` or `numpy.linalg.solve()` dependent on the flag `self.use_banded_solver`.

Variables

- **state** (*dict*) – method uses current state variables but does not modify them
- **use_banded_solver** (*bool*) – input flag whether to use `_solve_implicit_banded()` or `numpy.linalg.solve()` to do the matrix inversion
- **_diffTriDiag** (*array*) – the diffusion matrix which is given with the current state variable to the method solving the matrix problem

_update_diffusivity()

climlab.dynamics.meridional_moist_diffusion.**moist_amplification_factor**(*Tkelvin*, *relative_humidity=0.8*)

Compute the moisture amplification factor for the moist diffusivity given relative humidity and reference temperature profile.

10.4 climlab.model

This package contains ready-made models that can be run "off-the-shelf".

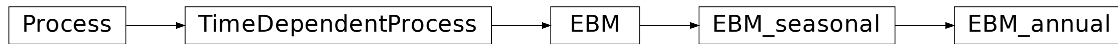
Example

```
import climlab
#  create a 1D Energy Balance Model
mymodel = climlab.EBM()
#  see what you just created
print(mymodel)
#  run the model
mymodel.integrate_years(2.)
#  display the current state
mymodel.state
#  see what diagnostics have been computed
mymodel.diagnostics.keys()
```

These modules are fully functional and tested. However users are encouraged to build their own models by explicitly creating individual processes and coupling together as subprocesses of a parent process.

See the documentation for the RRTMG scheme for an example of building a radiative-convective column model from individual components.

10.4.1 ebm



Convenience classes for pre-made Energy Balance Models in CLIMLAB.

These models all solve some form of the equation

$$C \frac{\partial}{\partial t} T_s(\phi,t) = (1-\alpha) S(\phi,t) - [A + B T_s] + \frac{1}{\cos\phi} \frac{\partial}{\partial \phi} \left[\cos\phi ~ D ~ \frac{\partial T_s}{\partial \phi} \right]$$

where

- ϕ is latitude
- T_s is a zonally averaged surface temperature
- C is a depth-integrated heat capacity
- α is an albedo (which may depend on latitude and/or temperature)
- $S(\phi, t)$ is the insolation
- $[A + BT_s]$ is a parameterization of the Outgoing Longwave Radiation to space
- the last term on the right hand side is a diffusive heat transport convergence with thermal diffusivity D in the same units as B

Three classes are provided, which differ in the type of insolation S:

- `climlab.EBM` uses a steady idealized annual insolation (second Legendre polynomial form)
- `climlab.EBM_annual` uses realistic steady annual-mean insolation
- `climlab.EBM_seasonal` uses realistic seasonally varying insolation

The `__init__` method of class `EBM` shows how these models are assembled from subprocesses representing each term in the above equation.

10.4.1.1 Building the Moist EBM

There is currently no ready-made convenience class for the **moist EBM**, but it can be readily built by swapping out the dry heat diffusion process `climlab.dynamics.MeridionalHeatDiffusion` with the moist equivalent `climlab.dynamics.MeridionalMoistDiffusion`.

This sort of mixing and matching of model components is at the heart of CLIMLAB design and functionality.

Example

```
import climlab
#  create and display a 1D Energy Balance Model
dry = climlab.EBM()
print(dry)
#  clone this model and swap out the diffusion subprocess
moist = climlab.process_like(dry)
diff = climlab.dynamics.MeridionalMoistDiffusion(state=moist.
↪state, timestep=moist.timestep)
moist.add_subprocess('diffusion', diff)
print(moist)
```

We can run both models out to equilibrium and compare the results as follows:

Example

```
#  Run both models out to quasi-equilibrium
#   print out the global mean planetary energy budget -- should
↪be very small
for m in [dry, moist]:
    m.integrate_years(10)
    print(climlab.global_mean(m.net_radiation))
#  plot and compare the temperatures
import matplotlib.pyplot as plt
plt.figure()
plt.plot(dry.lat, dry.Ts, label='Dry')
plt.plot(moist.lat, moist.Ts, label='Moist')
plt.legend()
plt.show()
#  plot and compare the heat transport
plt.figure()
plt.plot(dry.lat_bounds, dry.heat_transport, label='Dry')
plt.plot(moist.lat_bounds, moist.heat_transport, label='Moist')
plt.legend()
plt.show()
```

class climlab.model.ebm.**EBM**(*num_lat=90, num_lon=None, S0=1365.2, s2=-0.48, A=210.0, B=2.0, D=0.555, water_depth=10.0, Tf=-10.0, a0=0.3, a2=0.078, ai=0.62, timestep=350632.51200000005, T0=12.0, T2=-40.0, **kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

A parent class for all Energy-Balance-Model classes.

This class sets up a typical EnergyBalance Model with following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization through `AplusBT`
- Absorbed Shortwave Radiation (ASR) through `SimpleAbsorbedShortwave`
- solar insolation paramtrization through `P2Insolation`
- albedo parametrization in dependence of temperature through `StepFunctionAlbedo`
- energy diffusion through `MeridionalHeatDiffusion`

Initialization parameters

An instance of `EBM` is initialized with the following arguments *(for detailed information see Object attributes below)*:

Parameters

- **num_lat** (*int*) – number of equally spaced points for the latitue grid. Used for domain intialization of *zonal_mean_surface* (page 47)
  - default value: `90`
- **num_lon** (*int*) – number of equally spaced points in longitude
  - default value: `None`
- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `1365.2`

- **A** (*float*) – parameter for linear OLR parametrization `AplusBT`
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `210.0`
- **B** (*float*) – parameter for linear OLR parametrization `AplusBT`
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2\ ^{\circ}\mathrm{C}}$
  - default value: `2.0`
- **D** (*float*) – diffusion parameter for Meridional Energy Diffusion `MeridionalDiffusion`
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2\ ^{\circ}\mathrm{C}}$
  - default value: `0.555`
- **water_depth** (*float*) – depth of *zonal_mean_surface* (page 47) domain, which the heat capacity is dependent on
  - unit: meters
  - default value: `10.0`
- **Tf** (*float*) – freezing temperature
  - unit: $^{\circ}\mathrm{C}$
  - default value: `-10.0`
- **a0** (*float*) – base value for planetary albedo parametrization *StepFunctionAlbedo* (page 166)
  - unit: dimensionless
  - default value: `0.3`
- **a2** (*float*) – parabolic value for planetary albedo parametrization *StepFunctionAlbedo* (page 166)
  - unit: dimensionless
  - default value: `0.078`
- **ai** (*float*) – value for ice albedo paramerization in *StepFunctionAlbedo* (page 166)
  - unit: dimensionless
  - default value: `0.62`
- **timestep** (*float*) – specifies the EBM's timestep
  - unit: seconds
  - default value: (365.2422 * 24 * 60 * 60) / 90

    -> (90 timesteps per year)
- **T0** (*float*) – base value for initial temperature
  - unit $^{\circ}\mathrm{C}$
  - default value: `12`
- **T2** (*float*) – factor for 2nd Legendre polynomial *P2* (page 175) to calculate initial temperature
  - unit: dimensionless

– default value: `40`

Object attributes

Additional to the parent class `EnergyBudget` following object attributes are generated and updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with a couple of the initatilzation input arguments, namely `'S0'`, `'A'`, `'B'`, `'D'`, `'Tf'`, `'water_depth'`, `'a0'`, `'a2'` and `'ai'`.
- **domains** (*dict*) – If the object's `domains` and the `state` dictionaries are empty during initialization a domain `sfc` is created through *zonal_mean_surface()* (page 47). In the meantime the object's `domains` and `state` dictionaries are updated.
- **subprocess** (*dict*) – Several subprocesses are created (see above) through calling *add_subprocess()* (page 96) and therefore the subprocess dictionary is updated.
- **topdown** (*bool*) – is set to `False` to call subprocess compute methods first. See also *TimeDependentProcess* (page 103).
- ***diagnostics*** (page 97) (*dict*) – is initialized with keys: `'OLR'`, `'ASR'`, `'net_radiation'`, `'albedo'`, `'icelat'` and `'ice_area'` through *add_diagnostic()* (page 95).

Example Creation and integration of the preconfigured Energy Balance Model:

```
>>> import climlab
>>> model = climlab.EBM()

>>> model.integrate_years(2.)
Integrating for 180 steps, 730.4844 days, or 2.0 years.
Total elapsed time is 2.0 years.
```

For more information how to use the EBM class, see the *Tutorials* (page 25) chapter.

Attributes

S0

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`diffusive_heat_transport` (page 75)()	Compute instantaneous diffusive heat transport in unit PW on the staggered grid (bounds) through calculating:
`global_mean_temperature` (page 75)()	Convenience method to compute global mean surface temperature.
`inferred_heat_transport` (page 76)()	Calculates the inferred heat transport by integrating the TOA energy imbalance from pole to pole.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

S0

diffusive_heat_transport()

Compute instantaneous diffusive heat transport in unit PW on the staggered grid (bounds) through calculating:

$$H(\varphi) = -2\pi R^2 cos(\varphi) D \frac{dT}{d\varphi} \approx -2\pi R^2 cos(\varphi) D \frac{\Delta T}{\Delta\varphi}$$

Return type array of size `np.size(self.lat_bounds)`

THIS IS DEPRECATED AND WILL BE REMOVED IN THE FUTURE. Use the diagnostic `heat_transport` instead, which implements the same calculation.

global_mean_temperature()

Convenience method to compute global mean surface temperature.

Calls `global_mean()` (page 51) method which for the object attriute `Ts` which calculates the latitude weighted global mean of a field.

Example Calculating the global mean temperature of initial EBM temperature:

```
>>> import climlab
>>> model = climlab.EBM(T0=14., T2=-25)

>>> model.global_mean_temperature()
Field(13.99873037400856)
```

inferred_heat_transport()

Calculates the inferred heat transport by integrating the TOA energy imbalance from pole to pole.

The method is calculating

$$H(\varphi) = 2\pi R^2 \int_{-\pi/2}^{\varphi} cos\phi \; R_{TOA} d\phi$$

where R_{TOA} is the net radiation at top of atmosphere.

Returns total heat transport on the latitude grid in unit PW

Return type array of size `np.size(self.lat_lat)`

Example

```
import climlab
import matplotlib.pyplot as plt

# creating & integrating model
model = climlab.EBM()
model.step_forward()

# plot
fig = plt.figure( figsize=(6,4))
ax = fig.add_subplot(111)

ax.plot(model.lat, model.inferred_heat_transport())

ax.set_title('inferred heat transport')
ax.set_xlabel('latitude')
ax.set_xticks([-90,-60,-30,0,30,60,90])
ax.set_ylabel('energy (PW)')
plt.axhline(linewidth=2, color='grey', linestyle='dashed')
plt.show()
```

class climlab.model.ebm.**EBM_annual**(***kwargs*)

Bases: `climlab.model.ebm.EBM_seasonal` (page 78)

A class that implements Energy Balance Models with annual mean insolation.

The annual solar distribution is calculated through averaging the `DailyInsolation` (page 126) over time which has been used in used in the parent class `EBM_seasonal`. That is done by the subprocess `AnnualMeanInsolation` which is more realistic than the `P2Insolation` module used in the classical `EBM` class.

According to the parent class `EBM_seasonal` the model will not have an ice-albedo feedback, if albedo ice parameter `'ai'` is not given. For details see there.

Object attributes

Following object attributes are updated during initialization:

Variables **subprocess** (*dict*) – suprocess 'insolation' is overwritten by AnnualMeanInsolation

Example The EBM_annual class uses a different insolation subprocess than the EBM class:

```
>>> import climlab
>>> model_annual = climlab.EBM_annual()

>>> print model_annual
```

```
climlab Process of type <class 'climlab.model.ebm.EBM_annual'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.EBM_annual'>
   diffusion: <class 'climlab.dynamics.MeridionalHeatDiffusion'>
   LW: <class 'climlab.radiation.AplusBT'>
   albedo: <class 'climlab.surface.P2Albedo'>
   insolation: <class 'climlab.radiation.AnnualMeanInsolation'>
```

Attributes

S0

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 15 – continued from previous page

`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`diffusive_heat_transport`()	Compute instantaneous diffusive heat transport in unit PW on the staggered grid (bounds) through calculating:
`global_mean_temperature`()	Convenience method to compute global mean surface temperature.
`inferred_heat_transport`()	Calculates the inferred heat transport by integrating the TOA energy imbalance from pole to pole.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class `climlab.model.ebm.`**`EBM_seasonal`**(*a0=0.33*, *a2=0.25*, *ai=None*, ***kwargs*)

Bases: *`climlab.model.ebm.EBM`* (page 72)

A class that implements Energy Balance Models with realistic daily insolation.

This class is inherited from the general `EBM` class and uses the insolation subprocess `DailyInsolation` instead of `P2Insolation` to compute a realisitc distribution of solar radiation on a daily basis.

If argument for ice albedo `'ai'` is not given, the model will not have an albedo feedback.

An instance of `EBM_seasonal` is initialized with the following arguments:

Parameters

- **a0** (*float*) – base value for planetary albedo parametrization *`StepFunctionAlbedo`* (page 166) [default: 0.33]
- **a2** (*float*) – parabolic value for planetary albedo parametrization *`StepFunctionAlbedo`* (page 166) [default: 0.25]
- **ai** (*float*) – value for ice albedo paramerization in *`StepFunctionAlbedo`* (page 166) (optional)

Object attributes

Following object attributes are updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with `'a0'` and `'a2'`.
- **subprocess** (*dict*) – suprocess `'insolation'` is overwritten by `DailyInsolation` (page 126).

if `'ai'` *is not given*:

Variables

- **param** (*dict*) – `'ai'` and `'Tf'` are removed from the parameter dictionary (initialized by parent class `EBM` (page 72))
- **subprocess** (*dict*) – suprocess `'albedo'` is overwritten by `P2Albedo` (page 164).

if `'ai'` *is given*:

Variables

- **param** (*dict*) – The parameter dictionary is updated with `'ai'`.
- **subprocess** (*dict*) – suprocess `'albedo'` is overwritten by `StepFunctionAlbedo` (page 166) (which basically has been there before but now is updated with the new albedo parameter values).

Example The annual distribution of solar insolation:

```
import climlab
from climlab.utils import constants as const
import numpy as np
import matplotlib.pyplot as plt

# creating model
model = climlab.EBM_seasonal()
model.step_forward()

solar = model.subprocess['insolation'].insolation

# plot
fig = plt.figure( figsize=(6,4))
ax = fig.add_subplot(111)

season_days = const.days_per_year/4

for season in ['winter','spring','summer','autumn']:
    ax.plot(model.lat, solar, label=season)
    model.integrate_days(season_days)

ax.set_title('seasonal solar distribution')
ax.set_xlabel('latitude')
ax.set_xticks([-90,-60,-30,0,30,60,90])
ax.set_ylabel('solar insolation (W/m$^2$)')
ax.legend(loc='best')
plt.show()
```

Attributes

S0

`depth` Depth at grid centers (m)

`depth_bounds` Depth at grid interfaces (m)

`diagnostics` Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

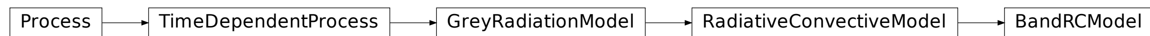
lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`diffusive_heat_transport`()	Compute instantaneous diffusive heat transport in unit PW on the staggered grid (bounds) through calculating:
`global_mean_temperature`()	Convenience method to compute global mean surface temperature.
`inferred_heat_transport`()	Calculates the inferred heat transport by integrating the TOA energy imbalance from pole to pole.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.4.2 column



Object-oriented code for radiative-convective models with grey-gas radiation.

Code developed by Brian Rose, University at Albany brose@albany.edu

Note that the column models by default represent global, time averages. Thus the insolation is a prescribed constant.

Here is an example to implement seasonal insolation at 45 degrees North

Example

```
import climlab

#  create the column model object
col = climlab.GreyRadiationModel()

#  create a new latitude axis with a single point
lat = climlab.domain.Axis(axis_type='lat', points=45.)

#  add this new axis to the surface domain
col.Ts.domain.axes['lat'] = lat

#  create a new insolation process using this domain
Q = climlab.radiation.insolation.DailyInsolation(domains=col.Ts.
↪domain, **col.param)

#  replace the fixed insolation subprocess in the column model
col.add_subprocess('insolation', Q)
```

This model is now a single column with seasonally varying insolation calculated for 45N.

class climlab.model.column.**BandRCModel**(***kwargs*)

Bases: *climlab.model.column.RadiativeConvectiveModel* (page 83)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`do_diagnostics`()	Set all the diagnostics from long and shortwave radiation.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class climlab.model.column.**GreyRadiationModel**(*num_lev=30*, *num_lat=1*, *lev=None*, *lat=None*, *water_depth=1.0*, *albedo_sfc=0.299*, *timestep=86400.0*, *Q=341.3*, *abs_coeff=0.0001229*, ***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`do_diagnostics` (page 83)()	Set all the diagnostics from long and shortwave radiation.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

do_diagnostics()
Set all the diagnostics from long and shortwave radiation.

class climlab.model.column.**RadiativeConvectiveModel**(*adj_lapse_rate=6.5*, ***kwargs*)
Bases: *climlab.model.column.GreyRadiationModel* (page 82)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`do_diagnostics`()	Set all the diagnostics from long and shortwave radiation.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

`climlab.model.column.`**`compute_layer_absorptivity`**(*abs_coeff*, *dp*)

Compute layer absorptivity from a constant absorption coefficient.

10.5 climlab.process

The base classes for all climlab processes.

10.5.1 diagnostic

class climlab.process.diagnostic.**DiagnosticProcess**(***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

A parent class for all processes that are strictly diagnostic, namely that do **not** contribute directly to tendencies of state variables.

During initialization following attribute is set:

Variables **time_type** (*str*) – is set to `'diagnostic'`

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

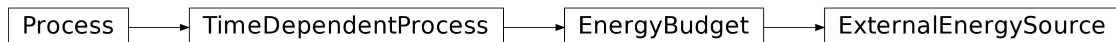
add_diagnostic(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
add_input(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.

Continued on next page

Table 20 – continued from previous page

`compute()`	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics([num_iter])`	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics(diaglist)`	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input(inputlist)`	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge([crit, verbose])`	Integrates the model until model states are converging.
`integrate_days([days, verbose])`	Integrates the model forward for a specified number of days.
`integrate_years([years, verbose])`	Integrates the model by a given number of years.
`remove_diagnostic(name)`	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess(name[, verbose])`	Removes a single subprocess from this process.
`set_state(name, value)`	Sets the variable `name` to a new state `value`.
`set_timestep([timestep, num_steps_per_year])`	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward()`	Updates state variables with computed tendencies.
`to_xarray([diagnostics])`	Convert process variables to `xarray.Dataset` format.

10.5.2 energy_budget



class `climlab.process.energy_budget.`**`EnergyBudget`**`(**kwargs)`

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

A parent class for explicit energy budget processes.

This class solves equations that include a heat capacitiy term like C
$fracdTdt =$
$textrmfluxconvergence$

In an Energy Balance Model with model state T this equation will look like this:

$$C$$
$$fracdTdt = R\downarrow - R\uparrow - H$$
$$fracdTdt =$$
$$fracR\downarrow C-$$
$$fracR\uparrow C-$$
$$fracHC$$

Every EnergyBudget object has a `heating_rate` dictionary with items corresponding to each state variable. The heating rate accounts the actual heating of a subprocess, namely the contribution to the energy budget of R

$downarrow, R$
$uparrow$ and H in this case. The temperature tendencies for each subprocess are then calculated through dividing the heating rate by the heat capacitiy C.

Initialization parameters n

An instance of `EnergyBudget` is initialized with the forwarded keyword arguments `**kwargs` of the corresponding children classes.

Object attributes n

Additional to the parent class `TimeDependentProcess` following object attributes are generated or modified during initialization:

Variables

- **time_type** (*str*) – is set to `'explicit'`
- **heating_rate** (*dict*) – energy share for given subprocess in unit
  $textrmW/$
  $textrmm^2$ stored in a dictionary sorted by model states

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.

Continued on next page

Table 21 – continued from previous page

declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.process.energy_budget.**ExternalEnergySource**(***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

A fixed energy source or sink to be specified by the user.

Object attributes

Additional to the parent class *EnergyBudget* (page 86) the following object attribute is modified during initialization:

Variables heating_rate (*dict*) – energy share dictionary for this subprocess is set to zero for every model state.

After initialization the user should modify the fields in the heating_rate dictionary, which contain heating rates in unit $\mathrm{W/m^2}$ for all state variables.

Example Creating an Energy Balance Model with a uniform external energy source of $10\ \mathrm{W/m^2}$ for all latitudes:

```
>>> import climlab
>>> from climlab.process.energy_budget import ExternalEnergySource
>>> import numpy as np

>>> # create model & external energy subprocess
>>> model = climlab.EBM(num_lat=36)
>>> ext_en = ExternalEnergySource(state= model.state,**model.param)

>>> # modify external energy rate
>>> ext_en.heating_rate.keys()
['Ts']

>>> np.squeeze(ext_en.heating_rate['Ts'])
Field([-0., -0., -0., -0., -0., -0., -0., -0., -0.,  0.,  0.,  0.,  0.
↪,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.
↪,
        0., -0., -0., -0., -0., -0., -0., -0., -0., -0.])
```

(continues on next page)

(continued from previous page)

```
>>> ext_en.heating_rate['Ts'][:]=10

>>> np.squeeze(ext_en.heating_rate['Ts'])
Field([ 10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  ␣
↪10.,
        10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  ␣
↪10.,
        10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  10.,  ␣
↪10.,
        10.,  10.,  10.])

>>> # add subprocess to model
>>> model.add_subprocess('ext_energy',ext_en)

>>> print model
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (36, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   ext_energy: <class 'climlab.process.energy_budget.
↪ExternalEnergySource'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.

Continued on next page

Table 22 – continued from previous page

add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

10.5.3 external_forcing

class climlab.process.external_forcing.**ExternalForcing**(***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

A Process class for user-defined tendencies of state variables. Useful for combining some prescribed external forcing with an interactive model.

Example The user can invoke the process on a dicionary of state variables mystate like this:

```
myforcing = climlab.process.ExternalForcing(state=mystate)
```

and then set the desired tendencies in the dictionary myforcing.forcing_tendencies, in units of [state variable unit] per second.

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.5.4 implicit

class climlab.process.implicit.**ImplicitProcess**(***kwargs*)

Bases: *climlab.process.time_dependent_process.TimeDependentProcess* (page 103)

A parent class for modules that use implicit time discretization.

During initialization following attributes are intitialized:

Variables

- **time_type** (*str*) – is set to 'implicit'
- **adjustment** (*dict*) – the model state adjustments due to this implicit subprocess

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 24 – continued from previous page

`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

`_compute`()

Computes the state variable tendencies in time for implicit processes.

To calculate the new state the `_implicit_solver()` method is called for daughter classes. This however returns the new state of the variables, not just the tendencies. Therefore, the adjustment is calculated which is the difference between the new and the old state and stored in the object's attribute adjustment.

Calculating the new model states through solving the matrix problem already includes the multiplication with the timestep. The derived adjustment is divided by the timestep to calculate the implicit subprocess tendencies, which can be handeled by the `compute()` (page 105) method of the parent `TimeDependentProcess` (page 103) class.

Variables `adjustment` (`dict`) – holding all state variables' adjustments of the implicit process which are the differences between the new states (which have been solved through matrix inversion) and the old states.

`_update_diagnostics`(*newstate*)

This method is called each timestep after the new state is computed with the implicit solver. Daughter classes can implement this method to compute any diagnostic quantities using the new state.

10.5.5 process

Process

class climlab.process.process.**Process**(*name='Untitled'*, *state=None*, *domains=None*, *subprocess=None*, *lat=None*, *lev=None*, *num_lat=None*, *num_levels=None*, *input=None*, *verbose=True*, ***kwargs*)

Bases: object

A generic parent class for all climlab process objects. Every process object has a set of state variables on a spatial grid.

For more general information about *Processes* and their role in climlab, see *Process* (page 16) section climlab-architecture.

Initialization parameters

An instance of Process is initialized with the following arguments *(for detailed information see Object attributes below)*:

Parameters

- **state** (Field (page 47)) – spatial state variable for the process. Set to None if not specified.
- **domains** (*_Domain* (page 42) or dict of *_Domain* (page 42)) – domain(s) for the process
- **subprocess** (*Process* (page 93) or dict of *Process* (page 93)) – subprocess(es) of the process
- **lat** (*array*) – latitudinal points (optional)
- **lev** – altitudinal points (optional)
- **num_lat** (*int*) – number of latitudional points (optional)
- **num_levels** (*int*) – number of altitudinal points (optional)
- **input** (*dict*) – collection of input quantities
- **verbose** (*bool*) – Flag to control text output during instantiation of the Process [default: True]

Object attributes

Additional to the parent class *Process* (page 93) following object attributes are generated during initialization:

Variables

- **domains** (*dict*) – dictionary of process *_Domain* (page 42)
- **state** (*dict*) – dictionary of process states (of type *Field* (page 47))
- **param** (*dict*) – dictionary of model parameters which are given through **kwargs
- ***diagnostics*** (page 97) (*dict*) – a dictionary with all diagnostic variables
- **_input_vars** (*dict*) – collection of input quantities like boundary conditions and other gridded quantities
- **creation_date** (*str*) – date and time when process was created
- **subprocess** (dict of *Process* (page 93)) – dictionary of suprocesses of the process

Attributes

depth **(page 97)** Depth at grid centers (m)

depth_bounds **(page 97)** Depth at grid interfaces (m)

diagnostics **(page 97)** Dictionary access to all diagnostic variables

input **(page 97)** Dictionary access to all input variables

lat **(page 97)** Latitude of grid centers (degrees North)

lat_bounds **(page 98)** Latitude of grid interfaces (degrees North)

lev **(page 98)** Pressure levels at grid centers (hPa or mb)

lev_bounds **(page 98)** Pressure levels at grid interfaces (hPa or mb)

lon **(page 98)** Longitude of grid centers (degrees)

lon_bounds **(page 98)** Longitude of grid interfaces (degrees)

Methods

add_diagnostic (page 95)(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
add_input (page 96)(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
add_subprocess (page 96)(name, proc)	Adds a single subprocess to this process.
add_subprocesses (page 97)(procdict)	Adds a dictionary of subproceses to this process.
declare_diagnostics (page 97)(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
declare_input (page 97)(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
remove_diagnostic (page 98)(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
remove_subprocess (page 99)(name[, verbose])	Removes a single subprocess from this process.
set_state (page 99)(name, value)	Sets the variable `name` to a new state `value`.
to_xarray (page 100)([diagnostics])	Convert process variables to `xarray.Dataset` format.

add_diagnostic(*name*, *value=None*)

Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.

Quantity is accessible in two ways:

- as a process attribute, i.e. `proc.name`
- as a member of the diagnostics dictionary, i.e. `proc.diagnostics['name']`

Use attribute method to set values, e.g. `` `proc.name = value ` ``

Parameters

- **name** (*str*) – name of diagnostic quantity to be initialized
- **value** (*array*) – initial value for quantity [default: None]

Example Add a diagnostic CO2 variable to an energy balance model:

```
>>> import climlab
>>> model = climlab.EBM()

>>> # initialize CO2 variable with value 280 ppm
>>> model.add_diagnostic('CO2',280.)
```

(continues on next page)

(continued from previous page)

```
>>> # access variable directly or through diagnostic dictionary
>>> model.CO2
280
>>> model.diagnostics.keys()
['ASR', 'CO2', 'net_radiation', 'icelat', 'OLR', 'albedo']
```

add_input (*name*, *value=None*)

Create a new input variable called `name` for this process and initialize it with the given `value`.

Quantity is accessible in two ways:

- as a process attribute, i.e. `proc.name`
- as a member of the input dictionary, i.e. `proc.input['name']`

Use attribute method to set values, e.g. `` `proc.name = value ` ``

Parameters

- **name** (*str*) – name of diagnostic quantity to be initialized
- **value** (*array*) – initial value for quantity [default: None]

add_subprocess (*name*, *proc*)

Adds a single subprocess to this process.

Parameters

- **name** (*string*) – name of the subprocess
- **proc** (*Process* (page 93)) – a Process object

Raises `ValueError` if `proc` is not a process

Example Replacing an albedo subprocess through adding a subprocess with same name:

```
>>> from climlab.model.ebm import EBM_seasonal
>>> from climlab.surface.albedo import StepFunctionAlbedo

>>> # creating EBM model
>>> ebm_s = EBM_seasonal()

>>> print ebm_s
```

```
climlab Process of type <class 'climlab.model.ebm.EBM_seasonal'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM_seasonal'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion
↪'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.DailyInsolation
↪'>
```

```
>>> #  creating and adding albedo feedback subprocess
>>> step_albedo = StepFunctionAlbedo(state=ebm_s.state, **ebm_s.
↪param)
```

(continues on next page)

(continued from previous page)

```
>>> ebm_s.add_subprocess('albedo', step_albedo)
>>>
>>> print ebm_s
```

```
climlab Process of type <class 'climlab.model.ebm.EBM_seasonal'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM_seasonal'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion
↪'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.DailyInsolation
↪'>
```

add_subprocesses (*procdict*)

Adds a dictionary of subproceses to this process.

Calls *add_subprocess()* (page 96) for every process given in the input-dictionary. It can also pass a single process, which will be given the name *default*.

Parameters procdict (*dict*) – a dictionary with process names as keys

declare_diagnostics (*diaglist*)

Add the variable names in `inputlist` to the list of diagnostics.

declare_input (*inputlist*)

Add the variable names in `inputlist` to the list of necessary inputs.

depth

Depth at grid centers (m)

Getter Returns the points of axis `'depth'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'depth'` axis can be found.

depth_bounds

Depth at grid interfaces (m)

Getter Returns the bounds of axis `'depth'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'depth'` axis can be found.

diagnostics

Dictionary access to all diagnostic variables

Type dict

input

Dictionary access to all input variables

That can be boundary conditions and other gridded quantities independent of the *process*

Type dict

lat
Latitude of grid centers (degrees North)

Getter Returns the points of axis `'lat'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lat'` axis can be found.

lat_bounds
Latitude of grid interfaces (degrees North)

Getter Returns the bounds of axis `'lat'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lat'` axis can be found.

lev
Pressure levels at grid centers (hPa or mb)

Getter Returns the points of axis `'lev'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lev'` axis can be found.

lev_bounds
Pressure levels at grid interfaces (hPa or mb)

Getter Returns the bounds of axis `'lev'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lev'` axis can be found.

lon
Longitude of grid centers (degrees)

Getter Returns the points of axis `'lon'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lon'` axis can be found.

lon_bounds
Longitude of grid interfaces (degrees)

Getter Returns the bounds of axis `'lon'` if availible in the process's domains.

Type array

Raises `ValueError` if no `'lon'` axis can be found.

remove_diagnostic(*name*)
Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.

Parameters **name** (*str*) – name of diagnostic quantity to be removed

Example Remove diagnostic variable 'icelat' from energy balance model:

```
>>> import climlab
>>> model = climlab.EBM()

>>> # display all diagnostic variables
>>> model.diagnostics.keys()
```

(continues on next page)

(continued from previous page)

```
['ASR', 'OLR', 'net_radiation', 'albedo', 'icelat']

>>> model.remove_diagnostic('icelat')
>>> model.diagnostics.keys()
['ASR', 'OLR', 'net_radiation', 'albedo']

>>> # Watch out for subprocesses that may still want
>>>  # to access the diagnostic 'icelat' variable !!!
```

remove_subprocess (*name*, *verbose=True*)

Removes a single subprocess from this process.

Parameters

- **name** (*string*) – name of the subprocess
- **verbose** (*bool*) – information whether warning message should be printed [default: True]

Example Remove albedo subprocess from energy balance model:

```
>>> import climlab
>>> model = climlab.EBM()

>>> print model
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion
↪'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>

>>> model.remove_subprocess('albedo')

>>> print model
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion
↪'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

set_state (*name*, *value*)

Sets the variable `name` to a new state `value`.

Parameters

- **name** (*string*) – name of the state
- **value** (*Field* (page 47) or *array*) – state variable

Raises ValueError if state variable value is not having a domain.

Raises ValueError if shape mismatch between existing domain and new state variable.

Example Resetting the surface temperature of an EBM to $-5°C$ on all latitues:

```
>>> import climlab
>>> from climlab import Field
>>> import numpy as np

>>> # setup model
>>> model = climlab.EBM(num_lat=36)

>>> # create new temperature distribution
>>> initial = -5 * ones(size(model.lat))
>>> model.set_state('Ts', Field(initial, domain=model.domains['Ts
↪']))

>>> np.squeeze(model.Ts)
Field([-5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -
↪5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -
↪5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5.])
```

to_xarray(*diagnostics=False*)

Convert process variables to xarray.Dataset format.

With diagnostics=True, both state and diagnostic variables are included.

Otherwise just the state variables are included.

Returns an xarray.Dataset object with all spatial axes, including 'bounds' axes indicating cell boundaries in each spatial dimension.

Example Create a single column radiation model and view as xarray object:

```
>>> import climlab
>>> state = climlab.column_state(num_lev=20)
>>> model = climlab.radiation.RRTMG(state=state)

>>> # display model state as xarray:
>>> model.to_xarray()
<xarray.Dataset>
Dimensions:        (depth: 1, depth_bounds: 2, lev: 20, lev_bounds:␣
↪21)
Coordinates:
  * depth          (depth) float64 0.5
  * depth_bounds   (depth_bounds) float64 0.0 1.0
  * lev            (lev) float64 25.0 75.0 125.0 175.0 225.0 275.0␣
↪325.0 ...
  * lev_bounds     (lev_bounds) float64 0.0 50.0 100.0 150.0 200.0␣
↪250.0 ...
Data variables:
    Ts             (depth) float64 288.0
    Tatm           (lev) float64 200.0 204.1 208.2 212.3 216.4 220.5␣
↪224.6 ...

>>> # take a single timestep to populate the diagnostic variables
>>> model.step_forward()
```

(continues on next page)

(continued from previous page)

```
>>> # Now look at the full output in xarray format
>>> model.to_xarray(diagnostics=True)
<xarray.Dataset>
Dimensions:           (depth: 1, depth_bounds: 2, lev: 20, lev_
↪bounds: 21)
Coordinates:
  * depth             (depth) float64 0.5
  * depth_bounds      (depth_bounds) float64 0.0 1.0
  * lev               (lev) float64 25.0 75.0 125.0 175.0 225.0 275.
↪0 325.0 ...
  * lev_bounds        (lev_bounds) float64 0.0 50.0 100.0 150.0 200.
↪0 250.0 ...
Data variables:
    Ts                (depth) float64 288.7
    Tatm              (lev) float64 201.3 204.0 208.0 212.0 216.1␣
↪220.2 ...
    ASR               (depth) float64 240.0
    ASRcld            (depth) float64 0.0
    ASRclr            (depth) float64 240.0
    LW_flux_down      (lev_bounds) float64 0.0 12.63 19.47 26.07 32.
↪92 40.1 ...
    LW_flux_down_clr  (lev_bounds) float64 0.0 12.63 19.47 26.07 32.
↪92 40.1 ...
    LW_flux_net       (lev_bounds) float64 240.1 231.2 227.6 224.1␣
↪220.5 ...
    LW_flux_net_clr   (lev_bounds) float64 240.1 231.2 227.6 224.1␣
↪220.5 ...
    LW_flux_up        (lev_bounds) float64 240.1 243.9 247.1 250.2␣
↪253.4 ...
    LW_flux_up_clr    (lev_bounds) float64 240.1 243.9 247.1 250.2␣
↪253.4 ...
    LW_sfc            (depth) float64 128.9
    LW_sfc_clr        (depth) float64 128.9
    OLR               (depth) float64 240.1
    OLRcld            (depth) float64 0.0
    OLRclr            (depth) float64 240.1
    SW_flux_down      (lev_bounds) float64 341.3 323.1 318.0 313.5␣
↪309.5 ...
    SW_flux_down_clr  (lev_bounds) float64 341.3 323.1 318.0 313.5␣
↪309.5 ...
    SW_flux_net       (lev_bounds) float64 240.0 223.3 220.2 217.9␣
↪215.9 ...
    SW_flux_net_clr   (lev_bounds) float64 240.0 223.3 220.2 217.9␣
↪215.9 ...
    SW_flux_up        (lev_bounds) float64 101.3 99.88 97.77 95.64␣
↪93.57 ...
    SW_flux_up_clr    (lev_bounds) float64 101.3 99.88 97.77 95.64␣
↪93.57 ...
    SW_sfc            (depth) float64 163.8
    SW_sfc_clr        (depth) float64 163.8
    TdotLW            (lev) float64 -1.502 -0.6148 -0.5813 -0.6173 -
↪0.6426 ...
    TdotLW_clr        (lev) float64 -1.502 -0.6148 -0.5813 -0.6173 -
↪0.6426 ...
    TdotSW            (lev) float64 2.821 0.5123 0.3936 0.3368 0.
↪3174 0.3299 ...
    TdotSW_clr        (lev) float64 2.821 0.5123 0.3936 0.3368 0.
↪3174 0.3299 ...
```

(continues on next page)

(continued from previous page)

```
```

climlab.process.process.**get_axes**(*process_or_domain*)

Returns a dictionary of all Axis in a domain or dictionary of domains.

Parameters **process_or_domain** (*Process* (page 93) or *_Domain* (page 42)) – a process or a domain object

Raises

exc *TypeError* if input is not or not having a domain

Returns dictionary of input's Axis

Return type dict

Example

```
>>> import climlab
>>> from climlab.process.process import get_axes

>>> model = climlab.EBM()

>>> get_axes(model)
{'lat': <climlab.domain.axis.Axis object at 0x7ff13b9dd2d0>,
 'depth': <climlab.domain.axis.Axis object at 0x7ff13b9dd310>}
```

climlab.process.process.**process_like**(*proc*)

Make an exact clone of a process, including state and all subprocesses.

The creation date is updated.

Parameters **proc** (*Process* (page 93)) – process

Returns new process identical to the given process

Return type *Process* (page 93)

Example

```
>>> import climlab
>>> from climlab.process.process import process_like

>>> model = climlab.EBM()
>>> model.subprocess.keys()
['diffusion', 'LW', 'albedo', 'insolation']

>>> albedo = model.subprocess['albedo']
>>> albedo_copy = process_like(albedo)

>>> albedo.creation_date
'Thu, 24 Mar 2016 01:32:25 +0000'

>>> albedo_copy.creation_date
'Thu, 24 Mar 2016 01:33:29 +0000'
```

10.5.6 time_dependent_process

class climlab.process.time_dependent_process.**TimeDependentProcess**(*time_type='explicit'*, *timestep=None*, *topdown=True*, ***kwargs*)

Bases: *climlab.process.process.Process* (page 93)

A generic parent class for all time-dependent processes.

TimeDependentProcess is a child of the *Process* (page 93) class and therefore inherits all those attributes.

Initialization parameters

An instance of TimeDependentProcess is initialized with the following arguments *(for detailed information see Object attributes below)*:

Parameters

- **timestep** (*float*) – specifies the timestep of the object (optional)
- **time_type** (*str*) – how time-dependent-process should be computed [default: 'explicit']
- **topdown** (*bool*) – whether geneterate *process_types* in regular or in reverse order [default: True]

Object attributes

Additional to the parent class *Process* (page 93) following object attributes are generated during initialization:

Variables

- **has_process_type_list** (*bool*) – information whether attribute *process_types* (which is needed for *compute()* (page 105) and build in _build_process_type_list()) exists or not. Attribute is set to 'False' during initialization.
- **topdown** (*bool*) – information whether the list *process_types* (which contains all processes and sub-processes) should be generated in regular or in reverse order. See _build_process_type_list().
- **timeave** (*dict*) – a time averaged collection of all states and diagnostic processes over the timeperiod that *integrate_years()* (page 106) has been called for last.
- **tendencies** (*dict*) – computed difference in a timestep for each state. See *compute()* (page 105) for details.
- **time_type** (*str*) – how time-dependent-process should be computed. Possible values are: 'explicit', 'implicit', 'diagnostic', 'adjustment'.
- **time** (*dict*) –

a collection of all time-related attributes of the process. The dictionary contains following items:

- 'timestep': see initialization parameter
- 'num_steps_per_year': see *set_timestep()* (page 107) and *timestep()* (page 108) for details
- 'day_of_year_index': counter how many steps have been integrated in current year
- 'steps': counter how many steps have been integrated in total
- 'days_elapsed': time counter for days
- 'years_elapsed': time counter for years
- 'days_of_year': array which holds the number of numerical steps per year, expressed in days

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

***timestep* (page 108)** The amount of time over which *step_forward()* (page 107) is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute (page 105)()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics (page 105)([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.

Continued on next page

Table 26 – continued from previous page

declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge (page 105)([crit, verbose])	Integrates the model until model states are converging.
integrate_days (page 106)([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years (page 106)([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state (page 107)(name, value)	Sets the variable name to a new state value.
set_timestep (page 107)([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of *timestep()* (page 108)
step_forward (page 107)()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

compute()

Computes the tendencies for all state variables given current state and specified input.

The function first computes all diagnostic processes. They don't produce any tendencies directly but they may affect the other processes (such as change in solar distribution). Subsequently, all tendencies and diagnostics for all explicit processes are computed.

Tendencies due to implicit and adjustment processes need to be calculated from a state that is already adjusted after explicit alteration. For that reason the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated by matrix inversions and similar to the explicit tendencies, the implicit ones are applied to the states temporarily. Subsequently, all instantaneous adjustments are computed.

Then the changes that were made to the states from explicit and implicit processes are removed again as this *compute()* (page 105) function is supposed to calculate only tendencies and not apply them to the states.

Finally, all calculated tendencies from all processes are collected for each state, summed up and stored in the dictionary self.tendencies, which is an attribute of the time-dependent-process object, for which the *compute()* (page 105) method has been called.

Object attributes

During method execution following object attributes are modified:

Variables

- **tendencies** (*dict*) – dictionary that holds tendencies for all states is calculated for current timestep through adding up tendencies from explicit, implicit and adjustment processes.
- ***diagnostics*** (page 97) (*dict*) – process diagnostic dictionary is updated by diagnostic dictionaries of subprocesses after computation of tendencies.

compute_diagnostics(*num_iter=3*)

Compute all tendencies and diagnostics, but don't update model state. By default it will call compute() 3 times to make sure all subprocess coupling is accounted for. The number of iterations can be changed with the input argument.

integrate_converge (*crit=0.0001*, *verbose=True*)
Integrates the model until model states are converging.

Parameters

- **crit** (*float*) – exit criteria for difference of iterated solutions [default: 0.0001]
- **verbose** (*bool*) – information whether total elapsed time should be printed [default: True]

Example

```
>>> import climlab
>>> model = climlab.EBM()

>>> model.global_mean_temperature()
Field(11.997968598413685)

>>> model.integrate_converge()
Total elapsed time is 10.0 years.

>>> model.global_mean_temperature()
Field(14.288155406577301)
```

integrate_days (*days=1.0*, *verbose=True*)
Integrates the model forward for a specified number of days.

It convertes the given number of days into years and calls *integrate_years()* (page 106).

Parameters

- **days** (*float*) – integration time for the model in days [default: 1.0]
- **verbose** (*bool*) – information whether model time details should be printed [default: True]

Example

```
>>> import climlab
>>> model = climlab.EBM()

>>> model.global_mean_temperature()
Field(11.997968598413685)

>>> model.integrate_days(80.)
Integrating for 19 steps, 80.0 days, or 0.219032740466 years.
Total elapsed time is 0.211111111111 years.

>>> model.global_mean_temperature()
Field(11.873680783355553)
```

integrate_years (*years=1.0*, *verbose=True*)
Integrates the model by a given number of years.

Parameters

- **years** (*float*) – integration time for the model in years [default: 1.0]
- **verbose** (*bool*) – information whether model time details should be printed [default: True]

It calls *step_forward()* (page 107) repetitively and calculates a time averaged value over the integrated period for every model state and all diagnostics processes.

Example

```
>>> import climlab
>>> model = climlab.EBM()

>>> model.global_mean_temperature()
Field(11.997968598413685)

>>> model.integrate_years(2.)
Integrating for 180 steps, 730.4844 days, or 2.0 years.
Total elapsed time is 2.0 years.

>>> model.global_mean_temperature()
Field(13.531055349437258)
```

set_state (*name*, *value*)

Sets the variable `name` to a new state `value`.

Parameters

- **name** (*string*) – name of the state
- **value** (*Field* (page 47) or *array*) – state variable

Raises `ValueError` if state variable `value` is not having a domain.

Raises `ValueError` if shape mismatch between existing domain and new state variable.

Example Resetting the surface temperature of an EBM to $-5°C$ on all latitues:

```
>>> import climlab
>>> from climlab import Field
>>> import numpy as np

>>> # setup model
>>> model = climlab.EBM(num_lat=36)

>>> # create new temperature distribution
>>> initial = -5 * ones(size(model.lat))
>>> model.set_state('Ts', Field(initial, domain=model.domains['Ts
↪']))

>>> np.squeeze(model.Ts)
Field([-5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -
↪5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -
↪5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5.])
```

set_timestep (*timestep=86400.0*, *num_steps_per_year=None*)

Calculates the timestep in unit seconds and calls the setter function of *timestep()* (page 108)

Parameters

- **timestep** (*float*) – the amount of time over which *step_forward()* (page 107) is integrating in unit seconds [default: 24*60*60]
- **num_steps_per_year** (*float*) – a number of steps per calendar year (optional)

If the parameter *num_steps_per_year* is specified and not `None`, the timestep is calculated accordingly and therefore the given input parameter *timestep* is ignored.

step_forward()

Updates state variables with computed tendencies.

Calls the *compute()* (page 105) method to get current tendencies for all process states. Multiplied with the timestep and added up to the state variables is updating all model states.

Example

```
>>> import climlab
>>> model = climlab.EBM()

>>> # checking time step counter
>>> model.time['steps']
0

>>> # stepping the model forward
>>> model.step_forward()

>>> # step counter increased
>>> model.time['steps']
1
```

timestep

The amount of time over which *step_forward()* (page 107) is integrating in unit seconds.

Getter Returns the object timestep which is stored in `self.param['timestep']`.

Setter Sets the timestep to the given input. See also *set_timestep()* (page 107).

Type float

climlab.process.time_dependent_process.**couple**(*proclist*, *name='Parent'*)

10.6 climlab.radiation

Modules for radiative transfer in vertical columns, along with processes for insolation and fixed relative humidity.

10.6.1 AplusBT

class climlab.radiation.aplusbt.**AplusBT**(*A=200.0*, *B=2.0*, ***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

The simplest linear longwave radiation module.

Calculates the Outgoing Longwave Radiation (OLR) $R\uparrow$ as

$$R\uparrow = A + B \cdot T$$

where T is the state variable.

Should be invoked with a single temperature state variable only.

Initialization parameters n

An instance of `AplusBT` is initialized with the following arguments:

Parameters

- **A** (*float*) – parameter for linear OLR parametrization n - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$ n - default value: `200.0`
- **B** (*float*) – parameter for linear OLR parametrization n - unit: $\frac{\mathrm{W}}{\mathrm{m}^2 {}^\circ\mathrm{C}}$ n - default value: `2.0`

Object attributes n

Additional to the parent class *EnergyBudget* (page 86) following object attributes are generated or modified during initialization:

Variables

- ***A*** (page 111) (*float*) – calls the setter function of *A()* (page 111)
- ***B*** (page 111) (*float*) – calls the setter function of *B()* (page 111)
- ***diagnostics*** (page 97) (*dict*) – key `'OLR'` initialized with value: *Field* (page 47) of zeros in size of `self.Ts`
- **OLR** (`Field` (page 47)) – the subprocess attribute `self.OLR` is created with correct dimensions

Warning: This module currently works only for a single state variable!

Example Simple linear radiation module (stand alone):

```
>>> import climlab

>>> # create a column atmosphere and scalar surface
>>> sfc, atm = climlab.domain.single_column()

>>> # Create a state variable
>>> Ts = climlab.Field(15., domain=sfc)

>>> # Make a dictionary of state variables
>>> s = {'Ts': Ts}

>>> # create process
>>> olr = climlab.radiation.AplusBT(state=s)

>>> print olr
climlab Process of type <class 'climlab.radiation.AplusBT.AplusBT'>.
State variables and domain shapes:
  Ts: (1,)
The subprocess tree:
top: <class 'climlab.radiation.AplusBT.AplusBT'>

>>> # to compute tendencies and diagnostics
>>> olr.compute()

>>> #  or to actually update the temperature
```

(continues on next page)

(continued from previous page)

```
>>> olr.step_forward()

>>> print olr.state
{'Ts': Field([ 5.69123176])}
```

Attributes

`A` **(page 111)** Property of AplusBT parameter A.

`B` **(page 111)** Property of AplusBT parameter B.

`depth` Depth at grid centers (m)

`depth_bounds` Depth at grid interfaces (m)

`diagnostics` Dictionary access to all diagnostic variables

`input` Dictionary access to all input variables

`lat` Latitude of grid centers (degrees North)

`lat_bounds` Latitude of grid interfaces (degrees North)

`lev` Pressure levels at grid centers (hPa or mb)

`lev_bounds` Pressure levels at grid interfaces (hPa or mb)

`lon` Longitude of grid centers (degrees)

`lon_bounds` Longitude of grid interfaces (degrees)

`timestep` The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.

Continued on next page

Table 27 – continued from previous page

remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

A

Property of AplusBT parameter A.

Getter Returns the parameter A which is stored in attribute `self._A`

Setter

- sets parameter A which is addressed as `self._A` to the new value
- updates the parameter dictionary `self.param['A']`

Type float

Example

```
>>> import climlab
>>> model = climlab.EBM()

>>> # getter
>>> model.subprocess['LW'].A
210.0
>>> # setter
>>> model.subprocess['LW'].A = 220
>>> # getter again
>>> model.subprocess['LW'].A
220

>>> # subprocess parameter dictionary
>>> model.subprocess['LW'].param['A']
220
```

B

Property of AplusBT parameter B.

Getter Returns the parameter B which is stored in attribute `self._B`

Setter

- sets parameter B which is addressed as `self._B` to the new value
- updates the parameter dictionary `self.param['B']`

Type float

class climlab.radiation.aplusbt.**AplusBT_CO2**(*CO2=300.0*, ***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

Linear longwave radiation module considering CO2 concentration.

This radiation subprocess is based in the idea to linearize the Outgoing Longwave Radiation (OLR) emitted to space according to the surface temperature (see *AplusBT* (page 108)).

To consider a the change of the greenhouse effect through range of CO_2 in the atmosphere, the parameters A and B are computed like the following:

$$A(c) = -326.4 + 9.161c - 3.164c^2 + 0.5468c^3$$
$$B(c) = 1.953 - 0.04866c + 0.01309c^2 - 0.002577c^3$$

where $c = \log \frac{p}{300}$ and p represents the concentration of CO_2 in the atmosphere.

For further reading see [Caldeira_1992].

Initialization parameters

An instance of `AplusBT_CO2` is initialized with the following argument:

Parameters **CO2** (*float*) – The concentration of CO_2 in the atmosphere. Referred to as p in the above given formulas.

- unit: ppm (parts per million)
- default value: `300.0`

Object attributes

Additional to the parent class *EnergyBudget* (page 86) following object attributes are generated or updated during initialization:

Variables

- ***CO2*** (page 114) (*float*) – calls the setter function of `CO2()` (page 114)
- ***diagnostics*** (page 97) (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.add_diagnostic('OLR', 0. * self.Ts)`
- **OLR** (`Field` (page 47)) – the subprocess attribute `self.OLR` is created with correct dimensions

Example Replacing an the regular AplusBT subprocess in an energy balance model:

```
>>> import climlab
>>> from climlab.radiation.AplusBT import AplusBT_CO2

>>> # creating EBM model
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> #  creating and adding albedo feedback subprocess
>>> LW_CO2 = AplusBT_CO2(CO2=400, state=model.state, **model.param)

>>> # overwriting old 'LW' subprocess with same name
>>> model.add_subprocess('LW', LW_CO2)

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT_CO2'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

Attributes

CO2 **(page 114)** Property of AplusBT_CO2 parameter CO2.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
add_input(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 28 – continued from previous page

compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

CO2

Property of AplusBT_CO2 parameter CO2.

Getter Returns the CO2 concentration which is stored in attribute `self._CO2`

Setter

- sets the CO2 concentration which is addressed as `self._CO2` to the new value
- updates the parameter dictionary `self.param['CO2']`

Type float

10.6.2 Boltzmann

class `climlab.radiation.boltzmann.`**`Boltzmann`**(*eps=0.65*, *tau=0.95*, ***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

A class for black body radiation.

Implements a radiation subprocess which computes longwave radiation with the Stefan-Boltzmann law for black/grey body radiation.

According to the Stefan Boltzmann law the total power radiated from an object with surface area A and temperature T (in unit Kelvin) can be written as

$$P = A\varepsilon\sigma T^4$$

where ε is the emissivity of the body.

As the *EnergyBudget* (page 86) of the Energy Balance Model is accounted in unit energy/area (W/m^2) the energy budget equation looks like this:

$$C\frac{dT}{dt} = R\downarrow - R\uparrow - H$$

The *Boltzmann* (page 114) radiation subprocess represents the outgoing radiation $R\uparrow$ which then can be written as

$$R\uparrow = \varepsilon\sigma T^4$$

with state variable T.

Initialization parameters n

An instance of `Boltzmann` is initialized with the following arguments:

Parameters

- **eps** (*float*) – emissivity of the planet's surface which is the effectiveness in emitting energy as thermal radiation [default: 0.65]
- **tau** (*float*) – transmissivity of the planet's atmosphere which is the effectiveness in transmitting the longwave radiation emitted from the surface [default: 0.95]

Object attributes n

During initialization both arguments described above are created as object attributes which calls their setter function (see below).

Variables

- ***eps*** (page 117) (*float*) – calls the setter function of *eps()* (page 117)
- ***tau*** (page 117) (*float*) – calls the setter function of *tau()* (page 117)
- ***diagnostics*** (page 97) (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.add_diagnostic('OLR', 0. * self.Ts)`
- **OLR** (`Field` (page 47)) – the subprocess attribute `self.OLR` is created with correct dimensions

Example Replacing an the regular AplusBT subprocess in an energy balance model:

```
>>> import climlab
>>> from climlab.radiation.Boltzmann import Boltzmann

>>> # creating EBM model
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> #  creating and adding albedo feedback subprocess
>>> LW_boltz = Boltzmann(eps=0.69, tau=0.98, state=model.state,
↪**model.param)

>>> # overwriting old 'LW' subprocess with same name
>>> model.add_subprocess('LW', LW_boltz)

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.Boltzmann.Boltzmann'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

eps **(page 117)** Property of emissivity parameter.

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

tau **(page 117)** Property of the transmissivity parameter.

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

eps

Property of emissivity parameter.

Getter Returns the albedo value which is stored in attribute `self._eps`

Setter

- sets the emissivity which is addressed as `self._eps` to the new value
- updates the parameter dictionary `self.param['eps']`

Type float

tau

Property of the transmissivity parameter.

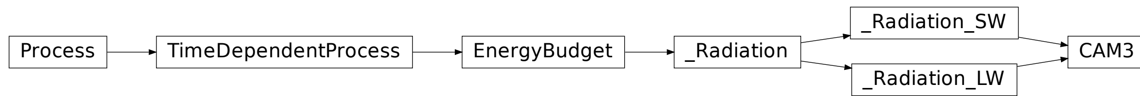
Getter Returns the albedo value which is stored in attribute `self._tau`

Setter

- sets the emissivity which is addressed as `self._tau` to the new value
- updates the parameter dictionary `self.param['tau']`

Type float

10.6.3 CAM3



climlab wrappers for the NCAR CAM3 radiation code

Input arguments and diagnostics follow specifications in `_Radiation`

Example Here is a quick example of setting up a single-column Radiative-Convective model with fixed relative humdity:

```
import climlab
alb = 0.25
#  State variables (Air and surface temperature)
state = climlab.column_state(num_lev=30)
#  Parent model process
rcm = climlab.TimeDependentProcess(state=state)
#  Fixed relative humidity
h2o = climlab.radiation.ManabeWaterVapor(state=state)
#  Couple water vapor to radiation
rad = climlab.radiation.CAM3(state=state, specific_humidity=h2o.q,
↪ albedo=alb)
#  Convective adjustment
conv = climlab.convection.ConvectiveAdjustment(state=state, adj_
↪lapse_rate=6.5)
#  Couple everything together
rcm.add_subprocess('Radiation', rad)
rcm.add_subprocess('WaterVapor', h2o)
rcm.add_subprocess('Convection', conv)
#  Run the model
rcm.integrate_years(1)
#  Check for energy balance
print rcm.ASR - rcm.OLR
```

class `climlab.radiation.cam3.cam3.`**`CAM3`**(***kwargs*)

Bases: *`climlab.radiation.radiation._Radiation_SW`* (page 142), *`climlab.radiation.radiation._Radiation_LW`* (page 141)

climlab wrapper for the CAM3 radiation code.

For some details about inputs and diagnostics, see the *radiation* module.

Attributes

`depth` Depth at grid centers (m)

`depth_bounds` Depth at grid interfaces (m)

`diagnostics` Dictionary access to all diagnostic variables

`input` Dictionary access to all input variables

`lat` Latitude of grid centers (degrees North)

`lat_bounds` Latitude of grid interfaces (degrees North)

`lev` Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

_cam3_to_climlab(*field*)

Output is either (KM, JM, 1) or (JM, 1). Transform this to...

- (KM,) or (1,) if JM==1
- (KM, JM) or (JM, 1) if JM>1

(longitude dimension IM not yet implemented).

_climlab_to_cam3(*field*)

Prepare field with proper dimension order. CAM3 code expects 3D arrays with (KM, JM, 1) and 2D arrays with (JM, 1).

climlab grid dimensions are any of:

- (KM,)

- (JM, KM)
- (JM, IM, KM)

(longitude dimension IM not yet implemented here).

_compute_heating_rates()

Computes energy flux convergences to get heating rates in unit W/m^2.

This method should be over-ridden by daughter classes.

_prepare_arguments()

class climlab.radiation.cam3.cam3.**CAM3_LW**(***kwargs*)

Bases: *climlab.radiation.cam3.cam3.CAM3* (page 118)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.

Continued on next page

Table 31 – continued from previous page

integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.radiation.cam3.cam3.**CAM3_SW**(***kwargs*)

Bases: *climlab.radiation.cam3.cam3.CAM3* (page 118)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.

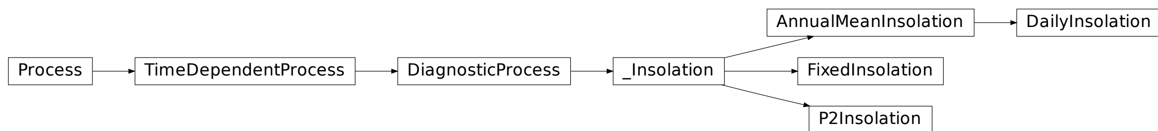
Continued on next page

Table 32 – continued from previous page

integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

climlab.radiation.cam3.cam3.**init_cam3**(*mod*)

10.6.4 insolation



Classes to provide insolation as input for other CLIMLAB processes.

Options include

- climlab.radiation.P2Insolation (idealized 2nd Legendre polynomial form)
- climlab.radiation.FixedInsolation (generic steady-state insolation)
- climlab.radiation.AnnualMeanInsolation (steady-state annual-mean insolation computed from orbital parameters and latitude)
- climlab.radiation.DailyInsolation (time-varying daily-mean insolation computed from orbital parameters, latitude and time of year)

All are subclasses of climlab.process.DiagnosticProcess and do add any tendencies to any state variables.

At least two diagnostics are provided:

- insolation, the incoming solar radiation in Wm^2
- coszen, cosine of the solar zenith angle

class climlab.radiation.insolation.**AnnualMeanInsolation**(*S0=1365.2, orb={'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}, **kwargs*)

Bases: climlab.radiation.insolation._Insolation

A class for latitudewise solar insolation averaged over a year.

This class computes the solar insolation for each day of the year and latitude specified in the domain on the basis of orbital parameters and astronomical formulas.

Therefore it uses the method *daily_insolation()* (page 155). For details how the solar distribution is dependend on orbital parameters see there.

The mean over the year is calculated from data given by *daily_insolation()* (page 155) and stored in the object's attribute `self.insolation`

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{W}{m^2}$
  - default value: `1365.2`
- **orb** (*dict*) – a dictionary with three orbital parameters (as provided by `OrbitalTable`):
  - `'ecc'` - eccentricity
    - unit: dimensionless
    - default value: `0.017236`
  - `'long_peri'` - longitude of perihelion (precession angle)
    - unit: degrees
    - default value: `281.37`
  - `'obliquity'` - obliquity angle
    - unit: degrees
    - default value: `23.446`

Object attributes

Additional to the parent class `_Insolation` following object attributes are generated and updated during initialization:

Variables

- ***insolation*** (page 122) (`Field` (page 47)) – Current insolation in W/m2
- **coszen** (`Field` (page 47)) – Cosine of the current solar zenith angle
- ***orb*** (page 125) (*dict*) – initialized with given argument `orb`

Example Create regular EBM and replace standard insolation subprocess by `AnnualMeanInsolation`:

```
>>> import climlab
>>> from climlab.radiation import AnnualMeanInsolation

>>> # model creation
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> # catch model domain for subprocess creation
>>> sfc = model.domains['Ts']

>>> # create AnnualMeanInsolation subprocess
>>> new_insol = AnnualMeanInsolation(domains=sfc, **model.param)

>>> # add it to the model
>>> model.add_subprocess('insolation',new_insol)

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.
↪AnnualMeanInsolation'>
```

Attributes

S0 Property of solar constant S0.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

orb **(page 125)** Property of dictionary for orbital parameters.

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

orb

Property of dictionary for orbital parameters.

orb contains: (for more information see `OrbitalTable`)

- `'ecc'` - eccentricity [unit: dimensionless]
- `'long_peri'` - longitude of perihelion (precession angle) [unit: degrees]
- `'obliquity'` - obliquity angle [unit: degrees]

Getter Returns the orbital dictionary which is stored in attribute `self._orb`.

Setter

- sets orb which is addressed as `self._orb` to the new value
- updates the parameter dictionary `self.param['orb']` and
- calls method `_compute_fixed()`

Type dict

class climlab.radiation.insolation.**DailyInsolation**(*S0=1365.2, orb={'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}, **kwargs*)

Bases: *climlab.radiation.insolation.AnnualMeanInsolation* (page 122)

A class to compute latitudewise daily solar insolation for specific days of the year.

This class computes the solar insolation on basis of orbital parameters and astronomical formulas.

Therefore it uses the method *daily_insolation()* (page 155). For details how the solar distribution is dependend on orbital parameters see there.

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `1365.2`
- **orb** (*dict*) – a dictionary with orbital parameters:
  - `'ecc'` - eccentricity
    - unit: dimensionless
    - default value: `0.017236`
  - `'long_peri'` - longitude of perihelion (precession angle)
    - unit: degrees
    - default value: `281.37`
  - `'obliquity'` - obliquity angle
    - unit: degrees
    - default value: `23.446`

Object attributes

Additional to the parent class `_Insolation` following object attributes are generated and updated during initialization:

Variables

- ***insolation*** (page 122) (`Field` (page 47)) – Current insolation in W/m2
- **coszen** (`Field` (page 47)) – Cosine of the current solar zenith angle
- ***orb*** (page 125) (*dict*) – initialized with given argument `orb`

Example Create regular EBM and replace standard insolation subprocess by `DailyInsolation`:

```
>>> import climlab
>>> from climlab.radiation import DailyInsolation

>>> # model creation
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> # catch model domain for subprocess creation
>>> sfc = model.domains['Ts']

>>> # create DailyInsolation subprocess and add it to the model
>>> model.add_subprocess('insolation',DailyInsolation(domains=sfc,␣
→**model.param))

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.DailyInsolation'>
```

Attributes

S0 Property of solar constant S0.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

orb Property of dictionary for orbital parameters.

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class `climlab.radiation.insolation.`**`FixedInsolation`**(*S0=341.3*, ***kwargs*)

Bases: `climlab.radiation.insolation._Insolation`

A class for fixed insolation at each point of latitude off the domain.

The solar distribution for the whole domain is constant and specified by a parameter.

Initialization parameters

Parameters **S0** (*float*) – solar constant

- unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
- default value: `const.S0/4 = 341.2`

Example

```
>>> import climlab
>>> from climlab.radiation.insolation import FixedInsolation

>>> model = climlab.EBM()
>>> sfc = model.Ts.domain

>>> fixed_ins = FixedInsolation(S0=340.0, domains=sfc)
```

(continues on next page)

(continued from previous page)

```
>>> print fixed_ins
climlab Process of type <class 'climlab.radiation.insolation.
↪FixedInsolation'>.
State variables and domain shapes:
The subprocess tree:
top: <class 'climlab.radiation.insolation.FixedInsolation'>
```

Attributes

S0 Property of solar constant S0.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.

Continued on next page

Table 35 – continued from previous page

remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.radiation.insolation.**P2Insolation**(*S0=1365.2*, *s2=-0.48*, ***kwargs*)

Bases: climlab.radiation.insolation._Insolation

A class for parabolic solar distribution over the domain's latitude on the basis of the second order Legendre Polynomial.

Calculates the latitude dependent solar distribution as

$$S(\varphi) = \frac{S_0}{4} \left(1 + s_2 P_2(x) \right)$$

where $P_2(x) = \frac{1}{2}(3x^2 - 1)$ is the second order Legendre Polynomial and $x = sin(\varphi)$.

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: 1365.2
- **s2** (*floar*) – factor for second legendre polynominal term
  - default value: -0.48

Example

```
>>> import climlab
>>> from climlab.radiation.insolation import P2Insolation

>>> model = climlab.EBM()
>>> sfc = model.Ts.domain

>>> p2_ins = P2Insolation(S0=340.0, s2=-0.5, domains=sfc)

>>> print p2_ins
climlab Process of type <class 'climlab.radiation.insolation.
↪P2Insolation'>.
State variables and domain shapes:
The subprocess tree:
top: <class 'climlab.radiation.insolation.P2Insolation'>
```

Attributes

S0 Property of solar constant S0.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

s2 **(page 131)** Property of second legendre polynomial factor s2.

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

s2

Property of second legendre polynomial factor s2.

s2 in following equation:

$$S(\varphi) = \frac{S_0}{4} \left(1 + s_2 P_2(x)\right)$$

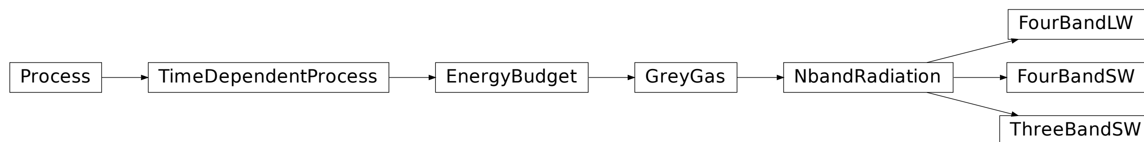
Getter Returns the s2 parameter which is stored in attribute `self._s2`.

Setter

- sets s2 which is addressed as `self._S0` to the new value
- updates the parameter dictionary `self.param['s2']` and
- calls method `_compute_fixed()`

Type float

10.6.5 nband



class `climlab.radiation.nband.`**`FourBandLW`**(***kwargs*)

Bases: *climlab.radiation.nband.NbandRadiation* (page 135)

Closely following SPEEDY / MITgcm longwave model band 0 is window region (between 8.5 and 11 microns) band 1 is CO2 channel (the band of strong absorption by CO2 around 15 microns) band 2 is weak H2O channel (aggregation of spectral regions with weak to moderate absorption by H2O) band 3 is strong H2O channel (aggregation of regions with strong absorption by H2O)

Attributes

absorptivity

band_fraction

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

emissivity

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

reflectivity

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

transmissivity

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`flux_components_bottom`()	Compute the contributions to the downwelling flux to surface due to emissions from each level.
`flux_components_top`()	Compute the contributions to the outgoing flux to space due to emissions from each level and the surface.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class `climlab.radiation.nband.`**`FourBandSW`**(*emissivity_sfc=0.0*, ***kwargs*)

Bases: `climlab.radiation.nband.NbandRadiation` (page 135)

A four-band mdoel for shortwave radiation.

The spectral decomposition used here is largely based on the "Moist Radiative-Convective Model" by Aarnout van Delden, Utrecht University a.j.vandelden@uu.nl http://www.staff.science.uu.nl/~delde102/RCM.htm

Four SW channels: channel 0 is Hartley and Huggins band (UV, 6%, <340 nm) channel 1 is part of visible with no O3 absorption (14%, 340 - 500 nm) channel 2 is Chappuis band (27%, 500 - 700 nm) channel 3 is near-infrared (53%, > 700 nm)

Attributes

absorptivity

band_fraction

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

emissivity

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

reflectivity

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

transmissivity

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`flux_components_bottom`()	Compute the contributions to the downwelling flux to surface due to emissions from each level.
`flux_components_top`()	Compute the contributions to the outgoing flux to space due to emissions from each level and the surface.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.

Continued on next page

Table 38 – continued from previous page

integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

emissivity

class climlab.radiation.nband.**NbandRadiation**(*absorber_vmr=None*, ***kwargs*)

Bases: climlab.radiation.greygas.GreyGas

Process for radiative transfer. Solves the discretized Schwarschild two-stream equations with the spectrum divided into N spectral bands.

Every NbandRadiation object has an attribute self.band_fraction with sum(self.band_fraction) == 1 that gives the fraction of the total beam in each band

Also a dictionary self.absorber_vmr that gives the volumetric mixing ratio of every absorbing gas on the same grid as temperature

and a dictionary self.absorption_cross_section that gives the absorption cross-section per unit mass for each gas in every spectral band

Attributes

absorptivity

band_fraction

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

emissivity

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

reflectivity

timestep The amount of time over which step_forward() is integrating in unit seconds.

transmissivity

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`flux_components_bottom`()	Compute the contributions to the downwelling flux to surface due to emissions from each level.
`flux_components_top`()	Compute the contributions to the outgoing flux to space due to emissions from each level and the surface.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

band_fraction

climlab.radiation.nband.**SPEEDY_band_fraction**(*T*)

Python / numpy implementation of the formula used by SPEEDY and MITgcm to partition longwave emissions into 4 spectral bands.

Input: temperature in Kelvin

returns: a four-element array of band fraction

Reproducing here the FORTRAN code from MITgcm/pkg/aim_v23/phy_radiat.F

```
EPS3=0.95 _d 0

DO JTEMP=200,320
  FBAND(JTEMP,0)= EPSLW
  FBAND(JTEMP,2)= 0.148 _d 0 - 3.0 _d -6 *(JTEMP-247)**2
```

(continues on next page)

(continued from previous page)

```
    FBAND(JTEMP,3)=(0.375 _d 0 - 5.5 _d -6 *(JTEMP-282)**2)*EPS3
    FBAND(JTEMP,4)= 0.314 _d 0 + 1.0 _d -5 *(JTEMP-315)**2
    FBAND(JTEMP,1)= 1. _d 0 -(FBAND(JTEMP,0)+FBAND(JTEMP,2)
&                          +FBAND(JTEMP,3)+FBAND(JTEMP,4))
  ENDDO

  DO JB=0,NBAND
    DO JTEMP=lwTemp1,199
      FBAND(JTEMP,JB)=FBAND(200,JB)
    ENDDO
    DO JTEMP=321,lwTemp2
      FBAND(JTEMP,JB)=FBAND(320,JB)
    ENDDO
  ENDDO
```

class climlab.radiation.nband.**ThreeBandSW**(*emissivity_sfc=0.0*, ***kwargs*)

Bases: `climlab.radiation.nband.NbandRadiation` (page 135)

A three-band mdoel for shortwave radiation.

The spectral decomposition used here is largely based on the "Moist Radiative-Convective Model" by Aarnout van Delden, Utrecht University a.j.vandelden@uu.nl http://www.staff.science.uu.nl/~delde102/RCM.htm

Three SW channels: channel 0 is Hartley and Huggins band (UV, 1%, 200 - 340 nm) channel 1 is Chappuis band (27%, 450 - 800 nm) channel 2 is remaining radiation (72%)

Attributes

absorptivity

band_fraction

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

emissivity

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

reflectivity

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

transmissivity

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
flux_components_bottom()	Compute the contributions to the downwelling flux to surface due to emissions from each level.
flux_components_top()	Compute the contributions to the outgoing flux to space due to emissions from each level and the surface.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

emissivity

10.6.6 Radiation

_Radiation, _Radiation_SW and _Radiation_LW are the base classes for radiative transfer modules

Currently this includes CAM3, RRTMG, RRTMG_LW, and RRTMG_SW

Basic characteristics:

State:

- Ts (surface radiative temperature)
- Tatm (air temperature)

Input arguments (both LW and SW):

- specific_humidity (kg/kg)

- `absorber_vmr = None` (dictionary of volumetric mixing ratios. Default values supplied if `None`)
- `cldfrac` (layer cloud fraction
- `clwp` (in-cloud liquid water path (g/m2))
- `ciwp = 0.`, # in-cloud ice water path (g/m2)
- `r_liq = 0.`, # Cloud water drop effective radius (microns)
- `r_ice = 0.`, # Cloud ice particle effective size (microns)
- `ozone_file = 'apeozone_cam3_5_54.nc'` (file with ozone distribution - ignored if `absorber_vmr` is given)

If `absorber_vmr = None` then ozone will be interpolated to the model grid from a climatology file, or set to zero if `ozone_file = None`.

Additional input arguments for SW:

- `albedo = None` (optional, single parameter to set all 4 albedo values)
- `aldif = 0.3`, (near-infrared albedo, diffuse)
- `aldir = 0.3`, (near-infrared albedo, direct)
- `asdif = 0.3`, (shortwave albedo, diffuse)
- `asdir = 0.3`, (shortwave albedo, direct)
- `S0 = const.S0`, (solar constant, W/m2)
- `insolation = const.S0/4.`, (time-mean insolaltion, W/m2)
- `coszen = None`, # cosine of the solar zenith angle
- `eccentricity_factor = 1.`, # instantaneous irradiance = S0 * eccentricity_factor

Additional input arguments for LW:

- `emissivity = 1.`, # surface emissivity

Shortave processes compute these diagnostics (minimum):

- `ASR` (W/m2, net Absorbed Shortwave Radiation at TOA, **positive down**)
- `ASRclr` (clear-sky component)
- `ASRcld` (cloud component, all-sky minus clear-sky)
- `SW_flux_up` (W/m2, defined at pressure level interfaces)
- `SW_flux_down` (W/m2, defined at pressure level interfaces)
- `SW_flux_net` (W/m2 **downward** net flux at pressure level interfaces)
- `SW_flux_up_clr` (clear-sky flux)
- `SW_flux_down_clr` (clear-sky flux)
- `SW_flux_net_clr` (clear-sky flux)
- `TdotSW` (K/day, radiative heating rate)
- `TdotSW_clr` (clear-sky heating rate)

Longwave processes compute these diagnostics (minimum):

- `OLR` (W/m2, net Outgoing Longwave radiation at TOA, **positive up**)
- `OLRclr` (clear-sky component)

- `OLRcld` (cloud component, all-sky minus clear-sky)
- `LW_flux_up` (W/m2, defined at pressure level interfaces)
- `LW_flux_down` (W/m2, defined at pressure level interfaces)
- `LW_flux_net` (W/m2 **upward** net flux at pressure level interfaces)
- `LW_flux_up_clr` (clear-sky flux)
- `LW_flux_down_clr` (clear-sky flux)
- `LW_flux_net_clr` (clear-sky flux)
- `TdotLW` (K/day, radiative heating rate)
- `TdotLW_clr` (clear-sky heating rate)

class climlab.radiation.radiation.**_Radiation**(*specific_humidity=None, absorber_vmr=None, cldfrac=0.0, clwp=0.0, ciwp=0.0, r_liq=0.0, r_ice=0.0, ozone_file='apeozone_cam3_5_54.nc', **kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

Abstact base class for SW and LW radiation processes.

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 41 – continued from previous page

compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.radiation.radiation.**_Radiation_LW**(*emissivity=1.0*, ***kwargs*)

Bases: *climlab.radiation.radiation._Radiation* (page 140)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.

Continued on next page

Table 42 – continued from previous page

add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

_compute_LW_flux_diagnostics()

class climlab.radiation.radiation.**_Radiation_SW**(*albedo=None*, *aldif=0.3*, *aldir=0.3*, *asdif=0.3*, *asdir=0.3*, *S0=1365.2*, *insolation=341.3*, *coszen=None*, *eccentricity_factor=1.0*, ***kwargs*)

Bases: *climlab.radiation.radiation._Radiation* (page 140)

Parent class for SW radiation modules

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

`_compute_SW_flux_diagnostics`()

climlab.radiation.radiation.**default_absorbers**(*Tatm*, *ozone_file='apeozone_cam3_5_54.nc'*, *verbose=True*)

Initialize a dictionary of well-mixed radiatively active gases All values are volumetric mixing ratios.

Ozone is set to a climatology.

All other gases are assumed well-mixed:

- CO2
- CH4
- N2O
- O2
- CFC11
- CFC12
- CFC22
- CCL4

Specific values are based on the AquaPlanet Experiment protocols, except for O2 which is set the realistic value 0.21 (affects the RRTMG scheme).

`climlab.radiation.radiation.`**`default_specific_humidity`**(*Tatm*)
Initialize a specific humidity distribution based on a prescribed relative humidity profile.

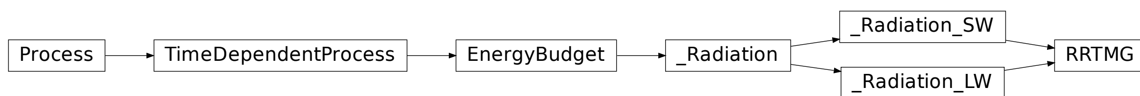
Input is air temperature array

Output is specific humidity on same grid

`climlab.radiation.radiation.`**`init_interface`**(*field*)
Return a Field object defined at the vertical interfaces of the input Field object.

10.6.7 RRTMG

10.6.7.1 RRTMG



class `climlab.radiation.rrtm.rrtmg.`**`RRTMG`**(*icld=1, irng=1, idrv=0, permuteseed_sw=150, permuteseed_lw=300, dyofyr=0, inflgsw=2, inflglw=2, iceflgsw=1, iceflglw=1, liqflgsw=1, liqflglw=1, tauc_sw=0.0, tauc_lw=0.0, ssac_sw=0.0, asmc_sw=0.0, fsfc_sw=0.0, tauaer_sw=0.0, ssaaer_sw=0.0, asmaer_sw=0.0, ecaer_sw=0.0, tauaer_lw=0.0, isolvar=0, indsolvar=array([0., 0.]), bndsolvar=array([0.]), solcycfrac=0.0, **kwargs*)
Bases: *climlab.radiation.radiation._Radiation_SW* (page 142), *climlab.radiation.radiation._Radiation_LW* (page 141)

Container to drive combined LW and SW radiation models.

For some details about inputs and diagnostics, see the *radiation* module.

Attributes

`depth` Depth at grid centers (m)

`depth_bounds` Depth at grid interfaces (m)

`diagnostics` Dictionary access to all diagnostic variables

`input` Dictionary access to all input variables

`lat` Latitude of grid centers (degrees North)

`lat_bounds` Latitude of grid interfaces (degrees North)

`lev` Pressure levels at grid centers (hPa or mb)

`lev_bounds` Pressure levels at grid interfaces (hPa or mb)

`lon` Longitude of grid centers (degrees)

`lon_bounds` Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.6.7.2 RRTMG_LW

class `climlab.radiation.rrtm.rrtmg_lw.`**`RRTMG_LW`**(*icld=1, irng=1, idrv=0, permuteseed=300, inflglw=2, iceflglw=1, liqflglw=1, tauc=0.0, tauaer=0.0, **kwargs*)

Bases: *`climlab.radiation.radiation._Radiation_LW`* (page 141)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

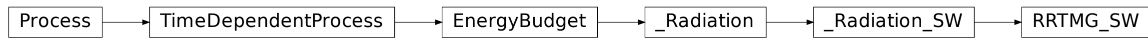
`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

_compute_heating_rates()

Prepare arguments and call the RRTGM_LW driver to calculate radiative fluxes and heating rates

_prepare_lw_arguments()

10.6.7.3 RRTMG_SW



class climlab.radiation.rrtm.rrtmg_sw.**RRTMG_SW**(*icld=1, irng=1, permuteseed=150, dyofyr=0, inflgsw=2, iceflgsw=1, liqflgsw=1, tauc=0.0, ssac=0.0, asmc=0.0, fsfc=0.0, iaer=0, tauaer=0.0, ssaaer=0.0, asmaer=0.0, ecaer=0.0, isolvar=-1, indsolvar=array([1., 1.]), bndsolvar=array([1.]), solcycfrac=1.0, **kwargs*)

Bases: `climlab.radiation.radiation._Radiation_SW` (page 142)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
add_input(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.

Continued on next page

Table 46 – continued from previous page

declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

_compute_heating_rates()

Prepare arguments and call the RRTGM_SW driver to calculate radiative fluxes and heating rates

_prepare_sw_arguments()

climlab wrapper for RRTMG radiation schemes.

This is implemented with classes RRTMG_LW, and RRTMG_SW, as well as a container class RRTMG that has LW and SW radiation as subprocesses.

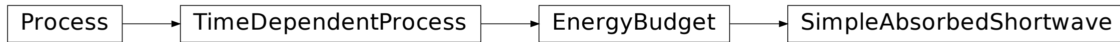
Input arguments and diagnostics follow specifications in _Radiation

See <http://rtweb.aer.com/rrtm_frame.html> for more information about the RRTMG code.

Example Here is a quick example of setting up a single-column Radiative-Convective model with fixed relative humidity:

```
import climlab
alb = 0.25
#  State variables (Air and surface temperature)
state = climlab.column_state(num_lev=30)
#  Fixed relative humidity
h2o = climlab.radiation.ManabeWaterVapor(name='WaterVapor',␣
↪state=state)
#  Couple water vapor to radiation
rad = climlab.radiation.RRTMG(name='Radiation', state=state,␣
↪specific_humidity=h2o.q, albedo=alb)
#  Convective adjustment
conv = climlab.convection.ConvectiveAdjustment(name='Convection',␣
↪state=state, adj_lapse_rate=6.5)
#  Couple everything together
rcm = climlab.couple([rad,h2o,conv], name='Radiative-Convective␣
↪Model')
#  Run the model
rcm.integrate_years(1)
#  Check for energy balance
print(rcm.ASR - rcm.OLR)
```

10.6.8 SimpleAbsorbedShortwave



class climlab.radiation.absorbed_shorwave.**SimpleAbsorbedShortwave**(*insolation=341.3*, *albedo=0.3*, ***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

A class for the shortwave radiation process in a one-layer EBM. The basic assumption is that all the shortwave absorption occurs at the surface.

Computes the diagnostic `ASR` (absorbed shortwave radiation) from the formula `self.ASR = (1-self.albedo) * self.insolation` and applies this as a tendency on the surface temperature `self.Ts`

`albedo` and `insolation` are given as inputs. These should either be scalars or have same dimensions as state variable `Ts`

User can supply constants, or link to diagnostics of specific insolation and albedo processes.

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 47 – continued from previous page

`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.6.9 transmissivity

Transmissivity

class climlab.radiation.transmissivity.**Transmissivity**(*absorptivity*, *reflectivity=None*, *axis=0*)

Bases: `object`

Class for calculating and store transmissivity between levels, and computing radiative fluxes between levels.

Input: numpy array of absorptivities. It is assumed that the last dimension is vertical levels.

Attributes: (all stored as numpy arrays):

- N: number of levels
- absorptivity: level absorptivity (N)
- transmissivity: level transmissivity (N)
- Tup: transmissivity matrix for upwelling beam (N+1, N+1)
- Tdown: transmissivity matrix for downwelling beam (N+1, N+1)

Example for N = 3 atmospheric layers:

tau is a vector of transmissivities

$$\tau = [1, \tau_0, \tau_1, \tau_2]$$

A is a matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \tau_0 & 1 & 1 & 1 \\ \tau_0 & \tau_1 & 1 & 1 \\ \tau_0 & \tau_1 & \tau_2 & 1 \end{bmatrix}$$

We then take the cumulative product along columns, and finally take the lower triangle of the result to get

$$Tup = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \tau_0 & 1 & 0 & 0 \\ \tau_1 \tau_0 & \tau_1 & 1 & 0 \\ \tau_2 \tau_1 \tau_0 0 & \tau_2 \tau_1 & \tau_2 & 1 \end{bmatrix}$$

and Tdown = transpose(Tup)

Construct an emission vector for the downwelling beam:

Edown = [E0, E1, E2, fromspace]

Now we can get the downwelling beam by matrix multiplication:

D = Tdown * Edown

For the upwelling beam, we start by adding the reflected part at the surface to the surface emissions:

Eup = [emit_sfc + albedo_sfc*D[0], E0, E1, E2]

So that the upwelling flux is

U = Tup * Eup

The total flux, positive up is thus

F = U - D

The absorbed radiation at the surface is then -F[0] The absorbed radiation in the atmosphere is the flux convergence:

-diff(F)

Methods

`flux_down` (page 151)(fluxDownTop[, emission])	Compute upwelling radiative flux at interfaces between layers.
`flux_up` (page 152)(fluxUpBottom[, emission])	Compute downwelling radiative flux at interfaces between layers.

flux_reflected_up	

`flux_down` (*fluxDownTop*, *emission=None*)

Compute upwelling radiative flux at interfaces between layers.

Inputs:

- fluxUpBottom: flux up from bottom
- emission: emission from atmospheric levels (N) defaults to zero if not given

Returns:

- vector of upwelling radiative flux between levels (N+1) element N is the flux up to space.

flux_reflected_up (*fluxDown*, *albedo_sfc=0.0*)

flux_up (*fluxUpBottom*, *emission=None*)
Compute downwelling radiative flux at interfaces between layers.

Inputs:

- fluxDownTop: flux down at top
- emission: emission from atmospheric levels (N) defaults to zero if not given

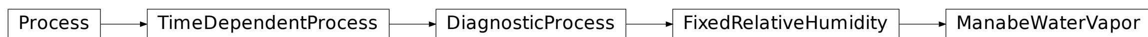
Returns:

- vector of downwelling radiative flux between levels (N+1) element 0 is the flux down to the surface.

climlab.radiation.transmissivity.**compute_T_vectorized** (*transmissivity*)

climlab.radiation.transmissivity.**tril** (*array*, *k=0*)
Lower triangle of an array. Return a copy of an array with elements above the k-th diagonal zeroed. Need a multi-dimensional version here because numpy.tril does not broadcast for numpy verison < 1.9.

10.6.10 water_vapor



class climlab.radiation.water_vapor.**FixedRelativeHumidity** (*relative_humidity=0.77*, *qStrat=5e-06*, ***kwargs*)
Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 85)

Compute water vapor mixing ratio profile Assuming constant relative humidity.

relative_humidity is the specified RH. Same value is applied everywhere. qStrat is the minimum specific humidity, ensuring that there is some water vapor in the stratosphere.

The attribute RH_profile can be modified to set different vertical profiles of relative humidity (see daughter class ManabeWaterVapor()).

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class `climlab.radiation.water_vapor.`**`ManabeWaterVapor`**(***kwargs*)

Bases: *`climlab.radiation.water_vapor.FixedRelativeHumidity`* (page 152)

Compute water vapor mixing ratio profile following Manabe and Wetherald JAS 1967 Fixed surface relative humidity and a specified fractional profile.

relative_humidity is the specified surface RH qStrat is the minimum specific humidity, ensuring that there is some water vapor in the stratosphere.

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.7 climlab.solar

Modules to calculate insolation and orbital variations.

These methods now accept and return `xarray` objects for easier data manipulation and plotting.

10.7.1 insolation

This module contains general-purpose routines for computing daily-average incoming solar radiation at the top of the atmosphere.

Example Compute the timeseries of insolation at 65N at summer solstice over the past 5 Myears:

```
import numpy as np
from climlab.solar.orbital import OrbitalTable
from climlab.solar.insolation import daily_insolation

# array with specified kyears (can be plain numpy or xarray.
↪DataArray)
years = np.linspace(-5000, 0, 5001)

# subset of orbital parameters for specified time
orb = OrbitalTable.interp(kyear=years)

# insolation values for past 5 Myears at 65N at summer solstice␣
↪(day 172)
S65 = daily_insolation(lat=65, day=172, orb=orb)
# returns an xarray.DataArray object with insolation values in W/
↪m2
```

Note: Ported and modified from MATLAB code daily_insolation.m

Original authors:

Ian Eisenman and Peter Huybers, Harvard University, August 2006

Available online at http://eisenman.ucsd.edu/code/daily_insolation.m

---

If using calendar days, solar longitude is found using an approximate solution to the differential equation representing conservation of angular momentum (Kepler's Second Law). Given the orbital parameters and solar longitude, daily average insolation is calculated exactly following [Berger_1978]. Further references: [Berger_1991].

`climlab.solar.insolation.`**`daily_insolation`**(*lat*, *day*, *orb={'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}*, *S0=1365.2*, *day_type=1*)

Compute daily average insolation given latitude, time of year and orbital parameters.

Orbital parameters can be interpolated to any time in the last 5 Myears with `climlab.solar.orbital.OrbitalTable` (see example above).

Longer orbital tables are available with `climlab.solar.orbital.LongOrbitalTable`

Inputs can be scalar, `numpy.ndarray`, or `xarray.DataArray`.

The return value will be `numpy.ndarray` if **all** the inputs are `numpy`. Otherwise `xarray.DataArray`.

Function-call argument

Parameters

- **lat** (*array*) – Latitude in degrees (-90 to 90).
- **day** (*array*) – Indicator of time of year. See argument `day_type` for details about format.

- **orb** (*dict*) – a dictionary with three members (as provided by `climlab.solar.orbital.OrbitalTable`)
  - `'ecc'` - eccentricity
    - unit: dimensionless
    - default value: `0.017236`
  - `'long_peri'` - longitude of perihelion (precession angle)
    - unit: degrees
    - default value: `281.37`
  - `'obliquity'` - obliquity angle
    - unit: degrees
    - default value: `23.446`
- **S0** (*float*) – solar constant
  - unit: W/m^2
  - default value: `1365.2`
- **day_type** (*int*) – Convention for specifying time of year (+/- 1,2) [optional].

  ***day_type=1* (default):** day input is calendar day (1-365.24), where day 1 is January first. The calendar is referenced to the vernal equinox which always occurs at day 80.

  day_type=2: day input is solar longitude (0-360 degrees). Solar longitude is the angle of the Earth's orbit measured from spring equinox (21 March). Note that calendar days and solar longitude are not linearly related because, by Kepler's Second Law, Earth's angular velocity varies according to its distance from the sun.

Raises `ValueError` if day_type is neither 1 nor 2

Returns

Daily average solar radiation in unit W/m^2.

Dimensions of output are `(lat.size, day.size, ecc.size)`

Return type array

Code is fully vectorized to handle array input for all arguments.

Orbital arguments should all have the same sizes. This is automatic if computed from `lookup_parameters()`

For more information about computation of solar insolation see the *Tutorials* (page 25) chapter.

`climlab.solar.insolation.`**`solar_longitude`**(*day*, *orb={'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}*, *days_per_year=None*)

Estimates solar longitude from calendar day.

Method is using an approximation from [Berger_1978] section 3 (lambda = 0 at spring equinox).

Function-call arguments

Parameters

- **day** (*array*) – Indicator of time of year.
- **orb** (*dict*) – a dictionary with three members (as provided by `OrbitalTable`)
  - `'ecc'` - eccentricity

    * unit: dimensionless
    * default value: `0.017236`
  - `'long_peri'` - longitude of perihelion (precession angle)
    * unit: degrees
    * default value: `281.37`
  - `'obliquity'` - obliquity angle
    * unit: degrees
    * default value: `23.446`
- **days_per_year** (*float*) – number of days in a year (optional) (default: 365.2422) Reads the length of the year from *constants* (page 172) if available.

Returns solar longitude `lambda_long` in dimension``( day.size, ecc.size )``

Return type array

Works for both scalar and vector orbital parameters.

10.7.2 orbital

The object `climlab.solar.orbital.OrbitalTable` is an `xarray.Dataset` holding orbital data (**eccentricity**, **obliquity**, and **longitude of perihelion**) for the past 5 Myears. The data are from [Berger_1991].

Data are read from the file `orbit91`, which was originally obtained from <https://www1.ncdc.noaa.gov/pub/data/paleo/climate_forcing/orbital_variations/insolation/> If the file isn't found locally, the module will attempt to read it remotely from the above URL.

A subclass `climlab.solar.orbital.long.OrbitalTable` works with La2004 orbital data for -51 to +21 Myears as calculated by [Laskar_2004]. See <http://vo.imcce.fr/insola/earth/online/earth/La2004/README.TXT>

(Breaking change from climlab 0.7.0 and previous)

Example Load orbital data from the past 5 Myears:

```
# Load the data
from climlab.solar.orbital import OrbitalTable

# Examine the xarray object
print(OrbitalTable)

# Get a timeseries of obliquity
print(OrbitalTable.obliquity)

# Get the orbital data for a specific year, 10 kyear before␣
↪present:
print(OrbitalTable.interp(kyear=-10))

# Get the long orbital table data
from climlab.solar.orbital.long import OrbitalTable as LongTable
print(LongTable)
```

10.7.3 orbital_cycles

OrbitalCycles

class climlab.solar.orbital_cycles.**OrbitalCycles**(*model*, *kyear_start=-20.0*, *kyear_stop=0.0*, *segment_length_years=100.0*, *orbital_year_factor=1.0*, *verbose=True*)

Bases: object

Automatically integrates a process through changes in orbital parameters.

OrbitalCycles is a module for setting up long integrations of climlab processes over orbital cycles.

The duration between integration start and end time is partitioned in time segments over which the orbital parameters are held constant. The process is integrated over every time segment and the process state `Ts` is stored for each segment.

The storage arrays are saving:

- **current model state** at end of each segment
- **model state averaged** over last integrated year of each segment
- **global mean** of averaged model state over last integrated year of each segment

Note: Input `kyear` is thousands of years after present. For years before present, use `kyear < 0`.

Initialization parameters

Parameters

- **model** (*TimeDependentProcess* (page 103)) – a time dependent process
- **kyear_start** (*float*) – integration start time.

  As time reference is present, argument should be < 0 for time before present.

  – *unit:* kiloyears

  – *default value:* `-20.`
- **kyear_stop** (*float*) – integration stop time.

  As time reference is present, argument should be ≤ 0 for time before present.

  – *unit:* kiloyears

  – *default value:* `0.`
- **segment_length_years** (*float*) – is the length of each integration with fixed orbital parameters. [default: 100.]

- **orbital_year_factor** (*float*) – is an optional speed-up to the orbital cycles. [default: 1.]
- **verbose** (*bool*) – prints product of calculation and information about computation progress [default: True]

Object attributes

Following object attributes are generated during initialization:

Variables

- ***model*** (page 70) (*TimeDependentProcess* (page 103)) – timedependent process to be integrated
- **kyear_start** (*float*) – integration start time
- **kyear_stop** (*float*) – integration stop time
- **segment_length_years** (*float*) – length of each integration with fixed orbital parameters
- **orbital_year_factor** (*float*) – speed-up factor to the orbital cycles
- **verbose** (*bool*) – print flag
- **num_segments** (*int*) – number of segments with fixed oribtal parameters, calculated through:

$$num_{seg} = \frac{-(kyear_{start} - kyear_{stop}) * 1000}{seg_{length} * orb_{factor}}$$

- **T_segments_global** (*array*) – storage for global mean temperature for final year of each segment
- **T_segments** (*array*) – storage for actual temperature at end of each segment
- **T_segments_annual** (*array*) – storage for timeaveraged temperature over last year of segment

  dimension: (size(Ts), num_segments)
- **orb_kyear** (*array*) – integration start time of all segments
- ***orb*** (page 125) (*dict*) – orbital parameters for last integrated segment

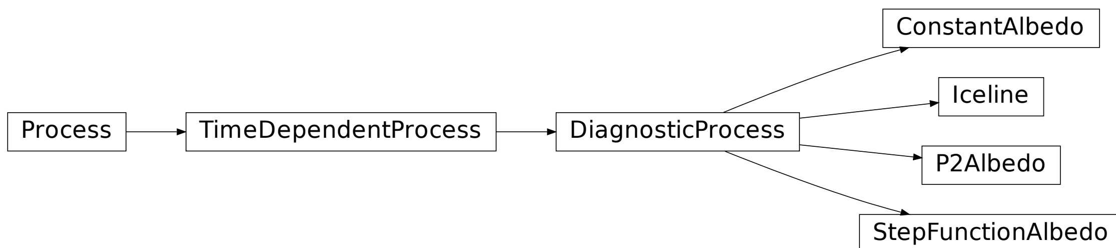
Example Integration of an energy balance model for 10,000 years with corresponding orbital parameters:

```
from climlab.model.ebm import EBM_seasonal
from climlab.solar.orbital_cycles import OrbitalCycles
from climlab.surface.albedo import StepFunctionAlbedo
ebm = EBM_seasonal()
print ebm

#  add an albedo feedback
albedo = StepFunctionAlbedo(state=ebm.state, **ebm.param)
ebm.add_subprocess('albedo', albedo)

#  start the integration
#  run for 10,000 orbital years, but only 1,000 model years
experiment = OrbitalCycles(ebm, kyear_start=-20, kyear_stop=-10,
                                        orbital_year_factor=10.)
```

10.8 climlab.surface

Modules for surface processes in climlab.

10.8.1 albedo



class climlab.surface.albedo.**ConstantAlbedo**(*albedo=0.33*, ***kwargs*)
Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 85)

A class for constant albedo values at all spatial points of the domain.

Initialization parameters

Parameters albedo (*float*) – albedo values [default: 0.33]

Object attributes

Additional to the parent class *DiagnosticProcess* (page 85) following object attributes are generated and updated during initialization:

Variables ***albedo*** (page 160) (Field (page 47)) – attribute to store the albedo value. During initialization the albedo() setter is called.

Example Creation of a constant albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import ConstantAlbedo

>>> # model creation
>>> model = climlab.EBM()

>>> sfc = model.domains['Ts']

>>> # subprocess creation
>>> const_alb = ConstantAlbedo(albedo=0.3, domains=sfc, **model.param)
```

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

Uniform prescribed albedo.

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

class `climlab.surface.albedo.`**`Iceline`**(*Tf=-10.0*, ***kwargs*)

Bases: `climlab.process.diagnostic.DiagnosticProcess` (page 85)

A class for an Iceline subprocess.

Depending on a freezing temperature it calculates where on the domain the surface is covered with ice, where there is no ice and on which latitude the ice-edge is placed.

Initialization parameters

Parameters **Tf** (*float*) – freezing temperature where sea water freezes and surface is covered

with ice

- unit: °C
- default value: -10

Object attributes

Additional to the parent class `DiagnosticProcess` (page 85) following object attributes are generated and updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with the input argument `'Tf'`.
- ***diagnostics*** (page 97) (*dict*) – keys `'icelat'` and `'ice_area'` initialized
- **icelat** (*array*) – the subprocess attribute `self.icelat` is created
- **ice_area** (*float*) – the subprocess attribute `self.ice_area` is created

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`find_icelines` (page 164)()	Finds iceline according to the surface temperature.

Continued on next page

Table 53 – continued from previous page

`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

find_icelines()

Finds iceline according to the surface temperature.

This method is called by the private function `_compute()` and updates following attributes according to the freezing temperature `self.param['Tf']` and the surface temperature `self.param['Ts']`:

Object attributes

Variables

- **noice** (`Field` (page 47)) – a Field of booleans which are `True` where $T_s \ge T_f$
- **ice** (`Field` (page 47)) – a Field of booleans which are `True` where $T_s < T_f$
- **icelat** (*array*) – an array with two elements indicating the ice-edge latitudes
- **ice_area** (*float*) – fractional area covered by ice (0 - 1)
- ***diagnostics*** (page 97) (*dict*) – keys `'icelat'` and `'ice_area'` are updated

class climlab.surface.albedo.**P2Albedo**(*a0=0.33*, *a2=0.25*, ***kwargs*)

Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 85)

A class for parabolic distributed albedo values across the domain on basis of the second order Legendre Polynomial.

Calculates the latitude dependent albedo values as

$$\alpha(\varphi) = a_0 + a_2 P_2(x)$$

where $P_2(x) = \frac{1}{2}(3x^2 - 1)$ is the second order Legendre Polynomial and $x = sin(\varphi)$.

Initialization parameters

Parameters

- **a0** (*float*) – basic parameter for albedo function [default: 0.33]
- **a2** (*float*) – factor for second legendre polynominal term in albedo function [default: 0.25]

Object attributes

Additional to the parent class *DiagnosticProcess* (page 85) following object attributes are generated and updated during initialization:

Variables

- ***a0*** (page 166) (*float*) – attribute to store the albedo parameter a0. During initialization the *a0()* (page 166) setter is called.
- ***a2*** (page 166) (*float*) – attribute to store the albedo parameter a2. During initialization the *a2()* (page 166) setter is called.
- ***diagnostics*** (page 97) (*dict*) – key `'albedo'` initialized
- ***albedo*** (page 160) (`Field` (page 47)) – the subprocess attribute `self.albedo` is created with correct dimensions (according to `self.lat`)

Example Creation of a parabolic albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import P2Albedo

>>> # model creation
>>> model = climlab.EBM()

>>> # modify a0 and a2 values in model parameter dictionary
>>> model.param['a0']=0.35
>>> model.param['a2']= 0.10

>>> # subprocess creation
>>> p2_alb = P2Albedo(domains=model.domains['Ts'], **model.param)

>>> p2_alb.a0
0.33
>>> p2_alb.a2
0.1
```

Attributes

a0 **(page 166)** Property of albedo parameter a0.

a2 **(page 166)** Property of albedo parameter a2.

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

a0

Property of albedo parameter a0.

Getter Returns the albedo parameter value which is stored in attribute `self._a0`

Setter

- sets albedo parameter which is addressed as `self._a0` to the new value
- updates the parameter dictionary `self.param['a0']`
- calls method `_compute_fixed()`

Type float

a2

Property of albedo parameter a2.

Getter Returns the albedo parameter value which is stored in attribute `self._a2`

Setter

- sets albedo parameter which is addressed as `self._a2` to the new value
- updates the parameter dictionary `self.param['a2']`
- calls method `_compute_fixed()`

Type float

class climlab.surface.albedo.**StepFunctionAlbedo**(*Tf=-10.0*, *a0=0.3*, *a2=0.078*, *ai=0.62*, ***kwargs*)

Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 85)

A step function albedo suprocess.

This class itself defines three subprocesses that are created during initialization:

- 'iceline' - *Iceline* (page 162)
- 'warm_albedo' - *P2Albedo* (page 164)
- 'cold_albedo' - *ConstantAlbedo* (page 160)

Initialization parameters

Parameters

- **Tf** (*float*) – freezing temperature for Iceline subprocess
  - unit: °C
  - default value: -10
- **a0** (*float*) – basic parameter for P2Albedo subprocess [default: 0.3]
- **a2** (*float*) – factor for second legendre polynominal term in P2Albedo subprocess [default: 0.078]
- **ai** (*float*) – ice albedo value for ConstantAlbedo subprocess [default: 0.62]

Additional to the parent class *DiagnosticProcess* (page 85) following object attributes are generated/updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with a couple of the initatilzation input arguments, namely 'Tf', 'a0', 'a2' and 'ai'.
- **topdown** (*bool*) – is set to False to call subprocess compute method first
- ***diagnostics*** (page 97) (*dict*) – key 'albedo' initialized
- ***albedo*** (page 160) (Field (page 47)) – the subprocess attribute self.albedo is created

Example Creation of a step albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import StepFunctionAlbedo
>>>
>>> model = climlab.EBM(a0=0.29, a2=0.1, ai=0.65, Tf=-2)
>>>
>>> step_alb = StepFunctionAlbedo(state= model.state, **model.param)
>>>
>>> print step_alb
climlab Process of type <class 'climlab.surface.albedo.
↪StepFunctionAlbedo'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
   iceline: <class 'climlab.surface.albedo.Iceline'>
   cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
   warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
```

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

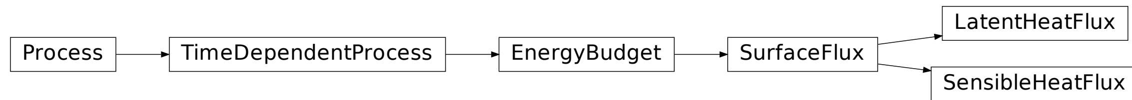
lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.
`integrate_days`([days, verbose])	Integrates the model forward for a specified number of days.
`integrate_years`([years, verbose])	Integrates the model by a given number of years.
`remove_diagnostic`(name)	Removes a diagnostic from the `process.diagnostic` dictionary and also delete the associated process attribute.
`remove_subprocess`(name[, verbose])	Removes a single subprocess from this process.
`set_state`(name, value)	Sets the variable `name` to a new state `value`.
`set_timestep`([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of `timestep()`
`step_forward`()	Updates state variables with computed tendencies.
`to_xarray`([diagnostics])	Convert process variables to `xarray.Dataset` format.

10.8.2 climlab.surface.turbulent



class climlab.surface.turbulent.**LatentHeatFlux**(*Cd=0.003*, ***kwargs*)

Bases: *climlab.surface.turbulent.SurfaceFlux* (page 171)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which `step_forward()` is integrating in unit seconds.

Methods

`add_diagnostic`(name[, value])	Create a new diagnostic variable called `name` for this process and initialize it with the given `value`.
`add_input`(name[, value])	Create a new input variable called `name` for this process and initialize it with the given `value`.
`add_subprocess`(name, proc)	Adds a single subprocess to this process.
`add_subprocesses`(procdict)	Adds a dictionary of subproceses to this process.
`compute`()	Computes the tendencies for all state variables given current state and specified input.
`compute_diagnostics`([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
`declare_diagnostics`(diaglist)	Add the variable names in `inputlist` to the list of diagnostics.
`declare_input`(inputlist)	Add the variable names in `inputlist` to the list of necessary inputs.
`integrate_converge`([crit, verbose])	Integrates the model until model states are converging.

Continued on next page

Table 56 – continued from previous page

integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.surface.turbulent.**SensibleHeatFlux**(*Cd=0.003*, ***kwargs*)

Bases: *climlab.surface.turbulent.SurfaceFlux* (page 171)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.
compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.

Continued on next page

Table 57 – continued from previous page

declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

class climlab.surface.turbulent.**SurfaceFlux**(*Cd=0.003*, ***kwargs*)

Bases: *climlab.process.energy_budget.EnergyBudget* (page 86)

Attributes

depth Depth at grid centers (m)

depth_bounds Depth at grid interfaces (m)

diagnostics Dictionary access to all diagnostic variables

input Dictionary access to all input variables

lat Latitude of grid centers (degrees North)

lat_bounds Latitude of grid interfaces (degrees North)

lev Pressure levels at grid centers (hPa or mb)

lev_bounds Pressure levels at grid interfaces (hPa or mb)

lon Longitude of grid centers (degrees)

lon_bounds Longitude of grid interfaces (degrees)

timestep The amount of time over which step_forward() is integrating in unit seconds.

Methods

add_diagnostic(name[, value])	Create a new diagnostic variable called name for this process and initialize it with the given value.
add_input(name[, value])	Create a new input variable called name for this process and initialize it with the given value.
add_subprocess(name, proc)	Adds a single subprocess to this process.
add_subprocesses(procdict)	Adds a dictionary of subproceses to this process.
compute()	Computes the tendencies for all state variables given current state and specified input.

Continued on next page

Table 58 – continued from previous page

compute_diagnostics([num_iter])	Compute all tendencies and diagnostics, but don't update model state.
declare_diagnostics(diaglist)	Add the variable names in inputlist to the list of diagnostics.
declare_input(inputlist)	Add the variable names in inputlist to the list of necessary inputs.
integrate_converge([crit, verbose])	Integrates the model until model states are converging.
integrate_days([days, verbose])	Integrates the model forward for a specified number of days.
integrate_years([years, verbose])	Integrates the model by a given number of years.
remove_diagnostic(name)	Removes a diagnostic from the process.diagnostic dictionary and also delete the associated process attribute.
remove_subprocess(name[, verbose])	Removes a single subprocess from this process.
set_state(name, value)	Sets the variable name to a new state value.
set_timestep([timestep, num_steps_per_year])	Calculates the timestep in unit seconds and calls the setter function of timestep()
step_forward()	Updates state variables with computed tendencies.
to_xarray([diagnostics])	Convert process variables to xarray.Dataset format.

10.9 climlab.utils

10.9.1 constants

Contains a collection of physical constants for the atmosphere and ocean.

```
import numpy as np

a = 6.373E6      # Radius of Earth (m)
Lhvap = 2.5E6    # Latent heat of vaporization (J / kg)
Lhsub = 2.834E6   # Latent heat of sublimation (J / kg)
Lhfus = Lhsub - Lhvap  # Latent heat of fusion (J / kg)
cp = 1004.     # specific heat at constant pressure for dry air (J / kg / K)
Rd = 287.         # gas constant for dry air (J / kg / K)
kappa = Rd / cp
Rv = 461.5       # gas constant for water vapor (J / kg / K)
cpv = 1875.   # specific heat at constant pressure for water vapor (J / kg / K)
Omega = 2 * np.math.pi / 24. / 3600.  # Earth's rotation rate, (s**(-1))
g = 9.8          # gravitational acceleration (m / s**2)
kBoltzmann = 1.3806488E-23  # the Boltzmann constant (J / K)
c_light = 2.99792458E8   # speed of light (m/s)
hPlanck = 6.62606957E-34  # Planck's constant (J s)
# sigma = 5.67E-8  # Stefan-Boltzmann constant (W / m**2 / K**4)
#  sigma derived from fundamental constants
sigma = (2*np.pi**5 * kBoltzmann**4) / (15 * c_light**2 * hPlanck**3)

S0 = 1365.2       # solar constant (W / m**2)
# value is consistent with Trenberth and Fasullo, Surveys of Geophysics 2012

ps = 1000.       # approximate surface pressure (mb or hPa)
```

(continues on next page)

(continued from previous page)

```
rho_w = 1000.     # density of water (kg / m**3)
cw = 4181.3       # specific heat of liquid water (J / kg / K)

tempCtoK = 273.15   # 0degC in Kelvin
tempKtoC = -tempCtoK  # 0 K in degC
mb_to_Pa = 100.  # conversion factor from mb to Pa

#  Some useful time conversion factors
seconds_per_minute = 60.
minutes_per_hour = 60.
hours_per_day = 24.

# the length of the "tropical year" -- time between vernal equinoxes
# This value is consistent with Berger (1978)
# "Long-Term Variations of Daily Insolation and Quaternary Climatic Changes"
days_per_year = 365.2422
seconds_per_hour = minutes_per_hour * seconds_per_minute
minutes_per_day = hours_per_day * minutes_per_hour
seconds_per_day = hours_per_day * seconds_per_hour
seconds_per_year = seconds_per_day * days_per_year
minutes_per_year = seconds_per_year / seconds_per_minute
hours_per_year = seconds_per_year / seconds_per_hour
#  average lenghts of months based on dividing the year into 12 equal parts
months_per_year = 12.
seconds_per_month = seconds_per_year / months_per_year
minutes_per_month = minutes_per_year / months_per_year
hours_per_month = hours_per_year / months_per_year
days_per_month = days_per_year / months_per_year

area_earth = 4 * np.math.pi * a**2

# present-day orbital parameters, in the same format generated by orbital.py
orb_present = {'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}
```

10.9.2 heat_capacity

Routines for calculating heat capacities for grid boxes.

`climlab.utils.heat_capacity.`**`atmosphere`**`(`*dp*`)`

Returns heat capacity of a unit area of atmosphere, in units J /m**2 / K.

$$C_a = \frac{c_p \cdot dp \cdot f_{\textrm{mb-to-Pa}}}{g}$$

where

variable	value	unit	description
C_a	*output*	$\textrm{J}/\textrm{m}^2/\textrm{K}$	heat capacity for atmospheric cell
c_p	1004.	$\textrm{J}/\textrm{kg}/\textrm{K}$	specific heat at constant pressure for dry air
dp	*input*	mb	pressure for atmospheric cell
$f_{\textrm{mb-to-Pa}}$	100	Pa/mb	conversion factor from mb to Pa
g	9.8	$\textrm{m}/\textrm{s}^2$	gravitational acceleration

Function-call argument

Parameters **dp** (*array*) – pressure intervals (*unit:* mb)

Returns the heat capacity for atmosphere cells correspoding to pressure input (*unit:* J /m**2 / K)

Return type array

Example Calculate atmospheric heat capacity for pressure intervals of 1, 10, 100 mb:

```
>>> from climlab.utils import heat_capacity

>>> pressure_interval = array([1,10,100]) # in mb
>>> heat_capacity.atmosphere(pressure_interval) # in J /m**2 / K
array([   10244.89795918,   102448.97959184,  1024489.79591837])
```

climlab.utils.heat_capacity.**ocean**(*dz*)

Returns heat capacity of a unit area of water, in units J /m**2 / K.

$$C_o = \rho_w \cdot c_w \cdot dz$$

where

variable	value	unit	description
C_o	*output*	$\mathrm{J/m^2/K}$	heat capacity for oceanic cell
c_w	4181.3	$\mathrm{J/kg/K}$	specific heat of liquid water
dz	*input*	m	water depth of oceanic cell
ρ_w	1000.	$\mathrm{kg/m^3}$	density of water

Function-call argument

Parameters **dz** (*array*) – water depth of ocean cells (*unit:* m)

Returns the heat capacity for ocean cells correspoding to depth input (*unit:* J /m**2 / K)

Return type array

Example Calculate atmospheric heat capacity for pressure intervals of 1, 10, 100 m:

```
>>> from climlab.utils import heat_capacity

>>> pressure_interval = array([1,10,100]) # in m
>>> heat_capacity.ocean(pressure_interval) # in J /m**2 / K
array([  4.18130000e+06,   4.18130000e+07,   4.18130000e+08])
```

climlab.utils.heat_capacity.**slab_ocean**(*water_depth*)

Returns heat capacity of a unit area slab of water, in units of J / m**2 / K.

Takes input argument `water_depth` and calls *ocean()* (page 174)

Function-call argument

Parameters **float** – water depth of slab ocean (*unit:* m)

Returns the heat capacity for slab ocean cell (*unit:* J / m**2 / K)

Return type float

10.9.3 legendre

Can calculate the first several Legendre polynomials, along with (some of) their first derivatives.

`climlab.utils.legendre.P0(x)`

$$P_0(x) = 1$$

`climlab.utils.legendre.P1(x)`

$$P_1(x) = 1$$

`climlab.utils.legendre.P2(x)`
The second Legendre polynomial.

$$P_2(x) = \frac{1}{2}(3x^2 - 1)$$

`climlab.utils.legendre.Pn(x)`
Calculate Legendre polyomials P0 to P28 and returns them in a dictionary `Pn`.

Parameters **x** (*float*) – argument to calculate Legendre polynomials

Return Pn dictionary which contains order of Legendre polynomials (from 0 to 28) as keys and the corresponding evaluation of Legendre polynomials as values.

Return type dict

`climlab.utils.legendre.Pnprime(x)`
Calculates first derivatives of Legendre polynomials and returns them in a dictionary `Pnprime`.

Parameters **x** (*float*) – argument to calculate first derivate of Legendre polynomials

Return Pn dictionary which contains order of Legendre polynomials (from 0 to 4 and even numbers until 14) as keys and the corresponding evaluation of first derivative of Legendre polynomials as values.

Return type dict

10.9.4 thermo

A collection of function definitions to handle common thermodynamic calculations for the atmosphere.

`climlab.utils.thermo.EIS(T0, T700)`
Convenience method, identical to thermo.estimated_inversion_strength(T0,T700)

`climlab.utils.thermo.Planck_frequency(nu, T)`
The Planck function B(nu,T): the flux density for blackbody radiation in frequency space nu is frequency in 1/s T is temperature in Kelvin

Formula (3.1) from Raymond Pierrehumbert, "Principles of Planetary Climate"

`climlab.utils.thermo.Planck_wavelength(l, T)`
The Planck function (flux density for blackbody radiation) in wavelength space l is wavelength in meters T is temperature in Kelvin

Formula (3.3) from Raymond Pierrehumbert, "Principles of Planetary Climate"

`climlab.utils.thermo.Planck_wavenumber(n, T)`
The Planck function (flux density for blackbody radition) in wavenumber space n is wavenumber in 1/cm T is temperature in Kelvin

Formula from Raymond Pierrehumbert, "Principles of Planetary Climate", page 140.

`climlab.utils.thermo.`**`T`**(*theta*, *p*)
Convenience method, identical to thermo.temperature_from_potential(theta,p).

`climlab.utils.thermo.`**`blackbody_emission`**(*T*)
Blackbody radiation following the Stefan-Boltzmann law.

`climlab.utils.thermo.`**`clausius_clapeyron`**(*T*)
Compute saturation vapor pressure as function of temperature T.

Input: T is temperature in Kelvin Output: saturation vapor pressure in mb or hPa

Formula from Rogers and Yau "A Short Course in Cloud Physics" (Pergammon Press), p. 16 claimed to be accurate to within 0.1% between -30degC and 35 degC Based on the paper by Bolton (1980, Monthly Weather Review).

`climlab.utils.thermo.`**`estimated_inversion_strength`**(*T0*, *T700*)
Compute the Estimated Inversion Strength or EIS, following Wood and Bretherton (2006, J. Climate)

Inputs: T0 is surface temp in Kelvin T700 is air temperature at 700 hPa in Kelvin

Output: EIS in Kelvin

EIS is a normalized measure of lower tropospheric stability acccounting for temperature-dependence of the moist adiabat.

`climlab.utils.thermo.`**`lifting_condensation_level`**(*T*, *RH*)
Compute the Lifiting Condensation Level (LCL) for a given temperature and relative humidity

Inputs: T is temperature in Kelvin RH is relative humidity (dimensionless)

Output: LCL in meters

This is height (relative to parcel height) at which the parcel would become saturated during adiabatic ascent.

Based on approximate formula from Bolton (1980 MWR) as given by Romps (2017 JAS)

For an exact formula see Romps (2017 JAS), doi:10.1175/JAS-D-17-0102.1

`climlab.utils.thermo.`**`mixing_ratio_from_vapor_pressure`**(*p*, *e*)
Water vapor mixing ratio p is air pressure e is vapor pressure p and e must be in same units (e.g. hPa)

`climlab.utils.thermo.`**`mmr_to_vmr`**(*mmr*, *gas*)
Convert mass mixing ratio to volume mixing ratio for named gas. (molecular weights are specific in climlab.utils.constants.py)

`climlab.utils.thermo.`**`potential_temperature`**(*T*, *p*)
Compute potential temperature for an air parcel.

Input: T is temperature in Kelvin p is pressure in mb or hPa

Output: potential temperature in Kelvin.

`climlab.utils.thermo.`**`pseudoadiabat`**(*T*, *p*)
Compute the local slope of the pseudoadiabat at given temperature and pressure

Inputs: p is pressure in hPa or mb T is local temperature in Kelvin

Output: dT/dp, the rate of temperature change for pseudoadiabatic ascent in units of K / hPa

The pseudoadiabat describes changes in temperature and pressure for an air parcel at saturation assuming instantaneous rain-out of the super-saturated water

Formula consistent with eq. (2.33) from Raymond Pierrehumbert, "Principles of Planetary Climate" which nominally accounts for non-dilute effects, but computes the derivative dT/dpa, where pa is the partial pressure of the non-condensible gas.

Integrating the result dT/dp treating p as total pressure effectively makes the dilute assumption.

`climlab.utils.thermo.`**`qsat`**(*T*, *p*)
Compute saturation specific humidity as function of temperature and pressure.

Input: T is temperature in Kelvin p is pressure in hPa or mb

Output: saturation specific humidity (dimensionless).

`climlab.utils.thermo.`**`rho_moist`**(*T*, *p*, *q*)
Density of moist air. T is air temperature (K) p is air pressure (hPa) q is specific humidity (hPa)

returns density in kg/m3

`climlab.utils.thermo.`**`temperature_from_potential`**(*theta*, *p*)
Convert potential temperature to in-situ temperature.

Input: theta is potential temperature in Kelvin p is pressure in mb or hPa

Output: absolute temperature in Kelvin.

`climlab.utils.thermo.`**`theta`**(*T*, *p*)
Convenience method, identical to thermo.potential_temperature(T,p).

`climlab.utils.thermo.`**`vapor_pressure_from_specific_humidity`**(*p*, *q*)
Vapor pressure (same units as input p) p is total air pressure q is specific humidity (dimensionless) – mass of vapor per unit mass moist air

`climlab.utils.thermo.`**`virtual_temperature_from_mixing_ratio`**(*T*, *w*)
Virtual temperature Tv T is air temperature (K) w is water vapor mixing ratio (dimensionless)

`climlab.utils.thermo.`**`vmr_to_mmr`**(*vmr*, *gas*)
Convert volume mixing ratio to mass mixing ratio for named gas. (molecular weights are specific in climlab.utils.constants.py)

10.9.5 walk

`climlab.utils.walk.`**`process_tree`**(*top*, *name='top'*)
Creates a string representation of the process tree for process top.

This method uses the `walk_processes()` (page 178) method to create the process tree.

Parameters

- **top** (`Process` (page 93)) – top process for which process tree string should be created
- **name** (`str`) – name of top process

Returns string representation of the process tree

Return type str

Example

```
>>> import climlab
>>> from climlab.utils import walk

>>> model = climlab.EBM()
>>> proc_tree_str = walk.process_tree(model, name='model')

>>> print proc_tree_str
model: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
```

(continues on next page)

(continued from previous page)

```
LW: <class 'climlab.radiation.AplusBT.AplusBT'>
albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
   iceline: <class 'climlab.surface.albedo.Iceline'>
   cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
   warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

climlab.utils.walk.**walk_processes**(*top*, *topname='top'*, *topdown=True*, *ignoreFlag=False*)

Generator for recursive tree of climlab processes

Starts walking from climlab process `top` and generates a complete list of all processes and sub-processes that are managed from `top` process. `level` indicades the rank of specific process in the process hierarchy:

Note:

- **level 0: `top` process**
  - **level 1: sub-processes of `top` process**
    - level 2: sub-sub-processes of `top` process (=subprocesses of level 1 processes)

The method is based on os.walk().

Parameters

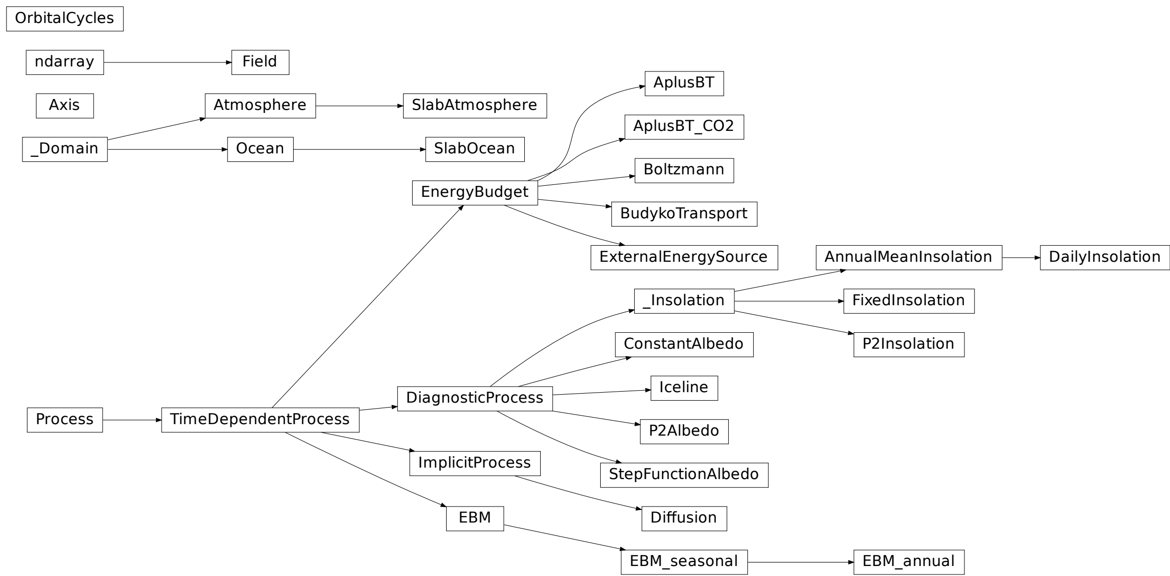
- **top** (*Process* (page 93)) – top process from where walking should start
- **topname** (*str*) – name of top process [default: 'top']
- **topdown** (*bool*) – whether geneterate *process_types* in regular or in reverse order [default: True]
- **ignoreFlag** (*bool*) – whether `topdown` flag should be ignored or not [default: False]

Returns name (str), proc (process), level (int)

Example

```
>>> import climlab
>>> from climlab.utils import walk

>>> model = climlab.EBM()

>>> for name, proc, top_proc in walk.walk_processes(model):
...     print name
...
top
diffusion
LW
iceline
cold_albedo
warm_albedo
albedo
insolation
```

10.10 Inheritance Diagram



CHAPTER 11

Contributing to CLIMLAB

This is an open project, and contributions of all kinds are welcome.

Here are some guidelines for how to get involved.

11.1 Usage in publications, teaching, etc.

If you use CLIMLAB in any way for published research or theses, online teaching materials, or anything else, we would appreciate hearing about it. Our goal is to maintain a list of links and references to use cases. This is essential information for our funders (NSF) but will also be a great resource for new users looking to find out more about what you can do with CLIMLAB.

The best way to report usage is through this open issue on the CLIMLAB github page. If you're not a github users you can report usage directly to Brian Rose (see Contact page).

For publications, please cite the CLIMLAB description paper in JOSS. The full citation is:

> Rose, (2018). CLIMLAB: a Python toolkit for interactive, process-oriented climate modeling. Journal of Open Source Software, 3(24), 659, https://doi.org/10.21105/joss.00659

11.2 Reporting bugs, issues, new feature requests, and documentation problems

These can all be raised as new issues at <https://github.com/brian-rose/climlab/issues>

If you are reporting a bug, try to include:

- Your operating system name and version
- The Python and CLIMLAB versions you are using
- Minimal code to reproduce the bug

If you aren't sure about any of this, please just post your issue anyway and we will assist.

Feel free to point out any inaccuracies or omissions in the documentation here as well.

11.3 Seeking help and support

Although CLIMLAB is offered to the community "as-is", we are very interested in helping people actually use it for scientific purposes.

Have a question about how to do something with CLIMLAB? First, make sure you've perused the documentation and the issue tracker at <https://github.com/brian-rose/climlab/issues>. Also look through published examples including the repository of lecture notes at <https://github.com/brian-rose/ClimateModeling_courseware>.

Then, feel free to ask questions by opening a new issue at <https://github.com/brian-rose/climlab/issues>. This requires a free github account but is the best way to engage the community for answers. We will do our best to respond. If the functionality you're looking for doesn't yet exist, we'll probably encourage you to get involved in developing the next big feature.

11.4 Contributing bug fixes and new features

We are thrilled to have any and all help. You may want to browse through <https://github.com/brian-rose/climlab/issues> to see if there is any low-hanging fruit already identified.

Contributions will happen through Pull Requests on github. You will need a free github account. Here's how to get started:

1. Follow these instructions to fork the main CLIMAB repo at <https://github.com/brian-rose/climlab>, clone it on your local machine, and keep your local master branch synced with the main repo.
2. Don't make any commits on your local master branch. Instead open a feature branch for every new development task:

```
git checkout -b cool_new_feature
```

(choose a more descriptive name for your new feature).

3. Work on your new feature, using `git add` to add your changes.
4. Build and Test your modified code! See *Building and Testing CLIMLAB* (page 183) below for instructions. Make sure to add new tests for your cool new feature.
5. When your feature is complete and tested, commit your changes:

```
git commit -m 'I made some cool new changes'
```

and push your branch to github:

```
git push origin cool_new_feature
```

6. At this point, you go find your fork on github and create a pull request. Clearly describe what you have done in the comments. We will gladly merge any pull requests that fix outstanding issues with the code or documentation. If you are adding a new feature, it is important to also add appropriate tests of the new feature to the automated test suite. If you don't know how to do this, submit your pull request anyway and we will assist.
7. After your pull request is merged, you can switch back to the master branch, rebase, and delete your feature branch. You will find your improvements are incorporated into CLIMLAB:

```
git checkout master
git fetch upstream
git rebase upstream/master
git branch -d cool_new_feature
```

11.5 Building and Testing CLIMLAB

CLIMLAB has an extensive set of tests designed to work with pytest. The test code is found in the `climlab/tests` directory inside the source repo.

To run the full set of tests on the currently installed version of CLIMLAB, you can always do this:

```
pytest -v --pyargs climlab
```

All tests should report `PASSED`.

CLIMLAB is a mix of pure Python and compiled Fortran. If you are developing new code that does not rely on the compiled components, it is useful to run tests directly from the source code directory. From the `climlab` root directory, do the following:

```
pytest -v -m "not compiled"
```

which excludes the tests marked as requiring the compiled components. Again, look for all tests to report `PASSED`. For more details see the pytest documentation.

If you are interacting with compiled components (e.g. RRTMG radiation), testing is a little more complicated. You will need to rebuild and install a new version. We use (and recommend) conda build to handle the dependencies including Fortran compiler.

To build CLIMLAB, do this from the root directory of the CLIMLAB source repo:

```
conda-build conda-recipe
```

This will automatically install all build dependencies in a temporary new conda environment, build all the Fortran extensions, bundle everything together, install the new pacakge in a temporary test environment, and run a minimal set of tests on the package (only the tests marked as `fast`). The whole procedure will take several minutes to run through.

Assuming the tests pass successfully, you will see a message like:

```
TEST END: /Users/br546577/anaconda3/conda-bld/osx-64/climlab-0.6.5.dev0-py36_5.tar.bz2
```

(though obviously with different paths and version numbers)

To fully test your new build (including the tests not marked as `fast`), you can now install it in a new test environment (with all dependencies) and run the full set of tests:

```
conda create --name newtest climlab --use-local
source activate newtest
pytest -v --pyargs climlab
```

Once you're happy with this you can safely delete the test environment with:

```
source deactivate
conda remove --name newtest --all
```

If you encounter problems with the conda build recipe, please raise an issue at <https://github.com/brian-rose/climlab/issues>. You could also take a look at the CLIMLAB recipe used on conda-forge, which might be a little more up-to-date.

11.6 Contributing improved documentation

The documentation is generated with Sphinx from docstrings in the source code itself, along with a small collection of ReStructuredText (.rst) files. You can help improve the documentation!

- Edit doctrings and/or .rst files in `climlab/docs/`
- Build the improved docs locally with:

```
make html
```

from the `climlab/docs` directory.

- The new and improved docs should now be available locally in the `climlab/docs/build/html` directory. Check them out in your web browser.
- Once you are satisfied, commit changes as described above and submit a new Pull Request describing your changes.

CHAPTER 12

References

- A. Berger. Long-term variations of daily insolation and quaternary climatic changes. *Journal of Atmospheric Science*, 35(12):2362–2367, 1978.
- A. Berger and M. F. Loutre. Insolation values for the climate of the last 10 million years. *Quaternary Science Reviews*, 10(4):297–317, 1991.
- M. I. Budyko. The effect of solar radiation variations on the climate of the earth. *Tellus*, 21(5):611–619, 1969.
- Ken Caldeira and James F. Kasting. Susceptibility of the early earth to irreversible glaciation caused by carbon dioxide clouds. *Nature*, 359:226–228, 1992.
- J. Laskar, P. Robutel, F. Joutel, M. Gastineau, A. C. M. Correia, and B. Levrard. A long-term numerical solution for the insolation quantities of the earth. *Astronomy & Astrophysics*, 428:261–285, 2004.

CHAPTER 13

License

The climlab package and associated documentation is freely available under MIT License


```
The MIT License (MIT)

Copyright (c) 2017 Brian E. J. Rose

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 14

Acknowledgement

Development of CLIMLAB and associated documentation is partially supported by the National Science Foundation under Grant Number AGS-1455071 to Brian Rose.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 15

Contact

The lead developer and maintainer of *climlab* is

Brian E. J. Rose
Department of Atmospheric and Environmental Sciences
University at Albany
brose@albany.edu

Community contributions are very welcome! Bugs can be reported through the issue tracker on github.

The documentation was originally created by **Moritz Kreuzer**, Potsdam Institut for Climate Impact Research (PIK). Other contributers include

- Ryan Abernathey
- Christopher Cardinale