climlab-0.3 Documentation

Release 0.3.2

Moritz Kreuzer

May 17, 2016

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* (page 11) 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 the requires the 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. Either they can be individially downloaded, compiled and installed according to the the instructions on the corresponding websites.

Easier to handle however are Python distributions like Anaconda or Enthought Canopy 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` (page 104) (or one of it's daughter classes to be specific).

Another example: the emitted energy of a surface can be computed through the `Boltzmann` (page 97) class which is also a climlab *Process* and implements the Stefan Boltzmann Law for a grey body. Like that, all events and procedures that *climlab* can model are organized in *Processes*.

Note: The implementation of a whole model, for example an Energy Balance Model (`EBM` (page 74)), is also an instance of the `Process` (page 84) class in *climlab*.

For more information about models, see the climlab *Models* (page 11) 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. More details about subprocesses can be found 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` (page 84). Most processes are timedependent and therefore an instance of the daughter class `TimeDependentProcess` (page 90).

3.1.1 Basic Dictionaries

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

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

[1] In the following the small written *process* refers to an instance of the `Process` (page 84) class.

- **`process.tendencies`** is an iterable dictionary of time tendencies (d/dt) for each state variable defined in `process.state`.

  Note: A non TimeDependentProcess (but instance of *`Process`* (page 84)) does not have this dictionary.

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

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

3.1.2 Subprocesses

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

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

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

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

A `subprocess` is similar to its `parent process` an instance of the *`Process`* (page 84) class. That means a `subprocess` has dictionaries and attributes with the same names as its `parent process`. Not necessary all will be the same or have the same entries, but a `subprocess` has at least the basic dictionaries and attributes created during initialization of the *`Process`* (page 84) instance.

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

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

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

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

3.1.3 Process Integration over time

A *`TimeDependentProcess`* (page 90) 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 the state variable S is discretized over time like

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

the state tendency can be calculated through

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

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

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

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

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, ...

$$\begin{aligned} \frac{dE}{dt} = C\frac{dT}{dt} &= \underbrace{SP_A(T)}_{\text{heating-rate of } SP_A} + \underbrace{SP_B(T)}_{\text{heating-rate of } SP_B} + ... \\ \Leftrightarrow \frac{dT}{dt} &= \underbrace{\frac{SP_A(T)}{C}}_{T \text{ tendency by } SP_A} + \underbrace{\frac{SP_B(T)}{C}}_{T \text{ tendency by } SP_B} + ... \end{aligned}$$

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* (page 8) below.

- **`compute()` (page 91) is a method that computes tendencies d/dt for all state variables**
  - it returns a dictionary of tendencies for all state variables

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

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

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

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

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

    In case subprocess SP_A itself has subprocesses, their T tendencies get included in tendency computation by *compute()* (page 91).

  - the method only computes d/dt but **does not apply changes** (which is done by *step_forward()* (page 93))
  - therefore, the method is relatively independent of the numerical scheme
  - method **will update** variables in `proc.diagnostic` dictionary. Therefore, it will also **gather all diagnostics** from the *subprocesses*

- ***step_forward()* (page 93) updates the state variables**
  - it calls *compute()* (page 91) to get current tendencies
  - the method multiplies state tendencies with the timestep and adds them up to the state variables
- *integrate_years()* (page 92) etc will automate time-stepping by calling the *step_forward* (page 93) method multiple times. It also does the computation of time-average diagnostics.
- *integrate_converge()* (page 91) calls *integrate_years()* (page 92) as long as the state variables keep changing over time.

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

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

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

Classification of Subprocess Types

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

Therefore, the *compute()* (page 91) method handles them seperately as well as in specific order. It calls private `_compute()` methods that are specified in daugther classes of *Process* (page 84) namely *DiagnosticProcess* (page 81), *EnergyBudget* (page 82) (which are explicit processes) or *ImplicitProcess* (page 83).

The description of *compute()* (page 91) reveals the details how the different process types are handeled:

> The function first computes all diagnostic processes. They don't produce any tendencies directly but they may effect the other processes (such as change in solar distribution). Subsequently, all tendencies and diagnostics for all explicit processes are computed.
>
> Tendencies due to implicit and adjustment processes need to be calculated from a state that is already adjusted after explicit alteration. For that reason the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated by matrix inversions and similar to the explicit tendencies, the implicit ones are applied to the states temporarily. Subsequently, all instantaneous adjustments are computed.
>
> Then the changes that were made to the states from explicit and implicit processes are removed again as this *compute()* (page 91) function is supposed to calculate only tendencies and not apply them to the states.
>
> Finally, all calculated tendencies from all processes are collected for each state, summed up and stored in the dictionary `self.tendencies`, which is an attribute of the time-dependent-process object, for which the *compute()* (page 91) method has been called.

3.2 Domain

A *Domain* defines an area or spatial base for a climlab `Process` (page 84) object. It consists of axes which are `Axis` (page 55) objects that define the dimensions of the *Domain*.

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

There are daughter classes `Atmosphere` (page 57) and `Ocean` (page 58) of the private `_Domain` (page 59) class implemented which themselves have daughter classes `SlabAtmosphere` (page 59) and `SlabOcean` (page 59).

Every `Process` (page 84) needs to be defined on a *Domain*. If none is given during initialization but latitude `lat` is specified, a default *Domain* is created.

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

- `single_column()` (page 62)
- `zonal_mean_column()` (page 62)
- `box_model_domain()` (page 60)

3.3 Axis

An `Axis` (page 55) is an object where information of a `_Domain` (page 59)‘s spacial dimension are specified.

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

The *axes* of a `_Domain` (page 59) are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of `_Domain` (page 59).

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 that `proc` is an instance of the `Process` (page 84) class here.

Example

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

# quick access
longwave_subp = model.subprocess.LW

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

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

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

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

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

# quick access
lat_points = model.lat
```

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

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

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

Warning: There may be other dictionaries that do have name conflicts: e.g. dictionary of tendencies `proc.tendencies`, with same keys as `proc.state`.
These will **not be accessible** as `proc.name`, but **will be accessible** as `proc.dict_name.name` (as well as regular dictionary interface `proc.dict_name['name']`).

CHAPTER

FOUR

MODELS

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

4.1 Energy Balance Model

Currently, there are three “standard” Energy Balance Models implemented in the *climlab* code. These are `EBM` (page 74), `EBM_seasonal` (page 79) and `EBM_annual` (page 79), which are explained below.

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

4.1.1 EBM Subprocesses

Insolation

- **`FixedInsolation` (page 103)** defines a constant solar value for all spatial points of the domain:

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

- **`P2Insolation` (page 103)** characterizes a parabolic solar distribution over the domain’s latitude on the basis of the second order Legendre Polynomial P_2:

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

  Variable φ represents the latitude.

- **`DailyInsolation` (page 101)** computes the daily solar insolation for each latitude of the domain on the basis of orbital parameters and astronomical formulas.
- **`AnnualMeanInsolation` (page 99)** computes a latitudewise yearly mean for solar insolation on the basis of orbital parameters and astronomical formulas.

Albedo

- **`ConstantAlbedo` (page 110)** defines constant albedo values at all spatial points of the domain:

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

- **`P2Albedo` (page 112)** initializes parabolic distributed albedo values across the domain on basis of the second order Legendre Polynomial P_2:

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

- **`Iceline` (page 111)** determines which part of the domain is covered with ice according to a given freezing temperature.

- ***StepFunctionAlbedo* (page 113)** implements an albedo step function in dependence of the surface temperature by using instances of the above described albedo classes as subprocesses.

Outgoing Longwave Radiation

- ***AplusBT* (page 93)** calculates the Outgoing Longwave Radiation (OLR) in form of a linear dependence of surface temperature T:

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

- ***AplusBT_CO2* (page 95)** calculates OLR in the same way as *AplusBT* (page 93) but uses parameters A and B dependent of the atmospheric CO_2 concentration c.

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

- ***Boltzmann* (page 97)** calculates OLR according to the Stefan-Boltzmann law for a grey body:

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

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* (page 70)** calculates the energy transport in a diffusion like process along the temperature gradient:

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

- ***BudykoTransport* (page 66)** calculates the energy transport for each latitude φ depending on the global mean temperature $\bar{T}$:

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

4.1.2 EBM templates

The preconfigured Energy Balance Models *EBM* (page 12), *EBM_seasonal* (page 13) and *EBM_annual* (page 13) use the described suprocesses above:

EBM

The *EBM* (page 74) class sets up a typical Energy Balance Model with following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via *AplusBT* (page 93)
- solar insolation paramterization via *P2Insolation* (page 103)
- albedo parametrization in dependence of temperature via *StepFunctionAlbedo* (page 113)
- energy diffusion via *MeridionalDiffusion* (page 70)

EBM_seasonal

The `EBM_seasonal` (page 79) class implements Energy Balance Models with realistic daily insolation. It uses following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via `AplusBT` (page 93)
- solar insolation paramterization via `DailyInsolation` (page 101)
- albedo parametrization in dependence of temperature via `StepFunctionAlbedo` (page 113)
- energy diffusion via `MeridionalDiffusion` (page 70)

EBM_annual

The `EBM_annual` (page 79) class that implements Energy Balance Models with annual mean insolation. It uses following subprocesses:

- Outgoing Longwave Radiation (OLR) parametrization via `AplusBT` (page 93)
- solar insolation paramterization via `AnnualMeanInsolation` (page 99)
- albedo parametrization in dependence of temperature via `StepFunctionAlbedo` (page 113)
- energy diffusion via `MeridionalDiffusion` (page 70)

Note: For information how to set up individual models or modify instances of the classes above, see the *Tutorials* (page 15) chapter.

4.2 Other Models

As noted in the *Introduction* (page 1), more model types are implemented in *climlab* but not covered in the documentation yet.

CHAPTER

FIVE

TUTORIALS

For a learning-by-doing approach, a couple of Tutorials come with the *climlab* package. They can be found in the package's *courseware* folder and are written in the Jupyter Notebook format. The first *climlab* notebooks have been used for teaching some basics of climate science and had also the initial purpose to document the `climlab` package.

Example usage

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

```
jupyter notebook Insolation.ipynb
```

Notebooks

A few notebooks which describe the basic usage of EBMs, can be found here:

5.1 Preconfigured Energy Balance Models

In this document the basic use of climlab's preconfigured EBM class is shown.

Contents are how to

- setup an EBM model
- show and access subprocesses
- integrate the model
- access and plot various model variables
- calculate the global mean of the temperature

```
In [1]: # import header

        %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import climlab
        from climlab import constants as const
        from climlab.domain.field import global_mean
```

5.1.1 Model Creation

The regular path for the EBM class is `climlab.model.ebm.EBM` but it can also be accessed through `climlab.EBM`

An EBM model instance is created through

```
In [2]: # model creation
        ebm_model = climlab.EBM()
```

By default many parameters are set during initialization:

```
num_lat=90, S0=const.S0, A=210., B=2., D=0.55, water_depth=10., Tf=-10,
a0=0.3, a2=0.078, ai=0.62, timestep=const.seconds_per_year/90., T0=12.,
T2=-40
```

For further details see the climlab documentation.

Many of the input parameters are stored in the following dictionary:

```
In [3]: # print model parameters
        ebm_model.param
```

```
Out[3]: {'A': 210.0,
         'B': 2.0,
         'D': 0.555,
         'S0': 1365.2,
         'Tf': -10.0,
         'a0': 0.3,
         'a2': 0.078,
         'ai': 0.62,
         'timestep': 350632.51200000005,
         'water_depth': 10.0}
```

The model consists of one state variable (surface temperature) and a couple of defined subprocesses.

```
In [4]: # print model states and suprocesses
        print ebm_model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

5.1.2 Model subprocesses

The subprocesses are stored in a dictionary and can be accessed through

```
In [5]: # access model subprocesses
        ebm_model.subprocess.keys()
```

```
Out[5]: ['diffusion', 'LW', 'albedo', 'insolation']
```

So to access the time type of the Longwave Radiation subprocess for example, type:

```
In [6]: # access specific subprocess through dictionary
        ebm_model.subprocess['LW'].time_type
```

```
Out[6]: 'explicit'
```

5.1.3 Model integration

The model time dictionary shows information about all the time related content and quantities.

```
In [7]: # accessing the model time dictionary
        ebm_model.time
```

```
Out[7]: {'day_of_year_index': 0,
         'days_elapsed': 0,
         'days_of_year': array([   0.        ,    4.05824667,    8.11649333,   12.17474
                  16.23298667,   20.29123333,   24.34948   ,   28.40772667,
                  32.46597333,   36.52422   ,   40.58246667,   44.64071333,
                  48.69896   ,   52.75720667,   56.81545333,   60.8737   ,
                  64.93194667,   68.99019333,   73.04844   ,   77.10668667,
                  81.16493333,   85.22318   ,   89.28142667,   93.33967333,
                  97.39792   ,  101.45616667,  105.51441333,  109.57266   ,
                 113.63090667,  117.68915333,  121.7474   ,  125.80564667,
                 129.86389333,  133.92214   ,  137.98038667,  142.03863333,
                 146.09688   ,  150.15512667,  154.21337333,  158.27162   ,
                 162.32986667,  166.38811333,  170.44636   ,  174.50460667,
                 178.56285333,  182.6211    ,  186.67934667,  190.73759333,
                 194.79584   ,  198.85408667,  202.91233333,  206.97058   ,
                 211.02882667,  215.08707333,  219.14532   ,  223.20356667,
                 227.26181333,  231.32006   ,  235.37830667,  239.43655333,
                 243.4948    ,  247.55304667,  251.61129333,  255.66954   ,
                 259.72778667,  263.78603333,  267.84428   ,  271.90252667,
                 275.96077333,  280.01902   ,  284.07726667,  288.13551333,
                 292.19376   ,  296.25200667,  300.31025333,  304.3685    ,
                 308.42674667,  312.48499333,  316.54324   ,  320.60148667,
                 324.65973333,  328.71798   ,  332.77622667,  336.83447333,
                 340.89272   ,  344.95096667,  349.00921333,  353.06746   ,
                 357.12570667,  361.18395333]),
         'num_steps_per_year': 90.0,
         'steps': 0,
         'timestep': 350632.51200000005,
         'years_elapsed': 0}
```

To integrate the model forward in time different methods are availible:

```
In [8]: # integrate model for a single timestep
        ebm_model.step_forward()
```

The model time step has increased from 0 to 1:

```
In [9]: ebm_model.time['steps']
```

```
Out[9]: 1
```

```
In [10]: # integrate model for a 50 days
         ebm_model.integrate_days(50.)
```

```
Integrating for 12 steps, 50.0 days, or 0.136895462792 years.
Total elapsed time is 0.144444444444 years.
```

```
In [11]: # integrate model for two years
         ebm_model.integrate_years(1.)
```

```
Integrating for 90 steps, 365.2422 days, or 1.0 years.
Total elapsed time is 1.14444444444 years.
```

```
In [12]: # integrate model until solution converges
         ebm_model.integrate_converge()
```

```
Total elapsed time is 9.14444444444 years.
```

Plotting model variables

A couple of interesting model variables are stored in a dictionary named `diagnostics`. It has following entries:

```
In [13]: ebm_model.diagnostics.keys()
```

```
Out[13]: ['ASR', 'OLR', 'icelat', 'net_radiation', 'albedo', 'insolation']
```

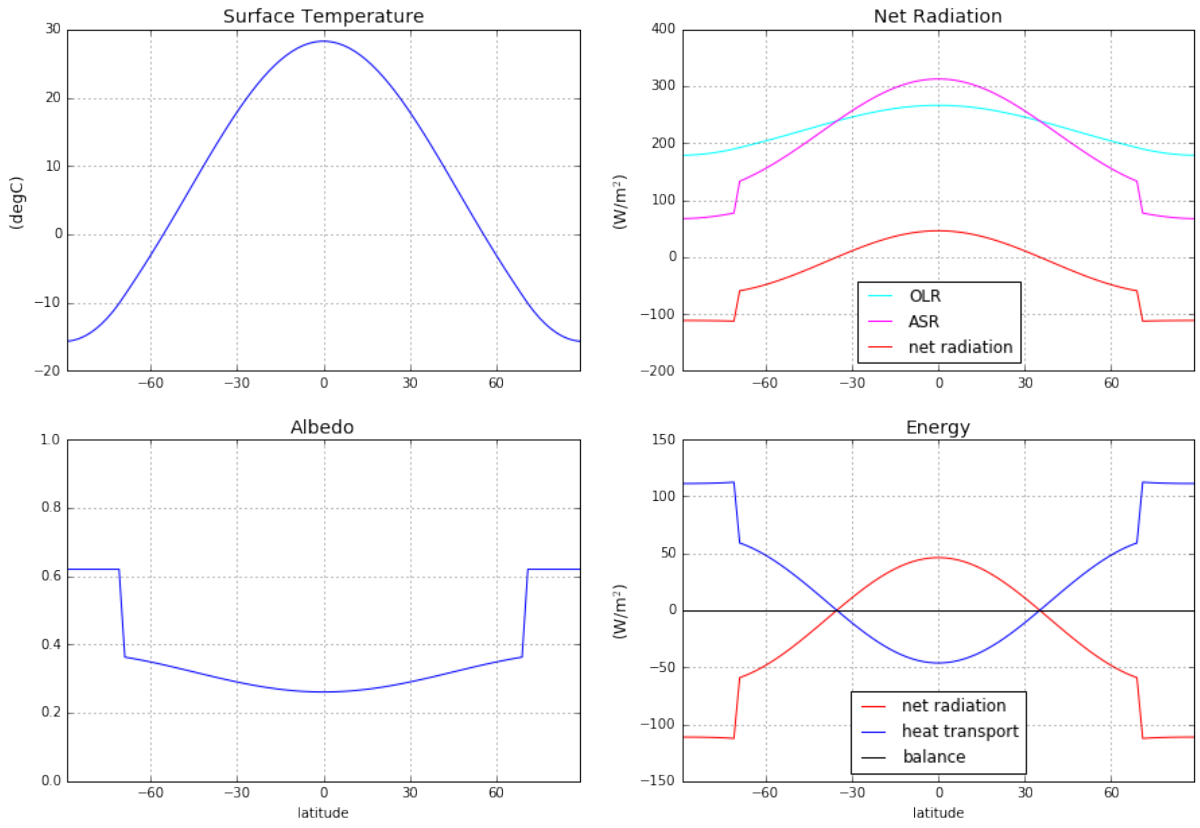
They can be accessed respecively to the keys through `ebm_model.diagnostics['ASR']`. Most of them can be also accessed directly as model attributes like:

```
In [14]: ebm_model.icelat
```

```
Out[14]: array([-70.,  70.])
```

The following code does the plotting for some model variables.

```
In [15]: # creating plot figure
         fig = plt.figure(figsize=(15,10))

         # Temperature plot
         ax1 = fig.add_subplot(221)
         ax1.plot(ebm_model.lat,ebm_model.Ts)

         ax1.set_xticks([-90,-60,-30,0,30,60,90])
         ax1.set_xlim([-90,90])
         ax1.set_title('Surface Temperature', fontsize=14)
         ax1.set_ylabel('(degC)', fontsize=12)
         ax1.grid()

         # Albedo plot
         ax2 = fig.add_subplot(223, sharex = ax1)
         ax2.plot(ebm_model.lat,ebm_model.albedo)

         ax2.set_title('Albedo', fontsize=14)
         ax2.set_xlabel('latitude', fontsize=10)
         ax2.set_ylim([0,1])
         ax2.grid()

         # Net Radiation plot
         ax3 = fig.add_subplot(222, sharex = ax1)
         ax3.plot(ebm_model.lat, ebm_model.OLR, label='OLR',
                                                color='cyan')
         ax3.plot(ebm_model.lat, ebm_model.ASR, label='ASR',
                                                color='magenta')
         ax3.plot(ebm_model.lat, ebm_model.ASR-ebm_model.OLR,
                                                label='net radiation',
                                                color='red')

         ax3.set_title('Net Radiation', fontsize=14)
         ax3.set_ylabel('(W/m$^2$)', fontsize=12)
         ax3.legend(loc='best')
         ax3.grid()

         # Energy Balance plot
         net_rad = np.squeeze(ebm_model.net_radiation)
         transport = ebm_model.heat_transport_convergence()

         ax4 = fig.add_subplot(224, sharex = ax1)
         ax4.plot(ebm_model.lat, net_rad, label='net radiation',
                                                       color='red')
         ax4.plot(ebm_model.lat, transport, label='heat transport',
                                                       color='blue')
         ax4.plot(ebm_model.lat, net_rad+transport, label='balance',
                                                      color='black')

         ax4.set_title('Energy', fontsize=14)
         ax4.set_xlabel('latitude', fontsize=10)
         ax4.set_ylabel('(W/m$^2$)', fontsize=12)
         ax4.legend(loc='best')
         ax4.grid()


         plt.show()
```



The energy balance is zero at every latitude. That means the model is in equilibrium. Perfect!

5.1.4 Global mean temperature

The model's state dictionary has following entries:

```
In [16]: ebm_model.state.keys()
```

```
Out[16]: ['Ts']
```

So the surface temperature can usually be accessed through `ebm_model.state['Ts']` but is also availible as a model attribute: `ebm_model.Ts`

The global mean of the model's surface temperature can be calculated through

```
In [17]: # calculate global mean temperature
         global_mean(ebm_model.Ts)
```

```
Out[17]: Field(14.288135944994657)
```

Note that in the **header** the `global_mean` method has been **imported**!

```
In [18]: print 'The global mean temperature is %s degC.' \
                                             %np.round(global_mean(ebm_model.Ts),2)

          print 'The modeled ice edge is at %s deg.' % np.max(ebm_model.icelat)
```

```
The global mean temperature is 14.29 degC.
The modeled ice edge is at 70.0 deg.
```

5.2 Boltzmann Outgoing Longwave Radiation

In this document an Energy Balance Model (EBM) is set up with the Outgoing Longwave Radiation (OLR) parametrized through the Stefan Boltzmann radiation of a grey body.

$$OLR(\varphi) = \sigma \cdot \varepsilon \cdot T_s(\varphi)^4$$

```
In [1]: # import header

        %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import climlab
        from climlab import constants as const
        from climlab.domain.field import global_mean
```

5.2.1 Model Creation

An EBM model instance is created through

```
In [2]: # model creation
        ebm_boltz = climlab.EBM(D=0.8, Tf=-2)
```

The model is set up by default with a linearized OLR parametrization (A+BT).

```
In [3]: # print model states and suprocesses
        print ebm_boltz
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

5.2.2 Create new subprocess

The creation of a subprocess needs some information from the model, especially on which model state the subprocess should be defined on.

```
In [4]: # create Boltzmann subprocess
        LW_boltz = climlab.radiation.Boltzmann(eps=0.65, tau=0.95,
                                               state=ebm_boltz.state,
                                               **ebm_boltz.param)
```

Note that the model's **whole state dictionary** is given as **input** to the subprocess. In case only the temperature field `ebm_boltz.state['Ts']` would be given, a new state dictionary would be created which holds the surface temperature with the key `'default'`. That raises an error as the Boltzmann process refers the temperature with key `'Ts'`.

Now the new OLR subprocess has to be merged into the model. Therefore, the `AplusBT` subprocess has to be removed first.

```
In [5]: # remove the old longwave subprocess
        ebm_boltz.remove_subprocess('LW')

        # add the new longwave subprocess
        ebm_boltz.add_subprocess('LW',LW_boltz)
```

Note that the new OLR subprocess has to have the **same key ''LW''** as the old one, as the model refers to this key for radiation balance computation.

That is why the old process has to be removed before the new one is added.

```
In [6]: print ebm_boltz
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.Boltzmann.Boltzmann'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

5.2.3 Model integration & Plotting

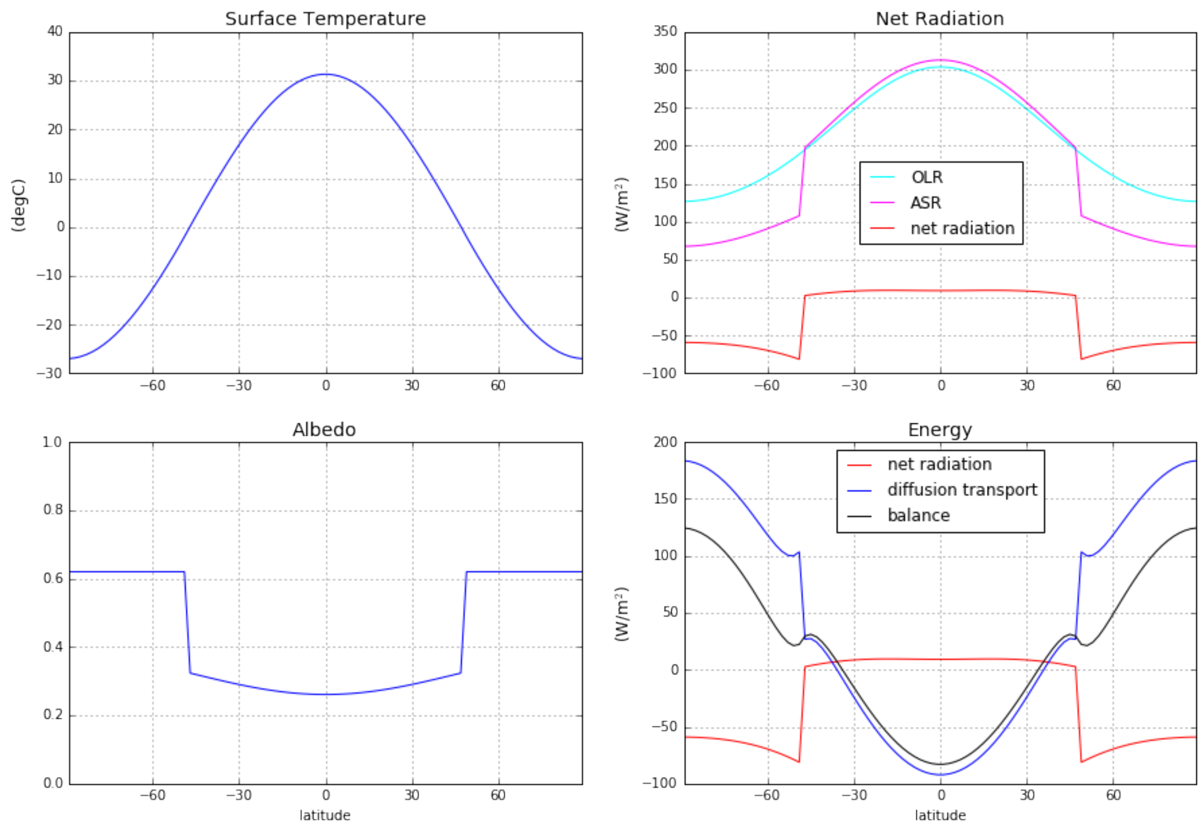
To visualize the model state at beginning of integration we first integrate the model only for one timestep:

```
In [7]: # integrate model for a single timestep
        ebm_boltz.step_forward()
```

The following code plots the current surface temperature, albedo and energy budget:

```
In [8]: # creating plot figure
        fig = plt.figure(figsize=(15,10))

        # Temperature plot
        ax1 = fig.add_subplot(221)
        ax1.plot(ebm_boltz.lat,ebm_boltz.Ts)

        ax1.set_xticks([-90,-60,-30,0,30,60,90])
        ax1.set_xlim([-90,90])
        ax1.set_title('Surface Temperature', fontsize=14)
        ax1.set_ylabel('(degC)', fontsize=12)
        ax1.grid()

        # Albedo plot
        ax2 = fig.add_subplot(223, sharex = ax1)
        ax2.plot(ebm_boltz.lat,ebm_boltz.albedo)

        ax2.set_title('Albedo', fontsize=14)
        ax2.set_xlabel('latitude', fontsize=10)
```

```
ax2.set_ylim([0,1])
ax2.grid()

# Net Radiation plot
ax3 = fig.add_subplot(222, sharex = ax1)
ax3.plot(ebm_boltz.lat, ebm_boltz.OLR, label='OLR',
                                        color='cyan')
ax3.plot(ebm_boltz.lat, ebm_boltz.ASR, label='ASR',
                                        color='magenta')
ax3.plot(ebm_boltz.lat, ebm_boltz.ASR-ebm_boltz.OLR,
                                        label='net radiation',
                                        color='red')

ax3.set_title('Net Radiation', fontsize=14)
ax3.set_ylabel('(W/m$^2$)', fontsize=12)
ax3.legend(loc='best')
ax3.grid()


# Energy Balance plot
net_rad = np.squeeze(ebm_boltz.net_radiation)
transport = ebm_boltz.heat_transport_convergence()

ax4 = fig.add_subplot(224, sharex = ax1)
ax4.plot(ebm_boltz.lat, net_rad, label='net radiation',
                                 color='red')
ax4.plot(ebm_boltz.lat, transport, label='diffusion transport',
                                   color='blue')
ax4.plot(ebm_boltz.lat, net_rad+transport, label='balance',
                                           color='black')

ax4.set_title('Energy', fontsize=14)
ax4.set_xlabel('latitude', fontsize=10)
ax4.set_ylabel('(W/m$^2$)', fontsize=12)
ax4.legend(loc='best')
ax4.grid()

plt.show()
```



The two right sided plots show that the model is not in equilibrium. The net radiation reveals that the model currently gains heat and therefore warms up at the poles and loses heat at the equator. From the Energy plot we can see that latitudinal energy balance is not met.

Now we integrate the model as long there are no more changes in the surface temperature and the model reached equilibrium:

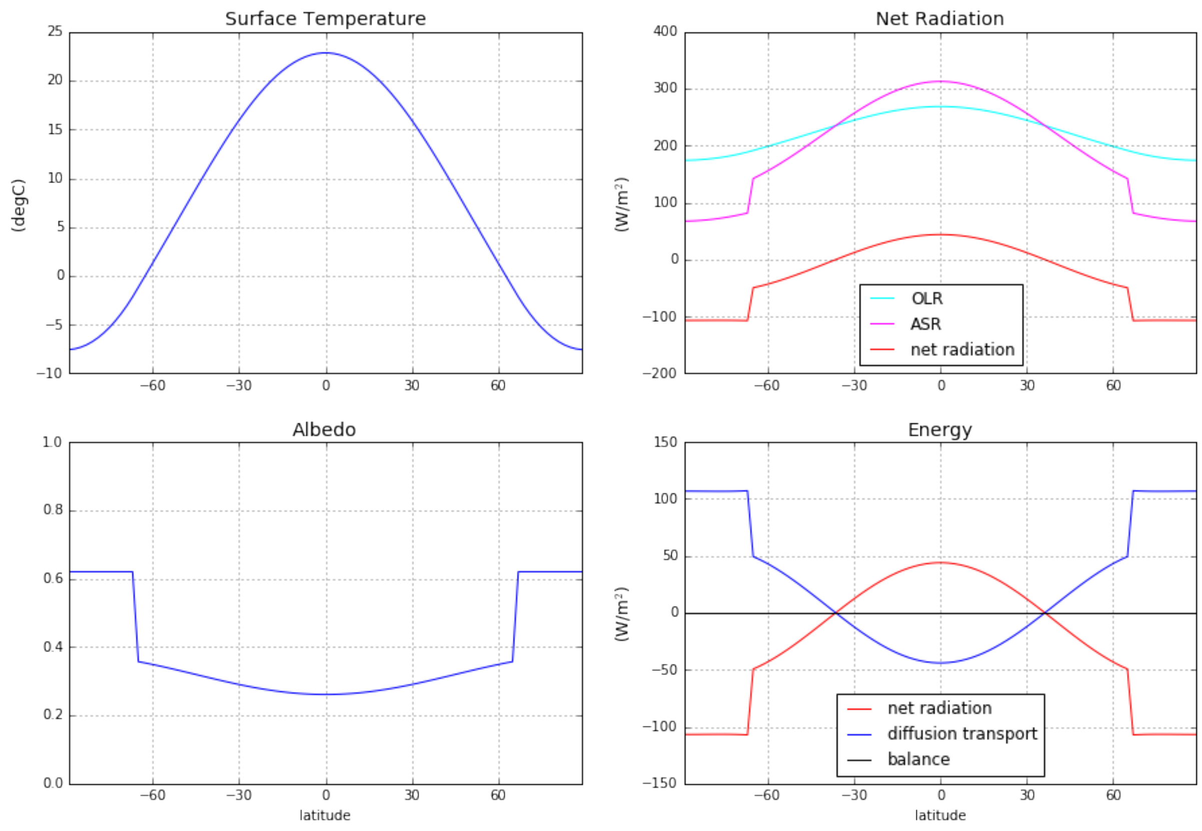
```
In [9]: # integrate model until solution converges
        ebm_boltz.integrate_converge()
```

```
Total elapsed time is 7.01111111111 years.
```

We run the same code as above to plot the results:

```
In [10]: # creating plot figure
         fig = plt.figure(figsize=(15,10))

         # Temperature plot
         ax1 = fig.add_subplot(221)
         ax1.plot(ebm_boltz.lat,ebm_boltz.Ts)

         ax1.set_xticks([-90,-60,-30,0,30,60,90])
         ax1.set_xlim([-90,90])
         ax1.set_title('Surface Temperature', fontsize=14)
         ax1.set_ylabel('(degC)', fontsize=12)
         ax1.grid()

         # Albedo plot
         ax2 = fig.add_subplot(223, sharex = ax1)
         ax2.plot(ebm_boltz.lat,ebm_boltz.albedo)

         ax2.set_title('Albedo', fontsize=14)
         ax2.set_xlabel('latitude', fontsize=10)
         ax2.set_ylim([0,1])
```

```
ax2.grid()

# Net Radiation plot
ax3 = fig.add_subplot(222, sharex = ax1)
ax3.plot(ebm_boltz.lat, ebm_boltz.OLR, label='OLR',
                                       color='cyan')
ax3.plot(ebm_boltz.lat, ebm_boltz.ASR, label='ASR',
                                       color='magenta')
ax3.plot(ebm_boltz.lat, ebm_boltz.ASR-ebm_boltz.OLR,
                                       label='net radiation',
                                       color='red')

ax3.set_title('Net Radiation', fontsize=14)
ax3.set_ylabel('(W/m$^2$)', fontsize=12)
ax3.legend(loc='best')
ax3.grid()


# Energy Balance plot
net_rad = np.squeeze(ebm_boltz.net_radiation)
transport = ebm_boltz.heat_transport_convergence()

ax4 = fig.add_subplot(224, sharex = ax1)
ax4.plot(ebm_boltz.lat, net_rad, label='net radiation',
                                 color='red')
ax4.plot(ebm_boltz.lat, transport, label='diffusion transport',
                                   color='blue')
ax4.plot(ebm_boltz.lat, net_rad+transport, label='balance',
                                           color='black')

ax4.set_title('Energy', fontsize=14)
ax4.set_xlabel('latitude', fontsize=10)
ax4.set_ylabel('(W/m$^2$)', fontsize=12)
ax4.legend(loc='best')
ax4.grid()

plt.show()
```



Now we can see that the latitudinal energy balance is statisfied. Each latitude gains as much heat (net radiation) as is transported out of it (diffusion transport). There is a net radiation surplus in the equator region, so more shortwave radiation is absorbed there than is emitted through longwave radiation. At the poles there is a net radiation deficit. That imbalance is compensated by the diffusive energy transport term.

5.2.4 Global mean temperature

We use climlab to compute the global mean temperature and print the ice edge latitude:

```
In [11]: print 'The global mean temperature is %s degC.' \
                              %np.round(global_mean(ebm_boltz.Ts),2)

         print 'The modeled ice edge is at %s deg.' %np.max(ebm_boltz.icelat)
```

```
The global mean temperature is 13.33 degC.
The modeled ice edge is at 66.0 deg.
```

5.3 Budyko Transport for Energy Balance Models

In this document an Energy Balance Model (EBM) is set up with the energy tranport parametrized through the the **budyko type parametrization** term (instead of the default diffusion term), which characterizes the local energy flux through the difference between local temperature and global mean temperature.

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

where $T(\varphi)$ is the surface temperature across the latitude φ, $\bar{T}$ the global mean temperature and $H(\varphi)$ is the transport of energy in an Energy Budget noted as:

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

```
In [1]: # import header

        %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import climlab
        from climlab import constants as const
        from climlab.domain.field import global_mean
```

5.3.1 Model Creation

An EBM model instance is created through

```
In [2]: # model creation
        ebm_budyko= climlab.EBM()
```

The model is set up by default with a meridional diffusion term.

```
In [3]: # print model states and suprocesses
        print ebm_budyko
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

5.3.2 Create new subprocess

The creation of a subprocess needs some information from the model, especially on which model state the subprocess should be defined on.

```
In [4]: # create Budyko subprocess
        budyko_transp = climlab.dynamics.BudykoTransport(b=3.81,
                                                         state=ebm_budyko.state,
                                                         **ebm_budyko.param)
```

Note that the model's **whole state dictionary** is given as **input** to the subprocess. In case only the temperature field `ebm_budyko.state['Ts']` is given, a new state dictionary would be created which holds the surface temperature with the key `'default'`. That raises an error as the budyko transport process refers the temperature with key `'Ts'`.

Now the new transport subprocess has to be merged into the model. The `diffusion` subprocess has to be removed.

```
In [5]: # add the new transport subprocess
        ebm_budyko.add_subprocess('budyko_transport',budyko_transp)

        # remove the old diffusion subprocess
        ebm_budyko.remove_subprocess('diffusion')
```

```
In [6]: print ebm_budyko
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   budyko_transport: <class 'climlab.dynamics.budyko_transport.BudykoTransport'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

5.3.3 Model integration & Plotting

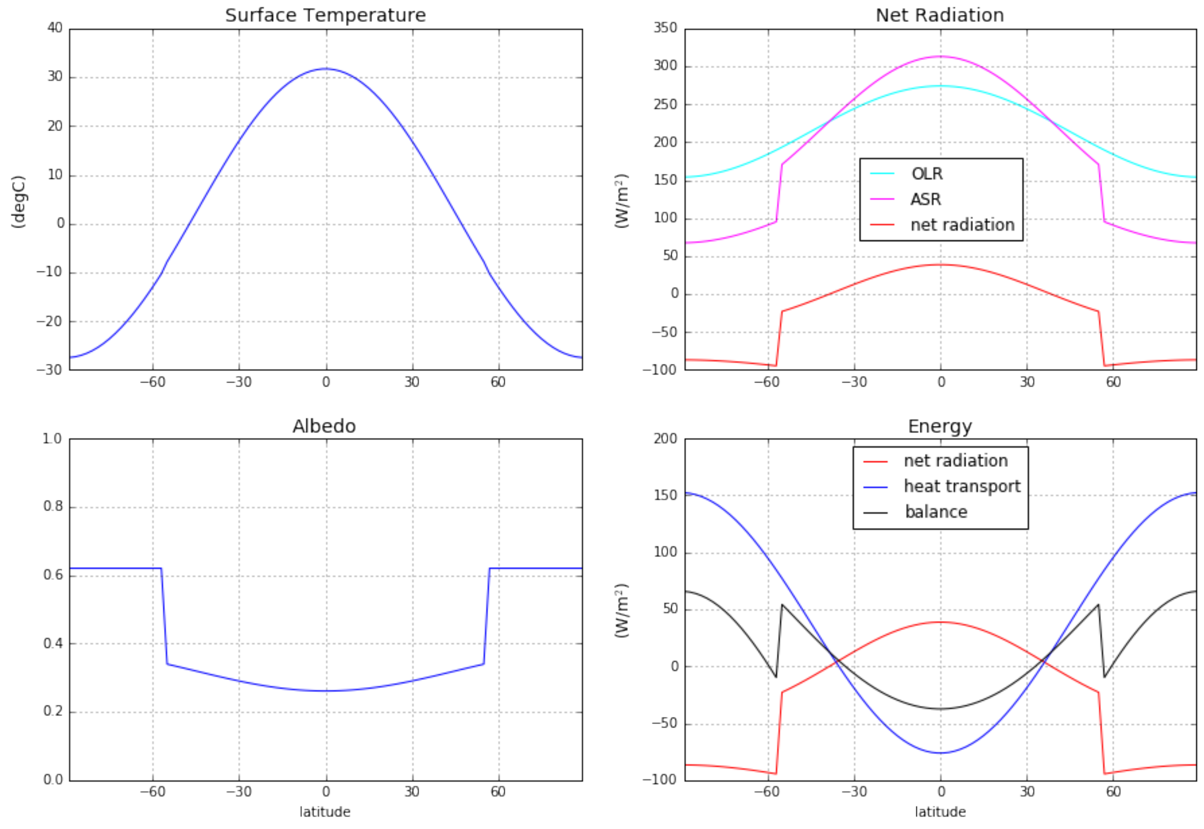
To visualize the model state at beginning of integration we first integrate the model only for one timestep:

```
In [7]: # integrate model for a single timestep
        ebm_budyko.step_forward()
```

The following code plots the current surface temperature, albedo and energy budget:

```
In [8]: # creating plot figure
        fig = plt.figure(figsize=(15,10))

        # Temperature plot
        ax1 = fig.add_subplot(221)
        ax1.plot(ebm_budyko.lat,ebm_budyko.Ts)

        ax1.set_xticks([-90,-60,-30,0,30,60,90])
        ax1.set_xlim([-90,90])
        ax1.set_title('Surface Temperature', fontsize=14)
        ax1.set_ylabel('(degC)', fontsize=12)
        ax1.grid()


        # Albedo plot
        ax2 = fig.add_subplot(223, sharex = ax1)
        ax2.plot(ebm_budyko.lat,ebm_budyko.albedo)

        ax2.set_title('Albedo', fontsize=14)
        ax2.set_xlabel('latitude', fontsize=10)
        ax2.set_ylim([0,1])
        ax2.grid()

        # Net Radiation plot
        ax3 = fig.add_subplot(222, sharex = ax1)
        ax3.plot(ebm_budyko.lat, ebm_budyko.OLR, label='OLR',
                                                  color='cyan')
        ax3.plot(ebm_budyko.lat, ebm_budyko.ASR, label='ASR',
                                                  color='magenta')
        ax3.plot(ebm_budyko.lat, ebm_budyko.ASR-ebm_budyko.OLR,
                                                  label='net radiation',
                                                  color='red')

        ax3.set_title('Net Radiation', fontsize=14)
        ax3.set_ylabel('(W/m$^2$)', fontsize=12)
        ax3.legend(loc='best')
        ax3.grid()
```

```
# Energy Balance plot
net_rad = ebm_budyko.net_radiation
transport = ebm_budyko.subprocess['budyko_transport'].heating_rate['Ts']

ax4 = fig.add_subplot(224, sharex = ax1)
ax4.plot(ebm_budyko.lat, net_rad, label='net radiation',
                                      color='red')
ax4.plot(ebm_budyko.lat, transport, label='heat transport',
                                      color='blue')
ax4.plot(ebm_budyko.lat, net_rad+transport, label='balance',
                                      color='black')

ax4.set_title('Energy', fontsize=14)
ax4.set_xlabel('latitude', fontsize=10)
ax4.set_ylabel('(W/m$^2$)', fontsize=12)
ax4.legend(loc='best')
ax4.grid()


plt.show()
```



The two right sided plots show that the model is not in equilibrium. The net radiation reveals that the model currently gains heat and therefore warms up at the poles and loses heat at the equator. From the Energy plot we can see that latitudinal energy balance is not met.

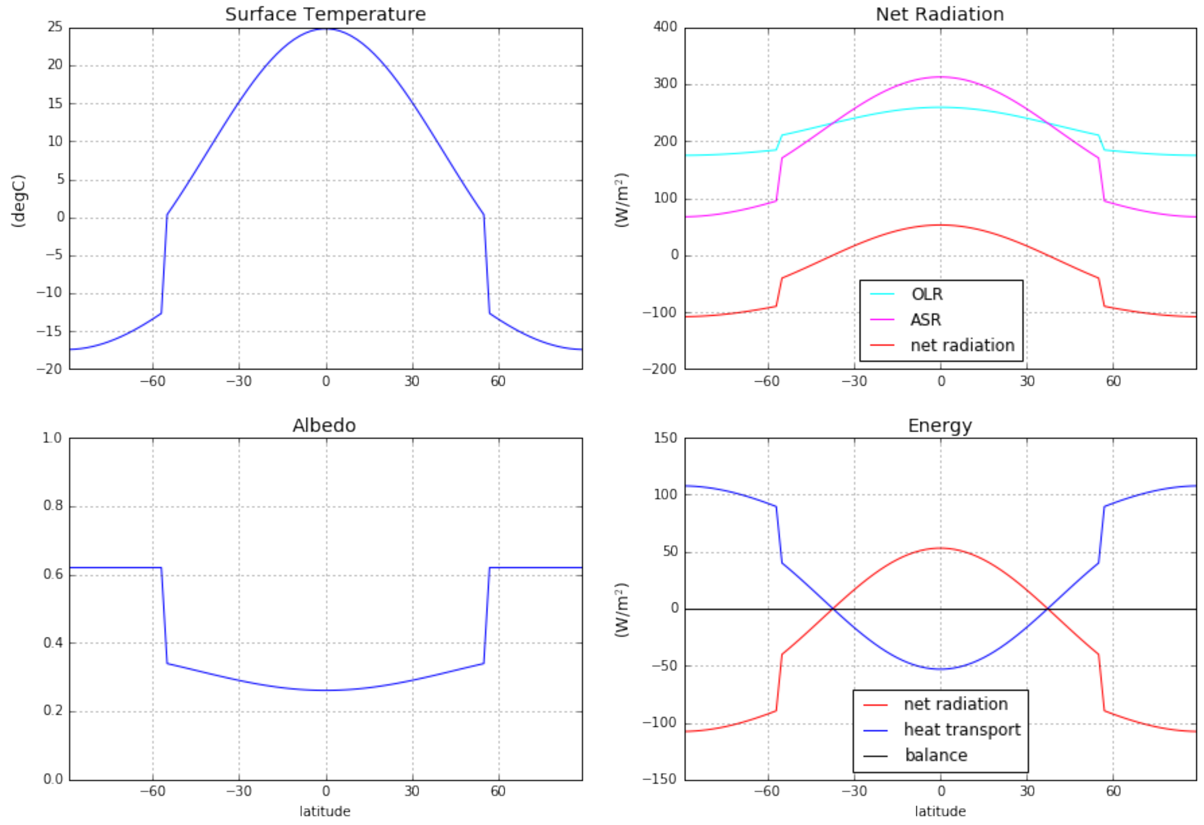
Now we integrate the model as long there are no more changes in the surface temperature and the model reached equilibrium:

```
In [9]: # integrate model until solution converges
        ebm_budyko.integrate_converge()
```

```
Total elapsed time is 7.01111111111 years.
```

```
In [10]: # creating plot figure
         fig = plt.figure(figsize=(15,10))

         # Temperature plot
         ax1 = fig.add_subplot(221)
         ax1.plot(ebm_budyko.lat,ebm_budyko.Ts)

         ax1.set_xticks([-90,-60,-30,0,30,60,90])
         ax1.set_xlim([-90,90])
         ax1.set_title('Surface Temperature', fontsize=14)
         ax1.set_ylabel('(degC)', fontsize=12)
         ax1.grid()


         # Albedo plot
         ax2 = fig.add_subplot(223, sharex = ax1)
         ax2.plot(ebm_budyko.lat,ebm_budyko.albedo)

         ax2.set_title('Albedo', fontsize=14)
         ax2.set_xlabel('latitude', fontsize=10)
         ax2.set_ylim([0,1])
         ax2.grid()

         # Net Radiation plot
         ax3 = fig.add_subplot(222, sharex = ax1)
         ax3.plot(ebm_budyko.lat, ebm_budyko.OLR, label='OLR',
                                                  color='cyan')
         ax3.plot(ebm_budyko.lat, ebm_budyko.ASR, label='ASR',
                                                  color='magenta')
         ax3.plot(ebm_budyko.lat, ebm_budyko.ASR-ebm_budyko.OLR,
                                                  label='net radiation',
                                                  color='red')

         ax3.set_title('Net Radiation', fontsize=14)
         ax3.set_ylabel('(W/m$^2$)', fontsize=12)
         ax3.legend(loc='best')
         ax3.grid()


         # Energy Balance plot
         net_rad = ebm_budyko.net_radiation
         transport = ebm_budyko.subprocess['budyko_transport'].heating_rate['Ts']

         ax4 = fig.add_subplot(224, sharex = ax1)
         ax4.plot(ebm_budyko.lat, net_rad, label='net radiation',
                                                      color='red')
         ax4.plot(ebm_budyko.lat, transport, label='heat transport',
                                                      color='blue')
         ax4.plot(ebm_budyko.lat, net_rad+transport, label='balance',
                                                      color='black')

         ax4.set_title('Energy', fontsize=14)
         ax4.set_xlabel('latitude', fontsize=10)
         ax4.set_ylabel('(W/m$^2$)', fontsize=12)
         ax4.legend(loc='best')
         ax4.grid()
```

```
        plt.show()
```



Now we can see that the latitudinal energy balance is statisfied. Each latitude gains as much heat (net radiation) as is transported out of it (diffusion transport). There is a net radiation surplus in the equator region, so more shortwave radiation is absorbed there than is emitted through longwave radiation. At the poles there is a net radiation deficit. That imbalance is compensated by the diffusive energy transport term.

5.3.4 Global mean temperature

We use climlab to compute the global mean temperature and print the ice edge latitude:

```
In [11]: print 'The global mean temperature is %s degC.' \
                                        %np.round(global_mean(ebm_budyko.Ts),2)

         print 'The modeled ice edge is at %s deg.' % np.max(ebm_budyko.icelat)
```

```
The global mean temperature is 10.87 degC with a model ice edge at 56.0 deg.
```

The temperature is a bit too cold for current climate as model parameters are not tuned. Sensitive parameters are `a0, a2, ai` and `Tf` (albedo), `A` and `B` (OLR), `b` (transport) and `num_lat` (grid resolution).

Some more notebooks which focus on specific aspects of Energy Balance Models:

5.4 Distribution of insolation

Here are some examples calculating daily average insolation at different locations and times.

These all use a function called `daily_insolation` in the module `insolation.py` to do the calculation. The code calculates daily average insolation anywhere on Earth at any time of year for a given set of orbital parameters.

To look at past orbital variations and their effects on insolation, we use the module `orbital.py` which accesses tables of values for the past 5 million years. We can easily lookup parameters for any point in the past and pass these to `daily_insolation`.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import netCDF4 as nc
        from climlab import constants as const
        from climlab.solar.insolation import daily_insolation
        from climlab.solar.orbital import OrbitalTable
```

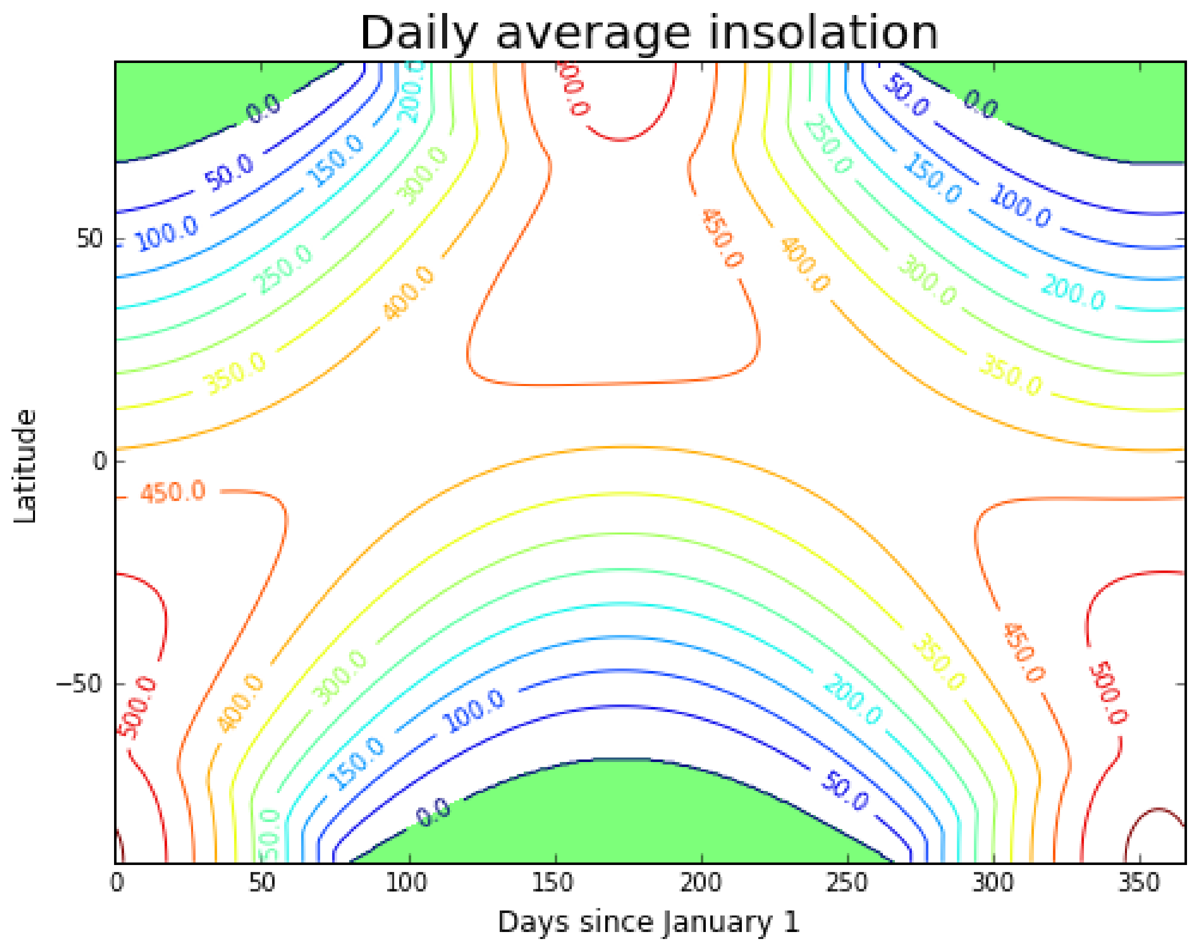
5.4.1 Present-day orbital parameters

Calculate an array of insolation over the year and all latitudes (for present-day orbital parameters).

```
In [2]: lat = np.linspace( -90., 90., 500. )
        days = np.linspace(0, const.days_per_year, 365. )
        Q = daily_insolation( lat, days )
```

And make a contour plot of Q as function of latitude and time of year.

```
In [3]: ax = plt.figure( figsize=(8,6) ).add_subplot(111)
        CS = ax.contour( days, lat, Q , levels = np.arange(0., 600., 50.) )
        ax.clabel(CS, CS.levels, inline=True, fmt='%r', fontsize=10)
        ax.set_xlabel('Days since January 1', fontsize=12 )
        ax.set_ylabel('Latitude', fontsize=12 )
        ax.set_title('Daily average insolation', fontsize=20 )
        ax.contourf ( days, lat, Q, levels=[-500., 0.] )
        plt.show()
```



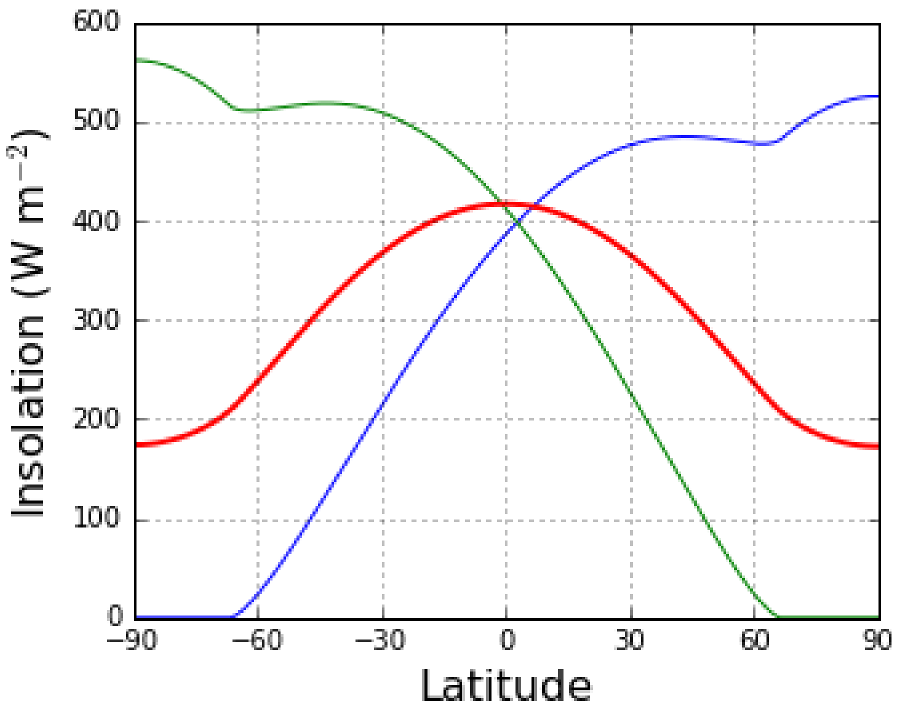
Take the area-weighted global, annual average of Q...

```
In [4]: print np.sum( np.mean( Q, axis=1 ) * np.cos( np.deg2rad(lat) ) ) / np.sum( np.co
```

```
341.384184481
```

Also plot the zonally averaged insolation at a few different times of the year:

```
In [5]: summer_solstice = 170
        winter_solstice = 353
        ax = plt.figure( figsize=(5,4) ).add_subplot(111)
        ax.plot( lat, Q[:,(summer_solstice, winter_solstice)] );
        ax.plot( lat, np.mean(Q, axis=1), linewidth=2 )
        ax.set_xbound(-90, 90)
        ax.set_xticks( range(-90,100,30) )
        ax.set_xlabel('Latitude', fontsize=16 );
        ax.set_ylabel('Insolation (W m$^{-2}$)', fontsize=16 );
        ax.grid()
        plt.show()
```



5.4.2 Past orbital parameters

The `orbital.py` code allows us to look up the orbital parameters for Earth over the last 5 million years.

Make reference plots of the variation in the three orbital parameter over the last 1 million years

```
In [6]: kyears = np.arange( -1000., 1.)
        table = OrbitalTable()
        orb = table.lookup_parameters( kyears )
```

```
Loading Berger and Loutre (1991) orbital parameter data from file /home/moritz/anaconda2
```

The Python object `orb` now holds 1 million years worth of orbital data, total of 1001 data points for each element: eccentricity `ecc`, obliquity angle `obliquity`, and solar longitude of perihelion `long_peri`.

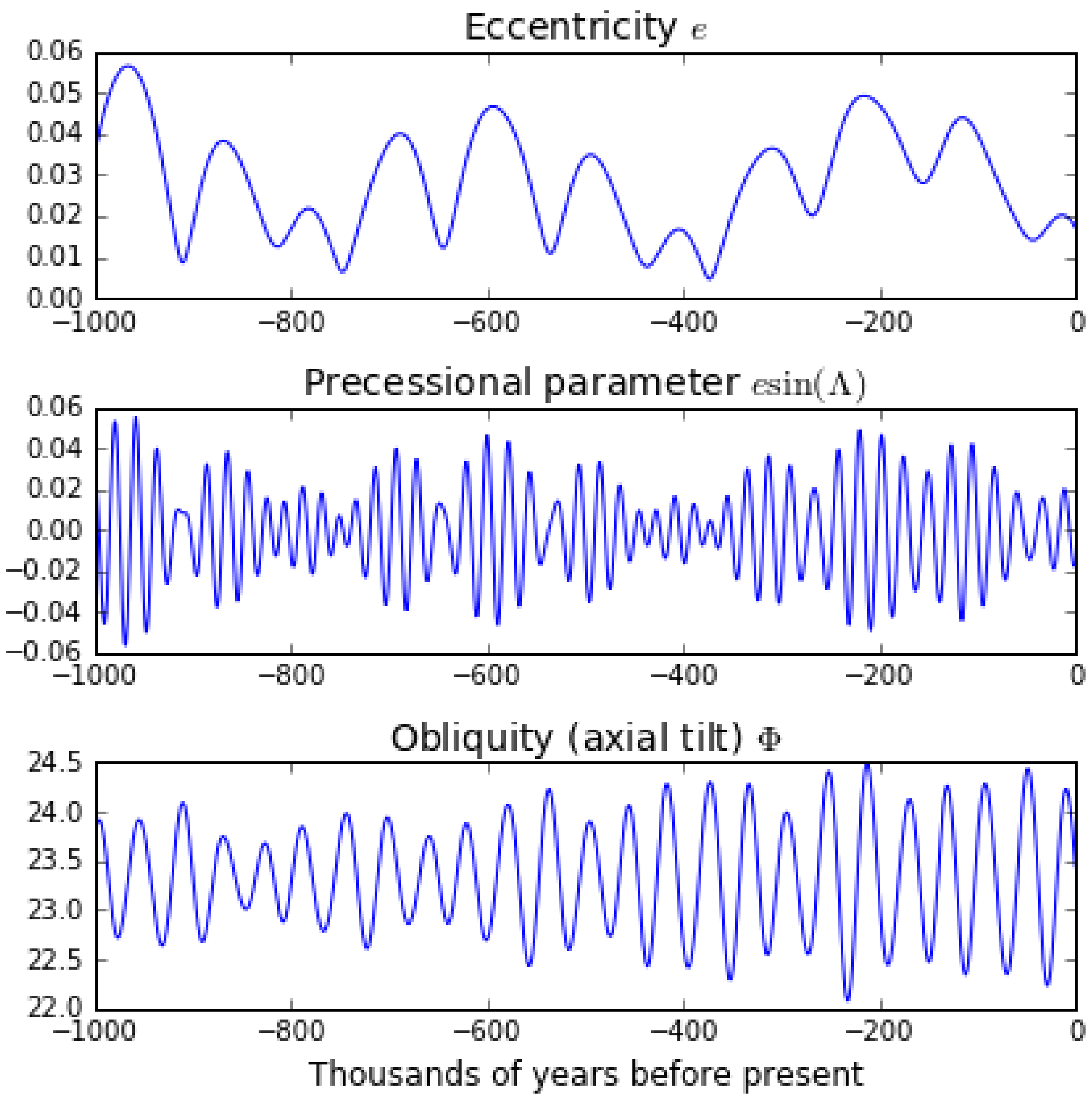
```
In [23]: orb
```

```
Out[23]: {'ecc': array([ 0.035765,  0.036953,  0.038114, ...,  0.018024,  0.017644,
                  0.017236]),
          'long_peri': array([ 122.46,  138.29,  154.17, ...,  247.23,  264.26,  281.37]
          'obliquity': array([ 23.778,  23.835,  23.877, ...,  23.697,  23.573,  23.446]
```

```
In [25]: print [np.shape( orb['ecc'] ),
                np.shape( orb['long_peri'] ),
                np.shape( orb['obliquity'] ) ]
```

```
[(1001,), (1001,), (1001,)]
```

```
In [8]: fig = plt.figure( figsize = (6,6) )
        ax1 = fig.add_subplot(3,1,1)
        ax1.plot( kyears, orb['ecc'] )
        ax1.set_title('Eccentricity $e$', fontsize=14 )
        ax2 = fig.add_subplot(3,1,2)
        ax2.plot( kyears, orb['ecc'] * np.sin( np.deg2rad( orb['long_peri'] ) ) )
        ax2.set_title('Precessional parameter $e \sin(\Lambda)$', fontsize=14 )
        ax3 = fig.add_subplot(3,1,3)
        ax3.plot( kyears, orb['obliquity'] )
        ax3.set_title('Obliquity (axial tilt) $\Phi$', fontsize=14 )
        ax3.set_xlabel( 'Thousands of years before present', fontsize=12 )

        plt.tight_layout()
        plt.show()
```



Annual mean insolation

Create a large array of insolation over the whole globe, whole year, and for every set of orbital parameters.

```
In [9]: lat = np.linspace(-90, 90, 181)
        days = np.linspace(1.,50.)/50 * const.days_per_year
        Q = daily_insolation(lat, days, orb)
        print Q.shape
```

```
(181, 50, 1001)
```

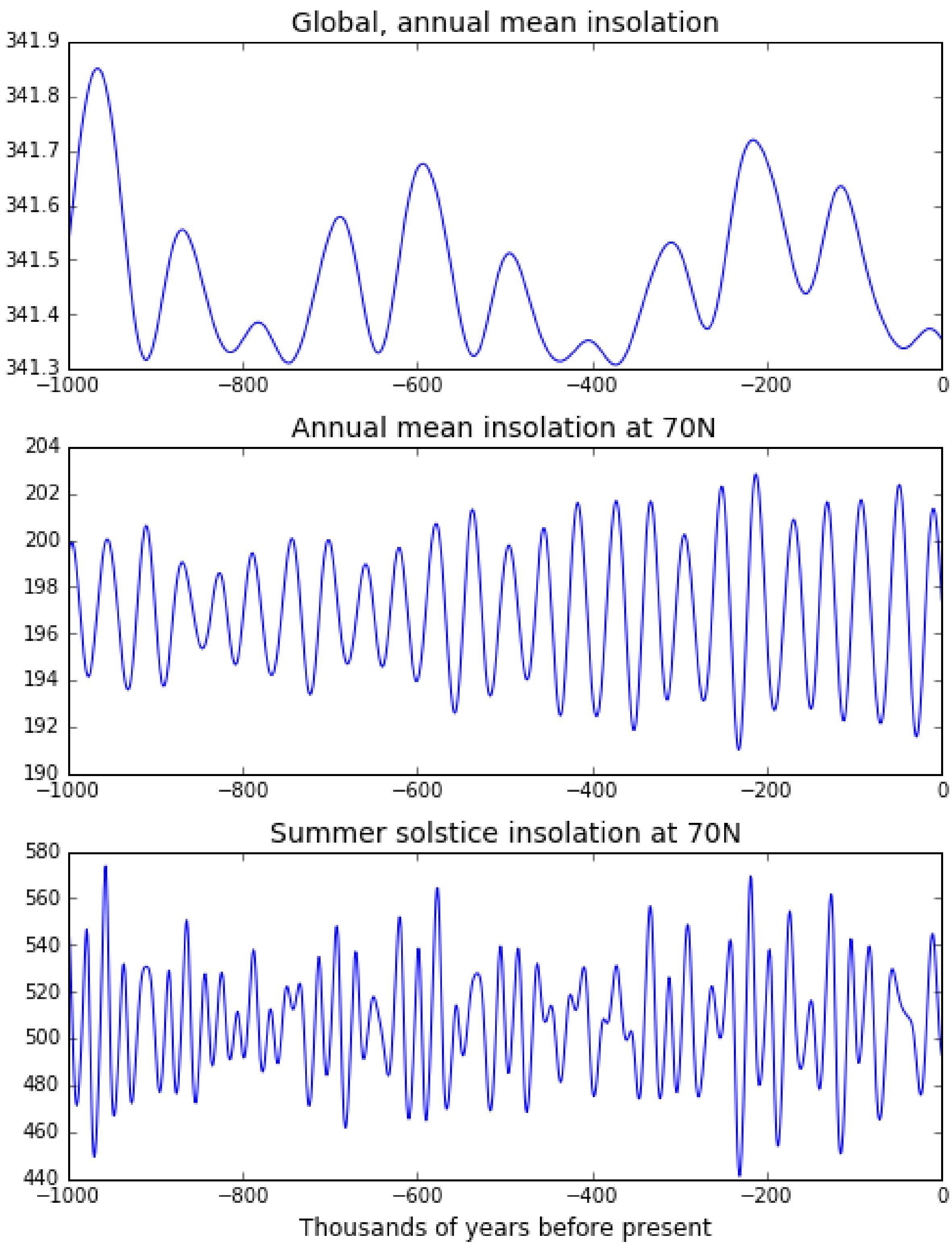
```
In [10]: Qann = np.mean(Q, axis=1)  # time average over the year
         print Qann.shape
         Qglobal = np.empty_like( kyears )
         for n in range( kyears.size ):   # global area-weighted average
             Qglobal[n] = np.sum( Qann[:,n] * np.cos( np.deg2rad(lat) ) ) \
                      / np.sum( np.cos( np.deg2rad(lat) ) )
         print Qglobal.shape
```

```
(181, 1001)
(1001,)
```

We are going to create a figure showing past time variations in three quantities:

1. Global, annual mean insolation
2. Annual mean insolation at high northern latitudes
3. Summer solstice insolation at high northern latitudes

```
In [32]: fig = plt.figure( figsize = (7,9) , dpi=80 )

         ax1 = fig.add_subplot(3,1,1)
         ax1.plot( kyears, Qglobal )
         ax1.set_title('Global, annual mean insolation', fontsize=14 )
         ax1.ticklabel_format( useOffset=False )

         ax2 = fig.add_subplot(3,1,2)
         ax2.plot( kyears, Qann[160,:] )
         ax2.set_title('Annual mean insolation at 70N', fontsize=14 )

         ax3 = fig.add_subplot(3,1,3)
         ax3.plot( kyears, Q[160,23,:] )
         ax3.set_xlabel( 'Thousands of years before present', fontsize=12 )
         ax3.set_title('Summer solstice insolation at 70N', fontsize=14 )

         plt.tight_layout()
         plt.show()
```



And comparing with the plots of orbital variations above, we see that

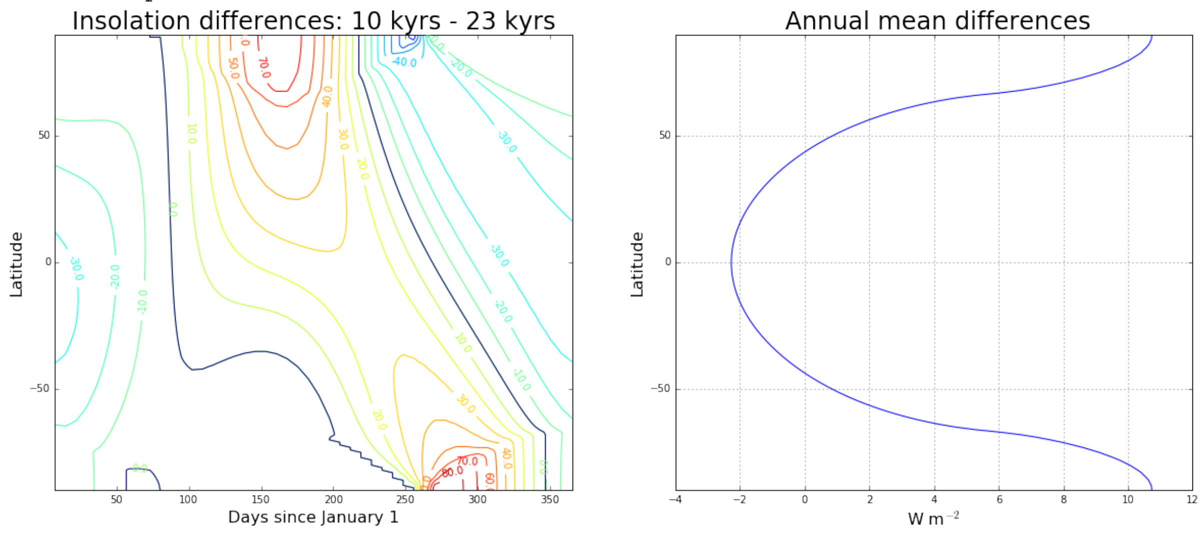
1. Global annual mean insolation variations on with eccentricity (slow), and the variations are very small!
2. Annual mean insolation varies with obliquity (medium). Annual mean insolation does NOT depend on precession!
3. Summer solstice insolation at high northern latitudes is affected by both precession and obliquity. The variations are large.

Insolation changes between the Last Glacial Maximum and the end of the last ice age

Last Glacial Maximum or "LGM" occurred around 23,000 years before present, when the ice sheets were at their greatest extent. By 10,000 years ago, the ice sheets were mostly gone and the last ice age was over. Let's plot the changes in the seasonal distribution of insolation from 23 kyrs to 10 kyrs.

```
In [12]: # present-day orbital parameters
         orb_0 = table.lookup_parameters( 0 )

         # orbital parameters for 10 kyrs before present
         orb_10 = table.lookup_parameters( -10 )

         # 23 kyrs before present
         orb_23 = table.lookup_parameters( -23 )

         # insolation arrays for each of the three sets of orbital parameters
         Q_0 = daily_insolation( lat, days, orb_0 )
         Q_10 = daily_insolation( lat, days, orb_10 )
         Q_23 = daily_insolation( lat, days, orb_23 )
```

```
In [13]: fig = plt.figure( figsize=(20,8) )

         ax1 = fig.add_subplot(1,2,1)
         Qdiff = Q_10 - Q_23
         CS1 = ax1.contour( days, lat, Qdiff, levels = np.arange(-100., 100., 10.) )
         ax1.clabel(CS1, CS1.levels, inline=True, fmt='%r', fontsize=10)
         ax1.contour( days, lat, Qdiff, levels = [0.], color = 'k' )
         ax1.set_xlabel('Days since January 1', fontsize=16 )
         ax1.set_ylabel('Latitude', fontsize=16 )
         ax1.set_title('Insolation differences: 10 kyrs - 23 kyrs', fontsize=24 )

         ax2 = fig.add_subplot(1,2,2)
         ax2.plot( np.mean( Qdiff, axis=1 ), lat )
         ax2.set_xlabel('W m$^{-2}$', fontsize=16 )
         ax2.set_ylabel( 'Latitude', fontsize=16 )
         ax2.set_title(' Annual mean differences', fontsize=24 )
         ax2.set_ylim((-90,90))
         ax2.grid()

         plt.show()
```



The annual mean plot shows a classic obliquity signal: at 10 kyrs, the axis close to its maximum tilt, around 24.2º. At 23 kyrs, the tilt was much weaker, only about 22.7º. In the annual mean, a stronger tilt means more sunlight to the poles and less to the equator. This is very helpful if you are trying to melt an ice sheet.

Finally, take the global average of the difference:

```
In [14]: print np.sum( np.mean(Qdiff,axis=1) * np.cos( np.deg2rad(lat) ) ) \
              / np.sum( np.cos(np.deg2rad(lat)) )
```

```
0.00651043078327
```

This confirms that the difference is tiny (and due to very small changes in the eccentricity). **Ice ages are driven by seasonal and latitudinal redistributions of solar energy**, NOT by changes in the total global amount of solar energy!

5.5 The seasonal cycle of surface temperature

Look at the observed seasonal cycle in the NCEP reanalysis data.

Read in the necessary data from the online server.

The catalog is here: http://www.esrl.noaa.gov/psd/thredds/dodsC/Datasets/ncep.reanalysis.derived/catalog.html

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import netCDF4 as nc
        from climlab import constants as const
        from climlab.model import ebm
        from climlab.solar.insolation import daily_insolation

        datapath = "http://ramadda.atmos.albany.edu:8080/repository/opendap/latest/Top/U
        endstr = "/entry.das"
```

```
In [2]: ncep_url = "http://www.esrl.noaa.gov/psd/thredds/dodsC/Datasets/ncep.reanalysis.
        ncep_air = nc.Dataset( ncep_url + "pressure/air.mon.1981-2010.ltm.nc" )
        ncep_Ts = nc.Dataset( ncep_url + "surface_gauss/skt.sfc.mon.1981-2010.ltm.nc" )
        lat_ncep = ncep_Ts.variables['lat'][:]; lon_ncep = ncep_Ts.variables['lon'][:]
        Ts_ncep = ncep_Ts.variables['skt'][:]
        print Ts_ncep.shape
```

```
(12, 94, 192)
```

Load the topography file from CESM, just so we can plot the continents.

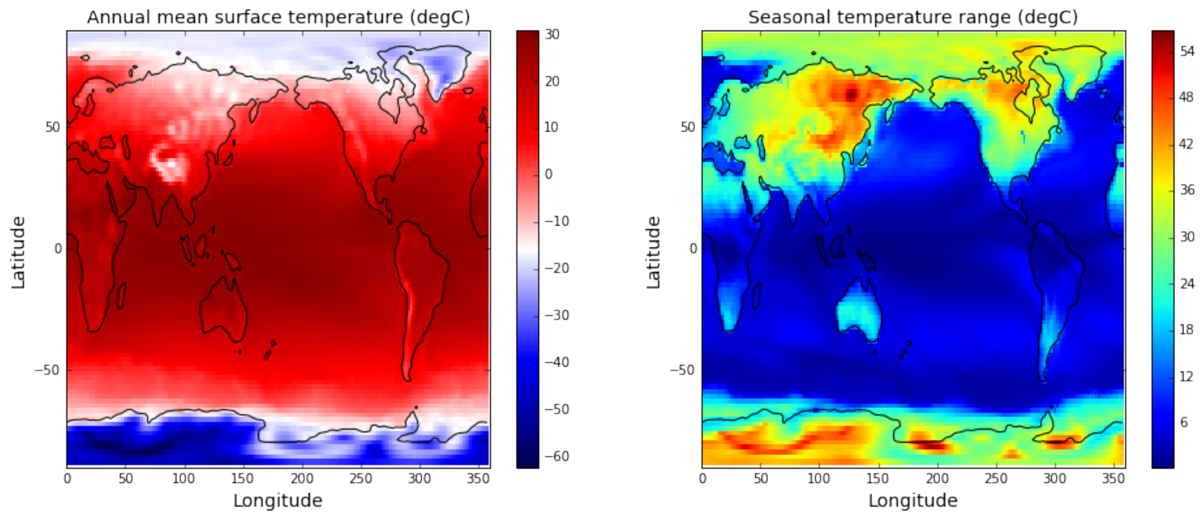
```
In [3]: topo = nc.Dataset( datapath + 'som_input/USGS-gtopo30_1.9x2.5_remap_c050602.nc'
        lat_cesm = topo.variables['lat'][:]
        lon_cesm = topo.variables['lon'][:]
```

Make two maps: one of annual mean surface temperature, another of the seasonal range (max minus min).

```
In [4]: maxTs = np.max(Ts_ncep,axis=0)
        minTs = np.min(Ts_ncep,axis=0)
```

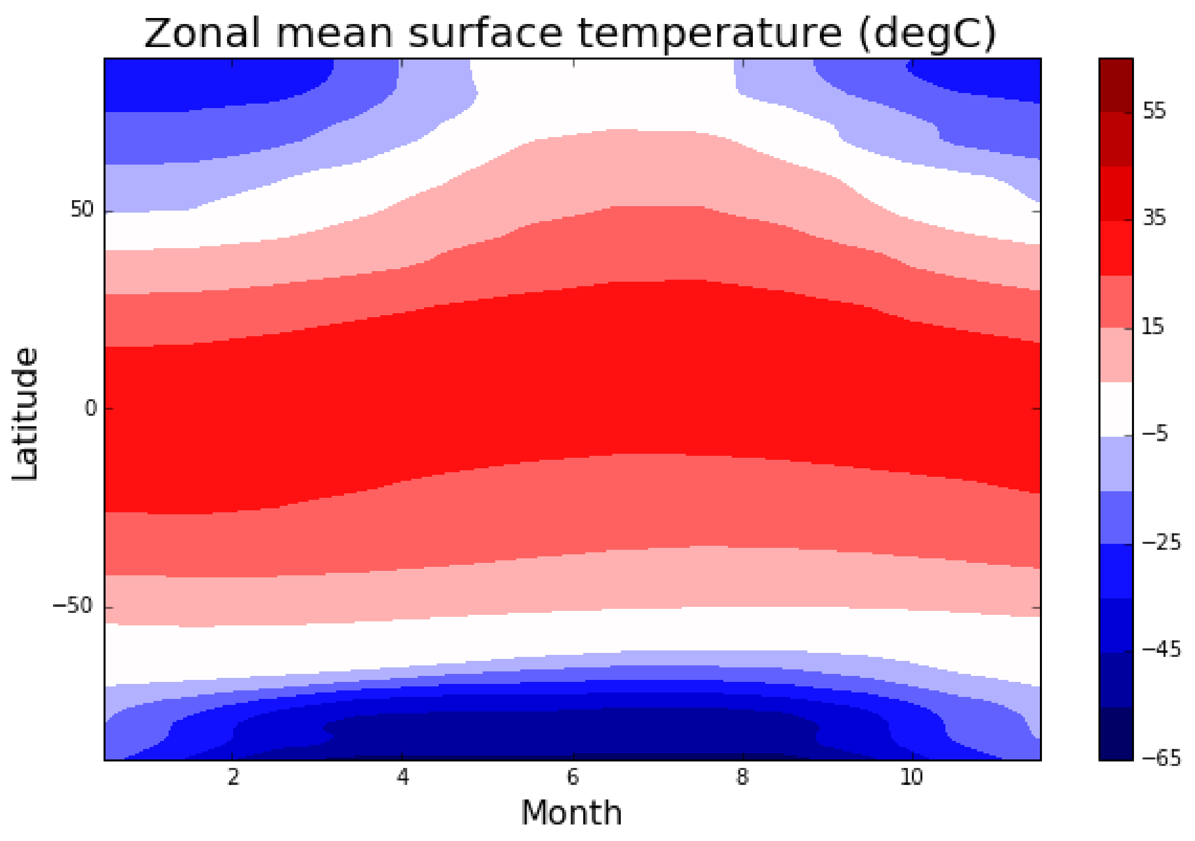
```
In [5]: fig = plt.figure( figsize=(16,6) )

        ax1 = fig.add_subplot(1,2,1)
        cax1 = ax1.pcolormesh( lon_ncep, lat_ncep, np.mean(Ts_ncep, axis=0), cmap=plt.cm
        cbar1 = plt.colorbar(cax1)
        ax1.set_title('Annual mean surface temperature (degC)', fontsize=14 )

        ax2 = fig.add_subplot(1,2,2)
        cax2 = ax2.pcolormesh( lon_ncep, lat_ncep, maxTs - minTs )
        cbar2 = plt.colorbar(cax2)
        ax2.set_title('Seasonal temperature range (degC)', fontsize=14)
```

```
for ax in [ax1,ax2]:
    ax.contour( lon_cesm, lat_cesm, topo.variables['LANDFRAC'][:], levels=[0.5],
    ax.axis([0, 360, -90, 90])
    ax.set_xlabel('Longitude', fontsize=14 ); ax.set_ylabel('Latitude', fontsize
```



Make a contour plot of the zonal mean temperature as a function of time of year

```
In [6]: [-65,-55,-45,-35,-25,-15,-5,5,15,25,35,45,55,65]

Out[6]: [-65, -55, -45, -35, -25, -15, -5, 5, 15, 25, 35, 45, 55, 65]

In [7]: np.arange(-65,75,10)

Out[7]: array([-65, -55, -45, -35, -25, -15,  -5,   5,  15,  25,  35,  45,  55,  65])

In [8]: Tmax = 65; Tmin = -Tmax; delT = 10
        clevels = np.arange(Tmin,Tmax+delT,delT)

        fig_zonobs = plt.figure( figsize=(10,6) )
        ax = fig_zonobs.add_subplot(111)
        cax = ax.contourf( np.arange(12)+0.5, lat_ncep, np.transpose(np.mean( Ts_ncep, a
                       cmap=plt.cm.seismic, vmin=Tmin, vmax=Tmax )
        ax.set_xlabel('Month', fontsize=16)
        ax.set_ylabel('Latitude', fontsize=16 )
        cbar = plt.colorbar(cax)
        ax.set_title('Zonal mean surface temperature (degC)', fontsize=20)
        plt.show()
```



5.5.1 Exploring the amplitude of the seasonal cycle with an EBM

We are looking at the 1D (zonally averaged) energy balance model with diffusive heat transport. The equation is

$$C\frac{\partial T(\phi,t)}{\partial t} = (1-\alpha)Q(\phi,t) - \Big(A + BT(\phi,t)\Big) + \frac{K}{\cos\phi}\frac{\partial}{\partial\phi}\left(\cos\phi\frac{\partial T}{\partial\phi}\right)$$

and the code in `climlab.model.ebm.py` solves this equation numerically.

One handy feature of `climlab` process code: the function `integrate_years()` automatically calculates the time averaged temperature. So if we run it for exactly one year, we get the annual mean temperature saved in the field `T_timeave`.

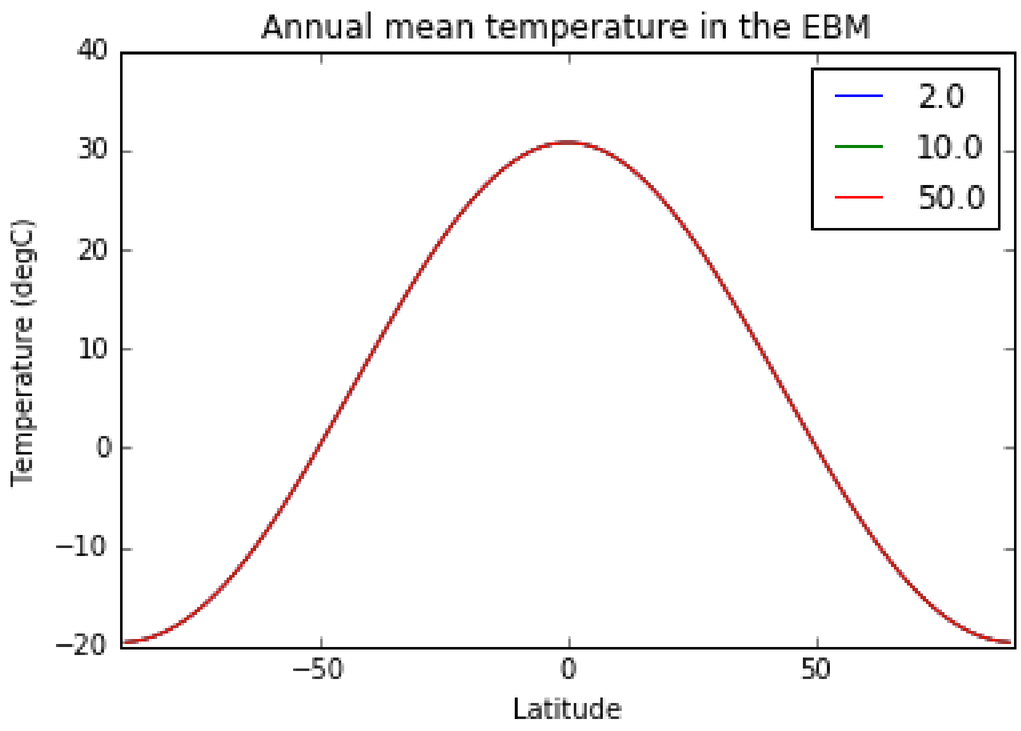
We will look at the seasonal cycle of temperature in three different models with different heat capacities (which we express through an equivalent depth of water in meters):

```
In [9]: model1 = ebm.EBM_seasonal()
        model1.integrate_years(1, verbose=True)

        water_depths = np.array([2., 10., 50.])

        num_depths = water_depths.size
        Tann = np.empty( [model1.lat.size, num_depths] )
        models = []

        for n in range(num_depths):
            models.append(ebm.EBM_seasonal(water_depth=water_depths[n]))
            models[n].integrate_years(20., verbose=False )
            models[n].integrate_years(1., verbose=False)
            Tann[:,n] = np.squeeze(models[n].timeave['Ts'])
```

```
Integrating for 90 steps, 365.2422 days, or 1 years.
Total elapsed time is 1.0 years.
```

All models should have the same annual mean temperature:

```
In [10]: lat = model1.lat

         plt.plot(lat, Tann)
         plt.xlim(-90,90)
         plt.xlabel('Latitude')
         plt.ylabel('Temperature (degC)')
         plt.title('Annual mean temperature in the EBM')
         plt.legend( water_depths.astype(str) )
         plt.show()
```



There is no automatic function in the `ebm.py` code to keep track of minimum and maximum temperatures (though we might add that in the future!)

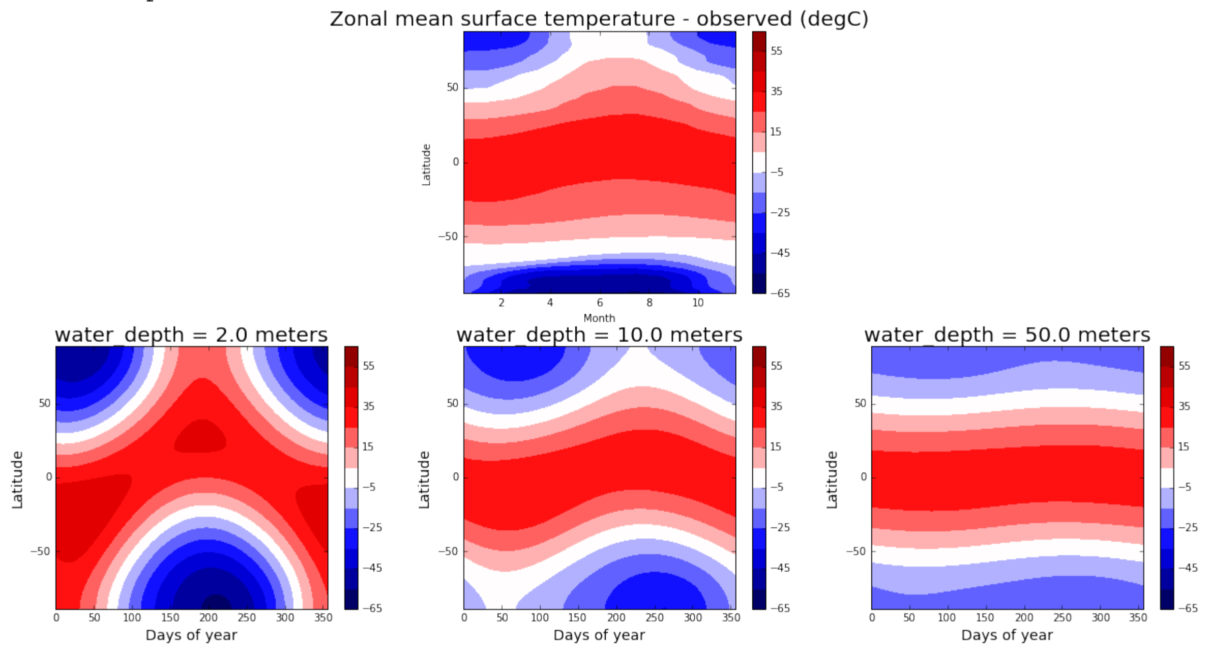
Instead we'll step through one year "by hand" and save all the temperatures.

```
In [11]: num_steps_per_year = int(model1.time['num_steps_per_year'])
         Tyear = np.empty((lat.size, num_steps_per_year, num_depths))
         for n in range(num_depths):
             for m in range(num_steps_per_year):
                 models[n].step_forward()
                 Tyear[:,m,n] = np.squeeze(models[n].state['Ts'])
```

Make a figure to compare the observed zonal mean seasonal temperature cycle to what we get from the EBM with different heat capacities:

```
In [12]: fig = plt.figure( figsize=(20,10) )

         ax = fig.add_subplot(2,num_depths,2)
         cax = ax.contourf( np.arange(12)+0.5, lat_ncep, np.transpose(np.mean( Ts_ncep,
                           cmap=plt.cm.seismic, vmin=Tmin, vmax=Tmax )
         ax.set_xlabel('Month')
         ax.set_ylabel('Latitude')
         cbar = plt.colorbar(cax)
         ax.set_title('Zonal mean surface temperature - observed (degC)', fontsize=20)
```

```
          for n in range(num_depths):
              ax = fig.add_subplot(2,num_depths,num_depths+n+1)
              cax = ax.contourf( 4*np.arange(num_steps_per_year), lat, Tyear[:,:,n], leve
                          cmap=plt.cm.seismic, vmin=Tmin, vmax=Tmax )
              cbar1 = plt.colorbar(cax)
              ax.set_title('water_depth = ' + str(models[n].param['water_depth']) + ' met
              ax.set_xlabel('Days of year', fontsize=14 )
              ax.set_ylabel('Latitude', fontsize=14 )

          #fig.set_title('Temperature in seasonal EBM with various water depths', fontsiz
          plt.show()
```

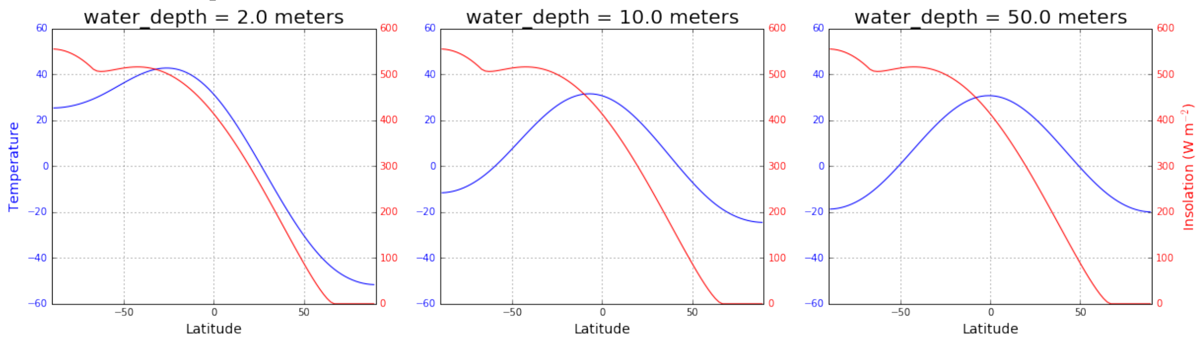


Which one looks more realistic? Depends a bit on where you look. But overall, the observed seasonal cycle matches the 10 meter case best. The effective heat capacity governing the seasonal cycle of the zonal mean temperature is closer to 10 meters of water than to either 2 or 50 meters.

Making an animation of the EBM solutions

```
In [13]: fpath = '/Users/Brian/Dropbox/PythonStuff/ebm_seasonal_frames/'

         fig = plt.figure( figsize=(20,5) )
         #for m in range(model2.time['num_steps_per_year']):
         for m in range(1):
             thisday = m * models[0].param['timestep'] / const.seconds_per_day
             Q = daily_insolation( lat, thisday )
             for n in range(num_depths):
                 c1 = 'b'
                 ax = fig.add_subplot(1,num_depths,n+1)
                 ax.plot( lat, Tyear[:,m,n], c1 )
                 ax.set_title('water_depth = ' + str(models[n].param['water_depth']) + '
                 ax.set_xlabel('Latitude', fontsize=14 )
                 if n is 0:
                     ax.set_ylabel('Temperature', fontsize=14, color=c1 )
                 ax.set_xlim([-90,90])
                 ax.set_ylim([-60,60])
                 for tl in ax.get_yticklabels():
                     tl.set_color(c1)
```

```
                ax.grid()

                c2 = 'r'
                ax2 = ax.twinx()
                ax2.plot( lat, Q, c2)
                if n is 2:
                    ax2.set_ylabel('Insolation (W m$^{-2}$)', color=c2, fontsize=14)
                for tl in ax2.get_yticklabels():
                    tl.set_color(c2)
                ax2.set_xlim([-90,90])
                ax2.set_ylim([0,600])

            filename = fpath + 'ebm_seasonal' + str(m).zfill(4) + '.png'
            #fig.savefig(filename)
            #fig.clear()
            plt.show()
```



5.5.2 The seasonal cycle for a planet with 90º obliquity

The EBM code uses our familiar `insolation.py` code to calculate insolation, and therefore it's easy to set up a model with different orbital parameters. Here is an example with **very** different orbital parameters: 90º obliquity. We looked at the distribution of insolation by latitude and season for this type of planet in the last homework.

```
In [14]: orb_highobl = {'ecc':0., 'obliquity':90., 'long_peri':0.}
         print orb_highobl
         model_highobl = ebm.EBM_seasonal(orb=orb_highobl)
         print model_highobl.param['orb']
```

```
'long_peri': 0.0, 'ecc': 0.0, 'obliquity': 90.0
'long_peri': 0.0, 'ecc': 0.0, 'obliquity': 90.0
```

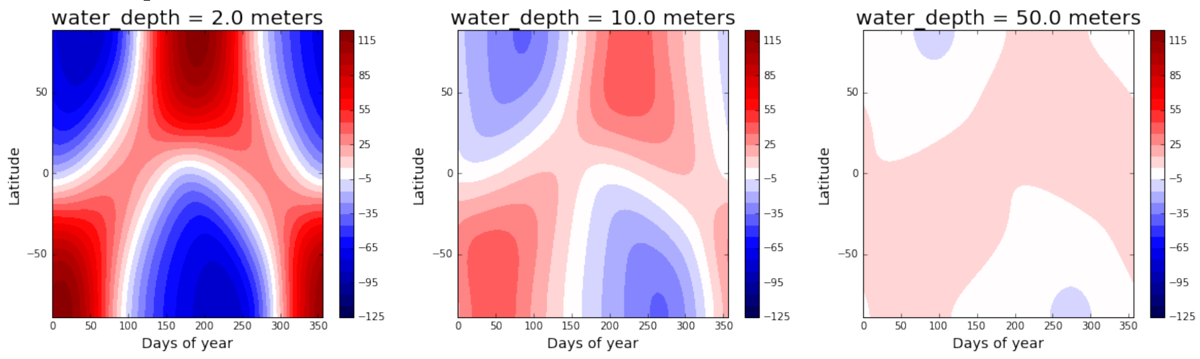
Repeat the same procedure to calculate and store temperature throughout one year, after letting the models run out to equilibrium.

```
In [15]: Tann_highobl = np.empty( [lat.size, num_depths] )
         models_highobl = []

         for n in range(num_depths):
             models_highobl.append(ebm.EBM_seasonal(water_depth=water_depths[n], orb=orb
             models_highobl[n].integrate_years(40., verbose=False )
             models_highobl[n].integrate_years(1.)
             Tann_highobl[:,n] = np.squeeze(models_highobl[n].timeave['Ts'])

         Tyear_highobl = np.empty([lat.size, num_steps_per_year, num_depths])
         for n in range(num_depths):
             for m in range(num_steps_per_year):
                 models_highobl[n].step_forward()
                 Tyear_highobl[:,m,n] = np.squeeze(models_highobl[n].state['Ts'])
```

```
Integrating for 90 steps, 365.2422 days, or 1.0 years.
Total elapsed time is 41.0 years.
Integrating for 90 steps, 365.2422 days, or 1.0 years.
Total elapsed time is 41.0 years.
Integrating for 90 steps, 365.2422 days, or 1.0 years.
Total elapsed time is 41.0 years.
```

And plot the seasonal temperature cycle same as we did above:

```
In [16]: fig = plt.figure( figsize=(20,5) )
         Tmax_highobl = 125; Tmin_highobl = -Tmax_highobl; delT_highobl = 10
         clevels_highobl = np.arange(Tmin_highobl, Tmax_highobl+delT_highobl, delT_highc

         for n in range(num_depths):
             ax = fig.add_subplot(1,num_depths,n+1)
             cax = ax.contourf( 4*np.arange(num_steps_per_year), lat, Tyear_highobl[:,:,
                 levels=clevels_highobl, cmap=plt.cm.seismic, vmin=Tmin_highobl, vmax=Tm
             cbar1 = plt.colorbar(cax)
             ax.set_title('water_depth = ' + str(models_highobl[n].param['water_depth'])
             ax.set_xlabel('Days of year', fontsize=14 )
             ax.set_ylabel('Latitude', fontsize=14 )

         plt.show()
```



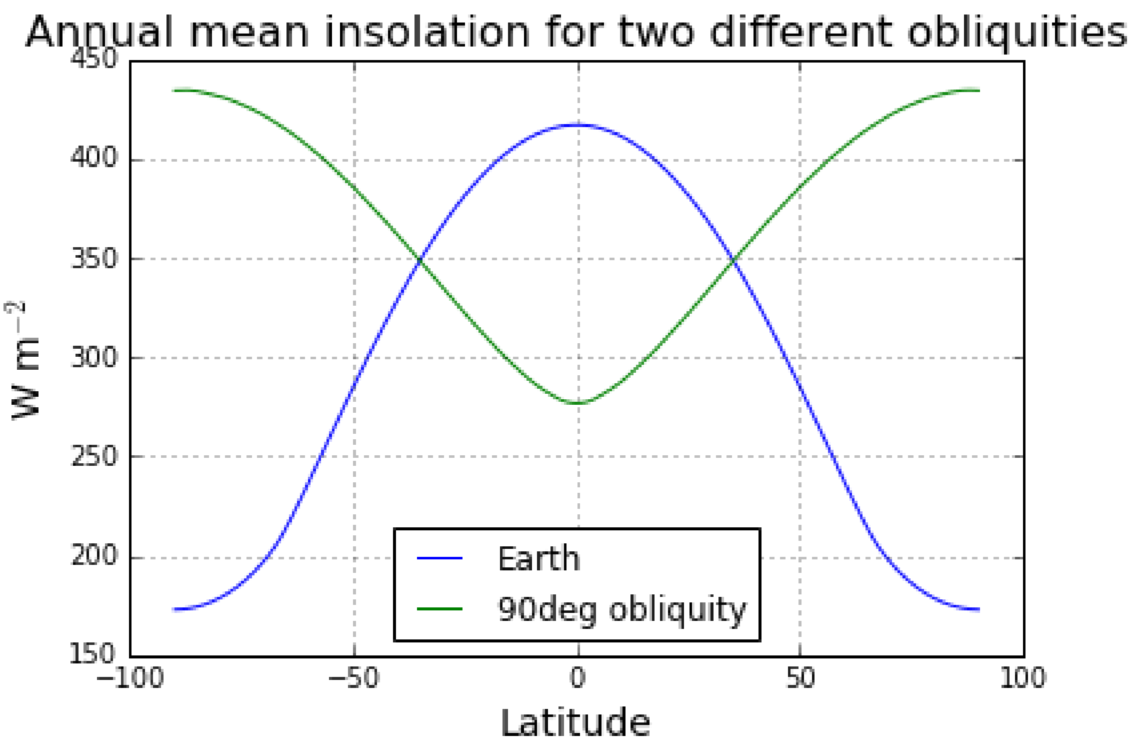
Note that the temperature range is much larger than for the Earth-like case above (but same contour interval, 10 degC).

Why is the temperature so uniform in the north-south direction with 50 meters of water?

To see the reason, let's plot the annual mean insolation at 90º obliquity, alongside the present-day annual mean insolation:

```
In [17]: lat2 = np.linspace(-90, 90, 181)
         days = np.linspace(1.,50.)/50 * const.days_per_year
         Q_present = daily_insolation( lat2, days )
         Q_highobl = daily_insolation( lat2, days, orb_highobl )
         Q_present_ann = np.mean( Q_present, axis=1 )
         Q_highobl_ann = np.mean( Q_highobl, axis=1 )
```

```
In [18]: plt.plot( lat2, Q_present_ann, label='Earth' )
         plt.plot( lat2, Q_highobl_ann, label='90deg obliquity' )
         plt.grid()
         plt.legend(loc='lower center')
         plt.xlabel('Latitude', fontsize=14 )
         plt.ylabel('W m$^{-2}$', fontsize=14 )
         plt.title('Annual mean insolation for two different obliquities', fontsize=16)
         plt.show()
```



Though this is a bit misleading, because our model prescribes an increase in albedo from the equator to the pole. So the absorbed shortwave gradients look even more different.

```
In [ ]:
```

5.6 Ice - Albedo Feedback and runaway glaciation

Here we will use the 1-dimensional diffusive Energy Balance Model (EBM) to explore the effects of albedo feedback and heat transport on climate sensitivity.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import climlab
        from climlab import constants as const
        from climlab import legendre
        from climlab.domain.field import global_mean
```

5.6.1 Annual-mean model with albedo feedback: adjustment to equilibrium

A version of the EBM in which albedo adjusts to the current position of the ice line, wherever $T < T_f$

```
In [2]: model1 = climlab.EBM_annual( num_points = 180, a0=0.3, a2=0.078, ai=0.62)
        print model1
```

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

```
In [3]: model1.integrate_years(5)
        Tequil = np.array(model1.Ts)
        ALBequil = np.array(model1.albedo)
        OLRequil = np.array(model1.OLR)
        ASRequil = np.array(model1.ASR)
```

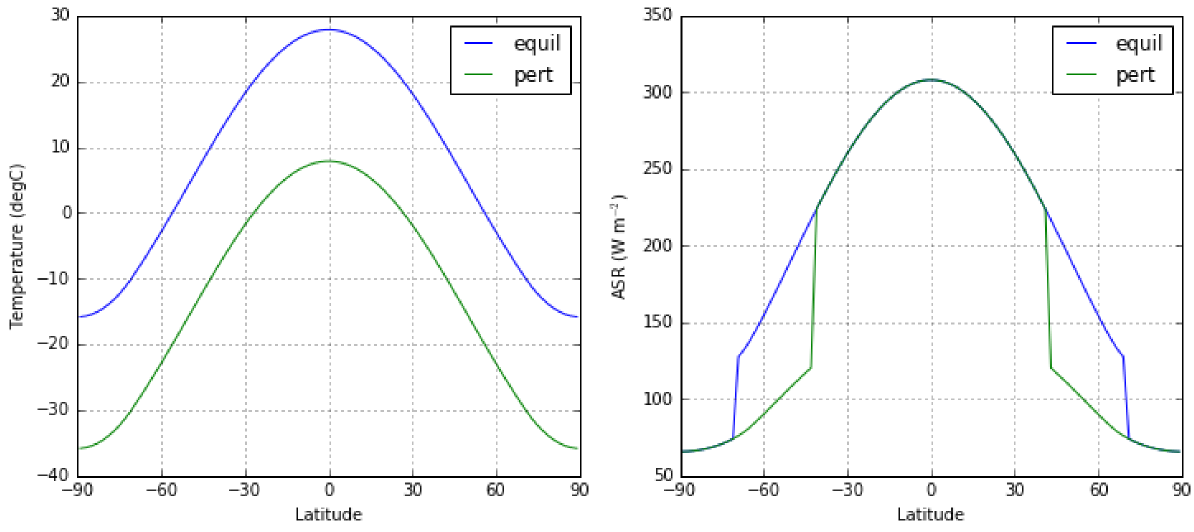
```
Integrating for 450 steps, 1826.211 days, or 5 years.
Total elapsed time is 5.0 years.
```

Let’s look at what happens if we perturb the temperature – make it 20ºC colder everywhere!

```
In [4]: model1.Ts -= 20.
        model1.compute_diagnostics()
```

Let’s take a look at how we have just perturbed the absorbed shortwave:

```
In [5]: my_ticks = [-90,-60,-30,0,30,60,90]
        lat = model1.domains['Ts'].axes['lat'].points

        fig = plt.figure( figsize=(12,5) )

        ax1 = fig.add_subplot(1,2,1)
        ax1.plot(lat, Tequil, label='equil')
        ax1.plot(lat, model1.Ts, label='pert' )
        ax1.grid()
        ax1.legend()
        ax1.set_xlim(-90,90)
        ax1.set_xticks(my_ticks)
        ax1.set_xlabel('Latitude')
        ax1.set_ylabel('Temperature (degC)')

        ax2 = fig.add_subplot(1,2,2)
        ax2.plot( lat, ASRequil, label='equil')
        ax2.plot( lat, model1.ASR, label='pert' )
        ax2.grid()
        ax2.legend()
        ax2.set_xlim(-90,90)
        ax2.set_xticks(my_ticks)
        ax2.set_xlabel('Latitude')
        ax2.set_ylabel('ASR (W m$^{-2}$)')

        plt.show()
```



So there is less absorbed shortwave now, because of the increased albedo. The global mean difference is:

```
In [6]: global_mean( model1.ASR - ASRequil )
```

```
Out[6]: Field(-20.37046205447195)
```

Less shortwave means that there is a tendency for the climate to cool down even more! In other words, the shortwave feedback is **positive**.

Recall that the net feedback for the EBM can be written

$\lambda = -B + \frac{\Delta <(1-\alpha) Q >}{\Delta <T>}$

where the second term is the change in the absorbed shortwave per degree global mean temperature change.

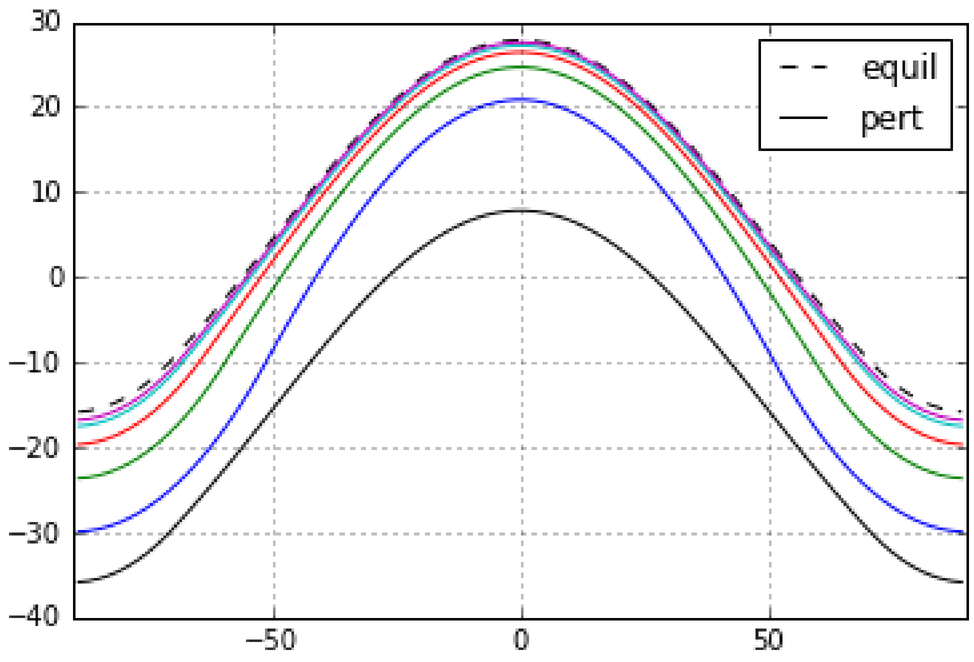
Plugging these numbers in gives

$\lambda = -2 + \frac{-20.4}{-20} = -2 + 1 = -1 \text{ W m}^{-2} \text{ }^{\circ}\text{C}^{-1}$

The feedback is negative, as we expect! The tendency to warm up from reduced OLR outweighs the tendency to cool down from reduced ASR. A negative net feedback means that the system will relax back towards the equilibrium.

Let's let the temperature evolve one year at a time and add extra lines to the graph:

```
In [7]: plt.plot( lat, Tequil, 'k--', label='equil' )
        plt.plot( lat, model1.Ts, 'k-', label='pert' )
        plt.grid()
        plt.xlim(-90,90)
        plt.legend()

        for n in range(5):
            model1.integrate_years(years=1.0, verbose=False)
            plt.plot(lat, model1.Ts)
```



Temperature drifts back towards equilibrium, as we expected!

What if we cool the climate **so much** that the entire planet is ice covered?

```
In [8]: model1.Ts -= 40.
        model1.compute_diagnostics()
```

Look again at the change in absorbed shortwave:

```
In [9]: global_mean( model1.ASR - ASRequil )
```

```
Out[9]: Field(-108.99200830729608)
```

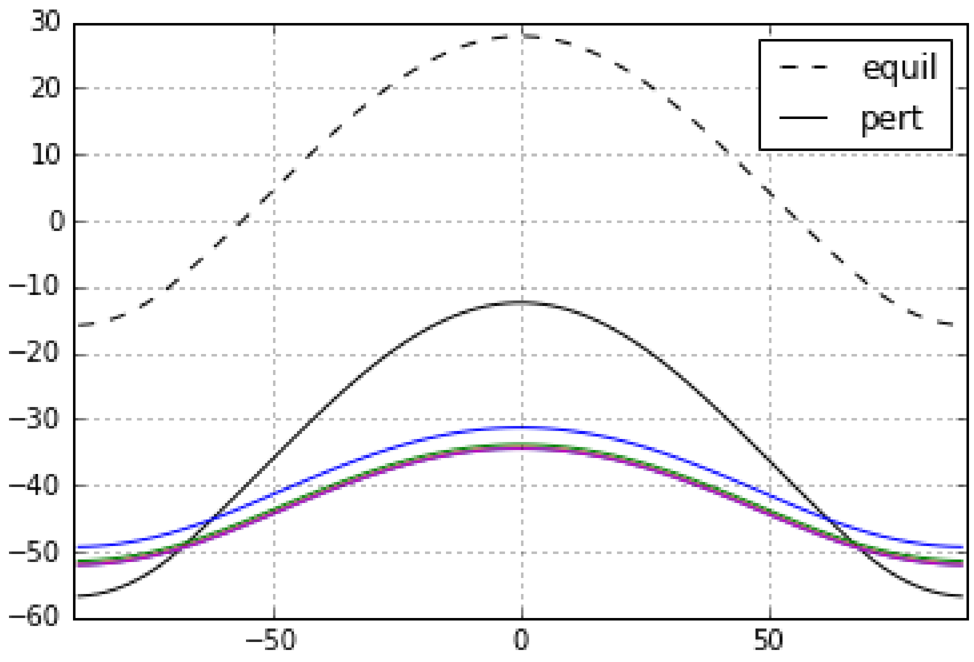
It's much larger because we've covered so much more surface area with ice!

The feedback calculation now looks like

$\lambda = -2 + \frac{-109}{-40} = -2 + 2.7 = +0.7$ W m^{-2} °C^{-1}

What? Looks like the **positive** albedo feedback is so strong here that it has outweighed the **negative** longwave feedback. What will happen to the system now? Let's find out...

```
In [10]: plt.plot( lat, Tequil, 'k--', label='equil' )
         plt.plot( lat, model1.Ts, 'k-', label='pert' )
         plt.grid()
         plt.xlim(-90,90)
         plt.legend()

         for n in range(5):
             model1.integrate_years(years=1.0, verbose=False)
             plt.plot(lat, model1.Ts)
```



Something **very different** happened! The climate drifted towards an entirely different equilibrium state, in which the entire planet is cold and ice-covered.

We will refer to this as the **SNOWBALL EARTH**.

Note that the warmest spot on the planet is still the equator, but it is now about -33ºC rather than +28ºC!

5.6.2 Here Comes the Sun! Where is the ice edge?

The ice edge in our model is always where the temperature crosses $T_f = -10°\mathrm{C}$. The system is at **equilibrium** when the temperature is such that there is a balance between ASR, OLR, and heat transport convergence everywhere.

Suppose that sun was hotter or cooler at different times (in fact it was significantly cooler during early Earth history). That would mean that the solar constant $S_0 = 4Q$ was larger or smaller. We should expect that the temperature (and thus the ice edge) should increase and decrease as we change S_0.

S_0 during the Neoproterozoic Snowball Earth events is believed to be about 93% of its present-day value, or about 1270 W m^{-2}.

We are going to look at how the **equilibrium** ice edge depends on S_0, by integrating the model out to equilibrium for lots of different values of S_0. We will start by slowly decreasing S_0, and then slowly increasing S_0.

```
In [11]: model2 = climlab.EBM_annual(num_points = 360, a0=0.3, a2=0.078, ai=0.62)
```

```
In [12]: S0array = np.linspace(1400., 1200., 200.)
         #S0array = np.linspace(1400., 1200., 10.)
         #print S0array
```

```
In [13]: model2.integrate_years(5)
```

```
Integrating for 450 steps, 1826.211 days, or 5 years.
Total elapsed time is 5.0 years.
```

```
In [14]: print model2.icelat
```

```
[-70.  70.]
```

```
In [15]: icelat_cooling = np.empty_like(S0array)
         icelat_warming = np.empty_like(S0array)
```

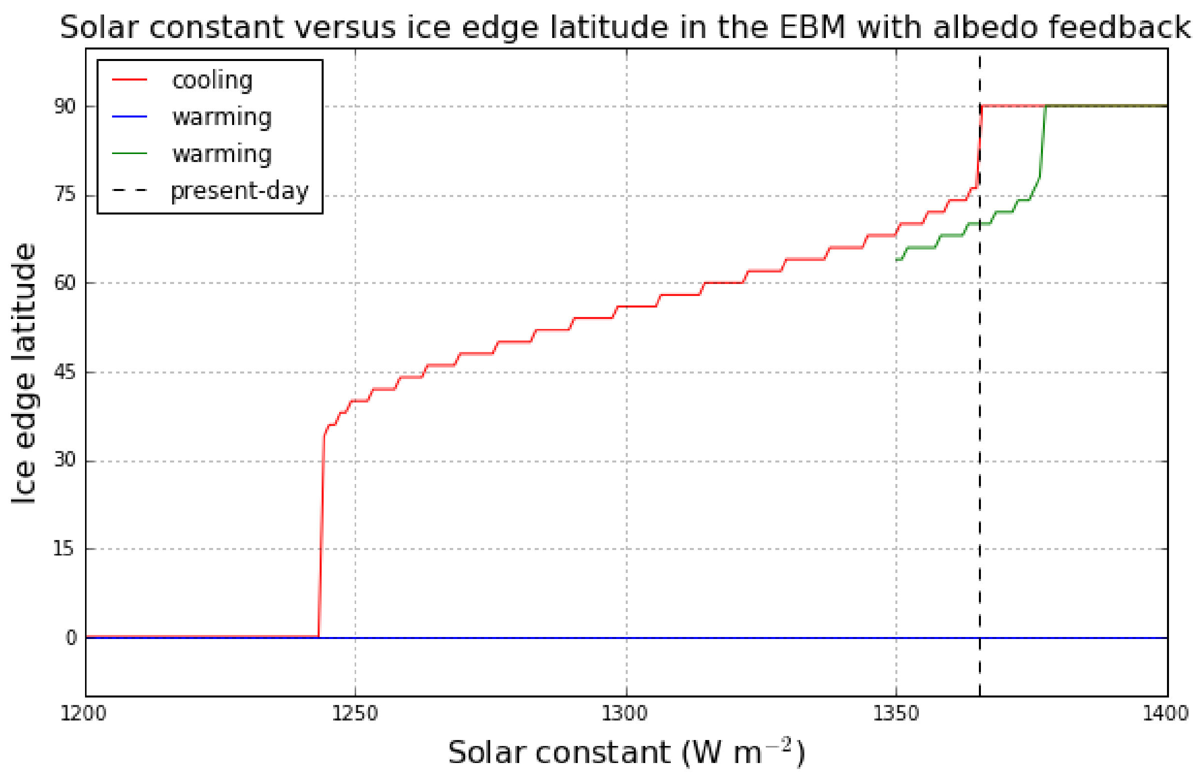
```
In [16]: # First cool....
         for n in range(S0array.size):
             model2.subprocess['insolation'].S0 = S0array[n]
             model2.integrate_years(10, verbose=False)
             icelat_cooling[n] = np.max(model2.icelat)
         # Then warm...
         for n in range(S0array.size):
             model2.subprocess['insolation'].S0 = np.flipud(S0array)[n]
             model2.integrate_years(10, verbose=False)
             icelat_warming[n] = np.max(model2.icelat)
```

For completeness: also start from present-day conditions and warm up.

```
In [17]: model3 = climlab.EBM_annual(num_points = 360, a0=0.3, a2=0.078, ai=0.62)
         S0array3 = np.linspace(1350., 1400., 50.)
         #S0array3 = np.linspace(1350., 1400., 5.)
         icelat3 = np.empty_like(S0array3)
```

```
In [18]: for n in range(S0array3.size):
             model3.subprocess['insolation'].S0 = S0array3[n]
             model3.integrate_years(10, verbose=False)
             icelat3[n] = np.max(model3.icelat)
```

```
In [19]: fig = plt.figure( figsize=(10,6) )
         ax = fig.add_subplot(111)
         ax.plot(S0array, icelat_cooling, 'r-', label='cooling' )
         ax.plot(S0array, icelat_warming, 'b-', label='warming' )
         ax.plot(S0array3, icelat3, 'g-', label='warming' )
         ax.set_ylim(-10,100)
         ax.set_yticks((0,15,30,45,60,75,90))
         ax.grid()
         ax.set_ylabel('Ice edge latitude', fontsize=16)
         ax.set_xlabel('Solar constant (W m$^{-2}$)', fontsize=16)
         ax.plot( [const.S0, const.S0], [-10, 100], 'k--', label='present-day' )
         ax.legend(loc='upper left')
         ax.set_title('Solar constant versus ice edge latitude in the EBM with albedo fe
         plt.show()
```



There are actually up to 3 different climates possible for a given value of S_0!

How to un-freeze the Snowball

The graph indicates that if the Earth were completely frozen over, it would be perfectly happy to stay that way even if the sun were brighter and hotter than it is today.

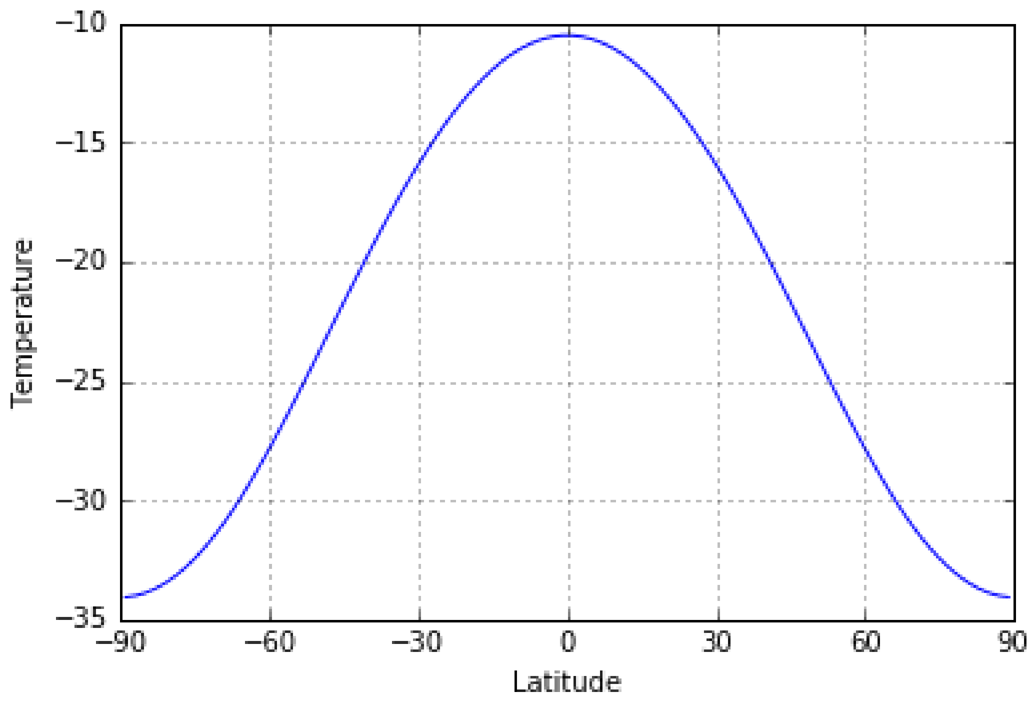
Our EBM predicts that (with present-day parameters) the equilibrium temperature at the equator in the Snowball state is about -33ºC, which is much colder than the threshold temperature $T_f = -10°$C. How can we melt the Snowball?

We need to increase the avaible energy sufficiently to get the equatorial temperatures above this threshold! That is going to require a much larger increase in S_0 (could also increase the greenhouse gases, which would have a similar effect)!

Let's crank up the sun to 1830 W m^{-2} (about a 34% increase from present-day).

```
In [20]: from climlab.process.process import process_like

         model4 = process_like(model2)  # initialize with cold Snowball temperature
         model4.subprocess['insolation'].S0 = 1830.
         model4.integrate_years(40)

         #lat = model4.domains['Ts'].axes['lat'].points
         plt.plot(model4.lat, model4.Ts)
         plt.xlim(-90,90)
         plt.ylabel('Temperature')
         plt.xlabel('Latitude')
         plt.grid()
         plt.xticks(my_ticks)
         plt.show()

         print('The ice edge is at ' + str(model4.icelat) + 'degrees latitude.' )
```

```
Integrating for 3600 steps, 14609.688 days, or 40 years.
Total elapsed time is 4044.99999998 years.
```



```
The ice edge is at [-0.  0.]degrees latitude.
```

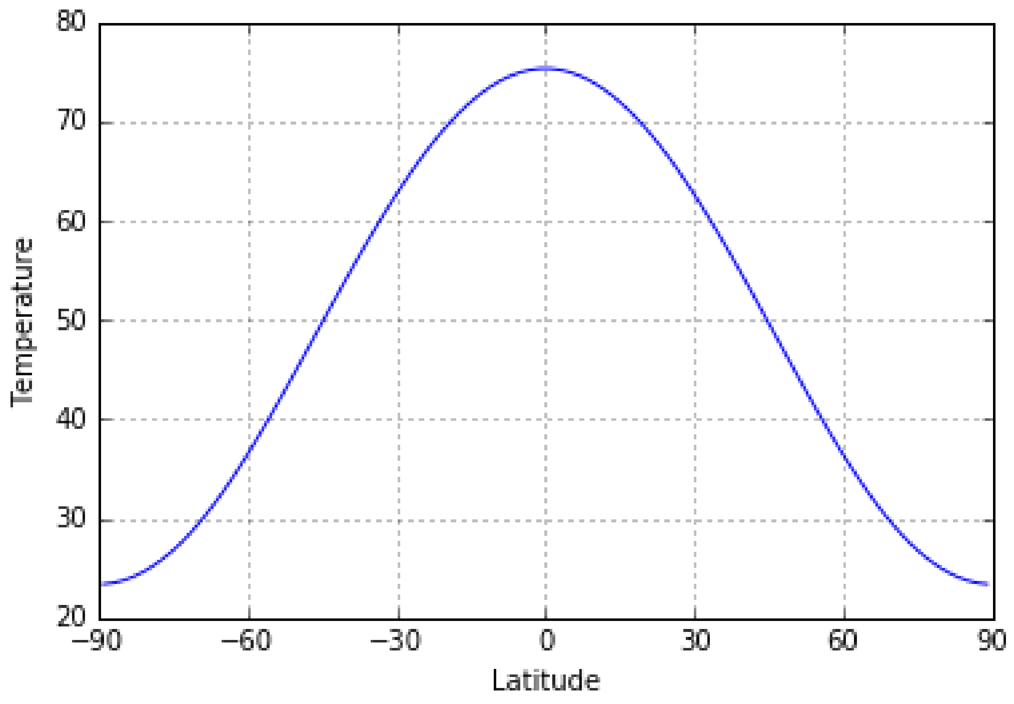
Still a Snowball... but just barely! The temperature at the equator is just below the threshold.

Try to imagine what might happen once it starts to melt. The solar constant is huge, and if it weren't for the highly reflective ice and snow, the climate would be really really hot!

We're going to increase S_0 one more time...

```
In [21]: model4.subprocess['insolation'].S0 = 1845.
         model4.integrate_years(10)

         plt.plot(lat, model4.state['Ts'])
         plt.xlim(-90,90)
         plt.ylabel('Temperature')
         plt.xlabel('Latitude')
         plt.grid()
         plt.xticks(my_ticks)
         plt.show()
```

```
Integrating for 900 steps, 3652.422 days, or 10 years.
Total elapsed time is 4054.99999998 years.
```



Suddenly the climate looks very very different again! The global mean temperature is

```
In [22]: print( model4.global_mean_temperature() )
```

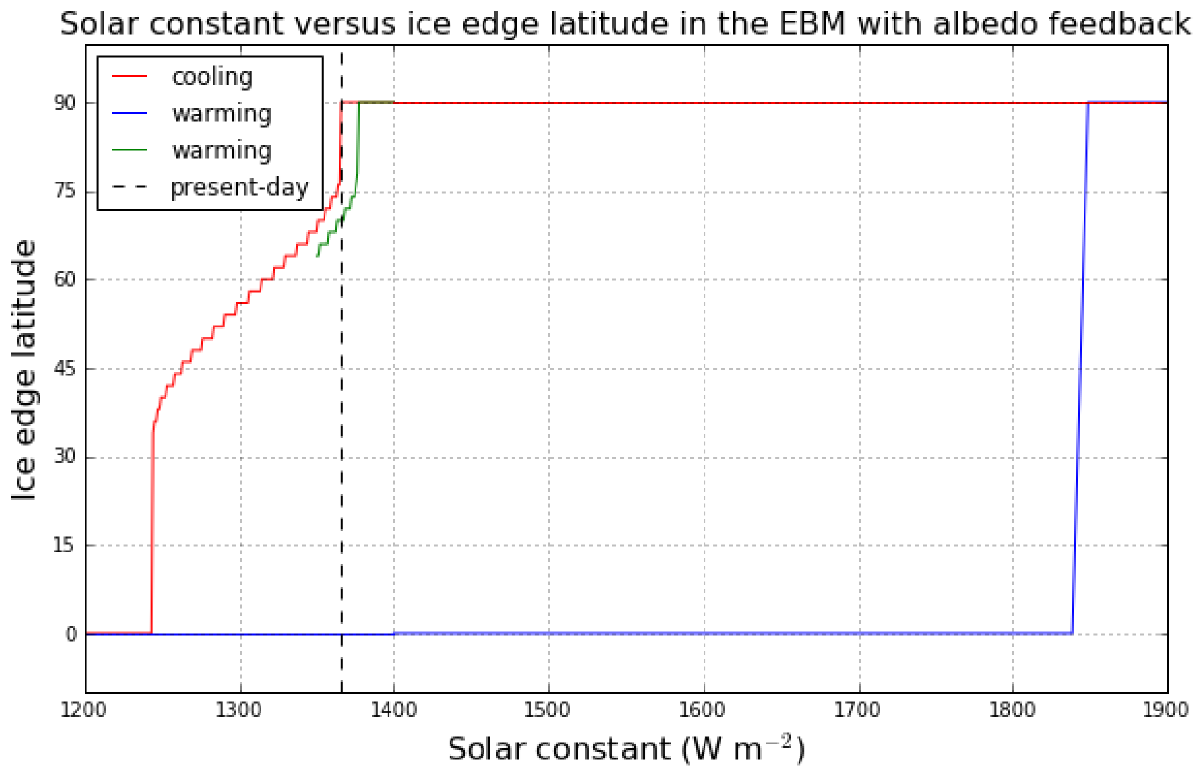
```
58.171701295
```

A roasty 60ºC, and the poles are above 20ºC. A tiny increase in S_0 has led to a very drastic change in the climate.

```
In [23]: S0array_snowballmelt = np.linspace(1400., 1900., 50)
         icelat_snowballmelt = np.empty_like(S0array_snowballmelt)
         icelat_snowballmelt_cooling = np.empty_like(S0array_snowballmelt)

         for n in range(S0array_snowballmelt.size):
             model2.subprocess['insolation'].S0 = S0array_snowballmelt[n]
             model2.integrate_years(10, verbose=False)
             icelat_snowballmelt[n] = np.max(model2.diagnostics['icelat'])

         for n in range(S0array_snowballmelt.size):
             model2.subprocess['insolation'].S0 = np.flipud(S0array_snowballmelt)[n]
             model2.integrate_years(10, verbose=False)
             icelat_snowballmelt_cooling[n] = np.max(model2.diagnostics['icelat'])
```

Now we will complete the plot of ice edge versus solar constant.

```
In [24]: fig = plt.figure( figsize=(10,6) )
         ax = fig.add_subplot(111)
         ax.plot(S0array, icelat_cooling, 'r-', label='cooling' )
         ax.plot(S0array, icelat_warming, 'b-', label='warming' )
         ax.plot(S0array3, icelat3, 'g-', label='warming' )
         ax.plot(S0array_snowballmelt, icelat_snowballmelt, 'b-' )
         ax.plot(S0array_snowballmelt, icelat_snowballmelt_cooling, 'r-' )
         ax.set_ylim(-10,100)
         ax.set_yticks((0,15,30,45,60,75,90))
         ax.grid()
         ax.set_ylabel('Ice edge latitude', fontsize=16)
         ax.set_xlabel('Solar constant (W m$^{-2}$)', fontsize=16)
         ax.plot( [const.S0, const.S0], [-10, 100], 'k--', label='present-day' )
         ax.legend(loc='upper left')
         ax.set_title('Solar constant versus ice edge latitude in the EBM with albedo fe
         plt.show()
```



The upshot:

- For extremely large S_0, the only possible climate is a hot Earth with no ice.
- For extremely small S_0, the only possible climate is a cold Earth completely covered in ice.
- For a large range of S_0 including the present-day value, more than one climate is possible!
- Once we get into a Snowball Earth state, getting out again is rather difficult!

```
In [ ]:
```

CHAPTER

SIX

APPLICATION PROGRAMMING INTERFACE

This chapter documents the source code of the `climlab` package. The focus is on the methods and functions that the user invokes while using the package. Nevertheless also the underlying code of the `climlab` architecture has been documented for a comprehensive understanding and traceability.

Until now only the Energy Balance Model revelent parts of `climlab` have been covered.

6.1 Subpackages

6.1.1 climlab.domain package

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`* (page 55) is an object where information of a spacial dimension of a *`_Domain`* (page 59) are specified.

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

The *axes* of a *`_Domain`* (page 59) are stored in the dictionary axes, so they can be accessed through `dom.axes` if `dom` is an instance of *`_Domain`* (page 59).

Initialization parameters

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

Parameters

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

- **bounds** (*array*) – array with specific bounds between points (optional)

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

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

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

Object attributes

Following object attributes are generated during initialization:

Variables

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

Axis Types

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

- **`'lev'`**
  - `'p'`
  - `'press'`
  - `'pressure'`
  - `'P'`
  - `'Pressure'`
  - `'Press'`
- **`'lat'`**
  - `'Latitude'`
  - `'latitude'`
- **`'lon'`**
  - `'Longitude'`
  - `'longitude'`
- **`'depth'`**
  - `'Depth'`
  - `'waterDepth'`
  - `'water_depth'`
  - `'slab'`

The **default units** are:

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

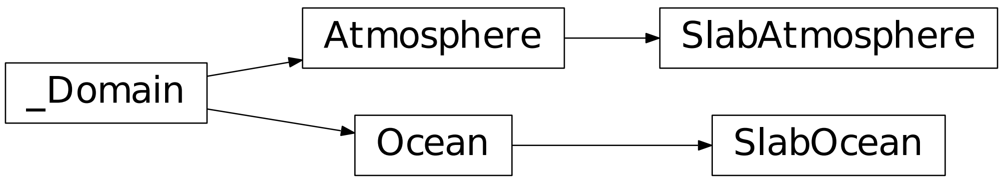
If bounds are not given during initialization, **default end points** are used:

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

Example Creation of a standalone Axis:

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

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

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

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

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

climlab.domain.domain module



class `climlab.domain.domain.`**`Atmosphere`**`(**kwargs)`
Bases: *`climlab.domain.domain._Domain`* (page 59)

Class for the implementation of an Atmosphere Domain.

Object attributes

Additional to the parent class `_Domain` (page 59) the following object attribute is modified during initialization:

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

Example Setting up an Atmosphere Domain:

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

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

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

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

`set_heat_capacity`()

Sets the heat capacity of the Atmosphere Domain.

Calls the utils heat capacity function *`atmosphere()`* (page 116) and gives the delta array of grid points of it's level axis `self.axes['lev'].delta` as input.

Object attributes

During method execution following object attribute is modified:

Variables ***`heat_capacity`*** (page 116) (*`array`*) – the ocean domain's heat capacity over the `'lev'` Axis.

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

Bases: *`climlab.domain.domain._Domain`* (page 59)

Class for the implementation of an Ocean Domain.

Object attributes

Additional to the parent class `_Domain` (page 59) the following object attribute is modified during initialization:

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

Example Setting up an Ocean Domain:

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

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

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

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

`set_heat_capacity`()

Sets the heat capacity of the Ocean Domain.

Calls the utils heat capacity function `ocean()` (page 116) and gives the delta array of grid points of it's depth axis `self.axes['depth'].delta` as input.

Object attributes

During method execution following object attribute is modified:

Variables ***`heat_capacity`*** (page 116) (*`array`*) – the ocean domain's heat capacity over the `'depth'` Axis.

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

Bases: `climlab.domain.domain.Atmosphere` (page 57)

A class to create a SlabAtmosphere Domain by default.

Initializes the parent `Atmosphere` (page 57) class with a simple axis for a Slab Atmopshere created by `make_slabatm_axis()` (page 61) which has just 1 cell in height by default.

Example Creating a SlabAtmosphere Domain:

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

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

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

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

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

Bases: `climlab.domain.domain.Ocean` (page 58)

A class to create a SlabOcean Domain by default.

Initializes the parent `Ocean` (page 58) class with a simple axis for a Slab Ocean created by `make_slabocean_axis()` (page 61) which has just 1 cell in depth by default.

Example Creating a SlabOcean Domain:

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

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

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

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

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

Bases: `object`

Private parent class for *Domains*.

A *Domain* defines an area or spatial base for a climlab `Process` (page 84) object. It consists of axes which are `Axis` (page 55) objects that define the dimensions of the *Domain*.

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

There are daughter classes `Atmosphere` (page 57) and `Ocean` (page 58) of the private `_Domain` (page 59) class implemented which themselves have daughter classes `SlabAtmosphere` (page 59) and `SlabOcean` (page 59).

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

- `single_column()` (page 62)
- `zonal_mean_column()` (page 62)
- `box_model_domain()` (page 60)

Initialization parameters

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

Parameters **`axes`** (dict or `Axis` (page 55)) – Axis object or dictionary of Axis object where domain will be defined on.

Object attributes

Following object attributes are generated during initialization:

Variables

- **`domain_type`** (`str`) – Set to `'undefined'`.
- **`axes`** (`dict`) – A dictionary of the domains axes. Created by `_make_axes_dict()` (page 60) called with input argument `axes`
- **`numdims`** (`int`) – Number of `Axis` (page 55) objects in `self.axes` dictionary.
- **`ax_index`** (`dict`) – A dictionary of domain axes and their corresponding index in an ordered list of the axes with:
  - `'lev'` or `'depth'` is last
  - `'lat'` is second last
- **`shape`** (`tuple`) – Number of points of all domain axes. Order in tuple given by `self.ax_index`.
- ***`heat_capacity`*** (page 116) (`array`) – the domain's heat capacity over axis specified in function call of `set_heat_capacity()` (page 60)

`_make_axes_dict`(*axes*)

Makes an axes dictionary.

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

Function-call argument

Parameters **`axes`** (dict or single instance of `Axis` (page 55) object or `None`) – axes input

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

Returns dictionary of input axes

Return type dict

`set_heat_capacity`()

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

Should be overridden by daugter classes.

`climlab.domain.domain.`**`box_model_domain`**(*num_points=2*, ***kwargs*)

Creates a box model domain (a single abstract axis).

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

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

Return type *`_Domain`* (page 59)

Example

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

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

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

Function-call argument

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

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

Return type *`Axis`* (page 55)

Example

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

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

>>> slab_atm_axis.axis_type
'lev'

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

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

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

Function-call argument

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

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

Return type *`Axis`* (page 55)

Example

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

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

>>> slab_ocean_axis.axis_type
'depth'

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

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

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

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

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

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

Function-call argument

Parameters

- **num_lev** (*int*) – number of pressure levels (evenly spaced from surface to TOA) [default: 30]
- **water_depth** (*float*) – depth of the ocean slab [default: 1.]
- **lev** (*Axis* (page 55) or pressure array) – specification for height axis (optional)

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

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

Return type `list` of *SlabOcean* (page 59), *SlabAtmosphere* (page 59)

Example

```
>>> from climlab import domain

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

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

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

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

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

- SlabOcean: one water cell and a latitude axis above (similar to *zonal_mean_surface()* (page 63))
- Atmosphere: a latitude axis and a level/height axis (two dimensional)

Function-call argument

Parameters

- **num_lat** (*int*) – number of latitude points on the axis [default: 90]
- **num_lev** (*int*) – number of pressure levels (evenly spaced from surface to TOA) [default: 30]
- **water_depth** (*float*) – depth of the water cell (slab ocean) [default: 10.]
- **lat** (*Axis* (page 55) or latitude array) – specification for latitude axis (optional)
- **lev** (*Axis* (page 55) or pressure array) – specification for height axis (optional)

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

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

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

Return type `list` of *SlabOcean* (page 59), *Atmosphere* (page 57)

Example

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

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

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

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

Creates a 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`* (page 55) or latitude array) – specification for latitude axis (optional)

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

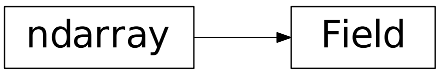
Returns surface domain

Return type *`SlabOcean`* (page 59)

Example

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

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

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* (page 59)) – the domain associated with that field (e.g. state variables)

Object attributes

Following object attribute is generated during initialization:

Variables ***domain*** (page 57) (*_Domain* (page 59)) – the domain associated with that field (e.g. state variables)

Example

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

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

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

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

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

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

>>> print s2
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

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

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

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

Parameters **field** (Field (page 63)) – input field

Raises ValueError if input field has no latitude axis

Returns latitude weighted global mean of the field

Return type float

Example initial global mean temperature of EBM model:

```
>>> import climlab
>>> from climlab.domain.field import global_mean

>>> model = climlab.EBM()

>>> global_mean(model.Ts)
Field(11.997968598413685)
```

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 top of atmosphere) [default: 30]
- **`num_lat`** (*`int`*) – number of latitude points on the axis [default: 1]
- **`lev`** (*`Axis`* (page 55) or pressure array) – specification for height axis (optional)
- **`lat`** (*`array`*) – size of array determines dimension of latitude (optional)
- **`water_depth`** (*`float`*) – *irrelevant*

Returns dictionary with two temperature *`Field`* (page 63) for atmospheric column `Tatm` and surface mixed layer `Ts`

Return type dict

Example

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

>>> print T_dict
{'Tatm': Field([ 200.        ,  202.68965517,  205.37931034,  208.06896552,
        210.75862069,  213.44827586,  216.13793103,  218.82758621,
        221.51724138,  224.20689655,  226.89655172,  229.5862069 ,
        232.27586207,  234.96551724,  237.65517241,  240.34482759,
        243.03448276,  245.72413793,  248.4137931 ,  251.10344828,
        253.79310345,  256.48275862,  259.17241379,  261.86206897,
        264.55172414,  267.24137931,  269.93103448,  272.62068966,
        275.31034483,  278.        ]), 'Ts': Field([ 288.])}
```

`climlab.domain.initial.`**`surface_state`**(*num_lat=90*, *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* (page 117) to calculate initial temperature
  - unit: dimensionless
  - default value: `-40`

Returns dictionary with temperature *Field* (page 63) for surface mixed layer `Ts`

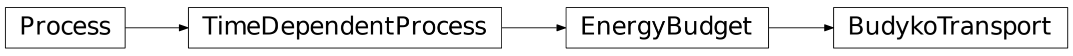
Return type dict

Example

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

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

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

6.1.2 climlab.dynamics package

climlab.dynamics.budyko_transport module



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

Bases: *climlab.process.energy_budget.EnergyBudget* (page 82)

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

Parameters b (*float*) – budyko transport parameter

- unit: $\text{W}/\left(\text{m}^2\ ^\circ\text{C}\right)$
- default value: `3.81`

As BudykoTransport is a *Process* (page 84) it needs a state do be defined on. See example for details.

Computation Details:

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

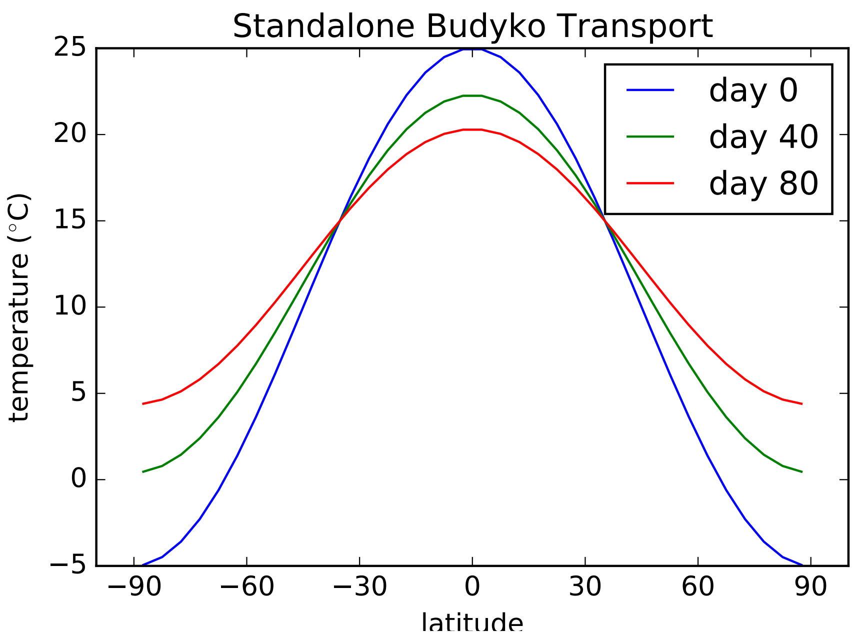
$$H = b[T - \bar{T}]$$

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

For further information see [Budyko_1969].

Example Budyko Transport as a standalone process:

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

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

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

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

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

### Integrate & Plot ###

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

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

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



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

$$\begin{tikzcd}\boxed{\text{Process}} \arrow[r] & \boxed{\text{TimeDependentProcess}} \arrow[r] & \boxed{\text{ImplicitProcess}} \arrow[r] & \boxed{\text{Diffusion}} \arrow[r] & \boxed{\text{MeridionalDiffusion}}\end{tikzcd}$$

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

Bases: *`climlab.process.implicit.ImplicitProcess`* (page 83)

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

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

Object attributes

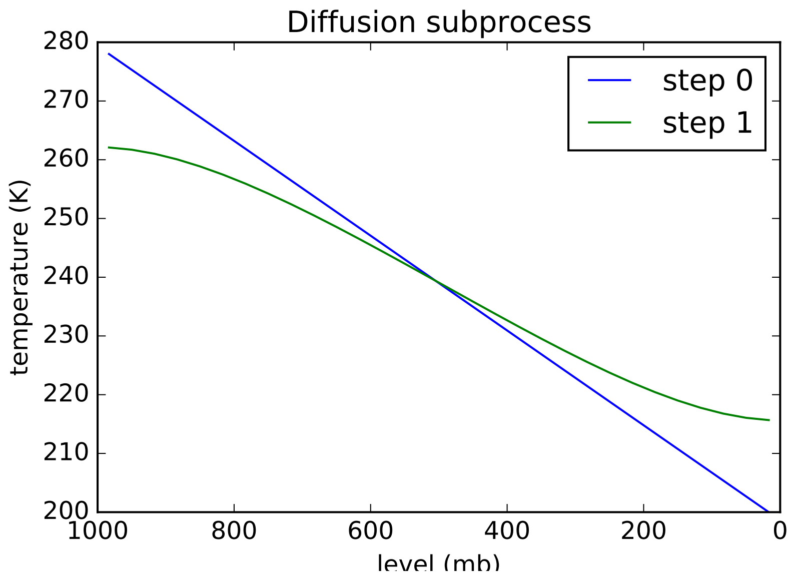
Additional to the parent class *ImplicitProcess* (page 83) following object attributes are generated or modified during initialization:

Variables

- **param** (*dict*) – parameter dictionary is extended by diffusivity parameter K (unit: $\frac{[\text{length}]^2}{\text{time}}$)
- **use_banded_solver** (*bool*) – input flag specifying numerical solving method (given during initialization)
- **diffusion_axis** (*str*) – dictionary key for axis where diffusion is occuring: specified during initialization or output of method *_guess_diffusion_axis()* (page 72)
- **K_dimensionless** (*array*) – diffusion parameter K multiplied by the timestep and divided by mean of diffusion axis delta in the power of two. Array has the size of diffusion axis bounds. $K_{\text{dimensionless}}[i] = K\frac{\Delta t}{(\Delta \text{bounds})^2}$
- **diffTriDiag** (*array*) – tridiagonal diffusion matrix made by *_make_diffusion_matrix()* (page 72) with input `self.K_dimensionless`

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

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

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

c.add_subprocess('diffusion',d)

### Integrate & Plot ###

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

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

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



_implicit_solver()

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

The method is called by the *_compute()* (page 83) function of the *ImplicitProcess* (page 83) class and solves the matrix problem

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

for diffusion matrix A and corresponding temperatures. T_{old} is in this case the current state variable which already has been adjusted by the explicit processes. T_{new} is the new state of the variable. To derive the temperature tendency of the diffusion process the adjustment has to be calculated and muliplied with the timestep which is done by the *_compute()* (page 83) function of the *ImplicitProcess* (page 83) class.

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

Variables

- **state** (*dict*) – method uses current state variables but does not modify them
- **use_banded_solver** (*bool*) – input flag whether to use *_solve_implicit_banded()* (page 73) 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* (page 68)

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

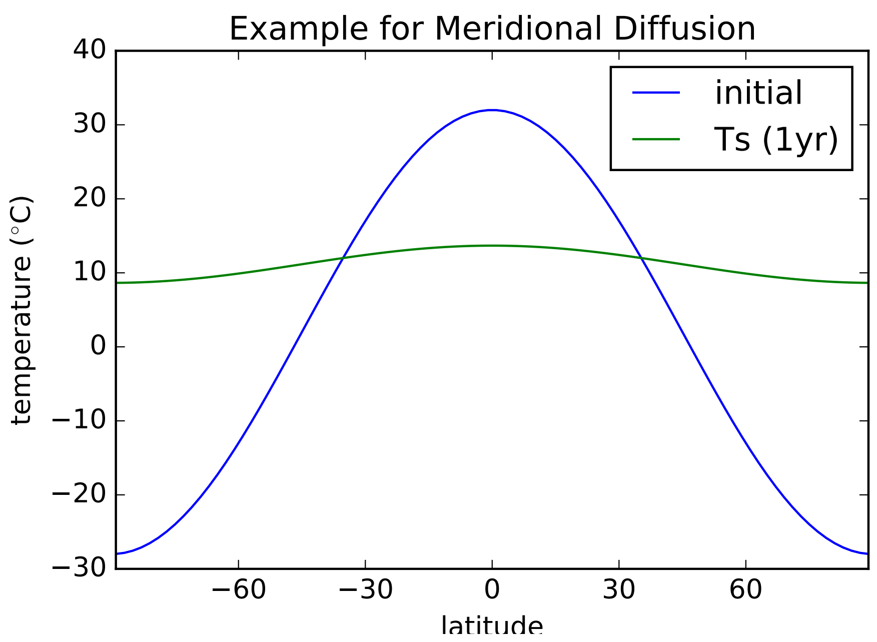
Additional to the parent class *Diffusion* (page 68) which is initialized with `diffusion_axis='lat'`, following object attributes are modified during initialization:

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()* (page 73)

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

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

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

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

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

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

d.integrate_years(1.)

import matplotlib.pyplot as plt

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



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

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

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

Parameters **process_or_domain** (*Process* (page 84), *_Domain* (page 59) 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

$$\text{diffTriDiag} = \begin{bmatrix} 1+\frac{s_1}{w_{2,0}} & -\frac{s_1}{w_{2,0}} & 0 & & ... & 0 \\ -\frac{s_1}{w_{2,1}} & 1+\frac{s_1+s_2}{w_{2,1}} & -\frac{s_2}{w_{2,1}} & 0 & ... & 0 \\ 0 & -\frac{s_2}{w_{2,2}} & 1+\frac{s_2+s_3}{w_{2,2}} & -\frac{s_3}{w_{2,2}} & ... & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & 0 & ... & -\frac{s_{n-2}}{w_{2,n-2}} & 1+\frac{s_{n-2}+s_{n-1}}{w_{2,n-2}} & -\frac{s_{n-1}}{w_{2,n-2}} \\ 0 & 0 & ... & 0 & -\frac{s_{n-1}}{w_{2,n-1}} & 1+\frac{s_{n-1}}{w_{2,n-1}} \end{bmatrix}$$

where

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

and following subsitute:

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

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

Calls `_make_diffusion_matrix()` (page 72) with appropriate weights for the meridional diffusion case.

Parameters

- **`K`** (`array`) – dimensionless diffusivities at cell boundaries of diffusion axis `lataxis`
- **`lataxis`** (`axis` (page 55)) – 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()` (page 72) results in a matrix like:

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

with the substitue of:

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

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

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

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

Parameters

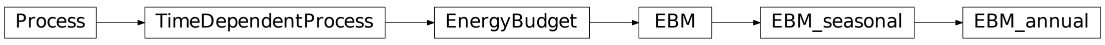
- **current** (*array*) – the current state of the variable for which matrix inversion should be computed
- **banded_matrix** (*array*) – complete diffusion matrix (*dimension: nxn*)

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

Return type array

6.1.3 climlab.model package

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`* (page 82)

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

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

•Outgoing Longwave Radiation (OLR) parametrization through *`AplusBT`* (page 93)

•solar insolation paramtrization through *`P2Insolation`* (page 103)

•albedo parametrization in dependence of temperature through *`StepFunctionAlbedo`* (page 113)

•energy diffusion through *`MeridionalDiffusion`* (page 70)

Initialization parameters

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

Parameters

- **num_lat** (*int*) – number of equally spaced points for the latitue grid. Used for domain intialization of *`zonal_mean_surface`* (page 63)
  - default value: `90`
- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `1365.2`
- **A** (*float*) – parameter for linear OLR parametrization *`AplusBT`* (page 93)
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `210.0`
- **B** (*float*) – parameter for linear OLR parametrization *`AplusBT`* (page 93)
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2\ ^\circ\mathrm{C}}$
  - default value: `2.0`
- **D** (*float*) – diffusion parameter for Meridional Energy Diffusion *`MeridionalDiffusion`* (page 70)
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2\ ^\circ\mathrm{C}}$
  - default value: `0.555`

- **`water_depth`** (*`float`*) – depth of *`zonal_mean_surface`* (page 63) 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 parametrization *`StepFunctionAlbedo`* (page 113)
  - unit: dimensionless
  - default value: `0.3`
- **`a2`** (*`float`*) – parabolic value for planetary albedo parametrization *`StepFunctionAlbedo`* (page 113)
  - unit: dimensionless
  - default value: `0.078`
- **`ai`** (*`float`*) – value for ice albedo paramerization in *`StepFunctionAlbedo`* (page 113)
  - 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`* (page 117) to calculate initial temperature
  - unit: dimensionless
  - default value: `40`

Object attributes

Additional to the parent class *`EnergyBudget`* (page 82) following object attributes are generated and updated during initialization:

Variables

- **`param`** (*`dict`*) – The parameter dictionary is updated with a couple of the initatilzation input arguments, namely `'S0'`, `'A'`, `'B'`, `'D'`, `'Tf'`, `'water_depth'`, `'a0'`, `'a2'` and `'ai'`.
- **`domains`** (*`dict`*) – If the object's `domains` and the `state` dictionaries are empty during initialization a domain `sfc` is created through *`zonal_mean_surface()`* (page 63). In the meantime the object's `domains` and `state` dictionaries are updated.
- **`subprocess`** (*`dict`*) – Several subprocesses are created (see above) through calling *`add_subprocess()`* (page 85) and therefore the subprocess dictionary is updated.

- **topdown** (*bool*) – is set to `False` to call subprocess compute methods first. See also *TimeDependentProcess* (page 90).
- **diagnostics** (*dict*) – is initialized with keys: `'OLR'`, `'ASR'`, `'net_radiation'`, `'albedo'` and `'icelat'` through *init_diagnostic()* (page 86).

Example Creation and integration of the preconfigured Energy Balance Model:

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

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

For more information how to use the EBM class, see the *Tutorials* (page 15) chapter.

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()* (page 64) method which for the object attriute `Ts` which calculates the latitude weighted global mean of a field.

Example Calculating the global mean temperature of initial EBM temperature:

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

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

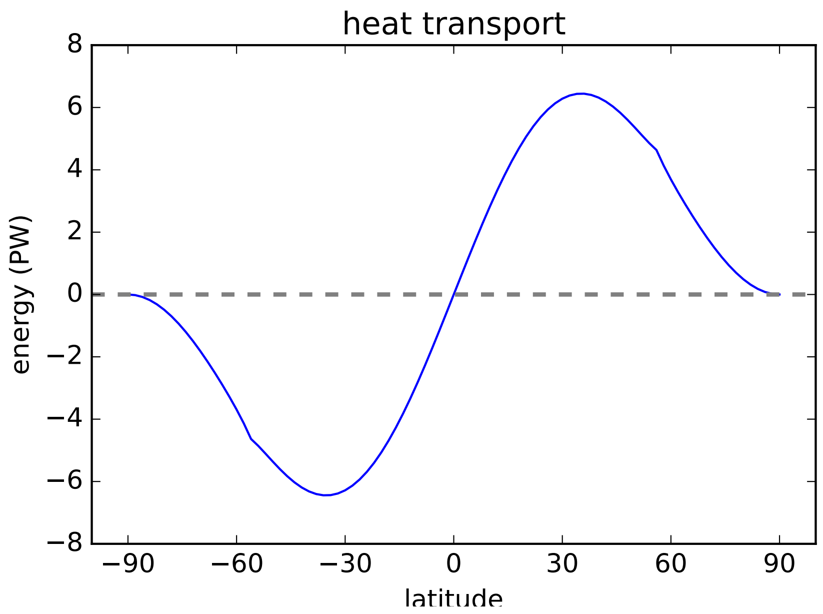
heat_transport()

Returns instantaneous heat transport in unit PW on the staggered grid (bounds) through calling *diffusive_heat_transport()* (page 76).

Example

```
import climlab
import matplotlib.pyplot as plt

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

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

bounds = model.domains['Ts'].axes['lat'].bounds
ax.plot(bounds, model.heat_transport())

ax.set_title('heat transport')
ax.set_xlabel('latitude')
ax.set_xticks([-90,-60,-30,0,30,60,90])
```

```
                ax.set_ylabel('energy (PW)')
                plt.axhline(linewidth=2, color='grey', linestyle='dashed')
                plt.show()
```



heat_transport_convergence()

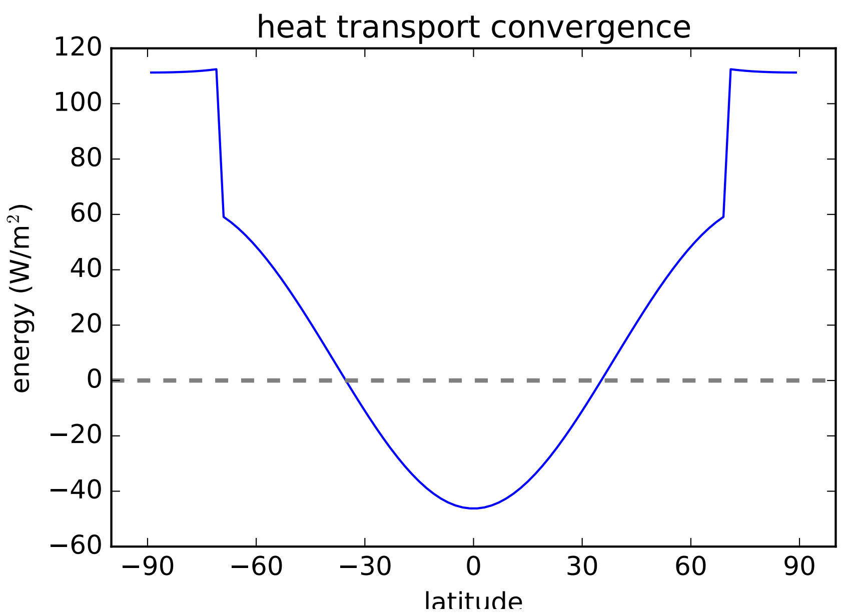
Returns instantaneous convergence of heat transport.

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

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.

Example

```
                import climlab
                import matplotlib.pyplot as plt

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

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

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

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



inferred_heat_transport()

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

The method is calculating

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

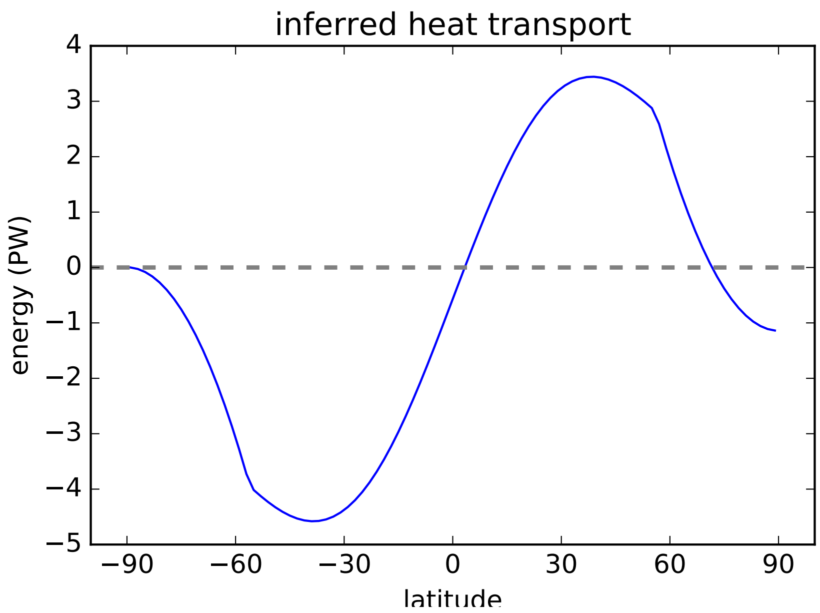
where R_{TOA} is the net radiation at top of atmosphere.

Returns total heat transport on the latitude grid in unit PW

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

Example

```
import climlab
import matplotlib.pyplot as plt

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

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

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

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



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

Bases: *`climlab.model.ebm.EBM_seasonal`* (page 79)

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

The annual solar distribution is calculated through averaging the *`DailyInsolation`* (page 101) over time which has been used in used in the parent class *`EBM_seasonal`* (page 79). That is done by the subprocess *`AnnualMeanInsolation`* (page 99) which is more realistic than the *`P2Insolation`* (page 103) module used in the classical *`EBM`* (page 74) class.

According to the parent class *`EBM_seasonal`* (page 79) 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`* (page 99)

Example The *`EBM_annual`* (page 79) class uses a different insolation subprocess than the *`EBM`* (page 74) class:

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

>>> print model_annual
```

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

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

Bases: *`climlab.model.ebm.EBM`* (page 74)

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

This class is inherited from the general `EBM` (page 74) class and uses the insolation subprocess `DailyInsolation` (page 101) instead of `P2Insolation` (page 103) to compute a realisitc distribution of solar radiation on a daily basis.

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

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

Parameters

- **a0** (*float*) – base value for planetary albedo parametrization `StepFunctionAlbedo` (page 113) [default: 0.33]
- **a2** (*float*) – parabolic value for planetary albedo parametrization `StepFunctionAlbedo` (page 113) [default: 0.25]
- **ai** (*float*) – value for ice albedo paramerization in `StepFunctionAlbedo` (page 113) (optional)

Object attributes

Following object attributes are updated during initialization:

Variables

- **param** (*dict*) – The parameter dictionary is updated with `'a0'` and `'a2'`.
- **subprocess** (*dict*) – suprocess `'insolation'` is overwritten by `DailyInsolation` (page 101).

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

Variables

- **param** (*dict*) – `'ai'` and `'Tf'` are removed from the parameter dictionary (initialized by parent class `EBM` (page 74))
- **subprocess** (*dict*) – suprocess `'albedo'` is overwritten by `P2Albedo` (page 112).

if `'ai'` *is given*:

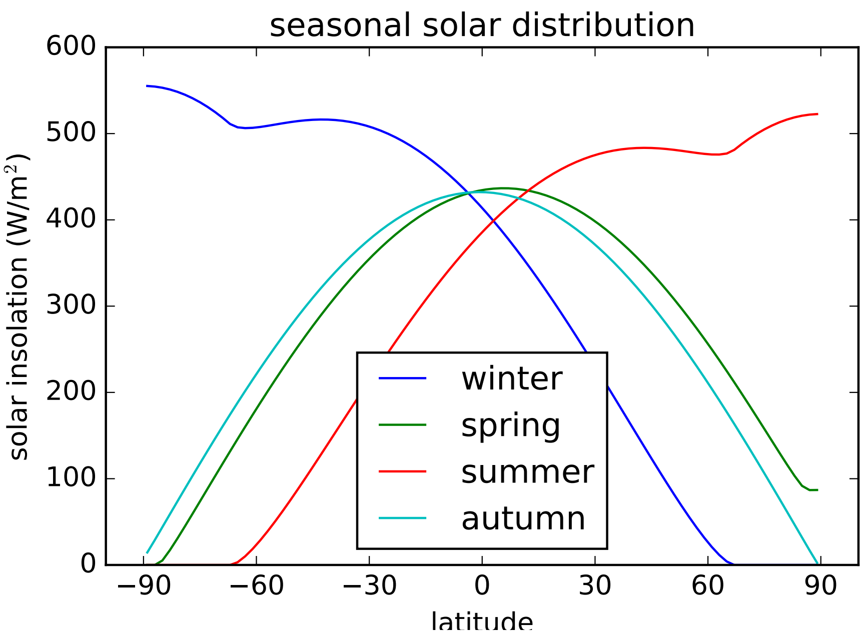
Variables

- **param** (*dict*) – The parameter dictionary is updated with `'ai'`.
- **subprocess** (*dict*) – suprocess `'albedo'` is overwritten by `StepFunctionAlbedo` (page 113) (which basically has been there before but now is updated with the new albedo parameter values).

Example The annual distribution of solar insolation:

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

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

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

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

season_days = const.days_per_year/4
```

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

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



6.1.4 climlab.process package

climlab.process.diagnostic module

$$\begin{tikzcd}\boxed{\text{Process}} \arrow[r] & \boxed{\text{TimeDependentProcess}} \arrow[r] & \boxed{\text{DiagnosticProcess}}\end{tikzcd}$$

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

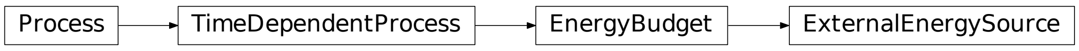
Bases: *`climlab.process.time_dependent_process.TimeDependentProcess`* (page 90)

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`* (page 90)

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`* (page 82)

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

Object attributes

Additional to the parent class *`EnergyBudget`* (page 82) the following object attribute is modified during initialization:

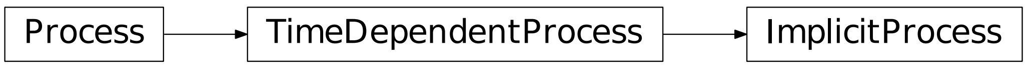
Variables **`heating_rate`** (*`dict`*) – energy share dictionary for this subprocess is set to zero for every model state.

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

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

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

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

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

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

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

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

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

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

climlab.process.implicit module



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

Bases: *`climlab.process.time_dependent_process.TimeDependentProcess`* (page 90)

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()` (page 91) method of the parent `TimeDependentProcess` (page 90) 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* (page 5) section climlab-architecture.

Initialization parameters

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

Parameters

- **`state`** (`Field` (page 63)) – spatial state variable for the process. Set to `None` if not specified.
- **`domains`** (`_Domain` (page 59) or dict of `_Domain` (page 59)) – domain(s) for the process
- **`subprocess`** (`Process` (page 84) or dict of `Process` (page 84)) – 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` (page 84) following object attributes are generated during initialization:

Variables

- **`domains`** (*`dict`*) – dictionary of process *`_Domain`* (page 59)
- **`state`** (*`dict`*) – dictionary of process states (of type *`Field`* (page 63))
- **`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`* (page 84)) – 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`* (page 84)) – a Process object

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

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

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

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

>>> print ebm_s
```

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

```
>>> #  creating and adding albedo feedback subprocess
>>> step_albedo = StepFunctionAlbedo(state=ebm_s.state, **ebm_s.param)
>>> ebm_s.add_subprocess('albedo', step_albedo)
>>>
>>> print ebm_s
```

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

```
                  albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
                     iceline: <class 'climlab.surface.albedo.Iceline'>
                     cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
                     warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
                  insolation: <class 'climlab.radiation.insolation.DailyInsolation'>
```

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

Calls *add_subprocess()* (page 85) 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.]

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

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

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

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

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

Example Remove diagnostic variable 'icelat' from energy balance model:

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

>>> # display all diagnostic variables
>>> model.diagnostics.keys()
['ASR', 'OLR', 'net_radiation', 'albedo', 'icelat']

>>> model.remove_diagnostic('icelat')
>>> model.diagnostics.keys()
['ASR', 'OLR', 'net_radiation', 'albedo']

>>> # Watch out for subprocesses that may still want
>>>  # to access the diagnostic 'icelat' variable !!!
```

remove_subprocess(*name*)

Removes a single subprocess from this process.

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

Example Remove albedo subprocess from energy balance model:

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

>>> print model
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>

>>> model.remove_subprocess('albedo')

>>> print model
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

set_state(*name*, *value*)

Sets the variable `name` to a new state `value`.

Parameters

- **name** (*string*) – name of the state
- **value** (*Field* (page 63) or *array*) – state variable

Raises `ValueError` if state variable `value` is not having a domain.

Raises `ValueError` if shape mismatch between existing domain and new state variable.

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

```
>>> import climlab
>>> from climlab import Field
>>> import numpy as np

>>> # setup model
>>> model = climlab.EBM(num_lat=36)

>>> # create new temperature distribution
>>> initial = -5 * ones(size(model.lat))
>>> model.set_state('Ts', Field(initial, domain=model.domains['Ts']))

>>> np.squeeze(model.Ts)
Field([-5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5., -5.,
       -5., -5., -5., -5., -5., -5., -5., -5., -5., -5.])
```

climlab.process.process.**get_axes**(*process_or_domain*)

Returns a dictionary of all Axis in a domain or dictionary of domains.

Parameters **process_or_domain** (*Process* (page 84) or *_Domain* (page 59)) – a process or a domain object

Raises

exc *TypeError* if input is not or not having a domain

Returns dictionary of input's Axis

Return type dict

Example

```
>>> import climlab
>>> from climlab.process.process import get_axes

>>> model = climlab.EBM()

>>> get_axes(model)
{'lat': <climlab.domain.axis.Axis object at 0x7ff13b9dd2d0>,
 'depth': <climlab.domain.axis.Axis object at 0x7ff13b9dd310>}
```

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

Copys the given process.

The creation date is updated.

Parameters **proc** (*Process* (page 84)) – process

Returns new process identical to the given process

Return type *Process* (page 84)

Example

```
>>> import climlab
>>> from climlab.process.process import process_like

>>> model = climlab.EBM()
>>> model.subprocess.keys()
['diffusion', 'LW', 'albedo', 'insolation']

>>> albedo = model.subprocess['albedo']
>>> albedo_copy = process_like(albedo)

>>> albedo.creation_date
'Thu, 24 Mar 2016 01:32:25 +0000'

>>> albedo_copy.creation_date
'Thu, 24 Mar 2016 01:33:29 +0000'
```

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`* (page 84)

A generic parent class for all time-dependent processes.

`TimeDependentProcess` is a child of the *`Process`* (page 84) class and therefore inherits all those attributes.

Initialization parameters

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

Parameters

- **`timestep`** (*`float`*) – specifies the timestep of the object (optional)
- **`time_type`** (*`str`*) – how time-dependent-process should be computed [default: 'explicit']
- **`topdown`** (*`bool`*) – whether geneterate *process_types* in regular or in reverse order [default: True]

Object attributes

Additional to the parent class *`Process`* (page 84) following object attributes are generated during initialization:

Variables

- **`has_process_type_list`** (*`bool`*) – information whether attribute *process_types* (which is needed for *`compute()`* (page 91) and build in `_build_process_type_list()`) exists or not. Attribute is set to `'False'` during initialization.
- **`topdown`** (*`bool`*) – information whether the list *process_types* (which contains all processes and sub-processes) should be generated in regular or in reverse order. See `_build_process_type_list()`.
- **`timeave`** (*`dict`*) – a time averaged collection of all states and diagnostic processes over the timeperiod that *`integrate_years()`* (page 92) has been called for last.
- **`tendencies`** (*`dict`*) – computed difference in a timestep for each state. See *`compute()`* (page 91) for details.
- **`time_type`** (*`str`*) – how time-dependent-process should be computed. Possible values are: `'explicit'`, `'implicit'`, `'diagnostic'`, `'adjustment'`.
- **`time`** (*`dict`*) –

  a collection of all time-related attributes of the process. The dictionary contains following items:

- `'timestep'`: see initialization parameter
- `'num_steps_per_year'`: see *set_timestep()* (page 93) and *timestep()* (page 93) 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. They don't produce any tendencies directly but they may effect the other processes (such as change in solar distribution). Subsequently, all tendencies and diagnostics for all explicit processes are computed.

Tendencies due to implicit and adjustment processes need to be calculated from a state that is already adjusted after explicit alteration. For that reason the explicit tendencies are applied to the states temporarily. Now all tendencies from implicit processes are calculated by matrix inversions and similar to the explicit tendencies, the implicit ones are applied to the states temporarily. Subsequently, all instantaneous adjustments are computed.

Then the changes that were made to the states from explicit and implicit processes are removed again as this *compute()* (page 91) function is supposed to calculate only tendencies and not apply them to the states.

Finally, all calculated tendencies from all processes are collected for each state, summed up and stored in the dictionary `self.tendencies`, which is an attribute of the time-dependent-process object, for which the *compute()* (page 91) 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 [default: 0.0001]
- **verbose** (*bool*) – information whether total elapsed time should be printed [default: True]

Example

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

>>> model.global_mean_temperature()
Field(11.997968598413685)

>>> model.integrate_converge()
Total elapsed time is 10.0 years.

>>> model.global_mean_temperature()
Field(14.288155406577301)
```

integrate_days (*days=1.0*, *verbose=True*)
Integrates the model forward for a specified number of days.

It convertes the given number of days into years and calls *integrate_years()* (page 92).

Parameters

- **days** (*float*) – integration time for the model in days [default: 1.0]
- **verbose** (*bool*) – information whether model time details should be printed [default: True]

Example

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

>>> model.global_mean_temperature()
Field(11.997968598413685)

>>> model.integrate_days(80.)
Integrating for 19 steps, 80.0 days, or 0.219032740466 years.
Total elapsed time is 0.211111111111 years.

>>> model.global_mean_temperature()
Field(11.873680783355553)
```

integrate_years (*years=1.0*, *verbose=True*)
Integrates the model by a given number of years.

Parameters

- **years** (*float*) – integration time for the model in years [default: 1.0]
- **verbose** (*bool*) – information whether model time details should be printed [default: True]

It calls *step_forward()* (page 93) repetitively and calculates a time averaged value over the integrated period for every model state and all diagnostics processes.

Example

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

>>> model.global_mean_temperature()
Field(11.997968598413685)

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

```
>>> model.global_mean_temperature()
Field(13.531055349437258)
```

set_timestep(*timestep=86400.0*, *num_steps_per_year=None*)

Calculates the timestep in unit seconds and calls the setter function of `timestep()` (page 93)

Parameters

- **timestep** (*float*) – the amount of time over which `step_forward()` (page 93) is integrating in unit seconds [default: 24*60*60]
- **num_steps_per_year** (*float*) – a number of steps per calendar year (optional)

If the parameter *num_steps_per_year* is specified and not `None`, the timestep is calculated accordingly and therefore the given input parameter *timestep* is ignored.

step_forward()

Updates state variables with computed tendencies.

Calls the `compute()` (page 91) method to get current tendencies for all process states. Multiplied with the timestep and added up to the state variables is updating all model states.

Example

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

>>> # checking time step counter
>>> model.time['steps']
0

>>> # stepping the model forward
>>> model.step_forward()

>>> # step counter increased
>>> model.time['steps']
1
```

timestep

The amount of time over which `step_forward()` (page 93) is integrating in unit seconds.

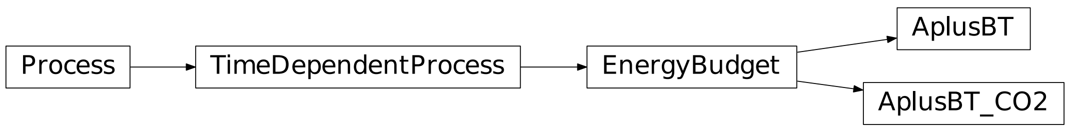
Getter Returns the object timestep which is stored in `self.param['timestep']`.

Setter Sets the timestep to the given input. See also `set_timestep()` (page 93).

Type float

6.1.5 climlab.radiation package

climlab.radiation.AplusBT module



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

Bases: `climlab.process.energy_budget.EnergyBudget` (page 82)

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

Object attributes

Additional to the parent class *EnergyBudget* (page 82) following object attributes are generated or modified during initialization:

Variables

- ***A*** (page 95) (*float*) – calls the setter function of *A()* (page 95)
- ***B*** (page 95) (*float*) – calls the setter function of *B()* (page 95)
- **diagnostics** (*dict*) – key `'OLR'` initialized with value: *Field* (page 63) of zeros in size of `self.Ts`
- ***OLR*** (page 95) (`Field` (page 63)) – the subprocess attribute `self.OLR` is created with correct dimensions

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

Example Simple linear radiation module (stand alone):

```
>>> import climlab

>>> # create a column atmosphere and scalar surface
>>> sfc, atm = climlab.domain.single_column()

>>> # Create a state variable
>>> Ts = climlab.Field(15., domain=sfc)

>>> # Make a dictionary of state variables
>>> s = {'Ts': Ts}

>>> # create process
>>> olr = climlab.radiation.AplusBT(state=s)

>>> print olr
climlab Process of type <class 'climlab.radiation.AplusBT.AplusBT'>.
State variables and domain shapes:
  Ts: (1,)
The subprocess tree:
top: <class 'climlab.radiation.AplusBT.AplusBT'>

>>> # to compute tendencies and diagnostics
>>> olr.compute()

>>> #  or to actually update the temperature
>>> olr.step_forward()

>>> print olr.state
{'Ts': Field([ 5.69123176])}
```

A

Property of AplusBT parameter A.

Getter Returns the parameter A which is stored in attribute `self._A`

Setter

- sets parameter A which is addressed as `self._A` to the new value
- updates the parameter dictionary `self.param['A']`

Type float

Example

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

>>> # getter
>>> model.subprocess['LW'].A
210.0
>>> # setter
>>> model.subprocess['LW'].A = 220
>>> # getter again
>>> model.subprocess['LW'].A
220

>>> # subprocess parameter dictionary
>>> model.subprocess['LW'].param['A']
220
```

B

Property of AplusBT parameter B.

Getter Returns the parameter B which is stored in attribute `self._B`

Setter

- sets parameter B which is addressed as `self._B` to the new value
- updates the parameter dictionary `self.param['B']`

Type float

OLR

class `climlab.radiation.AplusBT.`**`AplusBT_CO2`**(*CO2=300.0*, ***kwargs*)

Bases: `climlab.process.energy_budget.EnergyBudget` (page 82)

Linear longwave radiation module considering CO2 concentration.

This radiation subprocess is based in the idea to linearize the Outgoing Longwave Radiation (OLR) emitted to space according to the surface temperature (see `AplusBT` (page 93)).

To consider a the change of the greenhouse effect through range of CO_2 in the atmosphere, the parameters A and B are computed like the following:

$$A(c) = -326.4 + 9.161c - 3.164c^2 + 0.5468c^3$$
$$B(c) = 1.953 - 0.04866c + 0.01309c^2 - 0.002577c^3$$

where $c = \log \frac{p}{300}$ and p represents the concentration of CO_2 in the atmosphere.

For further reading see [Caldeira_1992].

Initialization parameters

An instance of `AplusBT_CO2` is initialized with the following argument:

Parameters **CO2** (*float*) – The concentration of CO_2 in the atmosphere. Referred to as p in the above given formulas.

- unit: ppm (parts per million)
- default value: `300.0`

Object attributes

Additional to the parent class *EnergyBudget* (page 82) following object attributes are generated or updated during initialization:

Variables

- ***CO2*** (page 97) (*float*) – calls the setter function of *CO2()* (page 97)
- **diagnostics** (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.init_diagnostic('OLR', 0. * self.Ts)`
- ***OLR*** (page 95) (`Field` (page 63)) – the subprocess attribute `self.OLR` is created with correct dimensions

Example Replacing an the regular AplusBT subprocess in an energy balance model:

```
>>> import climlab
>>> from climlab.radiation.AplusBT import AplusBT_CO2

>>> # creating EBM model
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> #  creating and adding albedo feedback subprocess
>>> LW_CO2 = AplusBT_CO2(CO2=400, state=model.state, **model.param)

>>> # overwriting old 'LW' subprocess with same name
>>> model.add_subprocess('LW', LW_CO2)

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
```

```
            The subprocess tree:
            top: <class 'climlab.model.ebm.EBM'>
               diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
               LW: <class 'climlab.radiation.AplusBT.AplusBT_CO2'>
               albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
                  iceline: <class 'climlab.surface.albedo.Iceline'>
                  cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
                  warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
               insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

CO2

Property of AplusBT_CO2 parameter CO2.

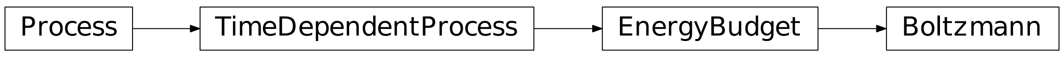
Getter Returns the CO2 concentration which is stored in attribute `self._CO2`

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`* (page 82)

A class for black body radiation.

Implements a radiation subprocess which computes longwave radiation with the Stefan-Boltzmann law for black/grey body radiation.

According to the Stefan Boltzmann law the total power radiated from an object with surface area A and temperature T (in unit Kelvin) can be written as

$$P = A\varepsilon\sigma T^4$$

where ε is the emissivity of the body.

As the *`EnergyBudget`* (page 82) of the Energy Balance Model is accounted in unit energy/area (W/m^2) the energy budget equation looks like this:

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

The *`Boltzmann`* (page 97) 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 [default: 0.65]
- **tau** (*float*) – transmissivity of the planet's atmosphere which is the effectiveness in transmitting the longwave radiation emitted from the surface [default: 0.95]

Object attributes

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

Variables

- ***eps*** (page 99) (*float*) – calls the setter function of `eps()` (page 99)
- ***tau*** (page 99) (*float*) – calls the setter function of `tau()` (page 99)
- **diagnostics** (*dict*) – the subprocess's diagnostic dictionary `self.diagnostic` is initialized through calling `self.init_diagnostic('OLR', 0. * self.Ts)`
- ***OLR*** (page 95) (`Field` (page 63)) – the subprocess attribute `self.OLR` is created with correct dimensions

Example Replacing an the regular AplusBT subprocess in an energy balance model:

```
>>> import climlab
>>> from climlab.radiation.Boltzmann import Boltzmann

>>> # creating EBM model
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> #  creating and adding albedo feedback subprocess
>>> LW_boltz = Boltzmann(eps=0.69, tau=0.98, state=model.state, **model.param)

>>> # overwriting old 'LW' subprocess with same name
>>> model.add_subprocess('LW', LW_boltz)

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.Boltzmann.Boltzmann'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
```

```
                    cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
                    warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
                 insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

`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.

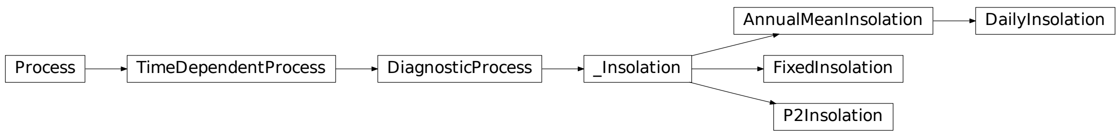
Getter Returns the albedo value which is stored in attribute `self._tau`

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` (page 104)

A class for latitudewise solar insolation averaged over a year.

This class computes the solar insolation for each day of the year and latitude specified in the domain on the basis of orbital parameters and astronomical formulas.

Therefore it uses the method `daily_insolation()` (page 105). For details how the solar distribution is dependend on orbital parameters see there.

The mean over the year is calculated from data given by `daily_insolation()` (page 105) 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* (page 108)):
  - '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* (page 104) following object attributes are generated and updated during initialization:

Variables

- ***insolation*** (page 104) (Field (page 63)) – 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*** (page 101) (*dict*) – initialized with given argument orb

Example Create regular EBM and replace standard insolation subprocess by AnnualMeanInsolation:

```
>>> import climlab
>>> from climlab.radiation import AnnualMeanInsolation

>>> # model creation
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> # catch model domain for subprocess creation
>>> sfc = model.domains['Ts']

>>> # create AnnualMeanInsolation subprocess
>>> new_insol = AnnualMeanInsolation(domains=sfc, **model.param)

>>> # add it to the model
>>> model.add_subprocess('insolation',new_insol)
>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.AnnualMeanInsolation'>
```

orb

Property of dictionary for orbital parameters.

orb contains: (for more information see *OrbitalTable* (page 108))

- '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* (page 99)

A class to compute latitudewise daily solar insolation for specific days of the year.

This class computes the solar insolation on basis of orbital parameters and astronomical formulas.

Therefore it uses the method *daily_insolation()* (page 105). For details how the solar distribution is dependend on orbital parameters see there.

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{W}{m^2}$
  - 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`* (page 104) following object attributes are generated and updated during initialization:

Variables

- ***`insolation`*** (page 104) (`Field` (page 63)) – 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`*** (page 101) (*`dict`*) – initialized with given argument `orb`

Example Create regular EBM and replace standard insolation subprocess by `DailyInsolation`:

```
>>> import climlab
>>> from climlab.radiation import DailyInsolation

>>> # model creation
>>> model = climlab.EBM()

>>> print model
```

```
climlab Process of type <class 'climlab.model.ebm.EBM'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

```
>>> # catch model domain for subprocess creation
>>> sfc = model.domains['Ts']

>>> # create DailyInsolation subprocess and add it to the model
>>> model.add_subprocess('insolation',DailyInsolation(domains=sfc, **model.param))

>>> print model
```

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

```
                    cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
                    warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
                 insolation: <class 'climlab.radiation.insolation.DailyInsolation'>
```

insolation

class `climlab.radiation.insolation.`**`FixedInsolation`**(*S0=341.3*, ***kwargs*)

Bases: *`climlab.radiation.insolation._Insolation`* (page 104)

A class for fixed insolation at each point of latitude off the domain.

The solar distribution for the whole domain is constant and specified by a parameter.

Initialization parameters

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

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

Example

```
>>> import climlab
>>> from climlab.radiation.insolation import FixedInsolation

>>> model = climlab.EBM()
>>> sfc = model.Ts.domain

>>> fixed_ins = FixedInsolation(S0=340.0, domains=sfc)

>>> print fixed_ins
climlab Process of type <class 'climlab.radiation.insolation.FixedInsolation'>.
State variables and domain shapes:
The subprocess tree:
top: <class 'climlab.radiation.insolation.FixedInsolation'>
```

insolation

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

Bases: *`climlab.radiation.insolation._Insolation`* (page 104)

A class for parabolic solar distribution over the domain's latitude on the basis of the second order Legendre Polynomial.

Calculates the latitude dependent solar distribution as

$$S(\varphi) = \frac{S_0}{4} \left(1 + s_2 P_2(x)\right)$$

where $P_2(x) = \frac{1}{2}(3x^2 - 1)$ is the second order Legendre Polynomial and $x = sin(\varphi)$.

Initialization parameters

Parameters

- **S0** (*float*) – solar constant
  - unit: $\frac{\mathrm{W}}{\mathrm{m}^2}$
  - default value: `1365.2`
- **s2** (*floar*) – factor for second legendre polynominal term
  - default value: `-0.48`

Example

```
>>> import climlab
>>> from climlab.radiation.insolation import P2Insolation

>>> model = climlab.EBM()
>>> sfc = model.Ts.domain

>>> p2_ins = P2Insolation(S0=340.0, s2=-0.5, domains=sfc)

>>> print p2_ins
climlab Process of type <class 'climlab.radiation.insolation.P2Insolation'>.
State variables and domain shapes:
The subprocess tree:
top: <class 'climlab.radiation.insolation.P2Insolation'>
```

insolation

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`* (page 81)

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`* (page 81) following object attributes are generated and updated during initialization:

Variables

- ***insolation*** (page 104) (`Field` (page 63)) – the array is initialized with zeros of the size of `self.domains['sfc']` or `self.domains['default']`.
- ***S0*** (page 105) (*float*) – initialized with given argument `S0`
- **diagnostics** (*dict*) – key `'insolation'` initialized with value: *`Field`* (page 63) of zeros in size of `self.domains['sfc']` or `self.domains['default']`

- ***insolation*** (page 104) – 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

6.1.6 climlab.solar package

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 [Berger_1978]. Further references: [Berger_1991].

`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()`* (page 108) (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* (page 108))
  - 'ecc' - eccentricity
    - unit: dimensionless
    - default value: 0.017236
  - 'long_peri' - longitude of perihelion (precession angle)
    - unit: degrees
    - default value: 281.37
  - 'obliquity' - obliquity angle
    - unit: degrees
    - default value: 23.446
- **S0** (*float*) – solar constant
  - unit: W/m^2
  - default value: 1365.2
- **day_type** (*int*) – Convention for specifying time of year (+/- 1,2) [optional].

  ***day_type=1* (default):** day input is calendar day (1-365.24), where day 1 is January first. The calendar is referenced to the vernal equinox which always occurs at day 80.

  day_type=2: day input is solar longitude (0-360 degrees). Solar longitude is the angle of the Earth's orbit measured from spring equinox (21 March). Note that calendar days and solar longitude are not linearly related because, by Kepler's Second Law, Earth's angular velocity varies according to its distance from the sun.

Raises ValueError if day_type is neither 1 nor 2

Returns

Daily average solar radiation in unit W/m^2.

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

Return type array

Code is fully vectorized to handle array input for all arguments.

Orbital arguments should all have the same sizes. This is automatic if computed from *lookup_parameters()* (page 108)

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

```
from climlab.solar.orbital import OrbitalTable
from climlab.solar.insolation import daily_insolation

# import orbital table
table = OrbitalTable()

# array with specified kyears
years = np.linspace(-5000, 0, 5001)

# orbital parameters for specified time
orb = table.lookup_parameters( years )

# insolation values for past 5 Myears at 65N at summer solstice
S65 = daily_insolation( 65, 172, orb )
```

For more information about computation of solar insolation see the *Tutorials* (page 15) chapter.

`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 [Berger_1978] section 3 (lambda = 0 at spring equinox).

Function-call arguments

Parameters

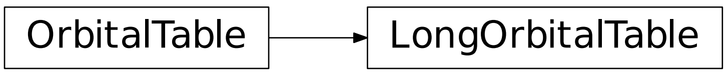
- **`day`** (*`array`*) – Indicator of time of year.
- **`orb`** (*`dict`*) – a dictionary with three members (as provided by *`OrbitalTable`* (page 108))
  - `'ecc'` - eccentricity
    - unit: dimensionless
    - default value: `0.017236`
  - `'long_peri'` - longitude of perihelion (precession angle)
    - unit: degrees
    - default value: `281.37`
  - `'obliquity'` - obliquity angle
    - unit: degrees
    - default value: `23.446`
- **`days_per_year`** (*`float`*) – number of days in a year (optional) (default: 365.2422) Reads the length of the year from *`constants`* (page 115) if available.

Returns solar longitude `lambda_long` in dimension``( 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`* (page 108) which holds orbital data, and includes a method *`lookup_parameters()`* (page 108) which interpolates the orbital data for a specific year (- works equally well for arrays of years).

The base class *`OrbitalTable()`* (page 108) is designed to work with 5 Myears of orbital data (**eccentricity, obliquity, and longitude of perihelion**) from [Berger_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()`* (page 107) works with La2004 orbital data for -51 to +21 Myears as calculated by [Laskar_2004]. See http://vo.imcce.fr/insola/earth/online/earth/La2004/README.TXT

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

Bases: *`climlab.solar.orbital.OrbitalTable`* (page 108)

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

Based on calculations by [Laskar_2004] http://vo.imcce.fr/insola/earth/online/earth/La2004/README.TXT

Usage is identical to parent class *`OrbitalTable()`* (page 108).

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

Invoking OrbitalTable() will load 5 million years of orbital data from [Berger_1991] and compute linear interpolants.

The data can be accessed through the method *`lookup_parameters()`* (page 108).

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). [default: 0]

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

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.

OrbitalCycles is a module for setting up long integrations of climlab processes over orbital cycles.

The duration between integration start and end time is partitioned in time segments over which the orbital parameters are held constant. The process is integrated over every time segment and the process state `Ts` is stored for each segment.

The storage arrays are saving:

- **current model state** at end of each segment
- **model state averaged** over last integrated year of each segment
- **global mean** of averaged model state over last integrated year of each segment

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

Initialization parameters

Parameters

- **`model`** (*`TimeDependentProcess`* (page 90)) – a time dependent process
- **`kyear_start`** (*`float`*) – integration start time.

  As time reference is present, argument should be <0 for time before present.

  – *unit:* kiloyears

  – *default value:* `-20.`
- **`kyear_stop`** (*`float`*) – integration stop time.

  As time reference is present, argument should be ≤ 0 for time before present.

  – *unit:* kiloyears

  – *default value:* `0.`
- **`segment_length_years`** (*`float`*) – is the length of each integration with fixed orbital parameters. [default: 100.]
- **`orbital_year_factor`** (*`float`*) – is an optional speed-up to the orbital cycles. [default: 1.]
- **`verbose`** (*`bool`*) – prints product of calculation and information about computation progress [default: True]

Object attributes

Following object attributes are generated during initialization:

Variables

- **`model`** (*`TimeDependentProcess`* (page 90)) – timedependent process to be integrated
- **`kyear_start`** (*`float`*) – integration start time
- **`kyear_stop`** (*`float`*) – integration stop time
- **`segment_length_years`** (*`float`*) – length of each integration with fixed orbital parameters
- **`orbital_year_factor`** (*`float`*) – speed-up factor to the orbital cycles

- **verbose** (*bool*) – print flag
- **num_segments** (*int*) – number of segments with fixed oribtal parameters, calculated through:

$$num_{seg} = \frac{-(kyear_{start} - kyear_{stop}) * 1000}{seg_{length} * orb_{factor}}$$

- **T_segments_global** (*array*) – storage for global mean temperature for final year of each segment
- **T_segments** (*array*) – storage for actual temperature at end of each segment
- **T_segments_annual** (*array*) – storage for timeaveraged temperature over last year of segment

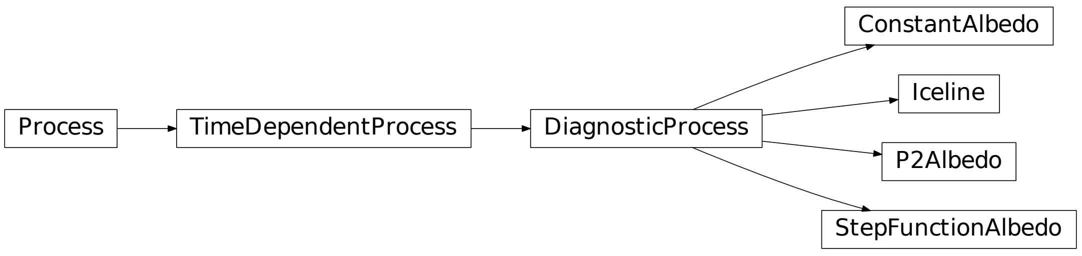
  dimension: (size(Ts), num_segments)
- **orb_kyear** (*array*) – integration start time of all segments
- ***orb*** (page 101) (*dict*) – orbital parameters for last integrated segment

Example Integration of an energy balance model for 10,000 years with corresponding orbital parameters:

```
from climlab.model.ebm import EBM_seasonal
from climlab.solar.orbital_cycles import OrbitalCycles
from climlab.surface.albedo import StepFunctionAlbedo
ebm = EBM_seasonal()
print ebm

#  add an albedo feedback
albedo = StepFunctionAlbedo(state=ebm.state, **ebm.param)
ebm.add_subprocess('albedo', albedo)

#  start the integration
#  run for 10,000 orbital years, but only 1,000 model years
experiment = OrbitalCycles(ebm, kyear_start=-20, kyear_stop=-10,
```

6.1.7 climlab.surface package

climlab.surface.albedo module



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

Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 81)

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

Initialization parameters

Parameters albedo (*float*) – albedo values [default: 0.33]

Object attributes

Additional to the parent class `DiagnosticProcess` (page 81) following object attributes are generated and updated during initialization:

Variables ***albedo*** (page 113) (`Field` (page 63)) – attribute to store the albedo value. During initialization the `albedo()` (page 111) setter is called.

Example Creation of a constant albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import ConstantAlbedo

>>> # model creation
>>> model = climlab.EBM()

>>> sfc = model.domains['Ts']

>>> # subprocess creation
>>> const_alb = ConstantAlbedo(albedo=0.3, domains=sfc, **model.param)
```

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` (page 81)

A class for an Iceline subprocess.

Depending on a freezing temperature it calculates where on the domain the surface is covered with ice, where there is no ice and on which latitude the ice-edge is placed.

Initialization parameters

Parameters **`Tf`** (*float*) – freezing temperature where sea water freezes and surface is covered with ice

- unit: °C
- default value: `-10`

Object attributes

Additional to the parent class `DiagnosticProcess` (page 81) 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*** (page 112) (*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` (page 63)) – a Field of booleans which are `True` where $T_s \ge T_f$
- **ice** (`Field` (page 63)) – a Field of booleans which are `True` where $T_s < T_f$
- ***icelat*** (page 112) (*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

icelat

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

Bases: *climlab.process.diagnostic.DiagnosticProcess* (page 81)

A class for parabolic distributed albedo values across the domain on basis of the second order Legendre Polynomial.

Calculates the latitude dependent albedo values as

$$\alpha(\varphi) = a_0 + a_2 P_2(x)$$

where $P_2(x) = \frac{1}{2}(3x^2 - 1)$ is the second order Legendre Polynomial and $x = sin(\varphi)$.

Initialization parameters

Parameters

- **a0** (*float*) – basic parameter for albedo function [default: 0.33]
- **a2** (*float*) – factor for second legendre polynominal term in albedo function [default: 0.25]

Object attributes

Additional to the parent class *DiagnosticProcess* (page 81) following object attributes are generated and updated during initialization:

Variables

- ***a0*** (page 113) (*float*) – attribute to store the albedo parameter a0. During initialization the *a0()* (page 113) setter is called.
- ***a2*** (page 113) (*float*) – attribute to store the albedo parameter a2. During initialization the *a2()* (page 113) setter is called.
- **diagnostics** (*dict*) – key `'albedo'` initialized
- ***albedo*** (page 113) (`Field` (page 63)) – the subprocess attribute `self.albedo` is created with correct dimensions (according to `self.lat`)

Example Creation of a parabolic albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import P2Albedo

>>> # model creation
>>> model = climlab.EBM()
>>> # modify a0 and a2 values in model parameter dictionary
>>> model.param['a0']=0.35
>>> model.param['a2']= 0.10

>>> # subprocess creation
>>> p2_alb = P2Albedo(domains=model.domains['Ts'], **model.param)

>>> p2_alb.a0
0.33
>>> p2_alb.a2
0.1
```

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

albedo

class `climlab.surface.albedo.`**`StepFunctionAlbedo`**(*Tf=-10.0*, *a0=0.3*, *a2=0.078*, *ai=0.62*, *****kwargs***)

Bases: *`climlab.process.diagnostic.DiagnosticProcess`* (page 81)

A step function albedo suprocess.

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

- `'iceline'` - *`Iceline`* (page 111)
- `'warm_albedo'` - *`P2Albedo`* (page 112)
- `'cold_albedo'` - *`ConstantAlbedo`* (page 110)

Initialization parameters

Parameters

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

- **ai** (*float*) – ice albedo value for ConstantAlbedo subprocess [default: 0.62]

Additional to the parent class *DiagnosticProcess* (page 81) 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*** (page 113) (Field (page 63)) – the subprocess attribute self.albedo is created

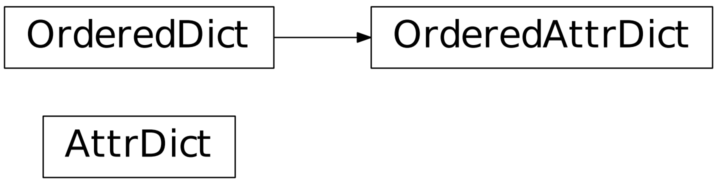
Example Creation of a step albedo subprocess on basis of an EBM domain:

```
>>> import climlab
>>> from climlab.surface.albedo import StepFunctionAlbedo
>>>
>>> model = climlab.EBM(a0=0.29, a2=0.1, ai=0.65, Tf=-2)
>>>
>>> step_alb = StepFunctionAlbedo(state= model.state, **model.param)
>>>
>>> print step_alb
climlab Process of type <class 'climlab.surface.albedo.StepFunctionAlbedo'>.
State variables and domain shapes:
  Ts: (90, 1)
The subprocess tree:
top: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
   iceline: <class 'climlab.surface.albedo.Iceline'>
   cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
   warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
```

albedo

6.1.8 climlab.utils package

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.

$$C_a = \frac{c_p \cdot dp \cdot f_{\text{mb-to-Pa}}}{g}$$

where

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

Example Calculate atmospheric heat capacity for pressure intervals of 1, 10, 100 mb:

```
>>> from climlab.utils import heat_capacity

>>> pressure_interval = array([1,10,100]) # in mb
>>> heat_capacity.atmosphere(pressure_interval) # in J /m**2 / K
array([   10244.89795918,   102448.97959184,  1024489.79591837])
```

`climlab.utils.heat_capacity.`**`ocean`**`(`*dz*`)`

Returns heat capacity of a unit area of water, in units J /m**2 / K.

$$C_o = \rho_w \cdot c_w \cdot dz$$

where

variable	value	unit	description
C_o	*output*	$\mathrm{J/m^2/K}$	heat capacity for oceanic cell
c_w	4181.3	$\mathrm{J/kg/K}$	specific heat of liquid water
dz	*input*	m	water depth of oceanic cell
ρ_w	1000.	$\mathrm{kg/m^3}$	density of water

Function-call argument

Parameters **dz** (*array*) – water depth of ocean cells (*unit:* m)

Returns the heat capacity for ocean cells correspoding to depth input (*unit:* J /m**2 / K)

Return type array

Example Calculate atmospheric heat capacity for pressure intervals of 1, 10, 100 m:

```
>>> from climlab.utils import heat_capacity

>>> pressure_interval = array([1,10,100]) # in m
>>> heat_capacity.ocean(pressure_interval) # in J /m**2 / K
array([  4.18130000e+06,   4.18130000e+07,   4.18130000e+08])
```

`climlab.utils.heat_capacity.`**`slab_ocean`**(*water_depth*)

Returns heat capacity of a unit area slab of water, in units of J / m**2 / K.

Takes input argument `water_depth` and calls *`ocean()`* (page 116)

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()`* (page 118) method to create the process tree.

Parameters

- **`top`** (*`Process`* (page 84)) – top process for which process tree string should be created
- **`name`** (*`str`*) – name of top process

Returns string representation of the process tree

Return type str

Example

```
>>> import climlab
>>> from climlab.utils import walk

>>> model = climlab.EBM()
>>> proc_tree_str = walk.process_tree(model, name='model')

>>> print proc_tree_str
model: <class 'climlab.model.ebm.EBM'>
   diffusion: <class 'climlab.dynamics.diffusion.MeridionalDiffusion'>
   LW: <class 'climlab.radiation.AplusBT.AplusBT'>
   albedo: <class 'climlab.surface.albedo.StepFunctionAlbedo'>
      iceline: <class 'climlab.surface.albedo.Iceline'>
      cold_albedo: <class 'climlab.surface.albedo.ConstantAlbedo'>
      warm_albedo: <class 'climlab.surface.albedo.P2Albedo'>
   insolation: <class 'climlab.radiation.insolation.P2Insolation'>
```

`climlab.utils.walk.`**`walk_processes`**(*top*, *topname='top'*, *topdown=True*, *ignoreFlag=False*)

Generator for recursive tree of climlab processes

Starts walking from climlab process `top` and generates a complete list of all processes and sub-processes that are managed from `top` process. `level` indicades the rank of specific process in the process hierarchy:

Note:

- **level 0: `top` process**
  - **level 1: sub-processes of `top` process**
    - level 2: sub-sub-processes of `top` process (=subprocesses of level 1 processes)

The method is based on os.walk().

Parameters

- **`top`** (*`Process`* (page 84)) – top process from where walking should start
- **`topname`** (*`str`*) – name of top process [default: 'top']
- **`topdown`** (*`bool`*) – whether geneterate *process_types* in regular or in reverse order [default: True]
- **`ignoreFlag`** (*`bool`*) – whether `topdown` flag should be ignored or not [default: False]

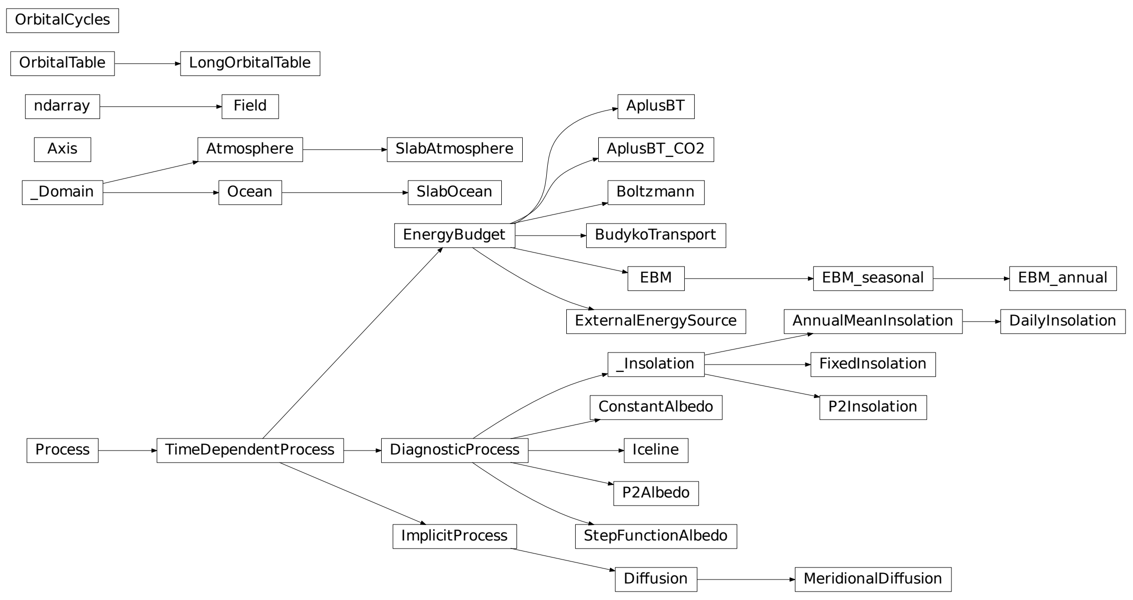
Returns name (str), proc (process), level (int)

Example

```
>>> import climlab
>>> from climlab.utils import walk

>>> model = climlab.EBM()

>>> for name, proc, top_proc in walk.walk_processes(model):
...     print name
...
top
diffusion
LW
iceline
```

```
                cold_albedo
                warm_albedo
                albedo
                insolation
```

6.2 Inheritance Diagram



CHAPTER SEVEN

REFERENCES

- A. Berger. Long-term variations of daily insolation and quaternary climatic changes. *Journal of Atmospheric Science*, 35(12):2362–2367, 1978.
- A. Berger and M. F. Loutre. Insolation values for the climate of the last 10 million years. *Quaternary Science Reviews*, 10(4):297–317, 1991.
- M. I. Budyko. The effect of solar radiation variations on the climate of the earth. *Tellus*, 21(5):611–619, 1969.
- Ken Caldeira and James F. Kasting. Susceptibility of the early earth to irreversible glaciation caused by carbon dioxide clouds. *Nature*, 359:226–228, 1992.
- J. Laskar, P. Robutel, F. Joutel, M. Gastineau, A. C. M. Correia, and B. Levrard. A long-term numerical solution for the insolation quantities of the earth. *Astronomy & Astrophysics*, 428:261–285, 2004.

CHAPTER

EIGHT

LICENSE

8.1 climlab

The climlab Python package is licensed under MIT License:

8.2 Documentation

The climlab Documentation is licensed under MIT License:

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