climlab-0.2.13 Documentation Documentation

Release 0.2.13a

Moritz Kreuzer

February 15, 2016

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

What is climlab?

CHAPTER

TWO

DOWNLOAD

2.1 Code

github

2.2 Dependencies

numpy, scipy, etc.

2.3 Installation

```
$ pip install climlab
```

2.4 Source Code

#.. autofunction:: climlab.model.ebm.EBM

CHAPTER

THREE

MODELS

CHAPTER

FOUR

TUTORIAL

Principles of the new *climlab* API design:

- *climlab.Process* object has several iterable dictionaries of named,

gridded variables:

- ***process.state***
  - state variables, usually time-dependent
- ***process.input***
  - boundary conditions and other gridded quantities independent of the

  process - often set by a parent *process*
- *process.param* (which are basically just scalar *input*)
- ***process.tendencies***
  - iterable *dict* of time-tendencies (d/dt) for each state variable
- ***process.diagnostics***
  - any quantity derived from current state

- The *process* is fully described by contents of *state*, *input* and *param*

dictionaries. *tendencies* and *diagnostics* are always computable from current state. - *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 accessible as attributes

of each variable

- e.g. Tatm.lat
- Shortcuts like *process.lat* will work where these are unambiguous

- **Many variables will be accessible as process attributes *process.name***
  - this restricts to unique field names in the above dictionaries
- **There may be other dictionaries that do have name conflicts**
  - e.g. dictionary of tendencies, with same keys as *process.state*
  - These will *not* be accessible as *process.name*
  - but *will* be accessible as *process.dict_name.name*

  (as well as regular dict interface)

- There will be a dictionary of named subprocesses *process.subprocess*
- Each item in subprocess dict will itself be a *climlab.Process* object
- For convenience with interactive work, each subprocess should be accessible

as *process.subprocess.name* as well as *process.subprocess['name']* - *process.compute()* is a method that computes tendencies (d/dt)

- returns a dictionary of tendencies for all state variables
- keys for this dictionary are same as keys of state dictionary
- tendency dictionary is the total tendency including all subprocesses
- method only computes d/dt, does not apply changes
- **thus method is relatively independent of numerical scheme**
  - may need to make exception for implicit scheme?
- **method *will* update variables in *process.diagnostic***
  - will also *gather all diagnostics* from *subprocesses*
- ***process.step_forward()* updates the state variables**
  - calls *process.compute()* to get current tendencies
  - implements a particular time-stepping scheme
  - user interface is agnostic about numerical scheme
- ***process.integrate_years()* etc will automate time-stepping**
  - also computation of time-average diagnostics.
- Every *subprocess* should work independently of its parent *process* given

appropriate *input*.

- investigating an individual *process* (possibly with its own

***subprocesses*) isolated from its parent needs to be as simple as doing:**

- *newproc = climlab.process_like(procname.subprocess['subprocname'])*
- *newproc.compute()*
- anything in the *input* dictionary of *subprocname* will remain fixed

To do: - use OrderedDict to hold the subprocess dictionary

- order of execution can then be controled by position in dictionary

4.1 Section

text

CHAPTER

FIVE

DOCUMENTATION FOR THE CODE

5.1 Subpackages

5.1.1 climlab.convection package

Submodules

climlab.convection.convadj module

climlab.convection.convadj.**Akamaev_adjustment**(*theta*, *q*, *beta*, *n_k*, *theta_k*, *s_k*, *t_k*)
Single column only.

class climlab.convection.convadj.**ConvectiveAdjustment**(*adj_lapse_rate=None*, ***kwargs*)
Bases: *climlab.process.time_dependent_process.TimeDependentProcess*

Convective adjustment process Instantly returns column to neutral lapse rate

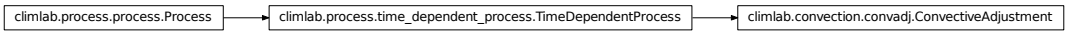
Adjustment includes the surface IF 'Ts' is included in the state dictionary. Otherwise only the atmopsheric temperature is adjusted.

climlab.convection.convadj.**convective_adjustment_direct**(*p*, *T*, *c*, *lapserate=6.5*)
Convective Adjustment to a specified lapse rate.

Input argument lapserate gives the lapse rate expressed in degrees K per km (positive means temperature increasing downward).

Default lapse rate is 6.5 K / km.

Returns the adjusted Column temperature. inputs: p is pressure in hPa T is temperature in K c is heat capacity in in J / m**2 / K



Module contents

5.1.2 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.

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:

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

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

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

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.

more details about domains come here

Initialization parameters

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

Parameters **axes** (dict or class:*climlab.domain.axis.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 class:*~climlab.domain.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 class:*~climlab.domain.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 class:*climlab.domain.axis.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 class:*climlab.domain.axis.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** (class:*climlab.domain.axis.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

```
from climlab import domain

sfc, atm = domain.single_column()
# or
sfc, atm = domain.single_column(num_lev=2, water_depth=10.)
print sfc, atm
```

`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`** (class:*climlab.domain.axis.Axis* or latitude array) – specification for latitude axis (optional)
- **`lev`** (class:*climlab.domain.axis.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`** (class:*climlab.domain.axis.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

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

A = np.linspace(0., 10., 30)
sfc, atm = domain.single_column()
s = field.Field(A, domain=atm)

print s
print s.domain

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

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

print s2
print s2.domain
```

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

Module contents

5.1.3 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

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

Variables

- **K** (*float*) – the diffusivity parameter, for units see below.
- **diffusion_axis** (axis) – axis on which the diffusion is occuring

- **use_banded_solver** (*boolean*) – input flag, whether to use `scipy.linalg.solve_banded()` instead of `numpy.linalg.solve()`

The diffusivity K should be given in units of $\frac{[\text{length}]^2}{\text{time}}$ where length is the unit of the spatial axis on which the diffusion is occuring.

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

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

```
import climlab
from climlab.dynamics.diffusion import Diffusion

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

print c.state
print d.state
c.step_forward()
print c.state
print d.state
```

_implicit_solver()

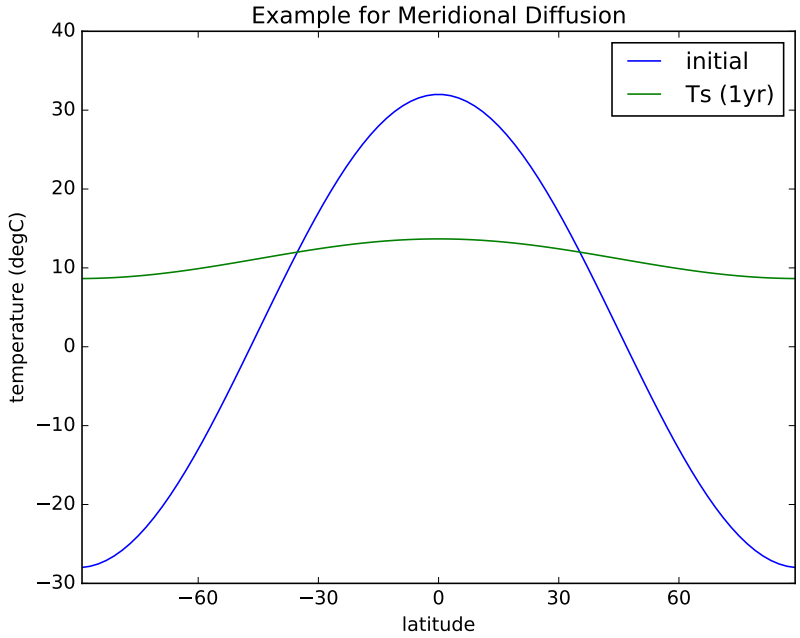
class climlab.dynamics.diffusion.**MeridionalDiffusion**(*K=None*, ***kwargs*)
Bases: *climlab.dynamics.diffusion.Diffusion*

Meridional diffusion process. K in units of 1 / s.

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*)

Input: a process, domain or dictionary of domains. If there is only one axis with length > 1 in the process or set of domains, return the name of that axis. Otherwise raise an error.

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

Builds the the diffusion matrix.

Function-all argument

Parameters

- **K** (*array*) – list of names of diagnostic variables
- **weight1** (*array*) –
- **weight2** (*array*) –

Object attributes

During method execution following object attribute is modified:

Variables **_diag_vars** (*list*) – extended by the list `diaglist` given as method argument

K is array of dimensionless diffusivities at cell boundaries: physical K (length**2 / time) / (delta length)**2 * (delta time)

$$M = \begin{bmatrix} 1+K^* & -K^* & 0 & 0 & ... & 0 \\ -K^* & 1+2K^* & -K^* & 0 & ... & 0 \\ 0 & -K^* & 1+2K^* & -K^* & ... & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & 0 & ... & -K^* & 1+2K^* & -K^* \\ 0 & 0 & ... & 0 & -K^* & 1+K^* \end{bmatrix}$$

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

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

Module contents

5.1.4 climlab.model package

Submodules

climlab.model.column module

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

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

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

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

Example

```
import climlab

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

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

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

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

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

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

class climlab.model.column.**BandRCModel**(***kwargs*)
Bases: *climlab.model.column.RadiativeConvectiveModel*

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

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

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

climlab.model.column.**compute_layer_absorptivity**(*abs_coeff*, *dp*)
Compute layer absorptivity from a constant absorption coefficient.

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*, *T_init_0=12.0*, *T_init_P2=-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*
  - default value: `0.3`
- **a2** (*float*) – parabolic value for planetary albedo parameterization *StepFunctionAlbedo*
  - default value: `0.078`
- **ai** (*float*) – value for ice albedo paramerization in *StepFunctionAlbedo*
  - default value: `0.62`
- **timestep** (*float*) – specifies the EBM's timestep
- **T_init_0** (*float*) – base value for initial temperature
  - unit °C
  - default value: `12`
- **T_init_P2** (*float*) – 2nd Legendre polynomial value for initial temperature
  - 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*.
- **_diag_vars** (*list*) – is updated with `['OLR','ASR','net_radiation','icelat']` through *add_diagnostics()*.

diffusive_heat_transport()

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

lating:

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

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 $\mathrm{W/m^2}$ 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 ehich 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 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).

climlab.model.stommelbox module

Module contents

5.1.5 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

Calculating the model state adjustments through solving the matrix problem already includes the multiplication with the timestep. The 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.

climlab.process.process module

Principles of the new *climlab* API design:

- *climlab.Process* object has several iterable dictionaries of named, gridded variables:
  - *process.state*
    - state variables, usually time-dependent
  - ***process.input***
    - boundary conditions and other gridded quantities independent of the

    process - often set by a parent *process*
  - *process.param* (which are basically just scalar *input*)
  - ***process.tendencies***
    - iterable *dict* of time-tendencies (d/dt) for each state variable
  - ***process.diagnostics***
    - any quantity derived from current state
- The *process* is fully described by contents of *state*, *input* and *param*

dictionaries. *tendencies* and *diagnostics* are always computable from current state. - *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 accessible as attributes

of each variable

- e.g. Tatm.lat
- Shortcuts like *process.lat* will work where these are unambiguous

- **Many variables will be accessible as process attributes *process.name***
  - this restricts to unique field names in the above dictionaries
- **There may be other dictionaries that do have name conflicts**
  - e.g. dictionary of tendencies, with same keys as *process.state*
  - These will *not* be accessible as *process.name*
  - but *will* be accessible as *process.dict_name.name*

  (as well as regular dict interface)
- There will be a dictionary of named subprocesses *process.subprocess*
- Each item in subprocess dict will itself be a *climlab.Process* object
- For convenience with interactive work, each subprocess should be accessible

as *process.subprocess.name* as well as *process.subprocess['name']* - *process.compute()* is a method that computes tendencies (d/dt)

- returns a dictionary of tendencies for all state variables
- keys for this dictionary are same as keys of state dictionary
- tendency dictionary is the total tendency including all subprocesses
- method only computes d/dt, does not apply changes
- **thus method is relatively independent of numerical scheme**
  - may need to make exception for implicit scheme?
- **method *will* update variables in *process.diagnostic***
  - will also *gather all diagnostics* from *subprocesses*
- ***process.step_forward()* updates the state variables**
  - calls *process.compute()* to get current tendencies
  - implements a particular time-stepping scheme
  - user interface is agnostic about numerical scheme
- ***process.integrate_years()* etc will automate time-stepping**
  - also computation of time-average diagnostics.
- Every *subprocess* should work independently of its parent *process* given

appropriate *input*.

- investigating an individual *process* (possibly with its own

***subprocesses*) isolated from its parent needs to be as simple as doing:**

- *newproc = climlab.process_like(procname.subprocess['subprocname'])*
- *newproc.compute()*
- anything in the *input* dictionary of *subprocname* will remain fixed

To do: - use OrderedDict to hold the subprocess dictionary

- order of execution can then be controled by position in dictionary

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.

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** –
- **lev** –
- **num_lat** –
- **num_levels** –
- **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
- **_diag_vars** (*frozenset*) – basically a list of names of 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_diagnostics(*diaglist*)

Updates the process's list of diagnostics.

Function-call argument

Parameters **diaglist** (*list*) – list of names of diagnostic variables

Object attributes

During method execution following object attribute is modified:

Variables **_diag_vars** (*frozenset*) – extended by the list diaglist given as method argument

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

depth_bounds

diagnostics
dictionary with all diagnostic variables

Getter Returns the content of `self._diag_vars`.

Type dict

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

lat_bounds

lev

lev_bounds

lon

lon_bounds

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

exc *ValueError* if state variable `value` is not

having a domain.

Raises

exc *ValueError* if shape mismatch between existing

domain and new state variable.

Example Resetting the surface temperature of an EBM to $-5°C$ on all latitues:

```
import climlab
from climlab import Field

model = climlab.EBM()
sfc = climlab.domain.zonal_mean_surface(num_lat=90, water_depth=10.)
lat = sfc.axes['lat'].points
initial = -5 * ones(size(lat))
model.set_state('Ts', Field(initial, domain=sfc))
```

`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 they 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 `func()` method has been called.

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** (*boolean*) – 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** – integration time for the model in days
- **verbose** (*boolean*) – 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** (*boolean*) – 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

5.1.6 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 Radation (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 during initialization:

Variables

- **A** (*float*) – calls the setter function of *A()*
- **B** (*float*) – calls the setter function of *B()*
- **_diag_vars** (*frozenset*) – extended by string `'OLR'`

Warning: This module currently works only for a single state variable!

Example Simple linear radiation module (stand alone):

```
import climlab
sfc, atm = climlab.domain.single_column()  # creates a column atmosphere and scalar surf

# Create a state variable
```

```
Ts = climlab.Field(15., domain=sfc)
# Make a dictionary of state variables
s = {'Ts': Ts}
olr = climlab.radiation.AplusBT(state=s)
print olr

# OR, we can pass a single state variable
olr = climlab.radiation.AplusBT(state=Ts)
print olr

# to compute tendencies and diagnostics
olr.compute()

#  or to actually update the temperature
olr.step_forward()
print olr.state
```

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()*
- **_diag_vars** (*frozenset*) – extended by string `'OLR'`

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

emission()

Calculates the Outgoing Longwave Radiation (OLR) of the AplusBT_CO2 subprocess.

Object attributes

During method execution following object attribute is modified:

Variables

- **OLR** (*float*) – the described formula is calculated and the result stored in the project attribute `self.OLR`
- ***diagnostics*** (*dict*) – the same result is written in `diagnostics` dictionary with the key `'OLR'`

Warning: This method currently works only for a single state variable!

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 ($\mathrm{W/m^2}$) 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()*
- **_diag_vars** (*frozenset*) – extended by string `'OLR'`

emission()

Calculates the Outgoing Longwave Radiation (OLR) of the Boltzmann radiation subprocess.

Object attributes

During method execution following object attribute is modified:

Variables

- **OLR** (*float*) – the described formula is calculated and the result stored in the project attribute `self.OLR`
- ***diagnostics*** (*dict*) – the same result is written in `diagnostics` dictionary with the key `'OLR'`

> **Warning:** This currently works only for a single state variable!

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.climtrad module

climlab.radiation.cloud module

Created on Tue Mar 10 09:35:48 2015

@author: Brian

climlab.radiation.cloud.**Reflection**(*tauN*, *coszen*)

climlab.radiation.cloud.**compute_beta**(*tauN*, *coszen*)

climlab.radiation.cloud.**compute_eps**(*W*)

climlab.radiation.cloud.**compute_tauN**(*W*)

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 annual mean solar insolation.

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.

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 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`

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 for daily solar insolation.

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: $\frac{\text{W}}{\text{m}^2}$
  - 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.

Defines a constant solar distribution for the whole domain.

Initialization parameters

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

- unit: $\frac{\text{W}}{\text{m}^2}$
- default value: `const.S0/4 = 341.2`

Example

```
import climlab

model = climlab.EBM()
sfc = model.Ts.domain
fixed_ins = climlab.radiation.insolation.FixedInsolation(S0=340.0, domains=sfc)
```

class climlab.radiation.insolation.**P2Insolation**(*S0=1365.2*, *s2=-0.48*, ***kwargs*)
Bases: *climlab.radiation.insolation._Insolation*

A class for parabolic solar distribution on 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.

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*** (*array*) – the array is initialized with zeros of the size of self.domains['sfc'] or self.domains['default'].

- ***S0*** (*float*) – initialized with given argument `S0`
- **`_diag_vars`** (*frozenset*) – extended by string `'insolation'`

`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.nband module

class `climlab.radiation.nband.`**`FourBandLW`**(***kwargs*)
Bases: *climlab.radiation.nband.NbandRadiation*

Closely following SPEEDY / MITgcm longwave model band 0 is window region (between 8.5 and 11 microns) band 1 is CO2 channel (the band of strong absorption by CO2 around 15 microns) band 2 is weak H2O channel (aggregation of spectral regions with weak to moderate absorption by H2O) band 3 is strong H2O channel (aggregation of regions with strong absorption by H2O)

class `climlab.radiation.nband.`**`FourBandSW`**(*emissivity_sfc=0.0*, ***kwargs*)
Bases: *climlab.radiation.nband.NbandRadiation*

A four-band mdoel for shortwave radiation.

The spectral decomposition used here is largely based on the "Moist Radiative-Convective Model" by Aarnout van Delden, Utrecht University a.j.vandelden@uu.nl http://www.staff.science.uu.nl/~delde102/RCM.htm

Four SW channels: channel 0 is Hartley and Huggins band (UV, 6%, <340 nm) channel 1 is part of visible with no O3 absorption (14%, 340 - 500 nm) channel 2 is Chappuis band (27%, 500 - 700 nm) channel 3 is near-infrared (53%, > 700 nm)

`emissivity`

class `climlab.radiation.nband.`**`NbandRadiation`**(*absorber_vmr=None*, ***kwargs*)
Bases: `climlab.radiation.greygas.GreyGas`

Process for radiative transfer. Solves the discretized Schwarschild two-stream equations with the spectrum divided into N spectral bands.

Every NbandRadiation object has an attribute `self.band_fraction` with sum(self.band_fraction) == 1 that gives the fraction of the total beam in each band

Also a dictionary `self.absorber_vmr` that gives the volumetric mixing ratio of every absorbing gas on the same grid as temperature

and a dictionary `self.absorption_cross_section` that gives the absorption cross-section per unit mass for each gas in every spectral band

band_fraction

climlab.radiation.nband.**SPEEDY_band_fraction**(*T*)

Python / numpy implementation of the formula used by SPEEDY and MITgcm to partition longwave emissions into 4 spectral bands.

Input: temperature in Kelvin

returns: a four-element array of band fraction

Reproducing here the FORTRAN code from MITgcm/pkg/aim_v23/phy_radiat.F

```
      EPS3=0.95 _d 0

      DO JTEMP=200,320
        FBAND(JTEMP,0)= EPSLW
        FBAND(JTEMP,2)= 0.148 _d 0 - 3.0 _d -6 *(JTEMP-247)**2
        FBAND(JTEMP,3)=(0.375 _d 0 - 5.5 _d -6 *(JTEMP-282)**2)*EPS3
        FBAND(JTEMP,4)= 0.314 _d 0 + 1.0 _d -5 *(JTEMP-315)**2
        FBAND(JTEMP,1)= 1. _d 0 -(FBAND(JTEMP,0)+FBAND(JTEMP,2)
     &                           +FBAND(JTEMP,3)+FBAND(JTEMP,4))
      ENDDO

      DO JB=0,NBAND
        DO JTEMP=lwTemp1,199
          FBAND(JTEMP,JB)=FBAND(200,JB)
        ENDDO
        DO JTEMP=321,lwTemp2
          FBAND(JTEMP,JB)=FBAND(320,JB)
        ENDDO
      ENDDO
```

class climlab.radiation.nband.**ThreeBandSW**(*emissivity_sfc=0.0*, ***kwargs*)

Bases: *climlab.radiation.nband.NbandRadiation*

A three-band mdoel for shortwave radiation.

The spectral decomposition used here is largely based on the "Moist Radiative-Convective Model" by Aarnout van Delden, Utrecht University a.j.vandelden@uu.nl http://www.staff.science.uu.nl/~delde102/RCM.htm

Three SW channels: channel 0 is Hartley and Huggins band (UV, 1%, 200 - 340 nm) channel 1 is Chappuis band (27%, 450 - 800 nm) channel 2 is remaining radiation (72%)

emissivity

climlab.radiation.radiation module

Radiation is the base class for all climlab radiation 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 - plus a bunch of others!!

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*

climlab.radiation.transmissivity module

class climlab.radiation.transmissivity.**Transmissivity**(*absorptivity*, *reflectivity=None*, *axis=0*)

Bases: object

Class for calculating and store transmissivity between levels, and computing radiative fluxes between levels.

Input: numpy array of absorptivities. It is assumed that the last dimension is vertical levels.

Attributes: (all stored as numpy arrays):

- •N: number of levels
- •absorptivity: level absorptivity (N)
- •transmissivity: level transmissivity (N)
- •Tup: transmissivity matrix for upwelling beam (N+1, N+1)
- •Tdown: transmissivity matrix for downwelling beam (N+1, N+1)

Example for N = 3 atmospheric layers:

tau is a vector of transmissivities

$$\tau = [1, \tau_0, \tau_1, \tau_2]$$

A is a matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \tau_0 & 1 & 1 & 1 \\ \tau_0 & \tau_1 & 1 & 1 \\ \tau_0 & \tau_1 & \tau_2 & 1 \end{bmatrix}$$

We then take the cumulative product along columns, and finally take the lower triangle of the result to get

$$Tup = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \tau_0 & 1 & 0 & 0 \\ \tau_1\tau_0 & \tau_1 & 1 & 0 \\ \tau_2\tau_1\tau_0 0 & \tau_2\tau_1 & \tau_2 & 1 \end{bmatrix}$$

and Tdown = transpose(Tup)

Construct an emission vector for the downwelling beam:

Edown = [E0, E1, E2, fromspace]

Now we can get the downwelling beam by matrix multiplication:

D = Tdown * Edown

For the upwelling beam, we start by adding the reflected part at the surface to the surface emissions:

Eup = [emit_sfc + albedo_sfc*D[0], E0, E1, E2]

So that the upwelling flux is

U = Tup * Eup

The total flux, positive up is thus

F = U - D

The absorbed radiation at the surface is then -F[0] The absorbed radiation in the atmosphere is the flux convergence:

-diff(F)

flux_down (*fluxDownTop*, *emission=None*)
Compute upwelling radiative flux at interfaces between layers.

Inputs:

•fluxUpBottom: flux up from bottom

•emission: emission from atmospheric levels (N) defaults to zero if not given

Returns:

- vector of upwelling radiative flux between levels (N+1) element N is the flux up to space.

flux_reflected_up (*fluxDown*, *albedo_sfc=0.0*)

flux_up (*fluxUpBottom*, *emission=None*)
Compute downwelling radiative flux at interfaces between layers.

Inputs:

•fluxDownTop: flux down at top

•emission: emission from atmospheric levels (N) defaults to zero if not given

Returns:

•vector of downwelling radiative flux between levels (N+1) element 0 is the flux down to the surface.

climlab.radiation.transmissivity.**compute_T_vectorized**(*transmissivity*)

climlab.radiation.transmissivity.**tril**(*array*, *k=0*)
Lower triangle of an array. Return a copy of an array with elements above the k-th diagonal zeroed. Need a multi-dimensional version here because numpy.tril does not broadcast for numpy verison < 1.9.

climlab.radiation.water_vapor module

class climlab.radiation.water_vapor.**FixedRelativeHumidity**(*relative_humidity=0.77*, *qStrat=5e-06*, ***kwargs*)
Bases: *climlab.process.diagnostic.DiagnosticProcess*

Compute water vapor mixing ratio profile Assuming constant relative humidity.

relative_humidity is the specified RH. Same value is applied everywhere. qStrat is the minimum specific humidity, ensuring that there is some water vapor in the stratosphere.

The attribute RH_profile can be modified to set different vertical profiles of relative humidity (see daughter class ManabeWaterVapor()).

class climlab.radiation.water_vapor.**ManabeWaterVapor**(**kwargs)
Bases: *climlab.radiation.water_vapor.FixedRelativeHumidity*

Compute water vapor mixing ratio profile following Manabe and Wetherald JAS 1967 Fixed surface relative humidity and a specified fractional profile.

relative_humidity is the specified surface RH qStrat is the minimum specific humidity, ensuring that there is some water vapor in the stratosphere.

Module contents

5.1.7 climlab.solar package

Submodules

climlab.solar.insolation module

insolation.py

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

Currently, only daily average insolation is computed.

Ported and modified from MATLAB code daily_insolation.m Original authors:

Ian Eisenman and Peter Huybers, Harvard University, August 2006

Available online at http://eisenman.ucsd.edu/code/daily_insolation.m

If using calendar days, solar longitude is found using an approximate solution to the differential equation representing conservation of angular momentum (Kepler's Second Law). Given the orbital parameters and solar longitude, daily average insolation is calculated exactly following Berger 1978.

References: Berger A. and Loutre M.F. (1991). Insolation values for the climate of

the last 10 million years. Quaternary Science Reviews, 10(4), 297-317.

Berger A. (1978). Long-term variations of daily insolation and Quaternary climatic changes. Journal of Atmospheric Science, 35(12), 2362-2367.

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 ecc,long_peri,obliquity = orbital.lookup_parameters(kyears)

Inputs: lat: Latitude in degrees (-90 to 90). day: Indicator of time of year, by default day 1 is Jan 1. orb: a dictionary with three members (as provided by orbital.py)

ecc: eccentricity (dimensionless) long_peri: longitude of perihelion (precession angle) (degrees) obliquity: obliquity angle (degrees)

S0: Solar constant in W/m^2, will try to read from constants.py day_type: 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.

Default values for orbital parameters are present-day

Output: Fsw = Daily average solar radiation in W/m^2.

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

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 orbital.OrbitalTable.lookup_parameters()

e.g. to compute the timeseries of insolation at 65N at summer solstice over the past 5 Myears from climlab.orbital import OrbitalTable table = OrbitalTable() years = np.linspace(0, 5000, 5001) orb = table.lookup_parameters(years) S65 = orbital.daily_insolation(65, 172, orb)

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

Estimate solar longitude (lambda = 0 at spring equinox) from calendar day using an approximation from Berger 1978 section 3.

Works for both scalar and vector orbital parameters.

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

climlab.solar.orbital module

orbital.py

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 Berger and Loutre (1991).

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 Laskar et al. (2004) http://www.imcce.fr/Equipes/ASD/insola/earth/La2004/README.TXT

References: Berger A. and Loutre M.F. (1991). Insolation values for the climate of

the last 10 million years. Quaternary Science Reviews, 10(4), 297-317.

Berger A. (1978). Long-term variations of daily insolation and Quaternary climatic changes. Journal of Atmospheric Science, 35(12), 2362-2367.

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

Bases: *climlab.solar.orbital.OrbitalTable*

Invoking LongOrbitalTable() will load orbital parameter tables for -51 to +21 Myears as calculated by Laskar et al. 2004 http://www.imcce.fr/Equipes/ASD/insola/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 Berger and Loutre 1991) and compute linear interpolants. The data can be accessed through the method OrbitalTable.lookup_parameters(kyear).

lookup_parameters (*kyear=0*)

Look up orbital parameters for given kyear measured from present. Input kyear is thousands of years after present. For years before present, use kyear < 0.

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.

climlab.solar.orbital_cycles module

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

example usage:

from climlab.model.ebm import EBM_seasonal from climlab.solar.orbital_cycles import OrbitalCycles from climlab.surface.albedo import StepFunctionAlbedo ebm = EBM_seasonal() print ebm # add an albedo feedback albedo = StepFunctionAlbedo(state=ebm.state, **ebm.param) ebm.add_subprocess('albedo', albedo) # start the integration # run for 10,000 orbital years, but only 1,000 model years experiment = OrbitalCycles(ebm, kyear_start=-20,

kyear_stop=-10, orbital_year_factor=10.)

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 integrate a process through changes in orbital parameters.

model is an instance of climlab.time_dependent_process

segment_length_years is the length of each integration with fixed orbital parameters. orbital_year_factor is an optional speed-up to the orbital cycles.

Module contents

5.1.8 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.

Defines constant albedo values for the whole 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.
- **_diag_vars** (*frozenset*) – extended by the list `['albedo',]`

Uniform prescribed albedo.

albedo

Property of albedo value.

Getter Returns the albedo value which is stored in attribute `self._albedo`

Setter

- sets albedo which is addressed as `self._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'`.
- **_diag_vars** (*frozenset*) – extended by the list `['noice','ice','icelat']`

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 $T_s \geq T_f$

- **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

class `climlab.surface.albedo.`**P2Albedo**(*a0=0.33*, *a2=0.25*, ***kwargs*)
Bases: *climlab.process.diagnostic.DiagnosticProcess*

A class for parabolic defined albedo values on basis of the second order Legendre Polynomial.

Defines parabolic shaped albedo values across the domain range.

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.
- **_diag_vars** (*frozenset*) – extended by the list `['albedo',]`

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
- **_diag_vars** (*frozenset*) – extended by the list `['albedo',]`

climlab.surface.surface_radiation module

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

Bases: *climlab.process.energy_budget.EnergyBudget*

climlab.surface.turbulent module

class climlab.surface.turbulent.**LatentHeatFlux**(*Cd=0.003*, ***kwargs*)
Bases: *climlab.surface.turbulent.SurfaceFlux*

class climlab.surface.turbulent.**SensibleHeatFlux**(*Cd=0.003*, ***kwargs*)
Bases: *climlab.surface.turbulent.SurfaceFlux*

class climlab.surface.turbulent.**SurfaceFlux**(*Cd=0.003*, ***kwargs*)
Bases: *climlab.process.energy_budget.EnergyBudget*

Module contents

5.1.9 climlab.tests package

Module contents

5.1.10 climlab.utils package

Submodules

climlab.utils.constants module

constants.py

A collection of physical constants for the atmosphere and ocean.

Part of the climlab package Brian Rose, University at Albany brose@albany.edu

climlab.utils.heat_capacity module

Routines for calculating heat capacities for grid boxes in units of J / m**2 / K

climlab.utils.heat_capacity.**atmosphere**(*dp*)
Heat capacity of a unit area of atmosphere, in units of J / m**2 / K Input is pressure intervals in units of mb.

climlab.utils.heat_capacity.**ocean**(*dz*)
Heat capacity of a unit area of water, in units of J / m**2 / K Input dz is water depth intervals in meters

climlab.utils.heat_capacity.**slab_ocean**(*water_depth*)
Heat capacity of a unit area slab of water, in units of J / m**2 / K Input is depth of water in meters.

climlab.utils.legendre module

The first several Legendre polynomials, along with (some of) their first derivatives.

climlab.utils.legendre.**P0**(*x*)

climlab.utils.legendre.**P1**(*x*)

climlab.utils.legendre.**P10**(*x*)

climlab.utils.legendre.**P10prime**(*x*)

climlab.utils.legendre.**P12**(*x*)

climlab.utils.legendre.**P12prime**(*x*)

climlab.utils.legendre.**P14**(*x*)

climlab.utils.legendre.**P14prime**(*x*)

climlab.utils.legendre.**P16**(*x*)

climlab.utils.legendre.**P18**(*x*)

climlab.utils.legendre.**P1prime**(*x*)

climlab.utils.legendre.**P2**(*x*)
The second Legendre polynomial.

climlab.utils.legendre.**P20**(*x*)

climlab.utils.legendre.**P22**(*x*)

climlab.utils.legendre.**P24**(*x*)

climlab.utils.legendre.**P26**(*x*)

climlab.utils.legendre.**P28**(*x*)

climlab.utils.legendre.**P2prime**(*x*)

climlab.utils.legendre.**P3**(*x*)

climlab.utils.legendre.**P3prime**(*x*)

climlab.utils.legendre.**P4**(*x*)

climlab.utils.legendre.**P4prime**(*x*)

climlab.utils.legendre.**P5**(*x*)

climlab.utils.legendre.**P6**(*x*)

climlab.utils.legendre.**P6prime**(*x*)

climlab.utils.legendre.**P8**(*x*)

climlab.utils.legendre.**P8prime**(*x*)

climlab.utils.legendre.**Pn**(*x*)
Calculate lots of Legendre polyomials and return them in a dictionary.

climlab.utils.legendre.**Pnprime**(*x*)
First derivatives of Legendre polynomials.

climlab.utils.thermo module

thermo.py

A collection of function definitions to handle common thermodynamic calculations for the atmosphere.

Code developed by Brian Rose, University at Albany brose@albany.edu in support of the class ATM/ENV 415: Climate Laboratory

climlab.utils.thermo.**EIS**(*T0*, *T700*)
Convenience method, identical to thermo.estimated_inversion_strength(T0,T700)

climlab.utils.thermo.**Planck_frequency**(*nu*, *T*)
The Planck function B(nu,T): the flux density for blackbody radiation in frequency space nu is frequency in 1/s T is temperature in Kelvin

Formula from Raymond Pierrehumbert, “Principles of Planetary Climate”

climlab.utils.thermo.**Planck_wavelength**(*l*, *T*)
The Planck function (flux density for blackbody radiation) in wavelength space l is wavelength in meters T is temperature in Kelvin

Formula from Raymond Pierrehumbert, "Principles of Planetary Climate"

climlab.utils.thermo.**Planck_wavenumber**(*n*, *T*)
The Planck function (flux density for blackbody radition) in wavenumber space n is wavenumber in 1/cm T is temperature in Kelvin

Formula from Raymond Pierrehumbert, "Principles of Planetary Climate"

climlab.utils.thermo.**T**(*theta*, *p*)
Convenience method, identical to thermo.temperature_from_potential(theta,p).

climlab.utils.thermo.**blackbody_emission**(*T*)
Blackbody radiation following the Stefan-Boltzmann law.

climlab.utils.thermo.**clausius_clapeyron**(*T*)
Compute saturation vapor pressure as function of temperature T.

Input: T is temperature in Kelvin Output: saturation vapor pressure in mb or hPa

Formula from Rogers and Yau "A Short Course in Cloud Physics" (Pergammon Press), p. 16 claimed to be accurate to within 0.1% between -30degC and 35 degC Based on the paper by Bolton (1980, Monthly Weather Review).

climlab.utils.thermo.**estimated_inversion_strength**(*T0*, *T700*)
Compute the "estimated inversion strength", T0 is surface temp, T700 is temp at 700 hPa, both in K. Following Wood and Bretherton, J. Climate 2006.

climlab.utils.thermo.**potential_temperature**(*T*, *p*)
Compute potential temperature for an air parcel.

Input: T is temperature in Kelvin p is pressure in mb or hPa

Output: potential temperature in Kelvin.

climlab.utils.thermo.**pseudoadiabat**(*T*, *p*)
Compute the local slope of the pseudoadiabat at given temperature and pressure

Inputs: p is pressure in hPa or mb T is local temperature in Kelvin

Output: dT/dp, the rate of temperature change for pseudoadiabatic ascent

the pseudoadiabat describes changes in temperature and pressure for an air parcel at saturation assuming instantaneous rain-out of the super-saturated water

Formula from Raymond Pierrehumbert, "Principles of Planetary Climate"

climlab.utils.thermo.**qsat**(*T*, *p*)
Compute saturation specific humidity as function of temperature and pressure.

Input: T is temperature in Kelvin p is pressure in hPa or mb

Output: saturation specific humidity (dimensionless).

climlab.utils.thermo.**temperature_from_potential**(*theta*, *p*)
Convert potential temperature to in-situ temperature.

Input: theta is potential temperature in Kelvin p is pressure in mb or hPa

Output: absolute temperature in Kelvin.

climlab.utils.thermo.**theta**(*T*, *p*)
Convenience method, identical to thermo.potential_temperature(T,p).

climlab.utils.walk module

climlab.utils.walk.**process_tree**(*top*, *name='top'*)
Create a string representation of the process tree for process top.

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

```
import climlab

model = climlab.EBM()
processes = []
for name, proc, top_proc in walk_processes(model):
    processes.append(proc)
```

Module contents

5.2 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

SIX

REFERENCES

CHAPTER

SEVEN

LICENSE

climlab licensed under MIT License

CHAPTER

EIGHT

CONTACT

climlab developed by Brian Rose documentation by Moritz Kreuzer

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

[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.

C

Symbols

A

B

C

D

E

F

G

H

I

T

W

Z