# climt Documentation

*Release 0.16.25*

Rodrigo Caballero

Feb 01, 2023

# Contents

climt (Climate Modelling and Diagnostics Toolkit) is a Python based library which provides a modular and intuitive approach to writing numerical models of the climate system. climt provides state-of-the art components and an easy-to-use interface to allow writing research quality models without the hassle of modifying Fortran code.

The modular nature of climt allows re-use of model code, allowing users to build progressively complicated models without having to rewrite the same code at each level of complexity.

climt uses sympl for its modelling infrastructure, making climt components and model scripts highly readable and self-documenting.

Contents:

# CHAPTER 1

# Introduction

climt (pronounced *klimt*) is an attempt to build a climate modelling infrastructure that is easy to use, easy to understand and easy to learn.

Most climate model components are written in fortran for performance reasons. For that very reason, it is difficult to change model configurations and behaviour easily, which is something scientists tend to do all the time during their research. The earth-science community is converging towards Python as the language of choice for data analysis tasks, thanks to Python's flexibility and emphasis on clean, readable code. climt aims to use Python for climate modelling for these very reasons – clearly documented components, self documenting model scripts, and a flexible configuration system will make climate modelling more reproducible and make the learning curve less steep.

climt is aimed at a wide spectrum of users – from students who are curious to learn about the climate system to researchers who want state-of-the-art components. climt aims to provide multiple levels of abstraction which will allow the user to tradeoff ease of use vs. flexibility, depending on their particular needs and experience in modelling and Python programming.

# CHAPTER 2

# Installation

## 2.1 Stable release

You can install climt by simply typing

```
$ pip install climt
```

This is the preferred method to install climt, as it will always install the most recent stable release. On Ubuntu Linux, you might need to prefix the above command with `sudo`. This command should work on Linux, Mac and Windows. For Mac and Windows, it is recommended to use the anaconda python distribution to make installation simpler.

**Note:** If you are not using Anaconda, please ensure you have the libpython library installed. See the next section for instructions to install libpython.

Since by default pip attempts to install a binary wheel, you won't need a compiler on your system.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 Installing from source

The sources for climt can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/CliMT/climt
```

Or download the tarball:

```
$ curl  -OL https://github.com/CliMT/climt/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ pip install -r requirements_dev.txt
$ python setup.py install
```

Both commands may require the use of `sudo`.

### 2.2.1 Dependencies for source installations

climt depends on some easily installable libraries. For an installation from source, it also requires that C and fortran compilers be installed.

On Ubuntu Linux, for example, these dependencies can be installed by:

```
$ sudo apt-get install gcc
$ sudo apt-get install gfortran
$ sudo apt-get install python-dev
$ sudo pip install -U cython
$ sudo pip install -U numpy
```

use `pip3` and `python3-dev` if you use Python 3.

On Mac OSX, it is recommended that you use anaconda as your python distribution. This will eliminate the need to install cython, numpy and python-dev. Once you have anaconda installed, you will need to do the following:

```
$ brew install gcc
$ export CC=gcc-x
$ export FC=gfortran-x
```

Where `gcc-x,gfortran-x` are the names of the C,Fortran compilers that Homebrew installs. Exporting the name of the compiler is essential on Mac since the default compiler that ships with Mac (called `gcc`, but is actually a different compiler) cannot compile OpenMP programs, like the dynamical core in climt.

CHAPTER 3

# Quickstart

Let us start with a simple climt model which is not very useful, but helps illustrate how to use climt:

```
In [1]: import climt

# Create some components
In [2]: radiation = climt.GrayLongwaveRadiation()

In [3]: surface = climt.SlabSurface()

# Get a state dictionary filled with required quantities
# for the components to run
In [4]: state = climt.get_default_state([radiation, surface])

# Run components
In [5]: tendencies, diagnostics = radiation(state)

# See output
In [6]: tendencies.keys()
Out[6]: dict_keys(['air_temperature'])

In [7]: tendencies['air_temperature']
Out[7]:
<xarray.DataArray (mid_levels: 28, lat: 1, lon: 1)>
array([[[-8.65953484e-06]],

       [[-8.82597967e-06]],

       [[-9.10930142e-06]],

       [[-9.50816216e-06]],

       [[-1.00211876e-05]],

       [[-1.06462877e-05]],
```

(continues on next page)

(continued from previous page)

```
       [[-1.13814120e-05]],

       [[-1.22239221e-05]],

       [[-1.31708319e-05]],

       [[-1.42185500e-05]],

...

       [[-2.69670979e-05]],

       [[-2.85017885e-05]],

       [[-2.99815186e-05]],

       [[-3.13776165e-05]],

       [[-3.26612456e-05]],

       [[-3.38033767e-05]],

       [[-3.47767484e-05]],

       [[-3.55573827e-05]],

       [[-3.61242997e-05]],

       [[-3.64632192e-05]]])
Dimensions without coordinates: mid_levels, lat, lon
Attributes:
    units:    degK s^-1
```

Here, all the essential aspects of creating and running a model in climt are present:

- Import the `climt` package
- Create one or many components
- Create a state dictionary using `get_default_state`
- Run the components
- Do something with the output

Variables `radiation` and `surface` are two components that we create. All climt components take a lot of optional arguments: However, by design, the default options (which are used if you don't specify any arguments) are meant to be scientifically meaningful.

The variables `state`, `tendencies` and `diagnostics` are dictionaries which contain quantities which act either as inputs **to** components or outputs **from** components.

The function `get_default_state()`, if called only with a list of components, will provide a set of quantities which represent a single column of the atmosphere. These default values may or may not be meaningful in your context, so it is best to see what they are and change them according to your needs.

**Note:** The square brackets are required in the call to `get_default_state`, even if it is one component: `climt`.

`get_default_state([radiation])` is the correct syntax.

Building more sophisticated models and running them is merely an extended version of the above simple example. climt makes heavy use of Sympl, and knowledge of Sympl is necessary to use climt to its full capabilities. So, do go through Sympl's docs!

# CHAPTER 4

# Interacting with climt

As we saw in the Quickstart section, climt has two kinds of entities for the user to interact with: model components and model state. Here, we will take a closer look at both these elements.

## 4.1 Model State

The model state is a dictionary whose keys are names of quantities and values are sympl DataArrays. The sympl DataArray is a thin wrapper over the xarray DataArray that makes it units aware. To ensure model scripts are readable not just by specialists, names of quantities use the descriptive CF Convention. Only in the case where the CF Convention names are really unwieldy, like `air_temperature_at_effective_cloud_top_defined_by_infrared_radiation` for example, we use more convenient names.

DataArrays are a more human-friendly way of handling numerical arrays. DataArrays label the dimensions of an array and provide various mathematical functions which can be directly applied to arrays. sympl DataArrays in addition allow conversion between units, a feature required to allow interoperability between components which expect inputs in different units.

Let's create a 3-d model state to see how useful DataArrays are:

```
In [1]: import climt

In [2]: import matplotlib.pyplot as plt

In [3]: import numpy as np

# Create some components
In [4]: radiation = climt.GrayLongwaveRadiation()

In [5]: convection = climt.DryConvectiveAdjustment()
```

We need to tell climt what the model dimensions are. This is done by the `get_grid` function. This function takes three arguments which are the number of grid points in the three directions, and provides a state dictionary containing the definition of a grid.

Passing this grid state dictionary onto `get_default_state` makes climt aware of the dimensions required by the model:

```
In [6]: grid = climt.get_grid(ny=3, nz=5)

# Get a state dictionary filled with required quantities
# for the components to run
In [7]: state = climt.get_default_state([radiation, convection], grid_state=grid)

In [8]: state['air_temperature']
Out[8]:
<xarray.DataArray 'air_temperature' (mid_levels: 5, lat: 3, lon: 1)>
array([[[290.],
        [290.],
        [290.]],

       [[290.],
        [290.],
        [290.]],

       [[290.],
        [290.],
        [290.]],

       [[290.],
        [290.],
        [290.]],

       [[290.],
        [290.],
        [290.]]])
Dimensions without coordinates: mid_levels, lat, lon
Attributes:
    units:    degK
```

climt **does not** interpret any of the dimension attributes in state quantities other than `units`. The values and labels of coordinates are mainly for users and components. For instance, `SimplePhysics` requires that the y dimension be called `latitude`. So, any model that uses `SimplePhysics` has to label one of the dimensions as latitude.

As you can see, `air_temperature` has

- a uniform value of 290
- coordinates of latitude and mid_levels
- units of *degK*, which is the notation used in climt (and Sympl) for *degrees Kelvin*.

It is also fairly easy to change units. The `to_units()` method can be used as below to return a DataArray with the equivalent temperature in degrees Farenheit:

```
In [9]: state['air_temperature'].to_units('degF')
Out[9]:
<xarray.DataArray 'air_temperature' (mid_levels: 5, lat: 3, lon: 1)>
array([[[62.33],
        [62.33],
        [62.33]],

       [[62.33],
        [62.33],
```

(continues on next page)

(continued from previous page)

```
         [62.33]],

        [[62.33],
         [62.33],
         [62.33]],

        [[62.33],
         [62.33],
         [62.33]],

        [[62.33],
         [62.33],
         [62.33]]])
Dimensions without coordinates: mid_levels, lat, lon
Attributes:
    units:     degF
```

**Note:** climt always names the vertical coordinate as `mid_levels` or `interface_levels`, however, the state dictionary will contain a key corresponding to the name of the vertical coordinate specified by `get_grid`.

As mentioned previously, DataArrays are a user-friendly way of handling numerical or numpy arrays. The numpy array underlying any DataArray is easily accessed using the `values` attribute:

```
In [10]: type(state['air_temperature'].values)
Out[10]: numpy.ndarray
```

and can also be modified easily:

```
In [11]: state['air_temperature'].values[:] = 291
```

The right hand side can also be any numpy array, as long as it has the same dimensions (or can be broadcasted to the same dimensions) as the current numpy array.

**Note:** It is recommended to use the syntax `...values[:] = ...` rather than `...values = ...`, as the former modifies the numpy array in-place. In either case, DataArrays check to ensure the dimensions (or shape) of the new data matches with the current dimensions.

You can perform any of the functions supported by xarray on the model state quantities.

```
In [12]: state['air_temperature'].sum()
Out[12]:
<xarray.DataArray 'air_temperature' ()>
array(4365.)
```

You can also directly plot DataArrays:

```
In [13]: state['air_temperature'].plot()
Out[13]: <matplotlib.collections.QuadMesh at 0x7fbb26261978>
```

DataArrays are a very powerful way of dealing with array-oriented data, and you should read more about xarray, and not just for using climt!

## 4.2 Model Components

Components are representations of physical processes. You can see all available components in climt in the section *Components*.

All components take some inputs from the model state, and return **outputs** or **tendencies** along with diagnostics (if any).

Diagnostics are quantities computed while calculating **outputs** or **tendencies**. For example, a radiation component calculates heating rates. However, in the process of calculating these heating rates, it also calculates the radiative flux at each interface level.

```
# These are the tendencies returned by radiation
In [14]: radiation.tendency_properties
Out[14]: {'air_temperature': {'units': 'degK s^-1'}}

# These are the diagnostics returned by radiation
In [15]: radiation.diagnostic_properties
Out[15]:
{'downwelling_longwave_flux_in_air': {'dims': ['interface_levels', '*'],
  'units': 'W m^-2',
  'alias': 'lw_down'},
 'upwelling_longwave_flux_in_air': {'dims': ['interface_levels', '*'],
  'units': 'W m^-2',
  'alias': 'lw_up'},
 'longwave_heating_rate': {'dims': ['mid_levels', '*'],
  'units': 'degK day^-1'}}

# These are the outputs returned by convection
In [16]: convection.output_properties
Out[16]: {'air_temperature': {'units': 'degK'}, 'specific_humidity': {'units': 'kg/kg
↪'}}

# convection returns no diagnostics
In [17]: convection.diagnostic_properties
Out[17]: {}
```

No component will return **both** outputs and tendencies. The tendency of a quantity $X$ is given by $\frac{dX}{dt}$, and so the units of a quantity returned as a tendency will always have per second as as suffix: i.e, if a component is returning `air_temperature` as a tendency, then its units will be `degK/s`.

# CHAPTER 5

# A Realistic Model

As mentioned before, climt includes some components which returns the a new version of the model state, and some which return just tendencies.

Since tendencies by themselves are not useful for much other than plotting, we need to couple them with numerical integration components to march the model forward in time. Again, we will use the grey radiation scheme as an example.

The following script is used to obtain the temperature profile of the atmosphere if no physical process other than radiation (specifically, grey gas radiation in this example) are present. The temperature profile obtained is called the **radiative equilibrium** profile.

As before, we will create the radiation component and the model state:

```
In [1]: import climt

In [2]: import matplotlib.pyplot as plt

In [3]: import numpy as np

# Two new imports
In [4]: from sympl import AdamsBashforth

In [5]: from datetime import timedelta

# Create some components
In [6]: radiation = climt.GrayLongwaveRadiation()

# Get a state dictionary filled with required quantities
# for the components to run
In [7]: state = climt.get_default_state([radiation])
```

We have two new imports, `AdamsBashforth` and `timedelta`. The former is a numerical integrator which will step the model forward in time, and the latter is a standard python module which will be used to represent the time step of the model.

Now, to create the integrator and the timestep:

```
In [8]: model_time_step = timedelta(hours=1)

In [9]: model = AdamsBashforth([radiation])
```

We now have a model ready to run! The integrator will return the new model state and any diagnostics that `radiation` has generated. We can then update the current model state with the new model state and continue to step the model forward in time:

```
In [10]: for step in range(10):
   ....:     diagnostics, new_state = model(state, model_time_step)
   ....:     ''' Update state with diagnostics.
   ....:     This updated state can be saved if necessary '''
   ....:     state.update(diagnostics)
   ....:     '''Update state quantities'''
   ....:     state.update(new_state)
   ....:     '''Update model time'''
   ....:     state['time'] += model_time_step
   ....:     '''See if the maximum temperature is changing'''
   ....:     print(state['time'], ': ', state['air_temperature'].max().values)
   ....:
2000-01-01 01:00:00 :  289.96882567458607
2000-01-01 02:00:00 :  289.9377097152726
2000-01-01 03:00:00 :  289.9066289418279
2000-01-01 04:00:00 :  289.8755896761563
2000-01-01 05:00:00 :  289.844584776716
2000-01-01 06:00:00 :  289.81362324988896
2000-01-01 07:00:00 :  289.78269480027757
2000-01-01 08:00:00 :  289.7518063986025
2000-01-01 09:00:00 :  289.72095925683254
2000-01-01 10:00:00 :  289.6901474644026
```

And voila, we have a model that actually evolves over time! Many example scripts that illustrate standard model configurations used in climate modelling are available in the github repository. These scripts include examples which setup graphics to view the evolution of the model over time.

**Note:** A more user friendly API called `Federation` will be available in a later version of climt. However, setting up models is easy enough even without `Federation` once you get used to the workflow.

CHAPTER 6

# Component Types

Deriving from Sympl, most components in climt are either `TendencyComponent`, `DiagnosticComponent`, `ImplicitTendencyComponent` or `Stepper`.

- `TendencyComponent` takes the model state as input and returns tendencies and optional diagnostics. A radiation component is a good example of such components: any radiation component returns the heating rate of each layer of the atmosphere (in $degK/s$)
- `DiagnosticComponent` takes the model state as input and returns some other diagnostic quantities. An example would be a component that takes the model state and returns the optical depth.
- `Stepper` takes the model state, and returns new values for some some quantities in the model state. A dynamical core is a good example, which returns new values of winds, temperature and pressure.
- `ImplicitTendencyComponent` takes the model state and outputs tendencies (like a `TendencyComponent`) but require the model timestep (like a `Stepper`) for various reasons, including to ensure that a vertical CFL criterion is met.

You can read more about this schema of model components in Sympl's documentation.

# CHAPTER 7

# Configuring climt

A typical climate model allows the user the following kinds of configuration:

- Algorithmic Configuration: Changing the "tuning" parameters of the algorithms underlying various components. For example, the kind of advection used in a dynamical core could be configured.
- Memory/Array Configuration: Deciding the grid shapes, and distribution over multiple processors, if using MPI.
- "Physical" Configuration: Choosing the physical constants that are used during the simulation. For example, gravitational acceleration or planetary rotation rate could be varied.
- Behavioural Configuration: Allows modification of component behaviour. For example, radiative transfer components are called only once every N (>> 1) time steps and the output is kept constant for the remaining N-1 time steps.
- Compositional Configuration: Describing the components that make up the model, the order in which components need to be called, among other things.

Climate models can be configured in many ways, including hard coded configurations, namelists, shell environment variables. These diverse ways of configuring a climate model make it difficult to keep track of all configurations and changes made.

climt aims to keep all configuration in the main run script, but separated logically to ensure the script is still intuitive to read.

## 7.1 Algorithmic Configuration

Configuring the algorithm used by each component is done by various keyword arguments passed to the component while creating it. See, for example, the documentation for `climt.RRTMGShortwave`.

## 7.2 Memory/Array Configuration

climt does not yet support MPI, so there is no API yet to handle distributed arrays. However, the shape of arrays used by a model can be set while calling `climt.get_default_state()`. See, for example, the configuration of arrays in a GCM.

## 7.3 Physical Configuration

climt provides an interface to set and reset constants required by various components. The constants are put into different categories (`boltzmann_constant` is a 'physical constant' whereas `planetary_rotation_rate` is a 'planetary constant', for example).

The constants can be reset to their default values so that climt is in a known state at the end of a simulation. In the future, climt will provide a context manager to clean up modified constants at the end of a run.

You can read more about this functionality in *General Utilities*.

## 7.4 Interfacial Configuration

Wrappers are the preferred way of changing the inputs or outputs of a component to make it apparently work in a different way.

- Piecewise constant output: Computationally expensive modules like radiative transfer are sometimes called only once every few timesteps, and the same values is used for the intermediate timesteps of a model. For example a GCM with a time step of 10 minutes might only call radiation after 1 hour of model time has elapsed. To allow for such behaviour, `sympl.UpdateFrequencyWrapper` can be used. See how this can be used practically in this example.
- TendencyComponent version: Spectral dynamical cores step the model forward in spectral space, and therefore, they do not play well with `Stepper` components that step forward the model in grid space. Typically, this is handled by finite differencing the output of Stepper components and providing them as time tendencies. `Stepper` components can be wrapped with `sympl.TimeDifferencingWrapper` which returns a component which provides the time differenced tendencies. The time differencing is done using a first order scheme:

  $\frac{dX}{dt} = (X_{out} - X_{in})/\delta t$.

  See how this is used in the Grey GCM.
- Scaled version: Very often, we perform experiments where we want to study the sensitivity of the simulation to a particular quantity or the effect of a certain quantity on the output (mechanism denial). This is in some instances done by scaling the quantity or setting it to zero (which is also a scaling). To allow for this kind of modification, `sympl.ScalingWrapper` can be used. This is a method available to all kinds of components (Stepper, TendencyComponent, etc.,). See the documentation for this method in the description of the base components in *Components*.

## 7.5 Compositional Configuration

This kind of configuration will allow the automatic building of models given certain components selected by the user. Currently, the user has to write the script to build the model and run it. It is clear that a lot of this code is repetitive and can be replaced by an entity (Which will be called `Federation`).

**Note:** This functionality is currently unavailble, and will be present in a future version of climt.

# CHAPTER 8

# Components

This page documents the different components available through climt.

## 8.1 Dynamics

| |
|---|
| `GFSDynamicalCore` |
| `GFSDynamicalCore.__call__` |

## 8.2 Radiation

| | |
|---|---|
| *`RRTMGLongwave`*([calculate_change_up_flux, . . . ]) | The Rapid Radiative Transfer Model (RRTMG). |
| *`RRTMGLongwave.__call__`*(state) | Gets tendencies and diagnostics from the passed model state. |
| *`RRTMGShortwave`*([cloud_overlap_method, . . . ]) | The Rapid Radiative Transfer Model (RRTMG). |
| *`RRTMGShortwave.__call__`*(state) | Gets tendencies and diagnostics from the passed model state. |
| *`GrayLongwaveRadiation`*([. . . ]) | |
| *`GrayLongwaveRadiation.__call__`*(state) | Gets tendencies and diagnostics from the passed model state. |
| *`Frierson06LongwaveOpticalDepth`*([. . . ]) | |
| *`Frierson06LongwaveOpticalDepth.__call__`*(state) | Gets diagnostics from the passed model state. |
| *`Instellation`*(**kwargs) | Calculates the zenith angle and star-planet correction factor given orbital parameters. |
| *`Instellation.__call__`*(state) | Gets diagnostics from the passed model state. |

### 8.2.1 climt.RRTMGLongwave

**class** climt.**RRTMGLongwave**(*calculate_change_up_flux=False, cloud_overlap_method=None, cloud_optical_properties='liquid_and_ice_clouds', cloud_ice_properties='ebert_curry_two', cloud_liquid_water_properties='radius_dependent_absorption', calculate_interface_temperature=True, mcica=False, random_number_generator='mersenne_twister', **kwargs*)

The Rapid Radiative Transfer Model (RRTMG).

This module wraps RRTMG for longwave radiation (i.e, emission from the earth's surface).

**__init__**(*calculate_change_up_flux=False, cloud_overlap_method=None, cloud_optical_properties='liquid_and_ice_clouds', cloud_ice_properties='ebert_curry_two', cloud_liquid_water_properties='radius_dependent_absorption', calculate_interface_temperature=True, mcica=False, random_number_generator='mersenne_twister', **kwargs*)

**Parameters**

- **calculate_change_up_flux** (*bool*) – calculate derivative of flux change with respect to surface temperature alone. Can be used to adjust fluxes in between radiation calls only due to change of surface temperature. Default value is `False`, meaning this quantity is not calculated.
- **cloud_overlap_method** (*string*) – Choose the method to do overlap with:
  - `clear_only` = Clear only (no clouds)
  - `random` = Random
  - `maximum_random` = Maximum/Random
  - `maximum` = Maximum.
- **cloud_optical_properties** (*string*) – Choose how cloud optical properties are calculated:
  - `direct_input` = Both cloud fraction and cloud optical depth are input directly. Other cloud properties (ie cloud particle size) are irrelevant.
  - `single_cloud_type` = Cloud fraction and cloud physical properties are input, ice and liquid clouds are treated together, cloud absorptivity is a constant value (0.060241). Not available with McICA.
  - `liquid_and_ice_clouds` = Cloud fraction and cloud physical properties are input, ice and liquid clouds are treated separately. Cloud optical depth is calculated from the cloud ice and water particle sizes and the mass content of cloud ice and cloud water.
- **cloud_ice_properties** (*string*) – set bounds on ice particle size. This is not used if 'cloud_optical_properties' == 'direct_input'
  - `ebert_curry_one` = ice particle has effective radius >= 10.0 micron [Ebert and Curry 1992]
  - `ebert_curry_two` = ice particle has effective radius between 13.0 and 130.0 micron [Ebert and Curry 1992]
  - `key_streamer_manual` = ice particle has effective radius between 5.0 and 131.0 micron [Key, Streamer Ref. Manual, 1996]
  - `fu` = ice particle has generalised effective size (dge) between 5.0 and 140.0 micron [Fu, 1996]. (dge = 1.0315 * r_ec)

Default value is 0.

- **`cloud_liquid_water_properties`** (*string*) – set treatment of cloud liquid water. This is not used if 'cloud_optical_properties' == 'direct_input'.
  - `radius_independent_absorption` = use radius independent absorption coefficient
  - `radius_dependent_absorption` = use radius dependent absorption coefficient (radius between 2.5 and 60 micron)
- **`calculate_interface_temperature`** (*bool*) – if `True`, the interface temperature is calculated internally using a weighted interpolation routine. If `False`, the quantity called `air_temperature_on_interface_levels` in the input state needs to be manually updated by user code.
- **`mcica`** (*bool*) –
  - mcica = True: use the McICA version of the longwave component of RRTMG
  - mcica = False: use the nomcica version of the longwave component of RRTMG
- **`random_number_generator`** (*string*) – Different methods of generating random numbers for McICA. * `kissvec` * `mersenne_twister`

### Methods

| | |
|---|---|
| `__init__`([calculate_change_up_flux, . . . ]) | **param calculate_change_up_flux** calculate derivative of flux change with respect to |
| `array_call`(state) | Gets tendencies and diagnostics from the passed model state. |

### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `name` |
| `num_longwave_bands` |
| `num_reduced_g_intervals` |
| `rrtm_iplon` |
| `tendencies_in_diagnostics` |
| `tendency_properties` |
| `tracer_tendency_time_unit` |
| `uses_tracers` |

## 8.2.2 climt.RRTMGLongwave.__call__

`RRTMGLongwave.`**`__call__`**(*state*)

Gets tendencies and diagnostics from the passed model state.

**Parameters** **`state`** (*dict*) – A model state dictionary satisfying the input_properties of this object.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

### 8.2.3 climt.RRTMGShortwave

**class** climt.**RRTMGShortwave**(*cloud_overlap_method=None*, *cloud_optical_properties='liquid_and_ice_clouds'*, *cloud_ice_properties='ebert_curry_two'*, *cloud_liquid_water_properties='radius_dependent_absorption'*, *solar_variability_method=0*, *use_solar_constant_from_fortran=False*, *ignore_day_of_year=False*, *facular_sunspot_amplitude=None*, *solar_variability_by_band=None*, *aerosol_type='no_aerosol'*, *mcica=False*, *random_number_generator='mersenne_twister'*, ***kwargs*)

The Rapid Radiative Transfer Model (RRTMG).

This module wraps RRTMG for shortwave radiation (i.e, emission from the sun).

**__init__**(*cloud_overlap_method=None*, *cloud_optical_properties='liquid_and_ice_clouds'*, *cloud_ice_properties='ebert_curry_two'*, *cloud_liquid_water_properties='radius_dependent_absorption'*, *solar_variability_method=0*, *use_solar_constant_from_fortran=False*, *ignore_day_of_year=False*, *facular_sunspot_amplitude=None*, *solar_variability_by_band=None*, *aerosol_type='no_aerosol'*, *mcica=False*, *random_number_generator='mersenne_twister'*, ***kwargs*)

**Parameters**

- **`cloud_overlap_method`** (*`int`*) – Choose the method to do overlap with:
  - 'clear_only' = Clear only (no clouds)
  - 'random' = Random
  - 'maximum_random' = Maximum/Random
  - 'maximum' = Maximum.
- **`cloud_optical_properties`** (*`string`*) – Choose how cloud optical properties are calculated:
  - `direct_input` = Cloud fraction, cloud optical depth, single scattering albedo, cloud asymmetry parameter and cloud forward scattering fraction are input. Cloud forward scattering fraction is used to scale the optical depth, single scattering albedo and asymmetry parameter. The latter three parameters are then used in the radiative transfer calculations. Other cloud properties (ie cloud particle size) are irrelevant.
  - `single_cloud_type` = Cloud fraction and cloud physical properties are input, ice and liquid clouds are treated together, cloud absorptivity is a constant value (0.060241). Not available with McICA.
  - `liquid_and_ice_clouds` = Cloud fraction and cloud physical properties are input, ice and liquid clouds are treated separately. Cloud optical depth, single scattering

albedo and cloud asymmetry parameter are calculated from the cloud ice and water particle sizes and the mass content of cloud ice and cloud water.

- **cloud_ice_properties** (*string*) – set bounds on ice particle size. This is not used if 'cloud_optical_properties' == 'direct_input'.
  - `ebert_curry_one` = ice particle has effective radius >= 10.0 micron [Ebert and Curry 1992] Not available with McICA.
  - `ebert_curry_two` = ice particle has effective radius between 13.0 and 130.0 micron [Ebert and Curry 1992]
  - `key_streamer_manual` = ice particle has effective radius between 5.0 and 131.0 micron [Key, Streamer Ref. Manual, 1996]
  - `fu` = ice particle has generalised effective size (dge) between 5.0 and 140.0 micron [Fu, 1996]. (dge = 1.0315 * r_ec)

  Default value is 0.
- **cloud_liquid_water_properties** (*string*) – set treatment of cloud liquid water. This is not used if 'cloud_optical_properties' == 'direct_input'.
  - `radius_independent_absorption` = use radius independent absorption coefficient Not available with McICA.
  - `radius_dependent_absorption` = use radius dependent absorption coefficient (radius between 2.5 and 60 micron)
- **solar_variability_method** (*int*) – set the solar variability model used by RRTMG.
  - solar_variability_method = -1:
    * If `use_solar_constant_from_fortran = True`: No solar variability and no solar cycle with a solar constant of 1368.22 $Wm^{-2}$.
    * If `use_solar_constant_from_fortran = False`: Solar variability defined by setting non-zero scale factors in `solar_variability_by_band`.
  - solar_variability_method = 0:
    * If `use_solar_constant_from_fortran = True`: No solar variability and no solar cycle with a solar constant of 1360.85 $Wm^{-2}$, with facular and sunspot effects fixed to the mean of solar cycles 13-24.
    * If `use_solar_constant_from_fortran = False`: No solar variability and no solar cycle.
  - solar_variability_method = 1: Solar variability using the NRLSSI2 solar model with solar cycle contribution determined by `solar_cycle_fraction` in the model state, and facular and sunspot adjustment scale factors specified in `facular_sunspot_amplitude`.
  - solar_variability_method = 2: Solar variability using the NRLSSI2 solar model using solar cycle determined by direct specification of **Mg** (facular) and **SB** (sunspot) indices provided in `facular_sunspot_amplitude`. `solar_constant` is ignored.
  - solar_variability_method = 3:
  - If `use_internal_solar_constant = True`: No solar variability and no solar cycle with a solar constant of 1360.85 $Wm^{-2}$.

  - If `use_internal_solar_constant = False`: scale factors in `solar_variability_by_band`.

- **`use_solar_constant_from_fortran`** (*bool*) – If `False`, the solar constant is taken from the constants library. The default value is `False`.
- **`ignore_day_of_year`** (*bool*) – If `True`, the solar output does not vary by day of year (i.e, higher close to the solstices and lesser close to the equinoxes). Default value is `False`.
- **`facular_sunspot_amplitude`** (*array of dimension 2*) – Facular and Sunspot amplitude variability parameters, described previously.
- **`solar_variability_by_band`** (*array of dimension 14 = number of spectral bands*) – scale factors for solar variability in all spectral bands.
- **`aerosol_type`** (*string*) – Type of aerosol inputs to RRTMG.
  - `no_aerosol`: No Aerosol.
  - `ecmwf`: ECMWF method. Requires aerosol optical depth at 55 micron as the state quantity `aerosol_optical_depth_at_55_micron`.
  - `all_aerosol_properties`: Input all aerosol optical properties.
- **`mcica`** (*bool*) –
  - mcica = True: use the McICA version for the shortwave component of RRTMG
  - mcica = False: use the nomcica version for the shortwave component of RRTMG
- **`random_number_generator`** (*string*) – Different methods of generating random numbers for McICA. * `kissvec` * `mersenne_twister`

## Methods

| | |
|---|---|
| `__init__`([cloud_overlap_method, . . . ]) | **param cloud_overlap_method** Choose the method to do overlap with: |
| `array_call`(state) | Get heating tendencies and shortwave fluxes. |

## Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `name` |
| `num_ecmwf_aerosols` |
| `num_reduced_g_intervals` |
| `num_shortwave_bands` |
| `rrtm_iplon` |
| `tendencies_in_diagnostics` |
| `tendency_properties` |
| `tracer_tendency_time_unit` |
| `uses_tracers` |

### 8.2.4 climt.RRTMGShortwave.__call__

RRTMGShortwave.**__call__**(*state*)

Gets tendencies and diagnostics from the passed model state.

**Parameters** **state** (*dict*) – A model state dictionary satisfying the input_properties of this object.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- KeyError – If a required quantity is missing from the state.
- InvalidStateError – If state is not a valid input for the TendencyComponent instance.

### 8.2.5 climt.GrayLongwaveRadiation

**class** climt.**GrayLongwaveRadiation**(*tendencies_in_diagnostics=False*, *name=None*)

**__init__**(*tendencies_in_diagnostics=False*, *name=None*)

Initializes the Stepper object.

**Parameters**

- **tendencies_in_diagnostics** (*bool, optional*) – A boolean indicating whether this object will put tendencies of quantities in its diagnostic output.
- **name** (*string, optional*) – A label to be used for this object, for example as would be used for Y in the name "X_tendency_from_Y". By default the class name in lowercase is used.

#### Methods

| | |
|---|---|
| *__init__*([tendencies_in_diagnostics, name]) | Initializes the Stepper object. |
| array_call(state) | Gets tendencies and diagnostics from the passed model state. |

#### Attributes

| |
|---|
| diagnostic_properties |
| input_properties |
| name |
| tendencies_in_diagnostics |
| tendency_properties |
| tracer_tendency_time_unit |

Continued on next page

Table 8 – continued from previous page

| |
|---|
| `uses_tracers` |

### 8.2.6 climt.GrayLongwaveRadiation.__call__

`GrayLongwaveRadiation.__call__`(*state*)

Gets tendencies and diagnostics from the passed model state.

**Parameters** **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

### 8.2.7 climt.Frierson06LongwaveOpticalDepth

**class** `climt.`**`Frierson06LongwaveOpticalDepth`**(*linear_optical_depth_parameter=0.1*, *longwave_optical_depth_at_equator=6*, *longwave_optical_depth_at_poles=1.5*, ***kwargs*)

**`__init__`**(*linear_optical_depth_parameter=0.1*, *longwave_optical_depth_at_equator=6*, *longwave_optical_depth_at_poles=1.5*, ***kwargs*)

**Parameters**

- **`linear_optical_depth_parameter`** (*`float, optional`*) – The constant $f_l$ which determines how much of the variation of $\tau$ with pressure is linear rather than quartic. $\tau = \tau_0[f_l \frac{p}{p_s} + (1-f_l)(\frac{p}{p_s})^4]$ Default is 0.1 as in [Frierson et al., 2006].
- **`longwave_optical_depth_at_equator`** (*`float, optional`*) – The value of $\tau_0$ at the equator. Default is 6 as in [Frierson et al., 2006].
- **`longwave_optical_depth_at_poles`** (*`float, optional`*) – The value of $\tau_0$ at the poles. Default is 1.5 as in [Frierson et al., 2006].

#### Methods

| | |
|---|---|
| *`__init__`*([linear_optical_depth_parameter, . . . ]) | **param linear_optical_depth_parameter** The constant $f_l$ which |
| `array_call`(state) | Gets diagnostics from the passed model state. |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |

### 8.2.8 climt.Frierson06LongwaveOpticalDepth.__call__

`Frierson06LongwaveOpticalDepth.`**`__call__`**`(`*state*`)`

Gets diagnostics from the passed model state.

**Parameters** **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.

**Returns** **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Return type** dict

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

### 8.2.9 climt.Instellation

**`class`** `climt.`**`Instellation`**`(`***kwargs*`)`

Calculates the zenith angle and star-planet correction factor given orbital parameters. Currently useful only for Earth-sun system.

**`__init__`**`(`***kwargs*`)`

Initializes the Stepper object.

#### Methods

| | |
|---|---|
| *`__init__`*(**kwargs) | Initializes the Stepper object. |
| `array_call`(state) | Calculate zenith angle. |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |

### 8.2.10 climt.Instellation.__call__

`Instellation.`**`__call__`**`(`*state*`)`

Gets diagnostics from the passed model state.

**Parameters** **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.

**Returns** **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Return type** dict

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

## 8.3 Convection

| | |
|---|---|
| *EmanuelConvection*([...]) | The Emanuel convection scheme from **'[Emanuel and Zivkovic-Rothman]'_** |
| *EmanuelConvection.__call__*(state, timestep) | Gets tendencies and diagnostics from the passed model state. |
| *DryConvectiveAdjustment*([...]) | A conservative scheme to keep the temperature profile close to the dry adiabat if it is super-adiabatic. |
| *DryConvectiveAdjustment.__call__*(state, timestep) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |

### 8.3.1 climt.EmanuelConvection

**class** climt.**EmanuelConvection**(*minimum_convecting_layer=1*, *autoconversion_water_content_threshold=0.0011*, *autoconversion_temperature_threshold=-55*, *entrainment_mixing_coefficient=1.5*, *downdraft_area_fraction=0.05*, *precipitation_fraction_outside_cloud=0.12*, *speed_water_droplets=50.0*, *speed_snow=5.5*, *rain_evaporation_coefficient=1.0*, *snow_evaporation_coefficient=0.8*, *convective_momentum_transfer_coefficient=0.7*, *downdraft_surface_velocity_coefficient=10.0*, *convection_bouyancy_threshold=0.9*, *mass_flux_relaxation_rate=0.1*, *mass_flux_damping_rate=0.1*, *reference_mass_flux_timescale=300.0*, ***kwargs*)

The Emanuel convection scheme from [Emanuel and Zivkovic-Rothman]

**__init__**(*minimum_convecting_layer=1*, *autoconversion_water_content_threshold=0.0011*, *autoconversion_temperature_threshold=-55*, *entrainment_mixing_coefficient=1.5*, *downdraft_area_fraction=0.05*, *precipitation_fraction_outside_cloud=0.12*, *speed_water_droplets=50.0*, *speed_snow=5.5*, *rain_evaporation_coefficient=1.0*, *snow_evaporation_coefficient=0.8*, *convective_momentum_transfer_coefficient=0.7*, *downdraft_surface_velocity_coefficient=10.0*, *convection_bouyancy_threshold=0.9*, *mass_flux_relaxation_rate=0.1*, *mass_flux_damping_rate=0.1*, *reference_mass_flux_timescale=300.0*, ***kwargs*)

**Parameters**

- **minimum_convecting_layer** (*int, optional*) – The least model level from which convection can be initiated. Normally set to `1` if using bulk PBL schemes. Else, it should be set to the first model level at which the temperature is defined.
- **autoconversion_water_content_threshold** (*float, optional*) –

The amount of water vapour in $kg/kg$ above which condensation occurs in warm (above freezing point of water) clouds. This value linearly reduces to zero between the freezing point and the `autoconversion_temperature_threshold`.

- **autoconversion_temperature_threshold** (*float, optional*) – The temperature in $°C$ below which all water vapour is converted to rain/snow.
- **entrainment_mixing_coefficient** (*float, optional*) – The coefficient of mixing for entrainment of environmental air into the cloud.
- **downdraft_area_fraction** (*float, optional*) – The fractional area covered by unsaturated downdrafts.
- **precipitation_fraction_outside_cloud** (*float, optional*) – The fraction of precipitation falling outside the cloud.
- **speed_water_droplets** (*float, optional*) – The speed of descent of water droplets in $Pa/s$.
- **speed_snow** (*float, optional*) – The speed of descent of snow in $Pa/s$.
- **rain_evaporation_coefficient** (*float, optional*) – Coefficient governing the rate of evaporation of rain.
- **snow_evaporation_coefficient** (*float, optional*) – Coefficient governing the rate of evaporation of snow.
- **convective_momentum_transfer_coefficient** (*float, optional*) – Coefficient **between 0 and 1** governing momentum transport by clouds. A value of 1 **shuts off** momentum transport.
- **downdraft_surface_velocity_coefficient** (*float, optional*) – Coefficient mulitplying the downdraft mass flux to calculate the downdraft velocity scale.
- **convection_bouyancy_threshold** (*float, optional*) – The maximum negative temperature perturbation in $degK$ a parcel can have below the temperature at its level of free convection. If difference is greater, and previous cloud base mass flux is zero, there is no convection.
- **mass_flux_relaxation_rate** (*float, optional*) – Coefficient governing the rate of relaxation to subcloud-layer quasi-equilibrium.
- **mass_flux_damping_rate** (*float, optional*) – Coefficient which damps the currently calculated mass flux towards the value from the previous time step.
- **reference_mass_flux_timescale** (*float, optional*) – Timescale used to calculate the actual damping coefficient along with `mass_flux_damping_rate` and the current time step.

### Methods

| | |
|---|---|
| `__init__`([minimum_convecting_layer, . . . ]) | **param minimum_convecting_layer** The least model level from which convection can be initiated. Normally set |

Continued on next page

Table 14 – continued from previous page

| | |
|---|---|
| `array_call`(raw_state, timestep) | Get convective heating and moistening. |

### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `name` |
| `tendencies_in_diagnostics` |
| `tendency_properties` |
| `tracer_tendency_time_unit` |
| `uses_tracers` |

## 8.3.2 climt.EmanuelConvection.__call__

`EmanuelConvection.`**`__call__`**(*state*, *timestep*)

Gets tendencies and diagnostics from the passed model state.

**Parameters**

- **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.
- **`timestep`** (*`timedelta`*) – The time over which the model is being stepped.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

## 8.3.3 climt.DryConvectiveAdjustment

**class** `climt.`**`DryConvectiveAdjustment`**(*tendencies_in_diagnostics=False*, *name=None*)

A conservative scheme to keep the temperature profile close to the dry adiabat if it is super-adiabatic.

**`__init__`**(*tendencies_in_diagnostics=False*, *name=None*)

Initializes the Stepper object.

**Parameters**

- **`tendencies_in_diagnostics`** (*`bool, optional`*) – A boolean indicating whether this object will put tendencies of quantities in its diagnostic output based on first order time differencing of output values.
- **`name`** (*`string, optional`*) – A label to be used for this object, for example as would be used for Y in the name "X_tendency_from_Y". By default the class name in lowercase is used.

#### Methods

| | |
|---|---|
| `__init__`([tendencies_in_diagnostics, name]) | Initializes the Stepper object. |
| `array_call`(state, time_step) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |
| `gas_constant`(q) | Calculate gas constant based on amount of q |
| `heat_capacity`(q) | Calculate heat capacity based on amount of q |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `output_properties` |
| `tendencies_in_diagnostics` |
| `time_unit_name` |
| `time_unit_timedelta` |
| `tracer_dims` |
| `uses_tracers` |

### 8.3.4 climt.DryConvectiveAdjustment.__call__

`DryConvectiveAdjustment.`**`__call__`**(*state*, *timestep*)

Gets diagnostics from the current model state and steps the state forward in time according to the timestep.

**Parameters**

- **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.
- **`timestep`** (*`timedelta`*) – The amount of time to step forward.

**Returns**

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the timestep after input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Stepper instance for other reasons.

## 8.4 Surface Processes

| | |
|---|---|
| *`SimplePhysics`*([simulate_cyclone, . . . ]) | Interface to the simple physics package. |
| *`SimplePhysics.__call__`*(state, timestep) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |
| *`SlabSurface`*([tendencies_in_diagnostics, name]) | Calculate the surface energy balance. |

Continued on next page

Table 18 – continued from previous page

| | |
|---|---|
| `SlabSurface.__call__`(state) | Gets tendencies and diagnostics from the passed model state. |
| `BucketHydrology`([soil_moisture_max, ... ]) | Manages surface energy and moisture balance This component assumes that the surface is a slab with some heat capacity and moisture holding capacity. |
| `BucketHydrology.__call__`(state, timestep) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |

### 8.4.1 climt.SimplePhysics

**class** climt.**SimplePhysics**(*simulate_cyclone=False, large_scale_condensation=True, boundary_layer=True, surface_fluxes=True, use_external_surface_temperature=True, use_external_surface_specific_humidity=False, top_of_boundary_layer=85000.0, boundary_layer_influence_height=20000.0, drag_coefficient_heat_fluxes=0.0011, base_momentum_drag_coefficient=0.0007, wind_dependent_momentum_drag_coefficient=6.5e-05, maximum_momentum_drag_coefficient=0.002, **kwargs*)

Interface to the simple physics package.

Reed and Jablonowski 2012: title = {Idealized tropical cyclone simulations of intermediate complexity: a test case for {AGCMs}} journal = {Journal of Advances in Modeling Earth Systems}

**__init__**(*simulate_cyclone=False, large_scale_condensation=True, boundary_layer=True, surface_fluxes=True, use_external_surface_temperature=True, use_external_surface_specific_humidity=False, top_of_boundary_layer=85000.0, boundary_layer_influence_height=20000.0, drag_coefficient_heat_fluxes=0.0011, base_momentum_drag_coefficient=0.0007, wind_dependent_momentum_drag_coefficient=6.5e-05, maximum_momentum_drag_coefficient=0.002, **kwargs*)

**Parameters**

- **simulate_cyclone** (*bool*) – Option indicating whether the package must simulate a tropical cyclone. This was the original test case this physics package was used for. Default value is False.
- **large_scale_condensation** (*bool*) – Option indicating whether the package must add moisture and heating tendencies due to large scale condensation. Default value is True.
- **boundary_layer** (*bool*) – Option indicating whether the package must simulate a simple boundary layer. **It is recommended that this option remain True unless another boundary layer component is being used**. Default value is True.
- **surface_fluxes** (*bool*) – Option indicating whether the package must calculate surface fluxes. **It is recommended that this option remain True unless the fluxes are being calculated by another component**. Default value is True.
- **use_external_surface_temperature** (*bool*) – Option indicating whether the package must use surface temperature available in the model state. If False, an internally generated surface temperature is used. Default value is True.
- **top_of_boundary_layer** (*float*) – The nominal top of the boundary layer in $Pa$.
- **boundary_layer_influence_height** (*float*) – The decay of the influence of the boundary layer above `top_of_boundary_layer` in $Pa$. The in-

fluence reduces to $1/e$ times the boundary layer value at a pressure given by `top_of_boundary_layer+boundary_layer_influence_height`.

- **drag_coefficient_heat_fluxes** (*float*) – The wind speed independent drag coefficient for latent and sensible heat fluxes.
- **base_momentum_drag_coefficient** (*float*) – The minimum drag coefficient for winds.
- **wind_dependent_momentum_drag_coefficient** (*float*) – The part of the momentum drag coefficient that depends on the surface wind speed. The total drag coefficient is given by `base_momentum_drag_coefficient + wind_dependent_momentum_drag_coefficient*u_base`, where `u_base` is the surface wind speed.
- **maximum_momentum_drag_coefficient** (*float*) – This drag coefficient is used for surface wind speeds exceeding $20m/s$.

### Methods

| | |
|---|---|
| `__init__`([simulate_cyclone, . . . ]) | **param simulate_cyclone** Option indicating whether the package must |
| `array_call`(state, timestep) | Calculate surface and boundary layer tendencies. |

### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `output_properties` |
| `tendencies_in_diagnostics` |
| `time_unit_name` |
| `time_unit_timedelta` |
| `tracer_dims` |
| `uses_tracers` |

## 8.4.2 climt.SimplePhysics.__call__

`SimplePhysics.`**`__call__`**(*state*, *timestep*)

Gets diagnostics from the current model state and steps the state forward in time according to the timestep.

**Parameters**

- **state** (*dict*) – A model state dictionary satisfying the input_properties of this object.
- **timestep** (*timedelta*) – The amount of time to step forward.

**Returns**

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the timestep after input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Stepper instance for other reasons.

### 8.4.3 climt.SlabSurface

**class** climt.**SlabSurface**(*tendencies_in_diagnostics=False*, *name=None*)

Calculate the surface energy balance.

This component assumes the surface is a slab of possibly varying heat capacity, and calculates the surface temperature.

**__init__**(*tendencies_in_diagnostics=False*, *name=None*)

Initializes the Stepper object.

**Parameters**

- **tendencies_in_diagnostics** (*bool, optional*) – A boolean indicating whether this object will put tendencies of quantities in its diagnostic output.
- **name** (*string, optional*) – A label to be used for this object, for example as would be used for Y in the name "X_tendency_from_Y". By default the class name in lowercase is used.

#### Methods

| | |
|---|---|
| `__init__`([tendencies_in_diagnostics, name]) | Initializes the Stepper object. |
| `array_call`(raw_state) | Gets tendencies and diagnostics from the passed model state. |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `name` |
| `tendencies_in_diagnostics` |
| `tendency_properties` |
| `tracer_tendency_time_unit` |
| `uses_tracers` |

### 8.4.4 climt.SlabSurface.__call__

SlabSurface.**__call__**(*state*)

Gets tendencies and diagnostics from the passed model state.

**Parameters** **state** (*dict*) – A model state dictionary satisfying the input_properties of this object.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.

- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

### 8.4.5 climt.BucketHydrology

**class** `climt.`**`BucketHydrology`**(*soil_moisture_max=0.15*, *beta_parameter=0.75*, *specific_latent_heat_of_water=2260000*, *bulk_coefficient=0.0011*, ***kwargs*)

Manages surface energy and moisture balance This component assumes that the surface is a slab with some heat capacity and moisture holding capacity. Calculates the sensible and latent heat flux, takes precipitation values as input.

**`__init__`**(*soil_moisture_max=0.15*, *beta_parameter=0.75*, *specific_latent_heat_of_water=2260000*, *bulk_coefficient=0.0011*, ***kwargs*)

Args: soil_moisture_max:

The maximum moisture that can be held by the surface_temperature

**beta_parameter:** A constant value that is used in the beta_factor calculation.

**bulk_coefficient:** The bulk transfer coefficient that is used to calculate maximum evaporation rate and sensible heat flux

#### Methods

| | |
|---|---|
| `__init__`([soil_moisture_max, . . . ]) | Args: soil_moisture_max: The maximum moisture that can be held by the surface_temperature beta_parameter: A constant value that is used in the beta_factor calculation. |
| `array_call`(state, timestep) | Calculates sensible and latent heat flux and returns surface temperature and soil moisture after timestep. |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `output_properties` |
| `tendencies_in_diagnostics` |
| `time_unit_name` |
| `time_unit_timedelta` |
| `tracer_dims` |
| `uses_tracers` |

### 8.4.6 climt.BucketHydrology.__call__

`BucketHydrology.__call__(state, timestep)`

Gets diagnostics from the current model state and steps the state forward in time according to the timestep.

**Parameters**

- **state** (*dict*) – A model state dictionary satisfying the input_properties of this object.
- **timestep** (*timedelta*) – The amount of time to step forward.

**Returns**

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the timestep after input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Stepper instance for other reasons.

## 8.5 Ice and Snow

| | |
|---|---|
| *IceSheet*([maximum_snow_ice_height]) | 1-d snow-ice energy balance model. |
| *IceSheet.__call__*(state, timestep) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |

### 8.5.1 climt.IceSheet

`class climt.IceSheet(maximum_snow_ice_height=10, **kwargs)`

1-d snow-ice energy balance model.

`__init__(maximum_snow_ice_height=10, **kwargs)`

**Parameters**

- **maximum_snow_ice_height** (*float*) – The maximum combined height of snow and ice handled by the model in $m$.
- **levels** (*int*) – The number of levels on which temperature must be output.

#### Methods

| | |
|---|---|
| *__init__*([maximum_snow_ice_height]) | **param maximum_snow_ice_height** The maximum combined height of snow and ice handled by the model in $m$. |

Continued on next page

Table 26 – continued from previous page

| | |
|---|---|
| `array_call`(raw_state, timestep) | Gets diagnostics from the current model state and steps the state forward in time according to the timestep. |
| `calculate_new_ice_temperature`(rho, ...[, ...]) | |

#### Attributes

| |
|---|
| `diagnostic_properties` |
| `input_properties` |
| `output_properties` |
| `tendencies_in_diagnostics` |
| `time_unit_name` |
| `time_unit_timedelta` |
| `tracer_dims` |
| `uses_tracers` |

### 8.5.2 climt.IceSheet.__call__

`IceSheet.`**`__call__`**(*state*, *timestep*)

Gets diagnostics from the current model state and steps the state forward in time according to the timestep.

**Parameters**

- **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.
- **`timestep`** (*`timedelta`*) – The amount of time to step forward.

**Returns**

- **diagnostics** (*dict*) – Diagnostics from the timestep of the input state.
- **new_state** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the timestep after input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the Stepper instance for other reasons.

## 8.6 Test Cases

| | |
|---|---|
| *`HeldSuarez`*([sigma_boundary_layer_top, k_f, ...]) | Provide the Held-Suarez forcing. |
| *`HeldSuarez.__call__`*(state) | Gets tendencies and diagnostics from the passed model state. |
| *`DcmipInitialConditions`*([condition_type, ...]) | Climt interface to the DCMIP initial conditions. |
| *`DcmipInitialConditions.__call__`*(state) | Gets diagnostics from the passed model state. |

### 8.6.1 climt.HeldSuarez

**class** climt.**HeldSuarez**(*sigma_boundary_layer_top=0.7*, *k_f=1.1574074074074073e-05*, *k_a=2.8935185185185185e-07*, *k_s=2.8935185185185184e-06*, *equator_pole_temperature_difference=60*, *delta_theta_z=10*, ***kwargs*)

Provide the Held-Suarez forcing.

Produces the forcings proposed by Held and Suarez for the intercomparison of dynamical cores of AGCMs. Relaxes the temperature field to a zonally symmetric equilibrium state, and uses Rayleigh damping of low-level winds to represent boundary-layer friction. Details can be found in [Held and Suarez (1994)].

#### References

**Held, I. and M. Suarez, 1994:** A Proposal for the Intercomparison of the Dynamical Cores of Atmospheric General Circulation Models. Bull. Amer. Meteor. Soc., 75, 1825-1830, doi: 10.1175/1520-0477(1994)075<1825:APFTIO>2.0.CO;2.

**__init__**(*sigma_boundary_layer_top=0.7*, *k_f=1.1574074074074073e-05*, *k_a=2.8935185185185185e-07*, *k_s=2.8935185185185184e-06*, *equator_pole_temperature_difference=60*, *delta_theta_z=10*, ***kwargs*)

**Parameters**

- **sigma_boundary_layer_top** (*float*) – The height of the boundary layer top in sigma coordinates. Corresponds to $sigma_b$ in Held and Suarez, 1994. Default is 0.7.
- **k_f** (*float*) – Velocity damping coefficient at the surface in $s^{-1}$. Default is $1\ day^{-1}$.
- **k_a** (*float*) – Parameter used in defining vertical profile of the temperature damping in $s^{-1}$, as outlined in Held and Suarez, 1994. Default is $1/40\ day^{-1}$.
- **k_s** (*float*) – Parameter used in defining vertical profile of the temperature damping in $s^{-1}$, as outlined in Held and Suarez, 1994. Default is $1/4\ day^{-1}$.
- **equator_pole_temperature_difference** (*float*) – Equator to pole temperature difference, in K. Default is 60K.
- **delta_theta_z** (*float*) – Parameter used in defining the equilibrium temperature profile as outlined in Held and Suarez, 1994, in K. Default is 10K.

#### Methods

| | |
|---|---|
| *__init__*([sigma_boundary_layer_top, k_f, …]) | **param sigma_boundary_layer_top** The height of the boundary |
| array_call(raw_state) | Get the Held-Suarez forcing tendencies |

#### Attributes

| |
|---|
| diagnostic_properties |
| input_properties |

Continued on next page

Table 30 – continued from previous page

| |
|---|
| `name` |
| `tendencies_in_diagnostics` |
| `tendency_properties` |
| `tracer_tendency_time_unit` |
| `uses_tracers` |

### 8.6.2 climt.HeldSuarez.__call__

`HeldSuarez.`**`__call__`**(*state*)

Gets tendencies and diagnostics from the passed model state.

**Parameters** **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.

**Returns**

- **tendencies** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the time derivative of those quantities in units/second at the time of the input state.
- **diagnostics** (*dict*) – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

### 8.6.3 climt.DcmipInitialConditions

**`class`** `climt.`**`DcmipInitialConditions`**(*condition_type='baroclinic_wave'*, *add_perturbation=True*, *moist=False*, ***kwargs*)

Climt interface to the DCMIP initial conditions. Currently only provides interfaces to tests 4 and 5.

**`__init__`**(*condition_type='baroclinic_wave'*, *add_perturbation=True*, *moist=False*, ***kwargs*)

Initialize the DCMIP module.

**Parameters**

- **`condition_type`** (*`str, optional`*) – The type of initial conditions desired. Can be one of `'baroclinic_wave'` or `'tropical_cyclone'`.
- **`add_perturbation`** (*`bool, optional`*) – Whether a perturbation must be added. Only applies to the baroclinic wave test.

#### Methods

| | |
|---|---|
| `__init__`([condition_type, add_perturbation, ...]) | Initialize the DCMIP module. |
| `array_call`(state) | Gets diagnostics from the passed model state. |

#### Attributes

| | |
|---|---|
| `diagnostic_properties` | |
| `input_properties` | |

### 8.6.4 climt.DcmipInitialConditions.__call__

`DcmipInitialConditions.`**`__call__`**`(`*state*`)`

Gets diagnostics from the passed model state.

**Parameters** **`state`** (*`dict`*) – A model state dictionary satisfying the input_properties of this object.

**Returns** **diagnostics** – A dictionary whose keys are strings indicating state quantities and values are the value of those quantities at the time of the input state.

**Return type** dict

**Raises**

- `KeyError` – If a required quantity is missing from the state.
- `InvalidStateError` – If state is not a valid input for the TendencyComponent instance.

# CHAPTER 9

# Component Manual

## 9.1 RRTMG - The Rapid Radiative Transfer Model

### 9.1.1 Introduction

#### Treatment of Clouds

There are two methods in RRTMG, which handle clouds in different ways.

#### McICA

McICA allows us to have fractional cloud areas, by randomly assigning some wavelengths to see cloud and other wavelengths to see no cloud. For example if we have a cloud fraction of 40%, then 40% of the wavelengths see cloud, while the other 60% do not.

#### nomcica

With nomcica, all of the wavelengths see some cloud. Using this method for the shortwave we can only have cloud area fractions of zero and one, representing clear and completely overcast skies. On the other hand, if we are calculating longwave fluxes, we can also use fractional cloud area fractions.

### 9.1.2 Calculation of radiative fluxes

#### Longwave

Two properties are needed to calculate the longwave radiative fluxes; the absorptivity (or transmittance which equals one minus the absorptivity) and the emissivity. Values of these two properties are needed for each model layer, for both cloudy and clear sky regimes and for each waveband.

```
radld = radld * (1 - atrans(lev)) * (1 - efclfrac(lev,ib)) + &
        gassrc * (1 - cldfrac(lev)) + bbdtot * atot(lev) * cldfrac(lev)
```

The radiative fluxes at each layer(`radld` on the left hand side of the equations) are calculated from the radiative fluxes from the layer above (`radld` on the right hand side of the equation) and the properties of the layer. The first term in the equation above is fraction of radiative flux from the layer above that is transmitted through the layer. `atrans(lev)` is the gaseous absorptivity and `efclfrac` is the absorptivity of the cloud weighted by the cloud area fraction (here for nomcica). The other two terms in the above equation are emission terms. The first of these represents emission from gases in the area of clear sky and the second represents emission from gases and cloud from the area of cloud. The equation for the upward longwave flux `radlu` is very similar: the flux is calculated from the radiative flux from the layer below and the properties of the layer.

These equations are used with `nomcica` when the `rrtmg_cloud_overlap_method` is set to `random` and the cloud area fraction is greater than $10^{-6}$. These calculations are in the `rrtmg_lw_rtrn.f90` file, in the downward radiative transfer loop and upward radiative transfer loop respectively. These equations are also used with `McICA`. In this case, `efclfrac` is either zero or the non-weighted absorptivity of the cloud and this is allocated randomly to each waveband, with the number of waveband receiving each depending on the cloud area fraction. For `McICA`, these calculations are in the `rrtmg_lw_rtrnmc.f90` file, in the downward loop and upward loop respectively. In both files, in the downward loop there are three different ways of calculating the absorptivity, which use different approximations for the exponential of the optical depth. The one that is used depends on the optical depth of the clear sky and of the total sky.

With `nomcica`, if the `rrtmg_cloud_overlap_method` is set to any of the other options except `random`, the `rrtmg_lw_rtrnmr.f90` file is called. Then, the radiative fluxes are calculated as follows, at the end of the downward radiative transfer loop.

```
cldradd = cldradd * (1._rb - atot(lev)) + cldfrac(lev) * bbdtot * atot(lev)
clrradd = clrradd * (1._rb-atrans(lev)) + (1._rb - cldfrac(lev)) * gassrc
radld = cldradd + clrradd
```

The downward radiative flux is split into the clear sky and cloudy components, `clrradd` and `cldradd` respectively. Both components contain a transmittance term from the clear or cloudy part, respectively, of the layer above and an emission term. The emission terms are identical to those described above. The fluxes `clrradd` and `cldradd` are modified by an amount that depends on the change in cloud fraction between layers before they are used for the calculation of fluxes in the layer below.

### 9.1.3 Clouds with McICA

A brief description of the different options that can be used in RRTMG with McICA and the input parameters required in each case.

#### Cloud properties

There are three options for the RRTMG `inflag`, as given in the `climt` dictionary: rrtmg_cloud_props_dict

```
  rrtmg_cloud_props_dict = {
      'direct_input': 0,
      'single_cloud_type': 1,
      'liquid_and_ice_clouds': 2
}
```

With McICA, we cannot use `single_cloud_type`, but can choose between `direct_input` and `liquid_and_ice_clouds`. If we choose `direct_input`, we input the `longwave_optical_thickness_due_to_cloud`, `shortwave_optical_thickness_due_to_cloud`, as well as the shortwave parameters `single_scattering_albedo_due_to_cloud`, `cloud_asymmetry_parameter` and `cloud_forward_scattering_fraction`. The `cloud_forward_scattering_fraction` is used to scale the other shortwave parameters (`shortwave_optical_thickness_due_to_cloud`, `single_scattering_albedo_due_to_cloud` and `cloud_asymmetry_parameter`), but it is not directly used in the radiative transfer calculations. If the `cloud_forward_scattering_fraction` is set to zero, no scaling is applied. The other cloud properties, namely `cloud_ice_particle_size` and `cloud_water_droplet_radius`, `mass_content_of_cloud_ice_in_atmosphere_layer`, and `mass_content_of_cloud_liquid_water_in_atmosphere_layer` are completely unused. The RRTMG `iceflag` and `liqflag` are irrelevant.

On the other hand, if we choose `liquid_and_ice_clouds`, any input values for `longwave_optical_thickness_due_to_cloud`, `shortwave_optical_thickness_due_to_cloud`, `single_scattering_albedo_due_to_cloud` and `cloud_asymmetry_parameter` are irrelevant. Instead, these parameters are calculated from the cloud ice and water droplet particle sizes (`cloud_ice_particle_size` and `cloud_water_droplet_radius`), as well as the cloud ice and water paths (`mass_content_of_cloud_ice_in_atmosphere_layer`, `mass_content_of_cloud_liquid_water_in_atmosphere_layer`). The methods used for the calculations depend on the cloud ice and water properties; `iceflag` and `liqflag` in RRTMG.

Regardless of the other cloud input type, `cloud_area_fraction_in_atmosphere_layer` is required, and is used in McICA to determine how much of the wavelength spectrum sees cloud and how much does not.

## Calculation of cloud properties

### Longwave

For the longwave, the only cloud property of interest for calculating radiative fluxes in RRTMG, is the optical depth. This is calculated at the end of the longwave cldprmc submodule as:

```
taucmc(ig,lay) = ciwpmc(ig,lay) * abscoice(ig) + &
                 clwpmc(ig,lay) * abscoliq(ig)
```

Values of cloud optical depth `taucmc` are calculated for each model layer (pressure), `lay`, and each g-interval, `ig`. The cloud ice and liquid absorption coefficients (`abscoice` and `abscoliq`) are multiplied by the cloud ice and liquid water paths (`ciwpmc` and `clwpmc`) respectively, to give the ice cloud optical depth and the liquid water cloud optical depth. The cloud ice and liquid water paths are input by the user, in `climt` as `mass_content_of_cloud_ice_in_atmosphere_layer` and `mass_content_of_cloud_liquid_water_in_atmosphere_layer` respectively. The cloud ice and liquid absorption coefficients are calculated based on the ice and liquid water particle sizes (specified by the user), and this calculation depends on the choice of `iceflag` and `liqflag`.

### Shortwave

For the shortwave, there are three cloud properties, which affect the radiative flux calculation in RRTMG, namely the optical depth, the single scattering albedo and the asymmetry parameter.

The shortwave optical depth is calculated as:

```
taucmc(ig,lay) = tauice + tauliq
```

with

```
tauice = (1 - forwice(ig) + ssacoice(ig)) * ciwpmc(ig,lay) * extcoice(ig)
tauliq = (1 - forwliq(ig) + ssacoliq(ig)) * clwpmc(ig,lay) * extcoliq(ig)
```

The single scattering albedo is calculated as:

```
ssacmc(ig,lay) = (scatice + scatliq) / taucmc(ig,lay)
```

with

```
scatice = ssacoice(ig) * (1._rb - forwice(ig)) / (1._rb - forwice(ig) * ssacoice(ig))␣
↪* tauice
scatliq = ssacoliq(ig) * (1._rb - forwliq(ig)) / (1._rb - forwliq(ig) * ssacoliq(ig))␣
↪* tauliq
```

The asymmetry parameter is given by:

```
asmcmc(ig,lay) = 1.0_rb / (scatliq + scatice) * ( &
                 scatliq * (gliq(ig) - forwliq(ig)) / (1.0_rb - forwliq(ig)) + &
                 scatice * (gice(ig) - forwice(ig)) / (1.0_rb - forwice(ig)) )
```

The original RRTMG code for these calculations is at the end of the shortwave cldprmc submodule.

Values of optical depth, single scattering albedo and asymmetry parameter are calculated for each model layer (pressure), `lay`, and each g-interval, `ig`. The cloud ice and liquid water paths (`ciwpmc` and `clwpmc`) are input by the user. The other parameters (`extcoice`, `extcoliq`, `ssacoice`, `ssacoliq`, `gice`, `gliq`, `forwice`, `forwliq`) are calculated based on the ice and liquid water particle sizes and this calculation depends on the choice of `iceflag` and `liqflag`.

## Cloud ice properties

There are four options for the RRTMG `iceflag`. These are given in the `climt` dictionary: rrtmg_cloud_ice_props_dict

```
rrtmg_cloud_ice_props_dict = {
    'ebert_curry_one': 0,
    'ebert_curry_two': 1,
    'key_streamer_manual': 2,
    'fu': 3
}
```

### ebert_curry_one

For the longwave, `ebert_curry_one` gives an absorption coefficient of

`abscoice = 0.005 + 1.0 / radice`

Here, `radice` is the ice particle size and the absorption coefficient is the same for all wavebands.

`ebert_curry_one` should not be used for the shortwave component with McICA.

### ebert_curry_two

`ebert_curry_two` is the default choice for cloud ice optical properties in `climt`. In this case, the longwave absorption coefficient is calculated in the lw_cldprmc file as follows.

```
abscoice(ig) = absice1(1,ib) + absice1(2,ib)/radice
```

The absorption coefficient `abscoice` is a function of g-interval `ig` and is made up of two contributions. The first of these `absice1(1, ib)` comes from a look up table and is given in [m$^2$/ g]. `ib` provides an index for the look up table, based on the waveband of the g-interval. `absice1(2,ib)` also comes from a look up table and is given in [microns m$^2$/ g]. It is divided by `radice`, the cloud ice particle size, providing an ice particle size dependence of the absorption coefficient. Although the syntax does not emphasise it, the absorption coefficient may also depend on model layer (pressure), as `radice` can have model layer dependence. `radice` comes from the input property labeled `cloud_ice_particle_size` in `climt`. The ice particle size dependent term is more important than the independent term (`absice1(2,ib)/radice` > `absice1(1,ib)`) at all wavebands for ice particle sizes less than 88 microns. Using `ebert_curry_two`, the ice particle size must be in the range [13, 130] microns, and even for larger particle sizes (> 88), the ice particle size dependent term is more important than the independent term for four of the five wavebands.

For the shortwave, the parameters (required for the optical depth, single scattering albedo and asymmetry) are calculated in the sw_cldprmc file as follows.

```
extcoice(ig) = abari(icx) + bbari(icx)/radice
ssacoice(ig) = 1._rb - cbari(icx) - dbari(icx) * radice
gice(ig) = ebari(icx) + fbari(icx) * radice
forwice(ig) = gice(ig)*gice(ig)
```

`abari`, `bbari`, `cbari`, `dbari`, `ebari` and `fbari` are all look up tables, containing five values, which correspond to five different wavebands. The choice of waveband is indicated by `icx`. The particle size dependence comes from `radice`, so each parameter consists of both a size independent and a size dependent contribution.

### key_streamer_manual

In this case, both the longwave absorption coefficient and three of the shortwave parameters (`excoice`, `ssacoice`, `gice`) are interpolated from look up tables. Comments in the RRTMG code state that these look up tables are for a spherical ice particle parameterisation. The look up tables contain 42 values for each of the 16 longwave and 14 shortwave wavebands. The 42 values correspond to different ice particle radii, evenly spaced in the range [5, 131] microns. Ice particles must be within this range, otherwise an error is thrown.

The shortwave parameter `forwice` is calculated as the square of `gice`.

### fu

The longwave absorption coefficient and shortwave parameters `extcoice`, `ssacoice` and `gice` are interpolated from look up tables. The look up tables differ to those in `key_streamer_manual`, and comments in the RRTMG code state that the look up tables for `fu` are for a hexagonal ice particle parameterisation. The look up tables for `fu` are slightly larger than those for `key_streamer_manual`, and the range of allowed values for the ice particle size is corresponding larger ([5, 140] microns).

The shortwave parameter `forwice` is calculated from `fdelta` (again taken from look up tables) and `ssacoice` as follows.

```
forwice(ig) = fdelta(ig) + 0.5_rb / ssacoice(ig)
```

The longwave and shortwave parameter calculations can be found in the longwave cldprmc and shortwave cldprmc subroutines respectively.

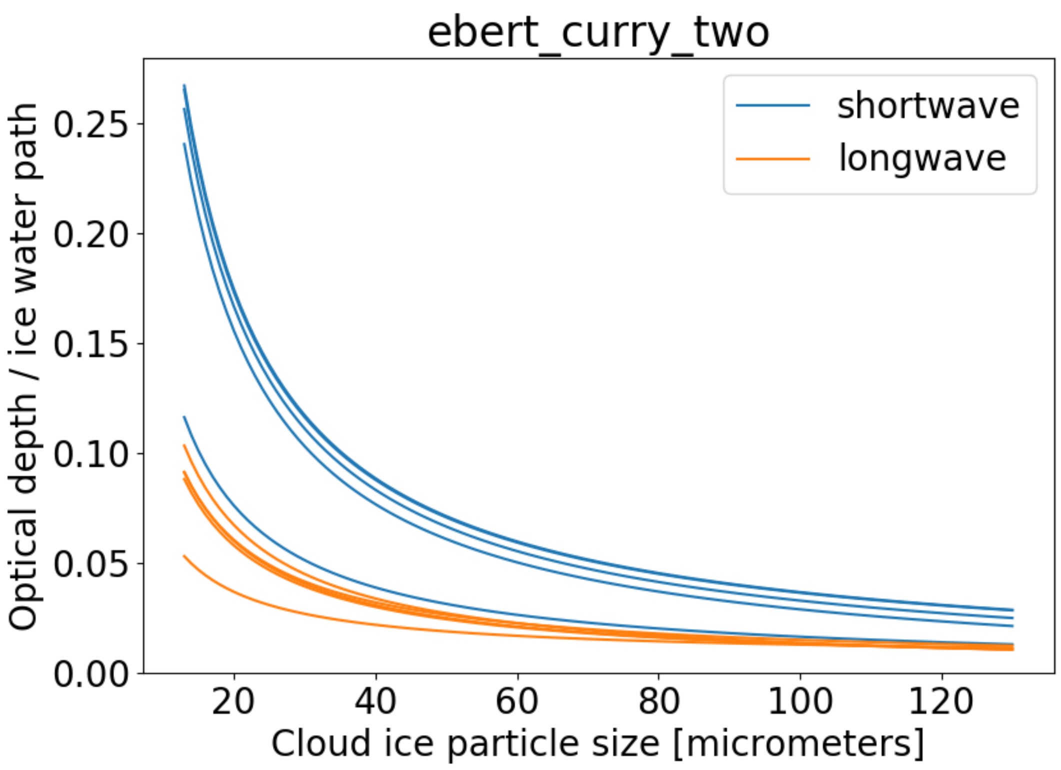

Fig. 1: *The dependence of cloud optical depth* `taucmc` *on cloud ice particle size (with an ice water path of 1), with different lines representing the different wavebands.*

### Cloud liquid properties

There are two options for the RRTMG `liqflag`. These are given in the `climt` dictionary: rrtmg_cloud_liquid_props_dict

```
rrtmg_cloud_liquid_props_dict = {
    'radius_independent_absorption': 0,
    'radius_dependent_absorption': 1
}
```

For `radius_independent_absorption`, the longwave absorption coefficient is 0.0903614 for all wavebands. This option should not be used for the shortwave.

`radius_dependent_absorption` is the default choice for cloud liquid water properties in `climt`. In this case, the longwave absorption coefficient and the shortwave parameters `extcoliq`, `ssacoliq`, and `gliq` are interpolated from look up tables. The look up tables have values for particle sizes in the range [2.5, 59.5] microns in 1 micron intervals (58 values) for each of the 16 longwave and 14 shortwave wavebands. The shortwave parameter `forwliq` is calculated as the square of `gliq`.

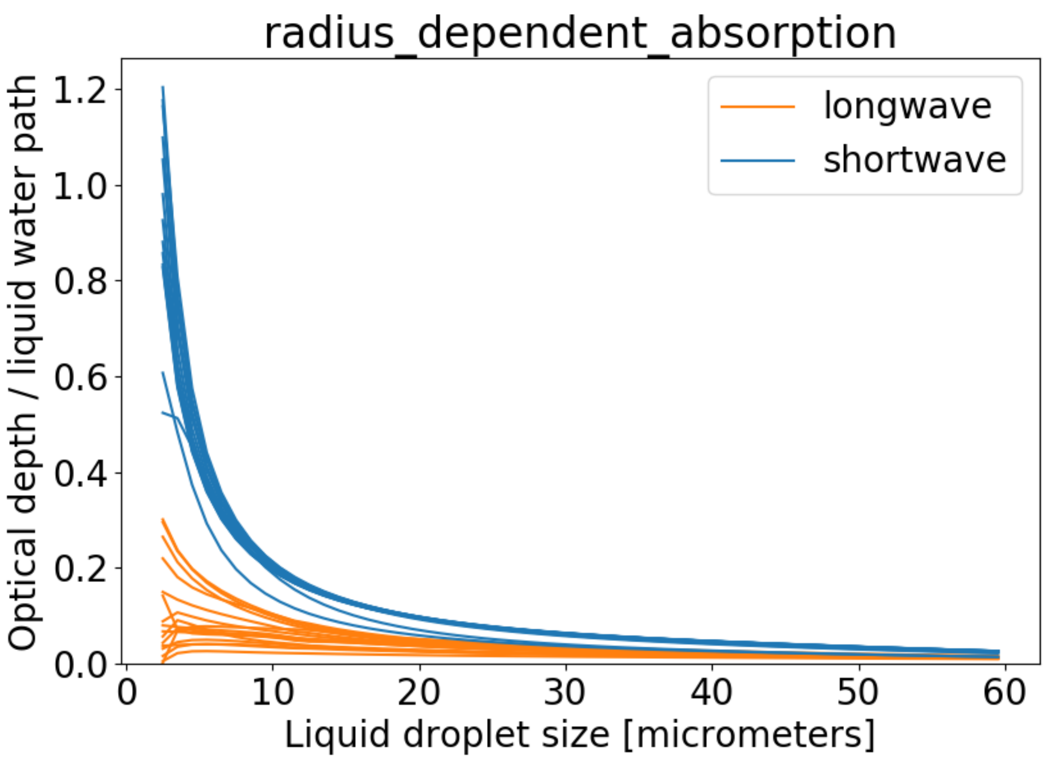

Fig. 2: *The dependence of cloud optical depth* `taucmc` *on cloud liquid water particle size (with a liquid water path of 1), with different lines representing the different wavebands.*

### Cloud overlap method

This is the RRTMG `icld` and is given in the climt dictionary: rrtmg_cloud_overlap_method_dict

```
rrtmg_cloud_overlap_method_dict = {
    'clear_only': 0,
    'random': 1,
    'maximum_random': 2,
    'maximum': 3
}
```

If we choose `clear_only`, there are no clouds, regardless of the other input. If we choose `random`, the g-intervals which see cloud are chosen randomly for each model layer. This means that there is a dependence on vertical resolution: if vertical resolution is increased, more layers contain the same cloud and a larger portion of the wavelength spectrum sees some of the cloud. With `maximum_random`, the g-intervals that see cloud in one model layer are the same as those that see cloud in a neighbouring model layer. This maximises the cloud overlap between neighbouring layers (within a single cloud). If the cloud area fraction changes between layers, the additional g-intervals that see (or don't see) cloud are assigned randomly. Therefore, if there are two clouds at different altitudes, separated by clear sky, the two clouds overlap randomly with respect to each other. If `maximum` is selected, cloud overlap is maximised both within and between clouds.

The implementation is in the mcica_subcol_gen_sw and mcica_subcol_gen_lw files, and consists firstly of assigning each g-interval a random number in the range [0, 1]. For `random`, and `maximum_random` (cases 1 and 2 in mcica_subcol_gen_sw and mcica_subcol_gen_lw), random numbers are generated for each layer, whereas for `maximum` (case 3) only one set of random numbers is generated and applied to all the layers. For `maximum_random`, the random numbers are recalculated to fulfill the assumption about overlap (this recalculation is described below). Whether a g-interval at a certain layer sees cloud depends on both the random number it has been assigned and the cloud fraction at that layer. For example, if the cloud area fraction is 30%, all g-intervals that have been assigned a random number > 0.7 (approximately 30% of the g-intervals) will see cloud. The other g-intervals will see clear-sky. If the cloud fraction is 20%, only g-intervals with a random number > 0.8 will see cloud. The recalculation of random numbers in the `maximum_random` case for a certain model layer (layer 2), considers the assigned random numbers and cloud area fraction of the layer above (layer 1). If the g-interval sees cloud in layer 1, its random number in layer 2 is changed so that it matches that in layer 1. This does not necessarily mean that it will see cloud in layer 2, because the cloud fraction could be smaller in layer 2 than layer 1 (so the requirement for seeing cloud would be increased). The random numbers for the g-intervals in layer 2, which do not see cloud in layer 1, are multiplied by one minus the cloud area fraction of layer 1, so that the set of random numbers assigned to layer 2 are still randomly distributed in the range [0, 1]. This is required so that the right proportion of g-intervals in layer 2 see cloud.

## 9.1.4 Differences in cloud input with nomcica

Regarding the options that can be used with nomcica, there are a few differences to those that can be used with McICA.

### Cloud properties

For the longwave, we can choose `single_cloud_type` in the rrtmg_cloud_props_dict. The longwave cloud optical depth is calculated as follows.

```
taucloud(lay,ib) = abscld1 * (ciwp(lay) + clwp(lay))
```

This gives us a cloud optical depth based on a single constant value, `abscld1` and the total cloud water path. Thus, for this option, the `mass_content_of_cloud_ice_in_atmosphere_layer`, and `mass_content_of_cloud_liquid_water_in_atmosphere_layer` are needed as input. `single_cloud_type` is not available for the shortwave.

If we choose `liquid_and_ice_clouds`, the calculations of the longwave and shortwave optical properties from the cloud mass and particle sizes are the same as for McICA, but are calculated for each waveband instead of each g-interval.

### Cloud overlap method

With nomcica choosing a cloud overlap of `random` in the rrtmg_cloud_overlap_method_dict is different to choosing either `maximum_random` or `maximum`. The latter two options do not differ. If we choose `random`, the radiative flux transmitted from one layer to the next does not care if it came from cloud or clear sky, whereas with `maximum_random`, the cloudy and clear fluxes are separated and treated separately from one model layer to the next.

# CHAPTER 10

# Initialisation

| | |
|---|---|
| `get_default_state`(component_list[, . . . ]) | Retrieves a reasonable initial state for the set of components given. |
| `get_grid`([nx, ny, nz, . . . ]) | **param nx** int, optional |

## 10.1 climt.get_default_state

`climt.`**`get_default_state`**`(`*component_list*, *grid_state=None*, *n_ice_interface_levels=30*`)`

Retrieves a reasonable initial state for the set of components given.

**Parameters**

- **`component_list`** (*`list`*) – Components for which to retrieve an initial state.
- **`grid_state`** (*`dict, optional`*) – An initial state containing grid quantities. If none is given, a default will be created.
- **`n_ice_interface_levels`** (*`int, optional`*) – Number of vertical interface levels to use for ice. Use None to disable the ice vertical grid.

**Returns** A reasonable initial state.

**Return type** default_state (dict)

## 10.2 climt.get_grid

`climt.`**`get_grid`**`(`*nx=None*, *ny=None*, *nz=28*, *n_ice_interface_levels=10*, *p_surf_in_Pa=None*, *p_toa_in_Pa=None*, *proportion_sigma_levels=0.1*, *proportion_isobaric_levels=0.25*, *x_name='lon'*, *y_name='lat'*, *latitude_grid='gaussian'*`)`

**Parameters**

- **nx** – int, optional Number of longitudinal points.
- **ny** – int, optional Number of latitudinal points.
- **nz** – int, optional Number of vertical mid levels.
- **n_ice_interface_levels** (*int, optional*) – Number of vertical interface levels to use for ice. Use None to disable the ice vertical grid.
- **p_surf_in_Pa** – float, optional Surface pressure in Pa.
- **x_name** – str, optional Name of latitudinal dimension
- **y_name** – str, optional Name of longitudinal dimension
- **latitude_grid** – 'gaussian' or 'regular' Type of spacing to use for the latitudinal grid.

**Returns**

**dict** A model state containing grid quantities.

**Return type** grid_state

# CHAPTER 11

# General Utilities

This documents some utility functions available in `climt`. Most of the constants functionality is inherited from `sympl`.

## 11.1 Constants

| | |
|---|---|
| *list_available_constants*() | Prints all the constants currently registered with sympl. |
| *set_constants_from_dict*(constant_descriptions) | Modify/Add constants in the library. |

### 11.1.1 climt.list_available_constants

climt.**list_available_constants**()

Prints all the constants currently registered with sympl.

### 11.1.2 climt.set_constants_from_dict

climt.**set_constants_from_dict**(*constant_descriptions*)

Modify/Add constants in the library.

**Parameters** **constant_descriptions** (*dict*) – Dictionary containing the description of the constants. The key should be the name of the constant, and the value should be a dictionary containing the following keys:

- **value (float):** The value assigned.
- **units (string):** The units of the value, e.g, m/s, J/kg.

## 11.2 Miscellaneous

| | |
|---|---|
| `mass_to_volume_mixing_ratio`(mass_mixing_ratio) | g/g or g/kg to mole/mole |
| `get_interface_values`(mid_level_values, ...) | Calculate interface values given mid-level values. |

## 11.2.1 climt.mass_to_volume_mixing_ratio

`climt.`**`mass_to_volume_mixing_ratio`**(*mass_mixing_ratio*, *molecular_weight=None*, *molecular_weight_air=28.964*)

g/g or g/kg to mole/mole

Converts from mass mixing ratio (mass per unit mass) to volume mixing ratio (volume per unit volume)

**Parameters**

- **`mass_mixing_ratio`** (*`array`*) – The quantity to be transformed in units of $g/g$.
- **`molecular_weight`** (*`float`*) – The molecular weight of the gas in $g/mol$.
- **`molecular_weight_air`** (*`float, optional`*) – The molecular weight of dry air. If it is not provided, the value for dry air on earth (28.964 g/mol) is used.

**Returns** The volume mixing ratio of the gas.

**Return type** volume_mixing_ratio (array)

**Raises** `ValueError` – if the molecular weight is not provided.

## 11.2.2 climt.get_interface_values

`climt.`**`get_interface_values`**(*mid_level_values*, *surface_value*, *mid_level_pressure*, *interface_level_pressure*)

Calculate interface values given mid-level values.

Given 3D values of a quantity on model mid levels (cell centers) and the 2D surface value, return the 3D values of that quantity on model full levels (cell interfaces). If the vertical dimension of `mid_level_values` is length K, the returned array will have a vertical dimension of length K+1.

Routine borrowed from CESM (radiation.F90 in rrtmg folder)

**Parameters**

- **`mid_level_values`** (*`array`*) – The values of the quantity on mid-levels.
- **`surface_value`** (*`array`*) – The value of the quantity at the surface. Must be in the same units as `mid_level_values`
- **`mid_level_pressure`** (*`array`*) – Pressure values on mid-levels. Can be in any units.
- **`interface_level_pressure`** (*`array`*) – Pressure values on interface levels. Must be in in the same units as `mid_level_pressure`.

**Returns** values of the quantity on mid-levels.

**Return type** interface_values (array)

# CHAPTER 12

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 12.1 Types of Contributions

### 12.1.1 Usage in Publications

If you use CliMT to perform research, your publication is a valuable resource for others looking to learn the ways they can leverage CliMT's capabilities. If you have used CliMT in a publication, please let us know so we can add it to the list.

### 12.1.2 Presenting CliMT to Others

CliMT is meant to be an accessible, community-driven model. You can help the community of users grow and be more effective in many ways, such as:

- Running a workshop
- Offering to be a resource for others to ask questions
- Presenting research that uses CliMT

If you or someone you know is contributing to the CliMT community by presenting it or assisting others with the model, please let us know so we can add that person to the contributors list.

### 12.1.3 Report Bugs

Report bugs at https://github.com/CliMT/climt/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 12.1.4 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 12.1.5 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 12.1.6 Write Documentation

CliMT could always use more documentation. You could:

- Clean up or add to the official CliMT docs and docstrings.
- Write useful and clear examples that are missing from the examples folder.
- Create a Jupyter notebook that uses CliMT and share it with others.
- Prepare reproducible model scripts to distribute with a paper using CliMT.
- Anything else that communicates useful information about CliMT.

### 12.1.7 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/CliMT/climt/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 12.2 Get Started!

Ready to contribute? Here's how to set up *climt* for local development.

1. Fork the *climt* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/climt.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv climt
$ cd climt/
$ python setup.py develop
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 climt tests
   $ python setup.py test or py.test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

## 12.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4 and 3.5. Check https://travis-ci.org/CliMT/climt/pull_requests and make sure that the tests pass for all supported Python versions.

## 12.4 Style

In the CliMT code, we follow PEP 8 style guidelines (tested by flake8). You can test style by running "tox -e flake8" from the root directory of the repository. There are some exceptions to PEP 8:

- All lines should be shorter than 80 characters. However, lines longer than this are permissible if this increases readability (particularly for lines representing complicated equations).
- Space should be assigned around arithmetic operators in a way that maximizes readability. For some cases, this may mean not including whitespace around certain operations to make the separation of terms clearer, e.g. "Cp*T + g*z + Lv*q".
- While state dictionary keys are full and verbose, within components they may be assigned to shorter names if it makes the code clearer.
- We can take advantage of known scientific abbreviations for quantities within components (e.g. "T" for "air_temperature") even thought they do not follow pothole_case.

## 12.5 Tips

To run a subset of tests:

```
$ py.test tests.test_timestepping
```

# CHAPTER 13

# Credits

## 13.1 Original Creator

- Rodrigo Caballero <rodrigo.caballero@misu.su.se>

## 13.2 Development Lead

- Joy Monteiro <joy.monteiro@misu.su.se>

## 13.3 Contributors

- Jeremy McGibbon <mcgibbon@uw.edu>
- Sergey Kosukhin <sergey.kosukhin@mpimet.mpg.de>
- Sally Dacie <sally.dacie@mpimet.mpg.de>
- Raymond Pierrehumbert <raymond.pierrehumbert@physics.ox.ac.uk>
- Dargan Frierson <dargan@atmos.washington.edu>
- Jonathan Mitchell <jonmitch@ucla.edu>
- Suhas D L <suhasd@iisc.ac.in>
- Abel Shibu <abel.shibu@students.iiserpune.ac.in>
- Monali Vadje <monali.sv4@gmail.com>

# CHAPTER 14

# History

## 14.1 Latest

- Removed dycore to move it to independent package

## 14.2 v.0.16.15

- Move to Github Actions tentatively finished!

## 14.3 v.0.16.11

- New component BucketHydrology that implements Manabe first generation land model
- BucketHydrology calculates the sensible and latent heat flux within the component
- Conservation test for the component also added
- Moving CI to Github Actions

## 14.4 v.0.16.8

- Fix timeout for all MAC builds

## 14.5 v.0.16.6

- Prevent MAC OS builds from timing out

## 14.6 v.0.16.5

- Fix formatting errors which prevent pypi deployment

## 14.7 v.0.16.4

- Fix MCICA for the shortwave component of RRTMG
- Revise random number generation for MCICA
- Improvement of the user interface to control MCICA

## 14.8 v.0.16.3

- update numpy requirement to avoid binary incompatibility error
- Fix error in documentation

## 14.9 v.0.16.2

- Fix wheel build on Mac

## 14.10 v.0.16.1

- Fixed issue with Mac build
- Few changes in the dry convection component. Significantly improves the performance.
- Changed logo!
- Fixed failing docs build

## 14.11 v0.16.0

- Added some documentation for using RRTMG with McICA
- CI Testing for Mac and py37 added.
- Refactored initialisation code
- Enable the McICA version of RRTMG Longwave for consistency with the Shortwave component.
- Fix bugs in IceSheet
- Add tests to verify conservation of quantities
- Fix bugs in initialisation
- Fix energy conservation in surface flux scheme
- Enable the McICA version of RRTMG Shortwave, so that partial cloud fractions can be used.

- Add GMD example scripts to repository.
- Fix docs to reflect API changes after refactor.
- Fix wrong initialisation to use sigma values instead of pressure values of optical depth for GrayLongwaveRadiation

## 14.12 Breaking Changes

- The flux outputs of GrayLongwaveRadiation have been renamed to eliminate *on_interface_levels* to keep consistency with other components.
- All arrays are now 3/2d by default based on their expected dimensions.
- horizontal dimensions are now *lon*, *lat*, but inputs used by components remain the same (*latitude*, *longitude*).

## 14.13 v.0.14.8

Many of the changes in this version come from changes in Sympl 0.4.0. We recommend reading those changes in the Sympl documentation.

- Updated component APIs to work with Sympl 0.4.0
- Many components which previously required horizontal dimensions now use wildcard matches for column dimensions.
- Switched many print statements to logging calls.
- Fixed bugs in some components

## 14.14 Breaking Changes

- get_constant and set_constant have been removed, use the ones in Sympl.
- Emanuel convection scheme can no longer be set to perform dry adiabatic adjustment to the boundary layer. This has been implemented in a separate component.
- ClimtPrognostic, ClimtImplicitPrognostic, ClimtDiagnostic, ClimtImplicit have been removed. Use the base types in Sympl.
- State initialization has been entirely re-worked. get_default_state now takes in an optional grid state instead of options to do with the state grid. A function get_grid is provided which can create a grid state, or one can be created manually. A grid state is a state containing air pressure and sigma on mid and interface levels, as well as surface pressure.
- Replaced references to “thermal_capacity” with references to “heat_capacity” in component quantity names.

## 14.15 v.0.14.7

- Fix issue with pip v10 and pandas 0.22 conflicts

## 14.16 v.0.14.3

- Fix release issue because of pip API change

## 14.17 v.0.14.1

- Fix appveyor fail due to pip changes

## 14.18 v.0.14.0

- Fixed broken version numbers

## 14.19 v.0.12.0

- new release to fix version numbers and create zenodo ID

## 14.20 v.0.9.4

- Added attributes to inputs/outputs/ etc., to work with ScalingWrapper Added tests as well.
- Added tests for constants functions
- Fixed requirements to ensure this version of climt installs the correct versions of sympl and numpy.

## 14.21 v.0.9.3

- Released because of a labelling issue. See 0.9.2 for details.

## 14.22 v.0.9.2

- Updated documentation
- Cleaned up examples
- Added (*)_properties as a property to all components
- The gas constant for dry air in the Emanuel scheme is now renamed _Rdair
- RRTMG LW and SW are now OpenMP parallel
- Added Instellation component to calculate zenith angle
- Added tests to increase coverage
- New constants handling functionality added
- Travis builds now use stages
- Appveyor CI up and running

- Pre-installation of cython and numpy no longer necessary for source builds
- Added snow-ice component
- Ozone profiles do not need to be specified externally
- Now also tested on Python 3.6

## 14.23 Breaking Changes

- API for constants setting changed to *set_constant_from_dict* and *add_constants_from_dict*
- *GfsDynamicalCore* renamed to *GFSDynamicalCore* for consistency
- *get_prognostic_version* method of *ClimtImplicit* renamed to *prognostic_version*, and no longer accepts timestep as an argument. The current timestep should be set in *ClimtImplicit.current_time_step* during each iteration.
- *RRTMGShortwave* now uses sympl's solar constant by default instead of from fortran.

## 14.24 v.0.9.1

- Held-Suarez and moist GCM with grey radiation work!
- Added DCMIP initial conditions, test 4 tried out.
- Dynamical core integrated now.
- BIG change in the build system. Tests pass on Mac as well
- Arrays can now have arbitrary dtype (to use qualitative, string, quantities)
- Added Emanuel Convection, surface energy balance model and ice sheet energy balance
- 2D coordinates are now supported for horizontal coordinates
- Replaced create_output_arrays() with a more general get_state_dict_for() and get_numpy_arrays_from_state() combination.
- State arrays now have coordinates
- Updated documentation
- RTD finally working, phew!
- Added RRTMG Longwave, Simple Physics
- Added helper functions to reduce boilerplate code in components

## 14.25 Breaking Changes

## 14.26 Latest

- method to obtain piecewise constant prognostic has been renamed to `piecewise_constant_version`
- Ozone profile has been modified
- Heating rate for RRTMG top-of-atmosphere is no longer manually set to zero
- Components no longer accept constants during initialisation. All constant handling is done internally.

## 14.27 v.0.9

- SlabSurface no longer uses depth_slab_surface as input
- changed order of outputs of GfsDynamicalCore and SimplePhysics to conform to TimeStepper order of diagnostics, new_state
- get_default_state now accepts mid_levels and interface_levels instead of z to specify vertical coordinates.
- mass_to_volume_mixing_ratio now uses numpy arrays instead of DataArrays.

# CHAPTER 15

# Indices and tables

- genindex
- modindex
- search

# Index