climlab-0.3.0.1 Documentation

Release 0.3.0.1a

Moritz Kreuzer

March 16, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 What is 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.

Most of the actual computation for simpler model components use vectorized `numpy` array functions. It should run out-of-the-box on a standard scientific Python distribution, such as `Anaconda` or `Enthought Canopy`.

1.2 What's new in version 0.3?

New in version 0.3, *climlab* now includes Python wrappers for more numerically intensive processes implemented in Fortran code (specifically the CAM3 radiation module). These require a Fortran compiler on your system, but otherwise have no other library dependencies. *climlab* uses a compile-on-demand strategy. The compiler is invoked automatically as necessary when a new process is created by the user.

1.3 Implementation of models

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

- 1D radiative and radiative-convective single column models, with various radiation schemes:
  - Grey Gas
  - Simplified band-averaged models (4 bands each in longwave and shortwave)
  - One GCM-level radiation module (CAM3)
- 1D diffusive energy balance models
- Seasonal and steady-state models
- Arbitrary combinations of the above, for example:
  - 2D latitude-pressure models with radiation, horizontal diffusion, and fixed relative humidity
- orbital / insolation calculations
- boundary layer sensible and latent heat fluxes

Note: For more details about the implemented Energy Balance Models, see the *Models* chapter.

1.4 Documentation

This documentation currently only covers all Energy Balance Model relevant parts of the code, which is just a part of the package. The whole package may be covered in a later release of the documentation.

CHAPTER

TWO

DOWNLOAD

2.1 Code

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

- Stable Releases
- Development Version

2.2 Dependencies

climlab is written in Python 2.7 and requires following *Python* packages to run:

- Numpy
- Scipy
- NetCDF4

Optional packages:

- Jupyter (to run Jupyter notebooks containing tutorials and introduction for climlab)
- Matplotlib (plotting libary)

The packages have to be installed on your machine. They can either be downloaded, compiled and installed individually.

Otherwise Python distributions like Anaconda or Enthought Canopy can be used which already include many popular Python packages.

2.2.1 Setup environment with Anaconda

An example is given here how to set up a python environment with Anaconda.

A new environment named `climlab_env` with all above packages is created like this:

```
$ conda create --name climlab_env numpy scipy netcdf4 jupyter matplotlib
```

All new packages which will be installed are displayed including their version number. Press `y` to proceed.

After the environment is build, it can be activated with

```
$ source activate climlab_env
```

and can be deactivated with

```
$ source deactivate
```

To install climlab in the new environment follow the steps below.

2.3 Installation

2.3.1 Stable Release

With the Python package pip which collects the current version from the Python Package Index, climlab can be easily installed on a machine through the terminal command

```
    $ pip install climlab
```

2.3.2 Development Version

Otherwise the package can be downloaded from the above referred link and installed manually through running from the package directory

```
    $ python setup.py install
```

for a regular system wide installation.

In case you want to develop new code, run following command (which also has an uninstall option):

```
    $ python setup.py develop
```

CHAPTER

THREE

ARCHITECTURE

The backbone of the climlab architecture are *Processes* and their relatives *TimeDependentProcesses*. 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 the Outgoing Longwave Radiation of a surface, based on the Stefan-Boltzmann Law for a grey body, is implemented in the `Boltzmann` which is also a climlab *Process*, and so on...

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

For more about models, see the climlab *Models* chapter.

A *Process* that represents a whole model will have a couple of *subprocesses* which will be *Processes* themselves. They represent a certain part of the model, for example the albedo or the insolation component. For more about subprocesses, see below.

A **Process** is 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.

3.1 Process

A process is an instance of the class `Process`. Most processes are timedependent and therefore instance of the daughter class `TimeDependentProcess`.

3.1.1 Basic Dictionaries

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

- **`process.state`** contains the process's state variables, which are usually time-dependent and which are major quantities that define the process's state and are usually time-dependent. 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) does not have this dictionary.

- **process.diagnostics** contains any quantity derived from current state. In an Energy Balance Model this dictionary can have entries like 'ASR', 'OLR', 'icelat', 'net_radiation', 'albedo', 'insolation'.
- **process.subprocess** holds subprocesses of the parent *process*. For details on subprocesses see below.

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

3.1.2 Subprocesses

Subprocesses are representing and modelling 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 themself, like albedo in this case.

A subprocess is same as the parent process an instance of the Process class. So it has dictionaries and attributes with same names as it's parent process. Not necessary all will be the same or having the same entries, but a subprocess has at least the basic dictionaries and attributes created during initialization of the Process 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 it's 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.

3.1.3 Process Integration over time

TimeDependentProcess -es can be integrated over time to see how the state variables and other diagnostic variables vary in time.

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 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

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

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

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

So the new state of S is calculated trough multiplying the process tendencies of S with the timestep and adding them up to the previous state of S.

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:

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

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* below.

- **`compute()` 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:

    $$\text{tendencies}(T) = \frac{SP_A(T)}{C} + \frac{SP_B(T)}{C} + ...$$
  - keys for this dictionary are 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 state variable T.

  - tendency dictionary is the total tendency including all subprocesses

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

  - method only computes d/dt but **does not apply changes** (which is done by `step_forward()`)
  - thus method is relatively independent of numerical scheme
  - method **will update** variables in `proc.diagnostic` dictionary. Therefore it will also *gather all diagnostics* from the *subprocesses*

- **`step_forward()` updates the state variables**
  - it calls `compute()` to get current tendencies
  - method multiplies state tendencies with the timestep and adds them up to the state variables
- `integrate_years()` etc will automate time-stepping through calling the `step_forward` method multiple times. It also does the computation of time-average diagnostics.
- `integrate_converge()` calls `integrate_years()` as long as the state variables keep changing over time.

  Example to integrate 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)
```

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()` method handles them seperately as well as in specific order. It calls private `_compute()` methods that are specified in daugther classes of `Process` namely `DiagnosticProcess`, `EnergyBudget` (which are explicit processes) or `ImplicitProcess`.

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

> The function first computes all diagnostic processes as they may effect all the other processes (such as change in solar distribution). After all the diagnostic processes don't produce any tendencies directly. 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. So the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated through matrix inversions and same like the explicit tendencies applied to the states temporarily. Subsequently all instantaneous adjustments are computed.
>
> Then the changes made to the states from explicit and implicit processes are removed again as this `compute()` function is supposed to calculate only tendencies and not applying 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()` method has been called.

3.2 Domain

A *Domain* defines an area or spatial base for a climlab `Process` object. It consists of axes which are `Axis` 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` and `Ocean` of the private `_Domain` class implemented which themselves have daughter classes `SlabAtmosphere` and `SlabOcean`.

Every `Process` 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()`
- `zonal_mean_column()`
- `box_model_domain()`

3.3 Axis

An `Axis` is an object where information of a spacial dimension of a `_Domain` 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` are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of `_Domain`.

3.4 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: `proc` is an instance of the `Process` class.

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 FOUR

MODELS

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

4.1 Energy Balance Model

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

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

4.1.1 EBM Subprocesses

Insolation

- **`FixedInsolation`** defines a constant solar factor for all spatial points of the domain.

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

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

$$S(lat) = \frac{S_0}{4}\left[1 + s_2 P_2(\sin lat)\right]$$

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

Albedo

- **`ConstantAlbedo`** defines constant albedo values at all spatial points of the domain:

$$\alpha(lat) = a_0$$

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

$$\alpha(lat) = a_0 + a_2 P_2(\sin lat)$$

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

Outgoing Longwave Radiation

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

$$OLR = A + B \cdot T$$

- **AplusBT_CO2** calculates OLR same as `AplusBT` but uses

parameters A and B dependent of the atmospheric CO2 concentration c.

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

- **Boltzmann** calculates OLR after the Stefan-Boltzmann law for a grey body like

$$OLR = \sigma \varepsilon T^4$$

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** calculates the energy transport for each latitude φ in relation to the global mean temperature $\bar{T}$:

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

4.1.2 EBM templates

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

EBM

The `EBM` class sets up a typical Energy Balance Model with following subprocesses:

- Outgoing Longwave Radiation (OLR) parameterization through `AplusBT`
- solar insolation paramterization through `P2Insolation`
- albedo parameterization in dependence of temperature through `StepFunctionAlbedo`
- energy diffusion through `MeridionalDiffusion`

EBM_seasonal

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

- Outgoing Longwave Radiation (OLR) parameterization through `AplusBT`
- solar insolation paramterization through `DailyInsolation`
- albedo parameterization in dependence of temperature through `StepFunctionAlbedo`
- energy diffusion through `MeridionalDiffusion`

EBM_annual

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

- Outgoing Longwave Radiation (OLR) parameterization through `AplusBT`
- solar insolation paramterization through `AnnualMeanInsolation`
- albedo parameterization in dependence of temperature through `StepFunctionAlbedo`
- energy diffusion through `MeridionalDiffusion`

Note: For information how to set up individual models or modify instances of the classes above, see the *Tutorial* chapter.

4.2 Other Models

As noted in the *Introduction* more model types are implemented in climlab but not covered in the documentation yet.

CHAPTER

FIVE

TUTORIAL

In this chapter you can find some tutorials and examples how to work with the *climlab* code.

5.1 Example usage

The directory `climlab/courseware/` contains a collection of IPython / 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
```

5.2 Notebooks

- Boltzmann OLR
- budyko transport
- seasonal/annual EBM
- Insolation
- Orbital Variations

APPLICATION PROGRAMMING INTERFACE

6.1 Subpackages

6.1.1 climlab.domain package

Submodules

climlab.domain.axis module

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` is an object where information of a spacial dimension of a `_Domain` 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` are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of `_Domain`.

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
- **`num_points`** (*`int`*) – number of points on axis
- **`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

Inputs for the `axis_type` like `'p'`, `'press'`, `'pressure'`, `'P'`, `'Pressure'` and `'Press'` are refered to as `'lev'`.

`'Latitude'` and `'latitude'` are refered to as `'lat'`.

`'Longitude'` and `'longitude'` are refered to as `'lon'`.

And `'depth'`, `'Depth'`, `'waterDepth'`, `'water_depth'` and `'slab'` are refered to as `'depth'`.

The **default units** are:

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

climlab.domain.domain module

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

Bases: `climlab.domain.domain._Domain`

Class for the implementation of a Atmosphere Domain.

Object attributes

Additional to the parent class `_Domain` the following object attribute is modified during initialization:

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

set_heat_capacity`()`

Sets the heat capacity of the Atmosphere Domain.

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

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

class `climlab.domain.domain.`**`Ocean`**`(**kwargs)`

Bases: `climlab.domain.domain._Domain`

Class for the implementation of an Ocean Domain.

Object attributes

Additional to the parent class `_Domain` the following object attribute is modified during initialization:

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

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

Calls the utils heat capacity function `ocean()` 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`** (*`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`

A class to create a SlabAtmosphere Domain by default.

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

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

A class to create a SlabOcean Domain by default.

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

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` object. It consists of axes which are `Axis` 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` and `Ocean` of the private `_Domain` class implemented which themselves have daughter classes `SlabAtmosphere` and `SlabOcean`.

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

- `single_column()`
- `zonal_mean_column()`
- `box_model_domain()`

Initialization parameters

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

Parameters **`axes`** (dict or `Axis`) – 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()` called with input argument `axes`
- **`numdims`** (*`int`*) – Number of `Axis` 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`** (*`array`*) – the domain's heat capacity over axis specified in function call of `set_heat_capacity()`

`_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` 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`

`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`

`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`

`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` 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`, `SlabAtmosphere`

Example

`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()`)

•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` or latitude array) – specification for latitude axis (optional)
- **`lev`** (`Axis` 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`, `Atmosphere`

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

Creates a Domain with one water cell and a latitude axis above.

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

Function-call argument

Parameters

- **num_lat** (*int*) – number of latitude points on the axis [default: 90]
- **water_depth** (*float*) – depth of the water cell (slab ocean) [default: 10.]
- **lat** (`Axis` 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`

climlab.domain.field module

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*) – the domain associated with that field (e.g. state variables)

Object attributes

Following object attribute is generated during initialization:

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

Example

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

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

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

Raises `ValueError` if input field has no latitude axis

Returns latitude weighted global mean of the field

Return type float

climlab.domain.initial module

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 TOA) [default: 30]
- **num_lat** (*int*) – number of latitude points on the axis [default: 1]
- **lev** (`Axis` or pressure array) – specification for height axis (optional)
- **lat** (*array*) – size of array determines dimension of latitude
- **water_depth** (*float*) – *irrelevant*

Returns dictionary with two temperature `Field` for atmospheric column `Tatm` and surface mixed layer `Ts`

Return type dict

`climlab.domain.initial.`**`surface_state`**(*num_lat=90*, *water_depth=10.0*, *T0=12.0*, *T2=-40.0*)

Sets up a state variable dictionary for a zonal-mean surface model (e.g. basic EBM).

Returns a single state variable *Ts*, the temperature of the surface mixed layer, initialized by a basic temperature and the second Legendre polynomial.

Function-call arguments

Parameters

- **num_lat** (*int*) – number of latitude points on the axis [default: 90]
- **water_depth** (*float*) – *irrelevant*
- **T0** (*float*) – base value for initial temperature
  - unit °C
  - default value: `12`
- **T2** (*float*) – factor for 2nd Legendre polynomial `P2` to calculate initial temperature
  - unit: dimensionless
  - default value: `40`

Returns dictionary with temperature `Field` for surface mixed layer `Ts`

Return type dict

Module contents

6.1.2 climlab.dynamics package

Submodules

climlab.dynamics.budyko_transport module

class `climlab.dynamics.budyko_transport.`**`BudykoTransport`**(*b=3.81*, ***kwargs*)

Bases: `climlab.process.energy_budget.EnergyBudget`

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

Variables **b** (*float*) – budyko transport parameter

In a global Energy Balance Model

$$C\frac{dT}{dt} = R\downarrow - R\uparrow - H$$

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

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

where T is a vector of the model temperature and $\bar{T}$ describes the mean value of T.

For further information see *[Budyko1969]*.

b

the budyko transport parameter in unit $\frac{\mathrm{W}}{\mathrm{m}^2\mathrm{K}}$

Getter returns the budyko transport parameter

Setter sets the budyko transport parameter

Type float

climlab.dynamics.diffusion module

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

Bases: `climlab.process.implicit.ImplicitProcess`

A parent class for one dimensional implicit diffusion modules.

Solves the one dimensional heat equation

$$\frac{dT}{dt} = \frac{d}{dy}\left[K \cdot \frac{dT}{dy}\right]$$

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()`

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` 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()`
- **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()` 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.

_implicit_solver()

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

The method is called by the `_compute()` function of the `ImplicitProcess` 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()` function of the `ImplicitProcess` 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

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

Bases: `climlab.dynamics.diffusion.Diffusion`

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 $1/s$

Object attributes

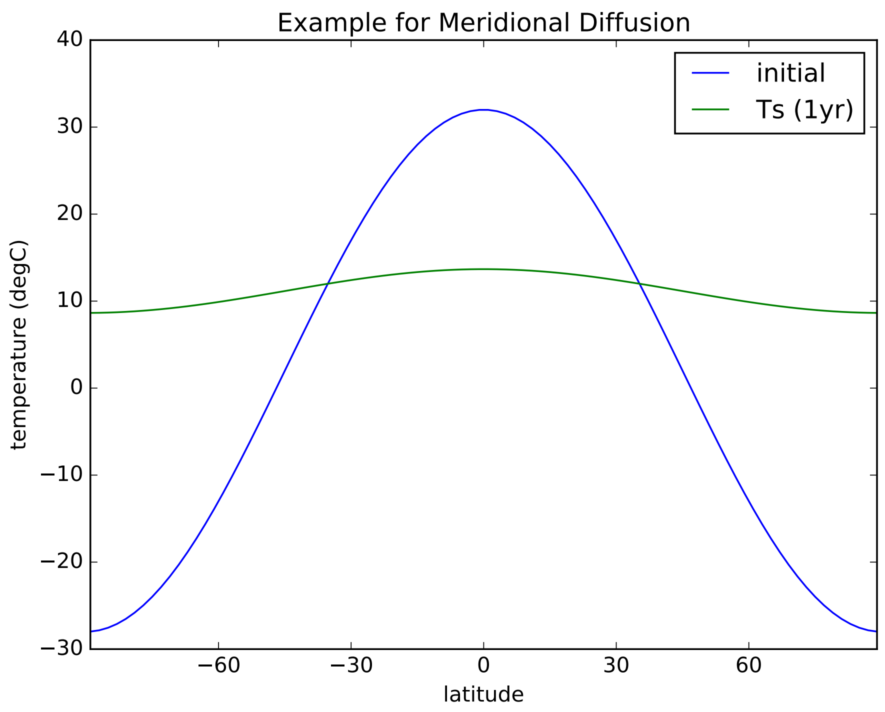
Additional to the parent class `Diffusion` which is initialized with `diffusion_axis='lat'`, following object attributes are modified during initialization:

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()
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 (degC)')
ax.plot(lat, initial,         label='initial')
ax.plot(lat, Ts, label='Ts (1yr)')
ax.legend(loc='best')
plt.show()
```



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, _Domain 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

$$\text{diffTriDiag} = \begin{bmatrix} 1+\frac{w_{1,1}K_1}{w_{2,0}} & -\frac{w_{1,1}K_1}{w_{2,0}} & 0 & & ... & \\ -\frac{w_{1,1}K_1}{w_{2,1}} & 1+\frac{w_{1,1}K_1+w_{1,2}K_2}{w_{2,1}} & -\frac{w_{1,2}K_2}{w_{2,1}} & 0 & ... & \\ 0 & -\frac{w_{1,2}K_2}{w_{2,2}} & 1+\frac{w_{1,2}K_2+w_{1,3}K_3}{w_{2,2}} & -\frac{w_{1,3}K_3}{w_{2,2}} & ... & \\ & & \ddots & \ddots & \ddots & \\ 0 & 0 & ... & -\frac{w_{1,n-2}K_{n-2}}{w_{2,n-2}} & 1+\frac{w_{1,n-2}K_{n-2}+w_{1,n-1}K_{n-1}}{w_{2,n-2}} & -w_1 \\ 0 & 0 & ... & 0 & -\frac{w_{1,n-1}K_{n-1}}{w_{2,n-1}} & 1+w \end{bmatrix}$$

where

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

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

Calls `_make_diffusion_matrix()` 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:

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

when bounds and points from `lataxis` are written as

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

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

$$\text{diffTriDiag} = \begin{bmatrix} 1+\frac{\cos(b_1)K_1}{\cos(p_0)} & -\frac{\cos(b_1)K_1}{\cos(p_0)} & 0 & & ... \\ -\frac{\cos(b_1)K_1}{\cos(p_1)} & 1+\frac{\cos(b_1)K_1+\cos(b_2)K_2}{\cos(p_1)} & -\frac{\cos(b_2)K_2}{\cos(b_1)} & 0 & ... \\ 0 & -\frac{\cos(b_2)K_2}{\cos(p_2)} & 1+\frac{\cos(b_2)K_2+\cos(b_3)K_3}{\cos(p_2)} & -\frac{\cos(b_3)K_3}{\cos(p_2)} & ... \\ & & \ddots & \ddots & \ddots \\ 0 & 0 & ... & -\frac{\cos(b_{n-2})K_{n-2}}{\cos(p_{n-2})} & 1+\frac{\cos(b_{n-2})K_{n-2}+\cos(b}{\cos(p_{n-2})} \\ 0 & 0 & ... & 0 & -\frac{\cos(b_{n-1})K_{n-}}{\cos(p_{n-1})} \end{bmatrix}$$

`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

Module contents

6.1.3 climlab.model package

Submodules

climlab.model.ebm module

class `climlab.model.ebm.`**`EBM`**(*num_lat=90*, *S0=1365.2*, *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.energy_budget.EnergyBudget`

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

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

•Outgoing Longwave Radiation (OLR) parameterization through `AplusBT`

•solar insolation paramterization through `P2Insolation`

•albedo parameterization in dependence of temperature through `StepFunctionAlbedo`

•energy diffusion through `MeridionalDiffusion`

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`
  – default value: `90`
- **`S0`** (*`float`*) – solar constant
  – unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  – default value: `1365.2`
- **`A`** (*`float`*) – parameter for linear OLR parameterization `AplusBT`
  – unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$

  - default value: `210.0`
- **B** (*float*) – parameter for linear OLR parameterization `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` domain, which the heat capacity is dependent on
  - unit: meters
  - default value: `10.0`
- **Tf** (*float*) – freezing temperature
  - unit: °C
  - default value: `-10.0`
- **a0** (*float*) – base value for planetary albedo parameterization `StepFunctionAlbedo`
  - unit: dimensionless
  - default value: `0.3`
- **a2** (*float*) – parabolic value for planetary albedo parameterization `StepFunctionAlbedo`
  - unit: dimensionless
  - default value: `0.078`
- **ai** (*float*) – value for ice albedo paramerization in `StepFunctionAlbedo`
  - 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 °C
  - default value: `12`
- **T2** (*float*) – factor for 2nd Legendre polynomial `P2` 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()`. In the meantime the object's `domains` and `state` dictionaries are updated.
- **subprocess** (*dict*) – Several subprocesses are created (see above) through calling `add_subprocess()` and therefore the subprocess dictionary is updated.
- **topdown** (*bool*) – is set to `False` to call subprocess compute methods first. See also `TimeDependentProcess`.
- **diagnostics** (*dict*) – is initialized with keys: `'OLR'`, `'ASR'`, `'net_radiation'`, `'albedo'` and `'icelat'` through `init_diagnostic()`.

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)`

global_mean_temperature()
Convenience method to compute global mean surface temperature.

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

heat_transport()
Returns instantaneous heat transport in unit PW on the staggered grid (bounds) through calling `diffusive_heat_transport()`.

heat_transport_convergence()
Returns instantaneous convergence of heat transport.

$$h(\varphi) = -\frac{1}{2\pi R^2 cos(\varphi)} \frac{dH}{d\varphi} \approx -\frac{1}{2\pi R^2 cos(\varphi)} \frac{\Delta H}{\Delta\varphi}$$

h is the *dynamical heating rate* in unit W/m^2 which is the convergence of energy transport into each latitude band, namely the difference between what's coming in and what's going out.

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)`

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

Bases: `climlab.model.ebm.EBM_seasonal`

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

The annual solar distribution is calculated through averaging the `DailyInsolation` 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`

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

Bases: `climlab.model.ebm.EBM`

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 parameterization `StepFunctionAlbedo`
  – default value: `0.33`
- **`a2`** (*float*) – parabolic value for planetary albedo parameterization `StepFunctionAlbedo`
  – default value: `0.25`
- **`ai`** (*float*) – value for ice albedo paramerization in `StepFunctionAlbedo`
  – default value: `None`

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`.

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

Variables

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

if `'ai'` *is given*:

Variables

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

Module contents

6.1.4 climlab.process package

Module contents

module content test

Submodules

climlab.process.diagnostic module

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

Bases: `climlab.process.time_dependent_process.TimeDependentProcess`

A parent class for all processes that are strictly diagnostic, namely no time dependence.

During initialization following attribute is set:

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

climlab.process.energy_budget module

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

Bases: `climlab.process.time_dependent_process.TimeDependentProcess`

A parent class for explicit energy budget processes.

This class solves equations that include a heat capacitiy term like $C\frac{dT}{dt}$ = flux convergence

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

$$C\frac{dT}{dt} = R\downarrow - R\uparrow - H$$
$$\frac{dT}{dt} = \frac{R\downarrow}{C} - \frac{R\uparrow}{C} - \frac{H}{C}$$

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

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

Object attributes

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 $\mathrm{W/m^2}$ stored in a dictionary sorted by model states

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

Bases: `climlab.process.energy_budget.EnergyBudget`

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

Initialization parameters

An instance of `ExternalEnergySource` is initialized with the forwarded keyword arguments `**kwargs` of hypthetical corresponding children classes (which are not existing in this case).

Object attributes

Additional to the parent class `EnergyBudget` 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.

climlab.process.implicit module

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

Bases: `climlab.process.time_dependent_process.TimeDependentProcess`

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

`_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()` method of the parent `TimeDependentProcess` 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.

climlab.process.process module

class `climlab.process.process.`**`Process`**(*state=None*, *domains=None*, *subprocess=None*, *lat=None*, *lev=None*, *num_lat=None*, *num_levels=None*, *input=None*, ***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* 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`*) – spatial state variable for the process. Set to `None` if not specified.
- **`domains`** (`_Domain` or dict of `_Domain`) – domain(s) for the process
- **`subprocess`** (`Process` or dict of `Process`) – 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

Object attributes

Additional to the parent class `Process` following object attributes are generated during initialization:

Variables

- **`domains`** (*`dict`*) – dictionary of process :class:‘~climlab.domain.domain._Domain‘s
- **`state`** (*`dict`*) – dictionary of process states (of type `Field`)
- **`param`** (*`dict`*) – dictionary of model parameters which are given through `**kwargs`
- **`diagnostics`** (*`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`) – dictionary of suprocesses of the process

`add_input`(*inputlist*)

Updates the process's list of inputs.

Parameters **`inputlist`** (*`list`*) – list of names of input variables

`add_subprocess`(*name*, *proc*)

Adds a single subprocess to this process.

Parameters

- **`name`** (*`string`*) – name of the subprocess
- **`proc`** (*`process`*) – a Process object

Raises

exc *ValueError* if `proc` is not a process

add_subprocesses (*procdict*)
Adds a dictionary of subproceses to this process.

Calls `add_subprocess()` 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

depth
Property of depth points of the process.

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
Property of depth bounds of the process.

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.

init_diagnostic (*name*, *value=0.0*)
Defines a new diagnostic quantity called `name` and initialize it with the given `value`.

Quantity is accessible and settable in two ways:

•as a process attribute, i.e. `proc.name`

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

Parameters

- **name** (*str*) – name of diagnostic quantity to be initialized
- **value** (*array*) – initial value for quantity - accepts also type float, int, etc. (*default*:`0.`)

input
dictionary with all input variables

That can be boundary conditions and other gridded quantities independent of the *process*

Getter Returns the content of `self._input_vars`.

Type dict

lat
Property of latitudinal points of the process.

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
Property of latitudinal bounds of the process.

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
Property of altitudinal points of the process.

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
Property of altitudinal bounds of the process.

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
Property of longitudinal points of the process.

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
Property of longitudinal bounds of the process.

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

remove_subprocess(*name*)
Removes a single subprocess from this process.

Parameters **name** (*string*) – name of the subprocess

set_state(*name, value*)
Sets the variable `name` to a new state `value`.

Parameters

- **name** (*string*) – name of the state
- **value** (`Field` 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:

`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` or `_Domain`) – 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

`climlab.process.process.`**`process_like`**`(`*proc*`)`

Copys the given process.

The creation date is updated.

Parameters **`proc`** (*`process`*) – process

Returns new process identical to the given process

Return type process

climlab.process.time_dependent_process module

class `climlab.process.time_dependent_process.`**`TimeDependentProcess`**`(`*time_type='explicit', timestep=None, topdown=True, **kwargs*`)`

Bases: `climlab.process.process.Process`

A generic parent class for all time-dependent processes.

`TimeDependentProcess` is a child of the `Process` 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
- **`time_type`** (*`str`*) – how time-dependent-process should be computed. Set to `'explicit'` by default.
- **`topdown`** (*`bool`*) – whether geneterate *process_types* in regular or in reverse order. Set to `True` by default.

Object attributes

Additional to the parent class `Process` following object attributes are generated during initialization:

Variables

- **`has_process_type_list`** (*`bool`*) – information whether attribute *process_types* (which is needed for `compute()` 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()` has been called for last.

- **tendencies** (*dict*) – computed difference in a timestep for each state. See `compute()` 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()` and `timestep()` 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

compute()

Computes the tendencies for all state variables given current state and specified input.

The function first computes all diagnostic processes as they may effect all the other processes (such as change in solar distribution). After all the diagnostic processes don't produce any tendencies directly. 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. So the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated through matrix inversions and same like the explicit tendencies applied to the states temporarily. Subsequently all instantaneous adjustments are computed.

Then the changes made to the states from explicit and implicit processes are removed again as this `compute()` function is supposed to calculate only tendencies and not applying 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()` 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** (*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
- **verbose** (*bool*) – information whether total elapsed time should be printed.

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()`.

Parameters

- **days** (*float*) – integration time for the model in days
- **verbose** (*bool*) – information whether model time details should be printed.

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
- **verbose** (*bool*) – information whether model time details should be printed.

It calls `step_forward()` repetitively and calculates a time averaged value over the integrated period for every model state and all diagnostics processes.

set_timestep(*timestep=86400.0*, *num_steps_per_year=None*)
Calculates the timestep in unit seconds and calls the setter function of `timestep()`

Parameters

- **timestep** (*float*) – the amount of time over which `step_forward()` is integrating in unit seconds
- **num_steps_per_year** (*float*) – a number of steps per calendar year

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()` 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.

timestep
The amount of time over which `step_forward()` 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()`.

Type float

6.1.5 climlab.radiation package

Submodules

climlab.radiation.AplusBT module

class `climlab.radiation.AplusBT.`**`AplusBT`**(*A=200.0*, *B=2.0*, ***kwargs*)

Bases: `climlab.process.energy_budget.EnergyBudget`

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

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

Parameters

- **A** (*float*) – parameter for linear OLR parameterization
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `200.0`
- **B** (*float*) – parameter for linear OLR parameterization
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2\ {}^{\circ}\mathrm{C}}$
  - default value: `2.0`

Object attributes

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

Variables

- **A** (*float*) – calls the setter function of `A()`
- **B** (*float*) – calls the setter function of `B()`
- **diagnostics** (*dict*) – key `'OLR'` initialized with value: `Field` of zeros in size of `self.Ts`
- **OLR** (*Field*) – 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):

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

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`

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`).

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 *[CaldeiraKasting1992]*.

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` following object attributes are generated or updated during initialization:

Variables

- **CO2** (*float*) – calls the setter function of `CO2()`
- **diagnostics** (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.init_diagnostic('OLR', 0. * self.Ts)`
- **OLR** (*Field*) – the subprocess attribute `self.OLR` is created with correct dimensions

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

climlab.radiation.Boltzmann module

class `climlab.radiation.Boltzmann.`**`Boltzmann`**(*eps=0.65*, *tau=0.95*, ***kwargs*)

Bases: `climlab.process.energy_budget.EnergyBudget`

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` 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` 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

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
  - unit: dimensionless
  - default value: `0.65`
- **`tau`** (*`float`*) – transmissivity of the planet's atmosphere which is the effectiveness in transmitting the longwave radiation emitted from the surface
  - unit: dimensionless
  - default value: `0.95`

Object attributes

During initialization both arguments described above are created as object attributes which calls their setter function (see below).

Variables

- **`eps`** (*`float`*) – calls the setter function of `eps()`
- **`tau`** (*`float`*) – calls the setter function of `tau()`

- **diagnostics** (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.init_diagnostic('OLR', 0. * self.Ts)`
- **OLR** (*Field*) – the subprocess attribute `self.OLR` is created with correct dimensions

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

climlab.radiation.insolation module

class `climlab.radiation.insolation.`**`AnnualMeanInsolation`**(*S0=1365.2*, *orb={'long_peri': 281.37, 'ecc': 0.017236, '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.

Therefor it uses the method `daily_insolation()`. 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()` and stored in the object's attribute `self.insolation`

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - 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`** (*`Field`*) – the solar distribution is calculated as a Field on the basis of the `self.domains['default']` domain and stored in the attribute `self.insolation`.
- **`orb`** (*`dict`*) – initialized with given argument `orb`

`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={'long_peri': 281.37, 'ecc': 0.017236, 'obliquity': 23.446}, **kwargs*)

Bases: `climlab.radiation.insolation.AnnualMeanInsolation`

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.

Therefor it uses the method `daily_insolation()`. 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** (*Field*) – the solar distribution is calculated as a Field on the basis of the `self.domains['default']` domain and stored in the attribute `self.insolation`.
- **orb** (*dict*) – initialized with given argument `orb`

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

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

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

class `climlab.radiation.insolation._Insolation`(*S0=1365.2*, ***kwargs*)

Bases: `climlab.process.diagnostic.DiagnosticProcess`

A private parent class for insolation processes.

Calling compute() will update self.insolation with current values.

Initialization parameters

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

Parameters **S0** (*float*) – solar constant

- unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
- default value: `1365.2`

Object attributes

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

Variables

- **insolation** (*Field*) – the array is initialized with zeros of the size of `self.domains['sfc']` or `self.domains['default']`.
- **S0** (*float*) – initialized with given argument `S0`
- **diagnostics** (*dict*) – key `'insolation'` initialized with value: `Field` of zeros in size of `self.domains['sfc']` or `self.domains['default']`
- **insolation** – the subprocess attribute `self.insolation` is created with correct dimensions

Note: `self.insolation` should always be modified with `self.insolation[:] = ...` so that links to the insolation in other processes will work.

S0

Property of solar constant S0.

The parameter S0 is stored using a python property and can be changed through `self.S0 = newvalue` which will also update the parameter dictionary.

Warning: changing `self.param['S0']` will not work!

Getter Returns the S0 parameter which is stored in attribute `self._S0`.

Setter

- sets S0 which is addressed as `self._S0` to the new value
- updates the parameter dictionary `self.param['S0']` and
- calls method `_compute_fixed()`

Type float

climlab.radiation.radiation module

Radiation is the base class climlab grey radiation and band modules

Basic characteristics:

State: - Ts (surface radiative temperature) - Tatm (air temperature)

Input (specified or provided by parent process): - flux_from_space

Diagnostics (minimum) - flux_to_sfc - flux_to_space - absorbed - absorbed_total

class `climlab.radiation.radiation.`**`Radiation`**`(`***kwargs*`)`

Bases: `climlab.process.energy_budget.EnergyBudget`

Base class for radiation models.

The following boundary values need to be specified by user or parent process: - flux_from_space

The following values are computed are stored in the .diagnostics dictionary: - flux_from_sfc - flux_to_sfc - flux_to_space - absorbed - absorbed_total (all in W/m2)

class `climlab.radiation.radiation.`**`RadiationLW`**`(`*emissivity_sfc=1.0*, *albedo_sfc=0.0*, ***kwargs*`)`

Bases: `climlab.radiation.radiation.Radiation`

class `climlab.radiation.radiation.`**`RadiationSW`**`(`*emissivity_sfc=0.0*, *albedo_sfc=1.0*, ***kwargs*`)`

Bases: `climlab.radiation.radiation.Radiation`

Module contents

6.1.6 climlab.solar package

Submodules

climlab.solar.insolation module

This module contains general-purpose routines for computing incoming solar radiation at the top of the atmosphere.

Currently, only daily average insolation is computed.

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 *[Berger1978]*. Further references: *[Berger1991]*.

`climlab.solar.insolation.`**`daily_insolation`**(*lat*, *day*, *orb={'long_peri': 281.37, 'ecc': 0.017236, 'obliquity': 23.446}*, *S0=None*, *day_type=1*)

Compute daily average insolation given latitude, time of year and orbital parameters.

Orbital parameters can be computed for any time in the last 5 Myears with `lookup_parameters()` (see example below).

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 `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: $\mathrm{W/m^2}$

    – 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()`

Example to compute the timeseries of insolation at 65N at summer solstice over the past 5 Myears

`climlab.solar.insolation.`**`solar_longitude`**(*day*, *orb={'long_peri': 281.37, 'ecc': 0.017236, 'obliquity': 23.446}*, *days_per_year=None*)

Estimates solar longitude from calendar day.

Method is using an approximation from *[Berger1978]* section 3 (lambda = 0 at spring equinox).

Function-call arguments

Parameters

- **`day`** (*`aray`*) – 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 (default: 365.2422).

  Reads the length of the year from `constants` if available.

Returns

solar longitude `lambda_long`

Dimensions of output are `( day.size, ecc.size )`

Return type array

Works for both scalar and vector orbital parameters.

climlab.solar.orbital module

This module defines the class `OrbitalTable` which holds orbital data, and includes a method `lookup_parameters()` which interpolates the orbital data for a specific year (- works equally well for arrays of years).

The base class `OrbitalTable()` is designed to work with 5 Myears of orbital data (**eccentricity, obliquity, and longitude of perihelion**) from *[Berger1991]*.

Data will be read from the file orbit91, which was originally obtained from ftp://ftp.ncdc.noaa.gov/pub/data/paleo/insolation/ If the file isn't found locally, the module will attempt to read it remotely from the above URL.

A subclass `LongOrbitalTable()` works with La2004 orbital data for -51 to +21 Myears as calculated by *[Laskar2004]*. See http://vo.imcce.fr/insola/earth/online/earth/La2004/README.TXT

class `climlab.solar.orbital.`**`LongOrbitalTable`**

Bases: `climlab.solar.orbital.OrbitalTable`

Loads orbital parameter tables for -51 to +21 Myears.

Based on calculations by ***[Laskar2004]*** http://vo.imcce.fr/insola/earth/online/earth/La2004/README.TXT

Usage is identical to parent class `OrbitalTable()`.

class `climlab.solar.orbital.`**`OrbitalTable`**

Invoking OrbitalTable() will load 5 million years of orbital data from *[Berger1991]* and compute linear interpolants.

The data can be accessed through the method `lookup_parameters()`.

Object attributes

Following object attributes are generated during initialization:

Variables

- **`kyear`** (*`array`*) – time table with negative values are before present (*unit:* kyears)
- **`ecc`** (*`array`*) – eccentricity over time (*unit:* dimensionless)
- **`long_peri`** (*`array`*) – longitude of perihelion (precession angle) (*unit:* degrees)
- **`obliquity`** (*`array`*) – obliquity angle (*unit:* degrees)
- **`kyear_min`** (*`float`*) – minimum value of time table (*unit:* kyears)
- **`kyear_max`** (*`float`*) – maximum value of time table (*unit:* kyears)

`lookup_parameters`(*kyear=0*)

Look up orbital parameters for given kyear measured from present.

Note: Input `kyear` is thousands of years after present. For years before present, use `kyear < 0`.

Function-call argument

Parameters **`kyear`** (*`array`*) – Time for which oribtal parameters should be given. Will handle scalar or vector input (for multiple years).

Returns

a three-member dictionary of orbital parameters:

- `'ecc'`: eccentricity (dimensionless)
- `'long_peri'`: longitude of perihelion relative to vernal equinox (degrees)
- `'obliquity'`: obliquity angle or axial tilt (degrees).

Each member is an array of same size as kyear.

Return type dict

climlab.solar.orbital_cycles module

Module for setting up long integrations of climlab processes over orbital cycles.

Example

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*)

Automatically integrates a process through changes in orbital parameters.

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`) – 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 if set to `True` (default).

Object attributes

Following object attributes are generated during initialization:

Variables

- **model** (`TimeDependentProcess`) – 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** (*dict*) – orbital parameters for last integrated segment

Module contents

6.1.7 climlab.surface package

Submodules

climlab.surface.albedo module

class `climlab.surface.albedo.`**`ConstantAlbedo`**(*albedo=0.33*, ***kwargs*)

Bases: `climlab.process.diagnostic.DiagnosticProcess`

A class for constant albedo values at all spatial points of the domain.

Initialization parameters

Parameters **albedo** (*float*) – albedo values

- unit: dimensionless

- default value: `0.33`

Object attributes

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

Variables albedo (*Field*) – attribute to store the albedo value. During initialization the `albedo()` setter is called.

Uniform prescribed albedo.

albedo

Property of albedo value.

Getter Returns the albedo value which is stored in diagnostic dict `self.diagnostic['albedo']`

Setter

- sets albedo which is addressed as `diagnostics['albedo']` to the new value through creating a Field on the basis of domain `self.domain['default']`
- updates the parameter dictionary `self.param['albedo']`

Type Field

class `climlab.surface.albedo.`**`Iceline`**(*Tf=-10.0*, ***kwargs*)

Bases: `climlab.process.diagnostic.DiagnosticProcess`

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` following object attributes are generated and updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with the input argument `'Tf'`.
- **diagnostics** (*dict*) – key `'icelat'` initialized
- **icelat** (*array*) – the subprocess attribute `self.icelat` is created

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*) – a Field of booleans which are `True` where $T_s \ge T_f$

- **ice** (*Field*) – a Field of booleans which are `True` where $T_s < T_f$
- **icelat** (*array*) – an array with two elements indicating the ice-edge latitudes
- **diagnostics** (*dict*) – key `'icelat'` is updated according to object attribute `self.icelat` during modification

class `climlab.surface.albedo.`**`P2Albedo`**(*a0=0.33*, *a2=0.25*, ***kwargs*)

Bases: `climlab.process.diagnostic.DiagnosticProcess`

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
  - unit: dimensionless
  - default value: `0.33`
- **a2** (*float*) – factor for second legendre polynominal term in albedo function
  - unit: dimensionless
  - default value: `0.25`

Object attributes

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

Variables

- **a0** (*float*) – attribute to store the albedo parameter a0. During initialization the `a0()` setter is called.
- **a2** (*float*) – attribute to store the albedo parameter a2. During initialization the `a2()` setter is called.
- **diagnostics** (*dict*) – key `'albedo'` initialized
- **albedo** (*Field*) – the subprocess attribute `self.albedo` is created with correct dimensions (according to `self.lat`)

`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`

A step function albedo suprocess.

This class itself defines three subprocesses that are created during initialization:

- `'iceline'` - `Iceline`
- `'warm_albedo'` - `P2Albedo`
- `'cold_albedo'` - `ConstantAlbedo`

Initialization parameters

Parameters

- **`Tf`** (*`float`*) – freezing temperature for Iceline subprocess
  - unit: °C
  - default value: `-10`
- **`a0`** (*`float`*) – basic parameter for P2Albedo subprocess
  - unit: dimensionless
  - default value: `0.3`
- **`a2`** (*`float`*) – factor for second legendre polynominal term in P2Albedo subprocess
  - unit: dimensionless
  - default value: `0.078`
- **`ai`** (*`float`*) – ice albedo value for ConstantAlbedo subprocess
  - unit: dimensionless
  - default value: `0.62`

Additional to the parent class `DiagnosticProcess` 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`** (*`dict`*) – key `'albedo'` initialized
- **`albedo`** (*`Field`*) – the subprocess attribute `self.albedo` is created

climlab.surface.surface_radiation module

class `climlab.surface.surface_radiation.`**`SurfaceRadiation`**(*albedo_sfc=None*, ***kwargs*)

Bases: `climlab.process.energy_budget.EnergyBudget`

Module contents

6.1.8 climlab.utils package

Submodules

climlab.utils.attr_dict module

class `climlab.utils.attr_dict.`**`AttrDict`**(**args*, ***kwargs*)

Bases: `dict`

Constructs a `dict` object with attribute access to data.

class `climlab.utils.attr_dict.`**`OrderedAttrDict`**(**args*, ***kwargs*)

Bases: `collections.OrderedDict`

Constructs an `OrderedDict` object with attribute access to data.

climlab.utils.constants module

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)

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}
```

climlab.utils.heat_capacity module

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*	$\textrm{J/m}^2\textrm{/K}$	heat capacity for atmospheric cell
c_p	1004.	$\textrm{J/kg/K}$	specific heat at constant pressure for dry air
dp	*input*	mb	pressure for atmospheric cell
$f_{\textrm{mb-to-Pa}}$	100	$\textrm{Pa/mb}$	conversion factor from mb to Pa
g	9.8	$\textrm{m/s}^2$	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

`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

`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()`

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

climlab.utils.legendre module

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

climlab.utils.walk module

`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()` method to create the process tree.

Parameters

- **`top`** (*`process`*) – 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

`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`*) – top process from where walking should start
- **`topname`** (*`str`*) – name of top process
- **`topdown`** (*`bool`*) – whether geneterate *process_types* in regular or in reverse order. Set to `True` by default.
- **`ignoreFlag`** (*`bool`*) – whether `topdown` flag should be ignored or not

Returns name (str), proc (process), level (int)

Example

Module contents

6.2 Inheritance Diagram

#.. inheritance-diagram:: climlab.convection.convadj climlab.domain.axis climlab.domain.domain climlab.domain.field climlab.process.diagnostic climlab.process.energy_budget climlab.process.implicit climlab.process.process climlab.process.time_dependent_process climlab.radiation.AplusBT climlab.radiation.cam3rad climlab.radiation._cam3_interface climlab.radiation.cloud climlab.radiation.greygas climlab.radiation.insolation climlab.radiation.nband climlab.radiation.radiation climlab.radiation.transmissivity climlab.radiation.water_vapor climlab.solar.insolation climlab.solar.orbital climlab.solar.orbital_cycles climlab.surface.albedo climlab.surface.surface_radiation climlab.surface.turbulent # :parts: 2

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

LICENSE

8.1 climlab

The climlab Python package is licensed under MIT License:

```
The MIT License (MIT)

Copyright (c) 2015 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.
```

8.2 Documentation

CHAPTER

NINE

CONTACT

9.1 climlab package

The climlab package has been developed by Brian Rose:

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

Bug reports can be reported through the issue tracker on github.

9.2 climlab documentation

The documentation has been built by Moritz Kreuzer using Sphinx. Based on some commentary strings in the source code and a couple of Jupyter Notebooks, this documentation has been developed.

Currently it covers only the Energy Balance Model relevant parts of the package.

Moritz Kreuzer
Potsdam Institut for Climate Impact Research (PIK)
Potsdam, Germany
kreuzer@pik-potsdam.de

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

[Berger1978] Berger A. 1978. “Long-term variations of daily insolation and Quaternary climatic changes.” *Journal of Atmospheric Science* 35(12):2362-2367.

[Berger1991] Berger A./Loutre M.F. 1991. “Insolation values for the climate of the last 10 million years.” *Quaternary Science Reviews* 10(4):297-317.

[Budyko1969] Budyko, M. I. 1969. “The effect of solar radiation variations on the climate of the Earth.” *Tellus* 21(5):611–619.

[CaldeiraKasting1992] Caldeira, Ken/Kasting, James. 1992. “Susceptibility of the early Earth to irreversible glaciation caused by carbon dioxide clouds.” *Nature* 359:226-228.

[Laskar2004] Laskar, J./P. Robutel/F. Joutel/M. Gastineau/A. C. M. Correia/B. Levrard. 2004. “A long-term numerical solution for the insolation quantities of the Earth.” *Astronomy & Astrophysics* 428:261–285.