doc Documentation

benni

Aug 23, 2019

Contents

This project is a python-implementation of low-dimensional energy balance models (EBMs), built up from a set of physical functions, combined to represent the behaviour of earth's climate through space and time. The implementations of this project are mainly based on a set of publications, especially from *Michail Budyko* and *William Sellers*. (For the full list see *References*)

The project was started as part of my bachelor thesis, **Benjamin Schmiedel (2019)**, at the **Institute of Environmental Physics, University of Heidelberg**, under the supervision of **Dr. Kira Rehfeld**, group leader of STACY (State and timescale-dependency of climate variability from the last Glacial to present day).

Continuing work was carried out by *me* as scientific assistant, supervised by Dr. Kira Rehfeld, and as intern at the **Geophysical Institute, University of Bergen**, supervised by **Ingo Bethke**. Funding through the **Emmy Noether programme** of the German Research foundation and the european **Erasmus+ Program** is gratefully acknowledged.

CHAPTER 1

Welcome to the lowEBMs documentation!

This documentation primarily aims on explaining the structure of the sourcecode and show how to use it for your own purpose. It is now the most well documented sourcecode, so if questions or concerns about implementations come up, please *contact* me.

1.1 Physical Background / Model Types

With this project different types of EBMs can be used to run simulations. However, the versatility of the resolution is limited to low dimensional EBMs from zero dimensionsal EBMs to one one dimensional EBMs resolved over the latitudes. It would be nice and is planned by me to extend this project to higher resolutions (for more information see *ToDo*)

1.1.1 Physical Background

In general, energy balance models describe the behaviour of a planet's energy balance over time. Here, the focus is obviously on the earth's energy balance, but EBMs are generally not restricted to describe the earth's energy balance.

Here shown is a 0D schematic of the earth's energy balance like it is often given in the standard literature. The radiative energy fluxes (in Wm^{-2}) of the earth are indicated with their strength and direction. However, EBMs describe the energy balance mostly with the crucial parts only, which means that small or strongly regional energy fluxes are neglected.

EBMs are commonly restricted to the **downward radiative energy flux** (R_{down}), the **upward radiative energy flux** (R_{up}), in the case of the treated 1D-EBMs to the **latitudinal transfer energy fluxes** ($F_{transfer}$) and in some cases of to additional **forcing energy flux** (F_{forced}) (e.g. Carbon Dioxide forcing). This is of course no necessity rather than a general identification of EBMs since they are specifically characterized by their simplicity.

The physical basis of EBMs can be expressed in a model equation which commonly has the following form:

$$C \cdot \frac{dT}{dt} = R_{down} + R_{up} + F$$

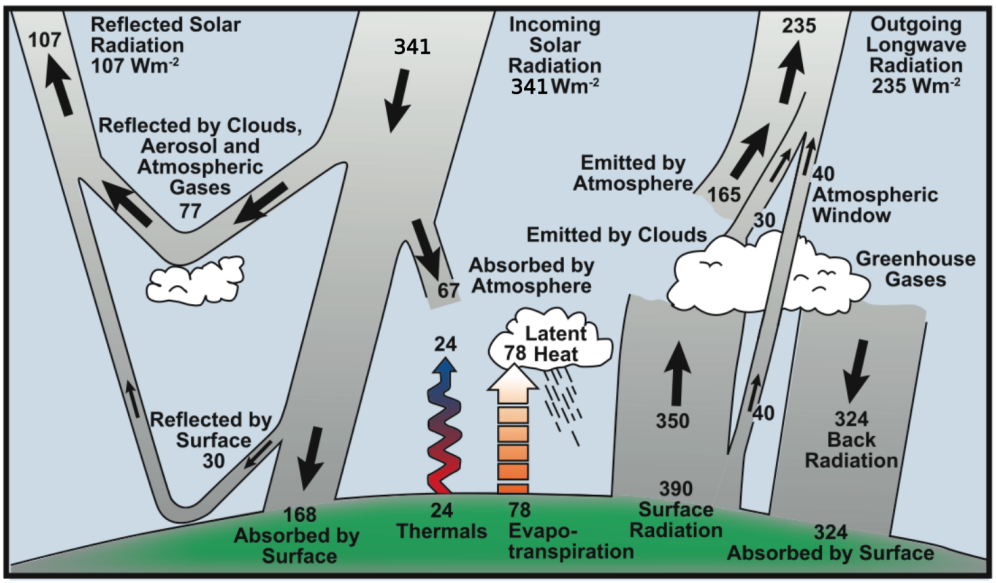


Fig. 1: Earth's energy balance [*IPCC, 2013*]

where C is the heatcapacity, $\frac{dT}{dt}$ the temperature tendency and R_{down}, R_{up}, F the energy fluxes which are included.

1.1.2 Model Types

0D-EBM

If one is interested in the 0D case, the model equation above suffices to describe the energy balance over time. However, F is neglected in general and only used in specific cases. By using the following discretizations:

$$R_{down} = (1 - \alpha) \cdot Q$$
$$R_{up} = -\epsilon\sigma T^4$$

with the albedo α, the solar insolation Q, the Stefan-Boltzmann constant σ, and the emissivity ϵ, the simplest form of an EBM is described by:

$$C \cdot \frac{dT}{dt} = R_{down} + R_{up} = (1 - \alpha) \cdot Q - \epsilon\sigma T^4$$

This equation can easily be solved analytically, but to observe the behaviour of the energy balance over time a numerical algorithm can be used to solve this equation. With the chapter *How to use* it will be investigated in detail how this project implements such an EBM. Additionally there is a demonstration file given once you have *installed* this project.

Note: The dependencies of parameters like α on variables like the temperature T are strongly related to the inbound type of *Functions* and is therefore not specified while formulating this model equations.

1D-EBM

The description of 1D EBMs does not differ much from 0D EBMs. In 1D EBMs the earth is commonly described by a grid of latitudinal bands. The model equation as introduced *above* can directly be transfered to be valid for each latitudinal band seperately.

As already mentioned, 1D EBMs use latitudinal transfer energy fluxes $F_{transfer}$ which consider an exchange of energy between latitudinal bands. This term is crucial, because the energy balance resolved over the latitudes shows strong differences between equator and poles due to the stronger insolation at the equator.

By identifying each latitudinal band and all its parameters with an index i, the simplest form of an 1D-EBM is described by:

$$C \cdot \frac{dT_i}{dt} = R_{down,i} + R_{up,i} + F_{transfer,i}$$

There are many different approaches to discretize these terms in 1D. Because this project was started to implement two specific EBMs, one developed by *Michail Budyko* and one by *William Seller*, both published in the late 1960s, these two discretizations will be shown.

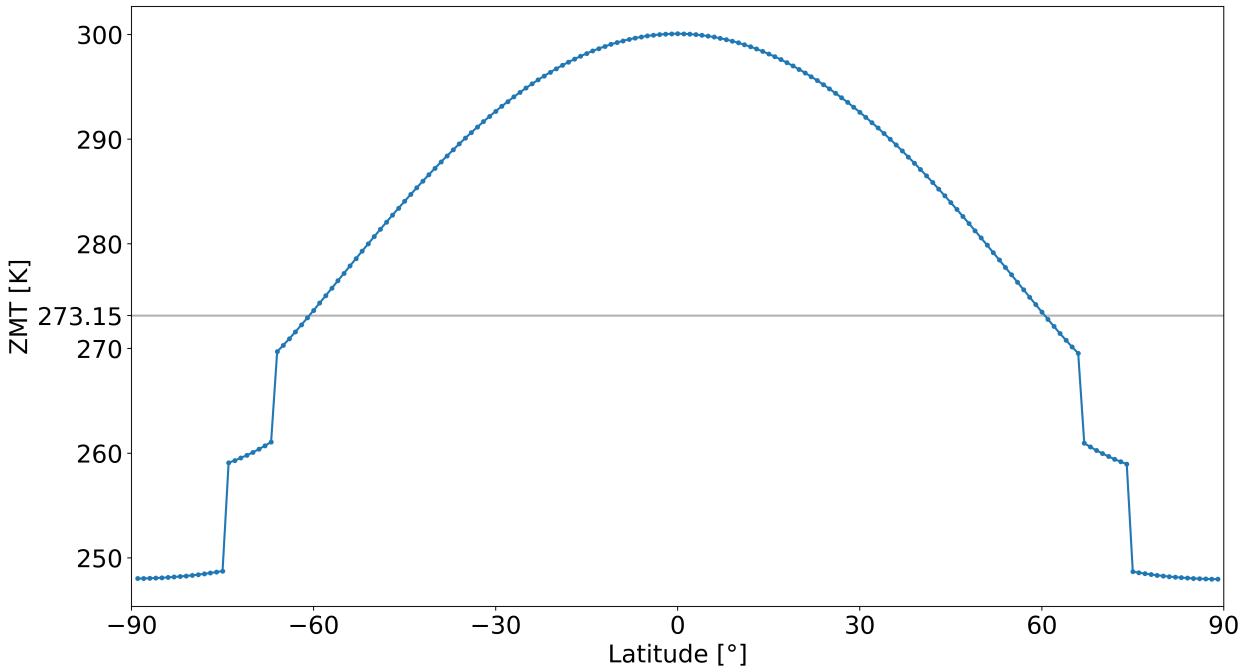
Budyko-type model

This EBM constructed by *Michail Budyko* uses various assumptions, supported by global earth observation data. The key features of this model are:

- An empirically determined upward radiation flux with linear dependence on temperature, in its simplest form described by $R_{up} = -(A + B \cdot T)$.
- An albedo seperated into three different regions with dependence on latitude (or by customization on temperature), with high albedo values towards the polar regions and low albedo values in the equatorial regions.
- A symmetric diffusive transfer energy flux with dependence on the difference of zonal (ZMT) to global (GMT) mean temperature.
- A grid resolving latitudinal bands of any width (in this project mostly used is a width of 1°)

The detailed physical formulation of the terms (and additional extensions) can be viewed along with the implementations (*Functions*).

An example zonal mean temperature distribution:



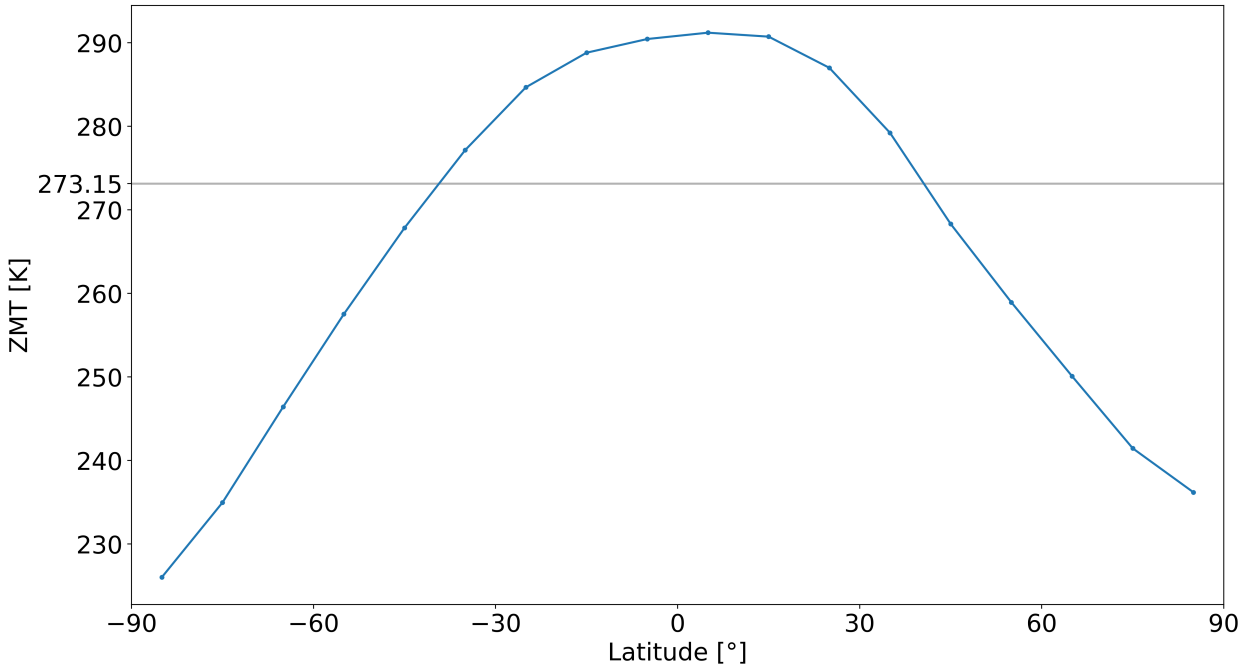
Sellers-type model

The EBM constructed by *William Seller* is adapted even more to global earth observation data than the Budyko-type model and thereby is constructed with more complex terms. The key features of this model are:

- The Stefan-Boltzmann radiation law as upward radiation flux extended with a term considering atmospheric attenuation.
- An albedo described by an empircal law with linear dependence on temperature and surface elevation.
- A transfer energy flux P seperated into three different components, the atmospheric water vapour transfer, the atmospheric sensible heat transfer and the oceanic sensible heat transfer. The total transfer energy flux F of one gridbox is given by the difference of northward and southward transfer energy flux P (the sum of those three components from the northern/southern boundary).
- A grid resolving the earth with latitudinal bands of 10° width. Hence, the earth is resolved by 18 latitudinal bands.

The detailed physical formulation of the terms (and additional extensions) can be viewed along with the implementations (*Functions*).

An example zonal mean temperature distribution:



1.2 Installation

1.2.1 Dependecies

To properly use this package there are several other packages required:

- Python (2.7 should work but I recommend higher versions, 3.5, 3.6 . . .)
- `numpy` (for mathematical calculations)
- `matplotlib` (for plotting)

- `netCDF4` (for comparison to observational data stored as netCDF-files)
- `tqdm` (for progress visualization)

1.2.2 Python Package

The simplest way to install `lowEBMs` is from source. To download and install `lowEBMs` with all its dependencies, go to the command line and type:

```
pip install lowEBMs
```

or (if you have python 2 and python 3 installed and want to install it on python 3):

```
pip install lowEBMs
```

Alternatively, you can clone the git repository of the source code and manually run the setup.py which installs the package with all its dependencies:

```
git clone https://github.com/BenniSchmiedel/Low-dimensional-EBMs.git

python setup.py install
```

(keep in mind to change to the directory where you cloned the repository to).

1.2.3 Export Tutorial Files

`lowEBMs` comes with a *list of tutorial files* supplemented in a subfolder of the package. When the package is installed via pip, it is automatically inbound in your specific python environment. To easily extract those *jupyter notebooks* and *configuration.ini* files to your preferred directory, do the following:

Open the terminal, change your directory to the one where you want the files and use:

```
python -c "from lowEBMs import Tutorial_copy; Tutorial_copy()"
```

Note: You can specify the output directory as argument with `Tutorial_copy(path='/outputdir')`

1.2.4 Export Forcing Files

There are also PMIP3 forcing datasets included which can be exported the same way as the tutorial files.

Open the terminal, change your directory to the one where you want the files and use:

```
python -c "from lowEBMs import Forcing_copy; Forcing_copy()"
```

Note: You can specify the output directory as argument with `Forcing_copy(path='/outputdir')`

1.3 How to use

Here described is how you use a given input, which initializes an EBM, to run a simulation with it.

We will write a small python script, which will do this in a few steps. As it is easier to visualize the output in a plot and modify it, I recommend to perform this steps in a jupyter notebook.

Skip detailed description

1.3.1 Step 0: Import packages

Before you can use any module of this package you have to import the core modules:

```
import matplotlib.pyplot as plt
import numpy as np
from lowEBMs.Packages.Configuration import importer
from lowEBMs.Packages.Variables import variable_importer
from lowEBMs.Packages.RK4 import rk4alg
from lowEBMs.Packages.ModelEquation import model_equation
```

1.3.2 First Step: Import model configuration

The way this project is built up enables to take any physical function implemented and merge them to formulate the desired EBM. The configuration has to be given manually and is stored in a **configuration.ini** file. Details on how to create and structure **.ini** files is given in *input*.

Important: The configuration.ini file will provide the physical sense of the EBM!

For now you can simply use the **EBM0D_simple_config.ini** file which imports a 0D EBM with a model run over 10 year and a stepsize of integration of 1 day. A demonstration on how to reproduce this **.ini** file is given in *Example Input 0D-EBM*.

To import the information from this file into the program use `importer()`:

```
configuration=importer('EBM0D_simple_config.ini')
```

Note: In case you work in another directory than the installation directory of the project or get the error 'File not found', add the additional argument path: `importer('filename',path='path/to/your/file')`. The path can be a relative or full path to where your **configuration.ini** is located.

`configuration` is an dictionary which contains all required input parameters. To seperate them for a clearer structure you can use:

```
eq=configuration['eqparam']
rk=configuration['rk4input']
fun=configuration['funccomp']
ini=configuration['initials']
```

Those are four dictionaries which contain the information needed for the base equation, the runge-kutta algorithm, the functions used and the initial conditions.

1.3.3 Second Step: Import variables

As next step the information from the configuration has to be imported into the programs variablespace. To do so use `variable_importer()`:

```
variable_importer(configuration)
```

1.3.4 Third Step: Start the simulation

Now we are ready to run the algorithm with `rk4alg()`. It requires the `model_equation` and the dictionaries we seperated before (maintain the order):

```
outputdata=rk4alg(model_equation,eq,rk,fun)
```

Depending on your settings the algorithm will need some time until it prints *Finished!*.

1.3.5 Final Step: Evaluating the output

The function `rk4alg` return three arrays, the **Time, zonal mean temperature (ZMT) and global mean temperature (GMT)**. Other variables of interest, for example the grid specifications, can be accessed by importing the *variables* variablespace and additional constants by importing the constants class:

```
from lowEBMs.Packages.Variables import Vars
    import lowEBMs.Packages.Constants as const
```

and then return the desired variables by their specified name, for example:

```
latitudinal_grid=Vars.Lat
```

For detailed information about output variables see section *output*.

You can plot the global temperature over time with (with time conversion):

```
plt.plot(Time/const.time_sec_year,GMT)
plt.xlabel('time [years]')
plt.ylabel('GMT [K]')
```

and you get something like this (for the simple 0D EBM):

1.3.6 Putting it together

The summary of what you need to get the model running. Import packages:

```
import matplotlib.pyplot as plt
import numpy as np
from lowEBMs.Packages.Configuration import importer
from lowEBMs.Packages.Variables import variable_importer
from lowEBMs.Packages.RK4 import rk4alg
from lowEBMs.Packages.ModelEquation import model_equation
```

and run the specific modules:

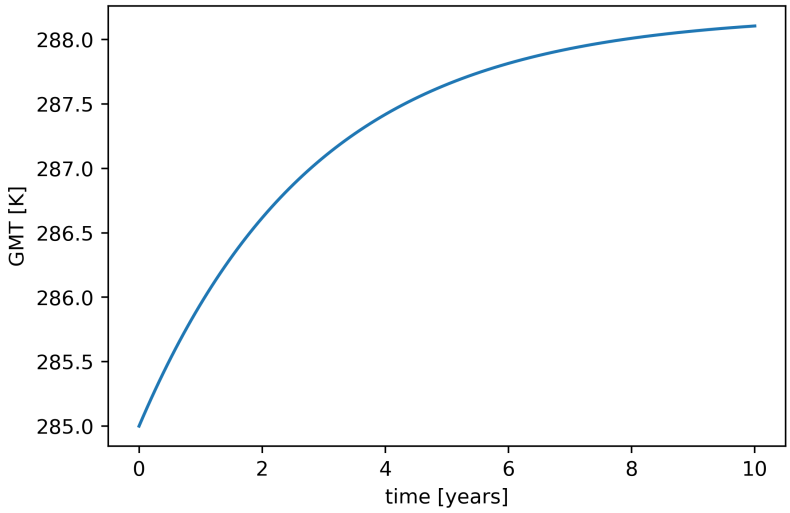


Fig. 2: with an initial temperature of 12°C (285K)

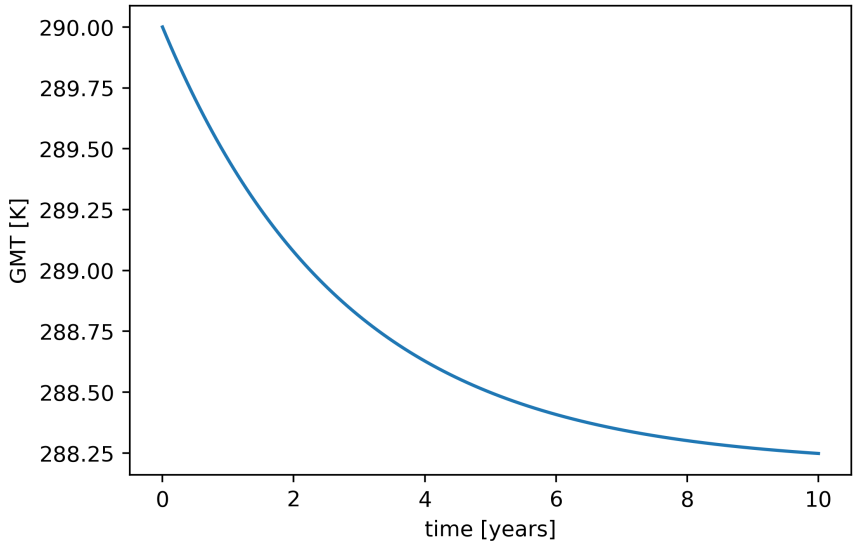


Fig. 3: with an initial temperature of 17°C (290K)

```
configuration=importer('EBM0D_simple_config.ini')
eq=configuration['eqparam']
rk=configuration['rk4input']
fun=configuration['funccomp']
variable_importer(configuration)
Time,ZMT,GMT=rk4alg(model_equation,eq,rk,fun)
```

This demonstration also exists as a jupyter notebook in the *'Tutorials/'* directive of this project (*EBM0D_simple.ipynb*).

1.4 Input

All the input required to run an EBM with this source code is provided by a **configuration.ini** file which you have to create. As already mentioned in the section *How to use*:

Important: The configuration.ini file will provide the model setup and physical sense of the EBM!

Here shown is, how this file is structured and which syntax has to be maintained to make it readable to the `importer` function.

There are four main components of the file, the modelequation parameters `[eqparam]`, the runge-kutta parameters `[rk4input]`, the initial condition parameters `[initials]` and physical functions with their specific parameters enumerated with `[func0]`, `[func1]` and so on.

Important: To define which function you add in `[func]`, insert the name of the function as parameter `func=name`, then add the required parameters below (for available options see *Configuration options* and for their physical background see *Functions*).

If you want to put together a new model simply create a textfile with the suffix **.ini** and the following style:

```
#Initial model setup & algorithm parameters
#-------------------------------------------
[eqparam]

[rk4input]

[initials]

#Model structure & physical equation parameters
#----------------------------------------------
[func1]

[func2]

.
.
.
```

Note: The order of your sections doesn't matter as long as the headers are correctly labeled.

Now each section has to be filled with parameters. `[eqparam]`, `[rk4input]` and `[initials]` always contain the same parameters since they define **how** the algorithm runs. The func-sections have to be modified since they define

which model equation the algorithm solves.

1.4.1 Configuration options

For a detailed definition of the options available for the model setup see here:

Configuration-Section Options

[eqparam], [rk4input] and [initials]

The options of these sections are always the same and are always required! After the `Configuration.importer` function processes the **.ini**-file, these options are distributed to the functions of the model algorithm. For more information about the parameters, for `[eqparam]` see *ModelEquation*, for `[rk4input]` see *variable_importer* and for `[initials]` see *variable_importer*:

```
[eqparam]
C_ao=70*4.2e6

[rk4input]
number_of_integration=365*10
stepsize_of_integration=60*60*24
spatial_resolution=0
both_hemispheres=True
latitudinal_circle=True
latitudinal_belt=False

eq_condition=False
eq_condition_length=100
eq_condition_amplitude=1e-3

data_readout=1
number_of_externals=0

[initials]
time=0
zmt=273+15
gmt=273+15
initial_temperature_cosine=False
initial_temperature_amplitude=30
initial_temperature_noise=True
initial_temperature_noise_amplitude=5
```

[func]

The options for a `[func]` section are all physical functions defined in *Functions* which describe an Energy flux. There are four classes which contain such functions, `Functions.flux_down`, `Functions.flux_up`, `Functions.transfer` and `Functions.forcing`.

The functions and examples of their required parameters are listed here after their class.

Important: There may only be one option each of `Functions.flux_down`, `Functions.flux_up` and `Functions.transfer` be used while multiple functions of `Functions.forcing` might be used.

---

flux_down Options

There is only one option, *Insolation*:

```
[func0]
func=flux_down.insolation
q=1
m=1
dq=0

albedo=albedo.dynamic_bud
albedoread=True
albedoparam=[273.15-5,273.15-15,0.32,0.5,0.75]

noise=False
noiseamp=342*0.03
noisedelay=1
seed=True
seedmanipulation=0

solarinput=True
convfactor=1
timeunit='annualmean'
orbital=False
orbitalyear=0
```

flux_up Options

Option 1, *Budyko clear sky*:

```
[func1]
func=flux_up.budyko_noclouds
activation=True
a=210
b=2.1
```

Option 2, *Budyko cloudy sky*:

```
[func1]
func=flux_up.budyko_clouds
activation=True
a=230.31
b=2.2274
a1=3.0*15.91
b1=0.1*15.91
fc=0.5
```

Option 3, *Stefan-Boltzmann radiation*:

```
[func1]
func=flux_up.planck
activation=True
grey=0.612
sigma=const.sigma
```

Option 4, *Sellers*:

```
[func1]
func=flux_up.sellers
activation=True
m=0.5
sig=const.sigma
gamma=1.9*10**(-15)
k=1
```

transfer Options

Option 1, *Budyko transfer*:

```
[func2]
func=transfer.budyko
beta=3.18
read=True
activated=True
```

Option 2, *Sellers transfer*:

```
[func2]
func=transfer.sellers
readout=True
activated=True
k_wv=10**5
k_h=10**6
k_o=10**2
g=9.81
a=2/100
eps=0.622
p=1000
e0=1700
L=const.Lhvap/1000
rd=const.Rd/1000
dy=1.11*10**6
dp=800
cp=const.cp
dz=2000
l_cover=0.5
radius=const.a
cp_w=4182
dens_w=998
factor_wv=1
factor_air=1
factor_oc=1
factor_kwv=1
factor_kair=1
```

forcing Options

Important: If you use multiple `forcing.random` and `forcing.predefiend` you have to increase the value of the option **forcingnumber** by 1, this will create an additional space in the output-array and an internal counter of

the forcings.

Option 1, *Random forcing*:

```
[func3]
func=forcing.random
forcingnumber=0
start=1958
stop=2018
steps=1/365
timeunit='year'
strength=10
frequency='rare'
behaviour='exponential'
lifetime=365
seed=None
sign='negative'
```

Option 2, *Imported predefined forcing*:

```
[func3]
func=forcing.predefined
forcingnumber=0
datapath="../Config/Data/"
name="EVA_800_2000.txt"
delimiter=","
header=1
footer=0
col_time=0
col_forcing=1
timeunit='year'
bp=False
time_start=0
k_output=1
m_output=0
k_input=1
m_input=0
```

Option 3, Imported 1D predefined forcing:

```
[func3]
func=forcing.predefined1
forcingnumber=0
datapath="../Config/Data/"
name="Forcingdata1D.csv"
delimiter=","
header=1
footer=0
col_time=0
colrange_forcing=[1,19]
timeunit='year'
bp=False
time_start=0
k_output=1
m_output=0
k_input=1
m_input=0
```

Option 4, *Imported CO2 forcing after Myhre*:

```
[func3]
func=forcing.co2_myhre
A=5.35
C_0=280
CO2_base=280
datapath="../Config/Data/"
name="CO2data.csv"
delimiter=","
header=0
footer=0
col_time=3
col_forcing=8
timeunit='year'
bp=False
time_start=0
```

Option 5, Imported orbital parameter data:

```
[func3]
func=forcing.orbital
datapath="../Config/Data/"
name="Orbitaldata.csv"
delimiter=","
header=0
footer=0
col_time=0
col_ecc=1
col_per=2
col_obl=3
timeunit='year'
bp=False
time_start=0
initial={'ecc': 0.017236, 'long_peri': 281.37, 'obliquity': 23.446}
perishift=180
```

Option 6, Imported total solar irradiance:

```
[func3]
func=forcing.solar
datapath="../Config/Data/"
name="tsi_SBF_11yr.txt"
delimiter=""
header=4
footer=0
col_time=0
col_forcing=1
timeunit='year'
bp=False
time_start=0
k_output=1
m_output=0
k_input=1
m_input=0
```

Option 7, Imported AOD forcing:

```
[func3]
func=forcing.aod
datapath="../Config/Data/"
name="AODdata.csv"
delimiter=","
header=0
footer=0
col_time=0
col_forcing=1
timeunit='year'
bp=False
time_start=0
k_output=1
m_output=0
k_input=1
m_input=0
```

1.4.2 Example Input 0D EBM

For the 0D-EBM (the *EBM0D_simple_config.ini*), the model setup might look like this:

```
#Initial model setup & algorithm parameters
#-----------------------------------------
[eqparam]
c_ao=70*4.2e6

[rk4input]
number_of_integration=365*10
stepsize_of_integration=60*60*24
spatial_resolution=0
both_hemispheres=True
latitudinal_circle=True
latitudinal_belt=False

eq_condition=False
eq_condition_length=100
eq_condition_amplitude=1e-3

data_readout=1
number_of_externals=0

[initials]
time=0
zmt=273+15
gmt=273+15
initial_temperature_cosine=False
initial_temperature_amplitude=30
initial_temperature_noise=True
initial_temperature_noise_amplitude=5
```

If you now want to give a model structure with a downward radiative energy flux and a upward radiative energy flux, this might look like this:

```
#Model structure & physical equation parameters
#---------------------------------------------
[func0]
```

(continues on next page)

(continued from previous page)

```
func=flux_down.insolation
q=342
m=1
dq=0

albedo=albedo.static
albedoread=True
albedoparam=[0.3]

noise=False
noiseamp=342*0.03
noisedelay=1
seed=True
seedmanipulation=0

sinusodial=False
convfactor=1
timeunit='annualmean'
orbital=False
orbitalyear=0

[func1]
func=flux_up.planck
activation=True
grey=0.612
sigma=const.sigma
```

1.5 Output

This chapter describes which variables can be printed out and how.

Note: Since longer model runs can be heavily memory-consuming the general frequency of data-readout can be adjusted in the *configuration.ini*. The parameter `data_readout` indicates on which step the data is read. 1 for every, 2 for every second ...

There are two types of data to print.

The first type are the **primary variables**: **time** and **temperature**. They are returned directly by the algorithm (for details see *How to use* or *RK4*).

The second type are secondary variables, such as the albedo or the insolation, which might be of interest to observe. They are stored by the *Variables* package within the class `Vars`. Most of them are written into the dictionary `Vars.Read` and callable with:

```
from Variables import Vars
Vars.Read
```

and contains the following variables, callable with:

```
cL,C,F,P,Transfer,alpha,BudTransfer,solar,noise,Rdown,Rup,ExternalOutput,CO2Output,
↪SolarOutput,AODOutput

Insolation_over_time=Vars.Read['solar']
```

Beneath the variables in `Vars.Read`, there are additional variables which can be printed, for example the following static variables:

```
Area=list
bounds=list
latlength=list
External_time_start=float
CO2_time_start=float
```

All of them are callable by:

```
from Variables import Vars
Vars.VARIABLENAME
```

The description of all output variables is given in chapter *Variables*.

1.6 Tutorials

Here given is a list of tutorial-/demonstration-EBMs which are supplemented within the installation directory under *~/lowEBMs/Tutorials/* or can otherwise be accessed from the git repository under https://github.com/BenniSchmiedel/Climate-Modelling/tree/master/lowEBMs/Tutorials.

EBM tutorial files:

- *Tutorials*
  - *0D EBM (simple)*
  - *0D EBM (CO_2 forced)*
  - *0D EBM (volcanic forced)*
  - *1D EBM Budyko-type (static albedo)*
  - *1D EBM Budyko-type (temperature-dependant albedo)*
  - *1D EBM Sellers-type (temperature-dependant albedo)*
  - *1D EBM Sellers-type, Volcanic Forcing*

For the physical background see *Model types* and the explanation of usage see *How to use*. The explanation of additional functions and ther usage can be looked up in the *functions' definitions* or in the referenced literature.

1.6.1 0D EBM (simple)

A 0D EBM equipped with:

- constant absorbed downward solar radiation flux
- upward radiation flux according to the Stefan-Boltzmann law

1.6.2 0D EBM (CO_2 forced)

A 0D EBM equipped with:

- constant absorbed downward solar radiation flux

- upward radiation flux according to the Stefan-Boltzmann law
- CO_2 radiative forcing according to estimates by *Myhre*

The tutorial-file of this EBM uses a CO_2-forcing based on 1958 - present atmospheric CO_2-concentrations (the *Keeling-curve*)

1.6.3 0D EBM (volcanic forced)

A 0D EBM equipped with:

- constant absorbed downward solar radiation flux
- upward radiation flux according to the Stefan-Boltzmann law
- volcanic radiative forcing given by the difference in Wm^{-2}

The tutorial-file of this EBM uses a randomly generated radiative forcing as volcanic-forcing. To truly consider volcanic radiative forcing the gas concentrations have to be converted into the amount of radiative forcing which is not implemented for now.

1.6.4 1D EBM Budyko-type (static albedo)

A 1D EBM equipped with:

- static albedo distribution with three regions of albedo regions → constant absorbed downward solar radiation flux
- upward radiation flux according to Budyko's radiation law
- a symmetric diffusive transfer energy flux according to Budyko

The parameters of the tutorial-file are chosen to reproduce the EBM as it was introduced by *Budyko (1968)*.

1.6.5 1D EBM Budyko-type (temperature-dependant albedo)

A 1D EBM equipped with:

- temperature dependant albedo distribution with three regions of albedo regions → dynamic absorbed downward solar radiation flux
- upward radiation flux given by the empirical law according to *Budyko*
- a symmetric diffusive transfer energy flux according to *Budyko*

The parameters of the tutorial-file are chosen to reproduce the EBM as it was introduced by *Budyko (1968)*. The temperature dependence of the albedo is defined in *Functions*.

1.6.6 1D EBM Sellers-type (temperature-dependant albedo)

A 1D EBM equipped with:

- dynamic albedo distribution with a continuous temperature dependant albedo function → dynamic absorbed downward solar radiation flux
- upward radiation flux given by an edited Stefan-Boltzmann radiation law according to *Sellers (1969)*
- transfer energy flux according to *Sellers*

- temperature distributions corrected by the average latitudinal band elevation

The parameters of the tutorial-file are chosen to reproduce the EBM as it was introduced by *Sellers (1969)*.

1.6.7 1D EBM Sellers-type, Volcanic Forcing

A 1D EBM equipped with:

- dynamic albedo distribution with a continuous temperature dependant albedo function → dynamic absorbed downward solar radiation flux
- upward radiation flux given by an edited Stefan-Boltzmann radiation law according to *Sellers (1969)*
- transfer energy flux according to *Sellers*
- temperature distributions corrected by the average latitudinal band elevation
- volcanic radiative forcing, imported from an external dataset

The parameters of the tutorial-file are chosen to reproduce the EBM as it was introduced by *Sellers (1969)*. The volcanic radiative forcing was created with the **ForcingGenerator**-module, an adopted simplified version of the EVA-Generator from *Toohey (2016)*.

1.7 Source Code

The project is seperated onto 5 python-packages with several submodules.

The coremodule is the numerical integrator, the Runge-Kutta 4th order scheme defined in `lowEBMs.Packages.RK`.

The structure of the model is provided by `lowEBMs.Packages.ModelEquation`. It builds up the EBM from a set of physical functions, specified in `lowEBMs.Packages.Function`.

In order to get a reasonable EBM structure you have to give a configurationfile (for details on how to create it, see *Input*) which is processed by `lowEBMs.Packages.Configuration`. Along with the basic configuration of the model many required variables are defined in `lowEBMs.Packages.Variables`, which may be running variables but also variables provided for later output (for details on what to print out, see *Output*).

1.7.1 lowEBMs.Packages

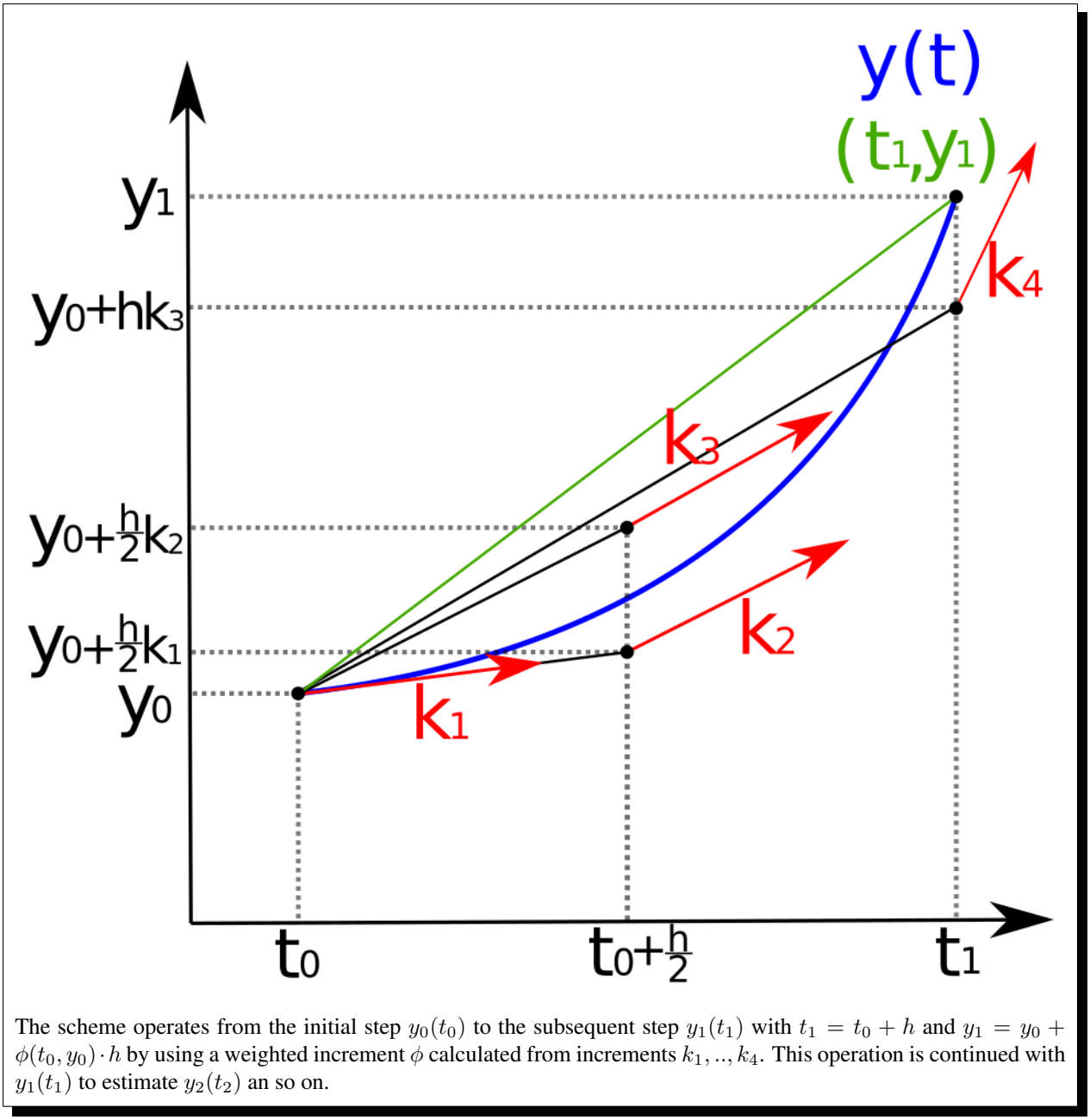
lowEBMs.Packages.RK4

Runge-Kutta 4th order scheme

The `lowEBMs.Packages.RK4` provides the numerical scheme to iteratively solve differential equations, hence the *model equation* which is parsed by `lowEBMs.Packages.ModelEquation`, initialized with the *configuration* provided by `lowEBMs.Packages.Configuration`.

For an example see *How to use*.

Operating principle RK4



The scheme operates from the initial step $y_0(t_0)$ to the subsequent step $y_1(t_1)$ with $t_1 = t_0 + h$ and $y_1 = y_0 + \phi(t_0, y_0) \cdot h$ by using a weighted increment ϕ calculated from increments $k_1, .., k_4$. This operation is continued with $y_1(t_1)$ to estimate $y_2(t_2)$ an so on.

The increments $k_1, .., k_4$ are obtained by solving the model equation, as defined in the physical background for the dynamical term $\frac{dT}{dt}$. The increments differ in their choice of inital conditions (point of evaluation of the model equation). One iterative step always goes through a cycle of evaluating the model equation four times. It starts with the calculation of k_1 at point $y_0(t_0)$ with:

$$k_1 = f(t_0, y_0),$$

where $f(t, y(t))$ is given by the deviation of $y(t)$, hence $\frac{dT}{dt} = \frac{1}{C} \cdot (R_{in} + R_{out} + ...)$ at $T_0(t_0)$.

Now the scheme continues the following procedure:

$$k_2 = f(t_0 + \frac{h}{2}, y_0 + \frac{h}{2} \cdot k1)$$
$$k_3 = f(t_0 + \frac{h}{2}, y_0 + \frac{h}{2} \cdot k2)$$
$$k_4 = f(t_0 + h, y_0 + h \cdot k3).$$

As final step of one iterative step the weighted increment ϕ is calculated by through:

$$\phi = \frac{1}{6} \cdot k_1 + \frac{1}{3} \cdot k_2 + \frac{1}{3} \cdot k_3 + \frac{1}{6} \cdot k_4$$

to estimate y_1 as final step of one iteration step:

$$y_1 = y_0 + \phi(t_0, y_0) \cdot h.$$

`lowEBMs.Packages.RK4.`**`rk4alg`**(*func*, *eqparam*, *rk4input*, *funccomp*, *progressbar=True*, *monthly=False*)

lowEBMs.Packages.ModelEquation

In `lowEBMs.Packages.ModelEquation` modules are defined which build the EBM from functions given to it. The operation of this modules is adding the given functions $F_1, F_2, ...F_i$ according to the following scheme (compare *physical background*):

$$y = \frac{1}{C_{ao}} \cdot (F_1 + F_2 + ... + F_i),$$

with the deviation function $y = \frac{dT}{dt}$ required by the `lowEBMs.Packages.RK4.rk4alg` and C_{ao} the heat capacity of the system which is passed to the right side of the model equation.

`lowEBMs.Packages.ModelEquation.`**`model_equation`**(*eqparam*, *funccomp*)

The module which builds and evaluates the EBM by adding functions parsed through the **funccomp**.

Input has to be given as *Dictionaries* supplied by `lowEBMs.Packages.Configuration.importer` from a specific **configuration.ini**.

Function-call arguments

Parameters

- **`eqparam`** (*dict*) – Configuration dictionary containing additional information for the model equation:
  - C_ao: The systems heat capacity (times the height of the system)
    - type: float
    - unit: Joule*Meter/Kelvin
    - value: > 0
- **`funccomp`** (*dict*) – Configuration 2D dictionary containing function names and function parameters used:
  - funcnames: a dictionary of names of functions defined in `lowEBMs.Packages.Functions` which are added up. See *here* for a list of functions

- funcparams: a dictionary of functions parameters corresponding to the functions chosen within **funcnames**. For details on the parameters see the specific function *here*

Returns The temperature gradient $\frac{dT}{dt}$ (Kelvin/seconds)

Return type float or array(float), depending on 0D EBM or 1D EBM. In 1D, output is an array containing the temperature gradient for each latitudinal belt.

lowEBMs.Packages.Functions

In `lowEBMs.Package.Functions` all physical equations are defined which describe the energy balance of the earth system.

This module is structured through classes which distinguish the type of energy flux or property definition. The classes contain several functions which follow a different approach of parameterizing the respective type of energy flux/ property.

The classes which define energy fluxes are:

flux_down	Class defining radiative fluxes directed downwards.
flux_up	Class defining radiative fluxes directed upwards.
transfer	Class defining latitudinal energy transfer transfer fluxes.
forcing	Class defining radiative forcing terms

Important: These contain the physical functions available for the EBM. To correctly run them they need parameters as input which are parsed by `Configuration.importer` but **have to be given manually into the configuration.ini**. To add a function, extend your *configuration.ini* with an enumerated [**func**_] section (with _ a number) and insert all parameters below which are given in the documentation here of the specific function. For examples see *Configuration Options*.

The classes which contain definitions of earth system properties are:

albedo	Class defining the albedo distributions
earthsystem	Class defining earthsystem properties

Additionally defined are tools for evaluation or simplification in the class:

`tools`

Here the full list of modules defined in `lowEBMs.Packages.Functions`:

flux_down

class lowEBMs.Packages.Functions.**flux_down**

Bases: `object`

Class defining radiative fluxes directed downwards.

Because the models in this project don't include atmospheric layers (for now), the only radiative flux directed downwards is the radiative energy coming from the sun. This function is the same for all implemented models and is described in `flux_down.insolation` which allows several adjustments.

insolation	Function defining the absorbed solar insolation.

lowEBMs.Packages.Functions.flux_down.**insolation**(*self*, *funcparam*)

Function defining the absorbed solar insolation. Physically there is an important difference between the insolation, which is denoted as Q and the absorbed insolation, which is the output of this function denoted as R_{down}. The absorbed insolation in it's simplest form is written (as introduced in the physical background):

$$R_{down} = (1 - \alpha) \cdot Q,$$

with the albedo α which is the reflected part of the insolation Q.

The definition of R_{down} in this function has several extensions:

$$R_{down} = m \cdot (1 - \alpha) \cdot (Q + dQ) + z,$$

with an energy offset dQ on Q, a factorial change of absorbed insolation m and a random noise factor z on the absorbed insolation. z is chosen as a normal distributed random number with `numpy.random.normal`.

This function allows the observation of the models behaviour to diverse manipulations of the solar insolation.

Input has to be given as *Dictionaries* supplied by `lowEBMs.Packages.Configuration.importer` from a specific **configuration.ini**.

Function-call arguments

Parameters **funcparams** (*dict*) – a dictionary of the functions parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *Q*: The value of solar insolation (only useful for 0D EBMs)
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: > 0 (standard 342)
- *m*: Factorial change of absorbed insolation
  - type: float
  - unit: -
  - value: > 0
- *dQ*: Additive energy offset on Q
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: any
- *albedo*: The name of albedo function which is called from `lowEBMs.Packages.Functions.albedo` to return the albedo value/distribution. See class albedo.
  - type: string
  - unit: -
  - value: albedo.static, albedo.static_bud, albedo.dynamic_bud, albedo.smooth, albedo.dynamical_sel

- *albedoread*: Indicates whether the albedo is provided as specific output
  - type: boolean
  - unit: -
  - value: True/Flase
- *albedoparam*: Provides an array of parameters the albedo function (see class albedo)
  - type: array
  - unit: -
  - value: depending on function chosen
- *noise*: Indicates whether solar noise is activated or not
  - type: boolean
  - unit: -
  - value: True/False
- *noiseamp*: Determines the strength of the random solar noise as one standard deviation of a normal distribution (for further information see `numpy.random.normal`)
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: >0 (e.g. noise with 1 percent of 342 is the value: 0.01*342)
- *noisedelay*: Determines how often this random factor is updated to a new random factor (one factor persists until it is replaced)
  - type: int eger
  - unit: number of iteration steps
  - value: minimum 1 (every iteration cycle)
- *seed*: Indicates whether a specific seed is used to ensure that the random numbers are the one created by this specific seed (useful for comparisons to other simulation with the same solar noise)
  - type: boolean
  - unit: -
  - value: True/False
- *seedmanipulation*: Defines the value for the seed
  - type: integer
  - unit: -
  - value: any (if 0 it is everytime another seed)
- *solarinput*: Indicates whether the solar insolation distribution from `climlab.solar.insolation` are used (recommended for 1D EBMs), which are called from `lowEBMs.Packages.Functions.earthsystem().solarradiation`
  - type: boolean
  - unit: -
  - value: True/False

- *convfactor*: Determines whether a conversation factor is used to change the solar insolation to another unit than Watt/m^2
  - type: float
  - unit: depending on the conversion applied
  - value: > 0
- *timeunit*: Determines which timeunit of the solarradiation shall be used for averaging (depending on how the *builtins.stepsize_of_integration* is chosen*)
  - type: string
  - unit: -
  - value: 'annualmean' (average annually and give Q as Watt/m^2), 'year', 'month', 'day', 'second'
- *orbital*: Indicates whether the solar insolation considers manipulation through orbital parameters over time (this will replace `lowEBMs.Packages.Functions.earthsystem().solarradiation` by `lowEBMs.Packages.Functions.earthsystem().solarradiation_orbital`
  - type: boolean
  - unit: -
  - value: True/False (if False, the year given in *orbitalyear* still matters)
- *orbitalyear*: Determines for which year (in ky) the orbitalparameters are taken (orbital parameters are provided by `climlab.solar.orbital` which is based on Berger (1991) and Laskar (2004)
  - type: integer
  - unit: $kyear$
  - value: -5000 to 0 (if 0, the year 1950 is used)

Returns The absorbed solar insolation R_{down}

Return type float / array(floats) (0D / 1D)

flux_up

class lowEBMs.Packages.Functions.**flux_up**

Bases: `object`

Class defining radiative fluxes directed upwards.

The equations used here are, expect from `flux_up.planck`, are estimated empirically and the standard parameters are mostly tailored to specific applications where they are used by the authors.

budyko_noclouds	An empirically determined upward radiative energy flux which approximates the top of the atmosphere radiation emitted to space to be dependant linear on temperature.

Continued on next page

Table 5 – continued from previous page

budyko_clouds	An empirically determined upward radiative energy flux which approximates the top of the atmosphere radiation emitted to space to be dependant linear on temperature.
planck	The stefan-boltzmann radiation for a grey body as radiative energy flux directed upward.
sellers	An empirically, by *William Sellers* adjusted stefan-boltzmann radiation as radiative energy flux directed upward.

lowEBMs.Packages.Functions.flux_up.**budyko_noclouds**(*self*, *funcparam*)

An empirically determined upward radiative energy flux which approximates the top of the atmosphere radiation emitted to space to be dependant linear on temperature. The presence of clouds is not specifically taken into account.

The upward radiative energy flux R_{up} of latitude ϕ is given by:

$$R_{up}(\phi) = -(A + B \cdot T(\phi))$$

with the temperature $T(\phi)$ and empirical constants A and B. The Temperature is hereby converted to Celcius because the constants are optimized for Celcius not Kelvin.

Function-call arguments

Parameters **funcparams** (*dict*) – a dictionary of the function's parameters directly parsed from lowEBMs.Packages.ModelEquation.model_equation

- *A*: Empirical offset parameter
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: any (standard 222.74)
- *B*: Empirical gradient parameter
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Celcius^{-1}$
  - value: any (standard 2.23)

Returns The upward radiative energy flux R_{up}

Return type float / array(floats) (0D / 1D)

lowEBMs.Packages.Functions.flux_up.**budyko_clouds**(*self*, *funcparam*)

An empirically determined upward radiative energy flux which approximates the top of the atmosphere radiation emitted to space to be dependant linear on temperature. The presence of clouds is specifically taken into account with a second temperature dependant term.

The upward radiative energy flux R_{up} of latitude ϕ is given by:

$$R_{up}(\phi) = -((A + B \cdot T(\phi)) - f_c \cdot (A_1 + B_1 \cdot T(\phi)))$$

with the temperature $T(\phi)$ and empirical constants A, B, A_1 and B_1. The Temperature is hereby converted to Celcius because the constants are optimized for Celcius not Kelvin

Function-call arguments

Parameters **funcparams** (*dict*) – a dictionary of the function's parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *A*: Empirical offset parameter
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: any (standard 222.74)
- *B*: Empirical gradient parameter
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Celcius^{-1}$
  - value: any (standard 2.23)
- *A1*: Empirical offset parameter cloud term
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: any (standard 47.73)
- *B1*: Empirical gradient parameter cloud term
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Celcius^{-1}$
  - value: any (standard 1.59)
- *f_c*: Cloud fraction
  - type: float
  - unit: -
  - value: $0 \leq$ f_c ≤ 1 (standard 0.5)

Returns The upward radiative energy flux R_{up}

Return type float / array(floats) (0D / 1D)

`lowEBMs.Packages.Functions.flux_up.`**planck** (*self*, *funcparam*)

The stefan-boltzmann radiation for a grey body as radiative energy flux directed upward. The ideal stefan-boltzmann radiation with a temperature to the power of 4 scaled with an emissivity factor ϵ.

The upward radiative energy flux R_{up} of latitude ϕ is given by:

$$R_{up}(\phi) = -\epsilon \cdot \sigma \cdot T(\phi)^4$$

with the temperature $T(\phi)$, the emissivity ϵ and stefan-boltzmann constant σ.

Function-call arguments

Parameters **funcparams** (*dict*) – a dictionary of the function's parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *grey*: The emissivity (greyness)
  - type: float
  - unit: -
  - value: $0 \leq$ grey ≤ 1 (standard 0.612)

- *sigma*: Stefan-boltzmann constant
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Kelvin^{-4}$
  - value: $5,67 \cdot 10^{-8}$ (use const.sigma to load it from `climlab.constants`)

Returns The upward radiative energy flux R_{up}

Return type float / array(floats) (0D / 1D)

`lowEBMs.Packages.Functions.flux_up.`**`sellers`**(*self*, *funcparam*)

An empirically, by *William Sellers* adjusted stefan-boltzmann radiation as radiative energy flux directed upward. The ideal stefan-boltzmann radiation with a temperature to the power of 4 and an additional tangens hyperbolicus term with the temperature to the power of 6 to take into account that cloud formation is temperature dependant.

The upward radiative energy flux R_{up} of latitude ϕ is given by:

$$R_{up}(\phi) = -\sigma \cdot T(\phi)^4 \cdot (1 - m \cdot tanh(\gamma \cdot T(\phi)^6)$$

with the temperature $T(\phi)$, the stefan-boltzmann constant σ, the atmospheric attenuation m and an empirical constant γ.

To make this function more adjustable there is an additional emissivity introduced (similar to `flux_up.planck`).

Function-call arguments

Parameters **`funcparams`** (*dict*) – a dictionary of the function's parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *m*: The atmospheric attenuation
  - type: float
  - unit: -
  - value: $0 \leq$ m ≤ 1 (standard 0.5)
- *sigma*: Stefan-boltzmann constant
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Kelvin^{-4}$
  - value: $5,67 \cdot 10^{-8}$ (use const.sigma to load it from `climlab.constants`)
- *gamma*: Empirical constant in the cloud term
  - type: float
  - unit: $Kelvin^{-6}$
  - value: $1.9 \cdot 10^{-15}$
- *grey*: The emissivity (greyness)
  - type: float
  - unit: -
  - value: $0 \leq$ grey ≤ 1 (standard 1)

Returns The upward radiative energy flux R_{up}

Return type float / array(floats) (0D / 1D)

transfer

class lowEBMs.Packages.Functions.**transfer**

Bases: object

Class defining latitudinal energy transfer transfer fluxes.

The equations used here are estimated empirically based on research of *Michail Budyko* and *William Sellers*.

budyko	A poleward energy transfer flux based on the local to global temperature difference introduced by *Michail Budyko*.
sellers	A energy transfer flux based on a combination of several transfer fluxes introduced by *William Sellers*.
watervapour_sel	The energy transfer flux through watervapour used in `transfer().sellers`.
sensibleheat_air_sel	The energy transfer flux through atmospheric sensible heat used in `transfer().sellers`.
sensibleheat_ocean_sel	The energy transfer flux through oceanic sensible heat used in `transfer().sellers`.

Note: Only `transfer().budyko` and `transfer().sellers` are transfer fluxes fully representing the globes meridional energy transfer, where `transfer().sellers` is built up from the three specific transfer fluxes `transfer().watervapour_sel`, `transfer().sensibleheat_air_sel` and `transfer().sensibleheat_ocean_sel`.

lowEBMs.Packages.Functions.transfer.**budyko**(*self*, *funcparam*)

A poleward energy transfer flux based on the local to global temperature difference introduced by *Michail Budyko*.

It can be shown that it is equivalent to the diffusive heat transfer of the globe (North, 1975b).

It is given by:

$$F_{transfer} = \beta \cdot (T(\phi) - T_g)$$

with the temperature $T(\phi)$ of latitude ϕ, the global mean temperature T_g and the transport parameter β.

Function-call arguments

Parameters **funcparams** (*dict*) – a dictionary of the function's parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *beta*: The transport parameter
  - type: float
  - unit: $Watt \cdot meter^{-2} \cdot Kelvin^{-1}$
  - value: any (standard 3.74)
- *Read*: Indicates whether the transfer flux is specifically provided as output
  - type: boolean
  - unit: -
  - value: True/False (standard True)

- *Activated*: Indicates whether the transfer flux is actually activated
  - type: boolean
  - unit: -
  - value: True/False (standard True)

Returns The Budyko energy transfer flux $F_{transfer}$

Return type array(floats) (1D)

`lowEBMs.Packages.Functions.transfer.`**`sellers`**(*self*, *funcparam*)

A energy transfer flux based on a combination of several transfer fluxes introduced by *William Sellers*.

It is defined as the difference of a sum of northward and a sum of southward transfer fluxes of one latitudinal belt. The sum (in one direction) P consists of `transfer().watervapour_sel`, `transfer().sensibleheat_air_sel` and `transfer().sensibleheat_ocean_sel`:

$$P(\phi) = L \cdot c_{wv}(\phi) + C_{air}(\phi) + F_{oc}(\phi)$$

with the energy transfer through watervapour $c_{wv}(\phi)$, the energy transfer through atmospheric sensible heat $C_{air}(\phi)$ and the energy transfer through oceanic sensible heat $F_{oc}(\phi)$ of latitude ϕ and the latent heat of condensation L.

The total energy flux, the difference of the southward and northward flux $P(\phi)$ weighted with the length of a latitudinal circle :math'l(phi)' and the area of the latitudinal belt $A(\phi)$, is given by:

$$F_{transfer} = (P(\phi) \cdot l(\phi) - P(\phi + d\phi) \cdot l(\phi + d\phi)) \cdot \frac{1}{A(\phi)}$$

where $P(\phi) \cdot l(\phi)$ is the sum of energy transfer from the latitudinal belt to the southern boundary and $P(\phi + d\phi) \cdot l(\phi + d\phi)$ the one to the nothern boundary ($d\phi$ indicates the step to the next northern gridpoint).

Note: The Sellers energy transfer flux comes with a large set of parameters, some given as scalars and some as distribution over the latitudes. In order to simplify the input of these parameters, the module `lowEBMs.Packages.Configuration.add_sellersparameters` can be called before running the algorithm which imports the parameter distributions into the *funcparam* dictionary. **Scalars are not included there!** The easiest way is to copy the prewritten configuration of this function from the *FunctionCalls.txt* in *lowEBMs/Turotials* and use `Configuration.add_sellersparameters`.

Function-call arguments

Parameters **`funcparams`** (*dict*) – a dictionary of the function's parameters directly parsed from `lowEBMs.Packages.ModelEquation.model_equation`

- *Readout*: Indicates whether **all** sellers transfer fluxes are provided as output
  - type: boolean
  - unit: -
  - value: True/False (standard True)
- *Activated*: Indicates if the transfer flux is actually activated
  - type: boolean
  - unit: -
  - value: True/False (standard True)
- *K_wv*: The thermal diffusivity of the watervapour term

  - type: float
  - unit: $meter^2 \cdot second^{-1}$
  - value: 10^5 (imported by `Configuration.add_sellersparameters`)
- *K_h*: The thermal diffusivity of the atmospheric sensible heat term
  - type: float
  - unit: $meter^2 \cdot second^{-1}$
  - value: 10^6 (imported by `Configuration.add_sellersparameters`)
- *K_o*: The thermal diffusivity of the oceanic sensible heat term
  - type: float
  - unit: $meter^2 \cdot second^{-1}$
  - value: 10^2 (imported by `Configuration.add_sellersparameters`)
- *g*: The gravitational acceleration
  - type: float
  - unit: $meter \cdot second^{-2}$
  - value: 9.81
- *a*: Empricial constant to calculate the meridional windspeed
  - type: float
  - unit: $meter \cdot second^{-1} \cdot Celcius^{-1}$
  - value: 10^-2 (imported by `Configuration.add_sellersparameters`)
- *eps*: Empirical constant of the saturation specific humidity
  - type: float
  - unit: -
  - value: 0.622
- *p*: The average sea level pressure
  - type: float
  - unit: $mbar$
  - value: 1000
- *e0*: The mean sea level saturation vapour pressure
  - type: float
  - unit: $mbar$
  - value: 17
- *L*: The latent heat of condensation
  - type: float
  - unit: $Joule \cdot gramm^{-1}$
  - value: $2.5 \cdot 10^3$
- *Rd*: The gas constant

  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 0.287
- *dy*: The width of an latitudinal belt
  - type: float
  - unit: $meter$
  - value: $1.11 \cdot 10^6$
- *dp*: The tropospheric pressure depth
  - type: float
  - unit: $mbar$
  - value: 700-900 (imported by `Configuration.add_sellersparameters`)
- *cp*: The specific heat capacity of air at constant pressure
  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 1.004
- *dz*: The average zonal ocean depth
  - type: float
  - unit: $meter$
  - value: 1000-4000 (imported by `Configuration.add_sellersparameters`)
- *l_cover*: The proportion of ocean covered surface
  - type: float
  - unit: -
  - value: 0.5
- *re*: The earth's radius
  - type: float
  - unit: $meter$
  - value: $6.371 \cdot 10^6$
- *cp_w*: The specific heat capacity of sea water
  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 4182
- *dens_w*: The density of water
  - type: float
  - unit: $gramm \cdot meter^{-3}$
  - value: $0.997 \cdot 10^6$
- *factor_wv*: A tuning factor applied to the watervapour term

  - type: float
  - unit: -
  - value: any
- *factor_air*: A tuning factor applied to the atmospheric sensible heat term
  - type: float
  - unit: -
  - value: any
- *factor_oc*: A tuning factor applied to the oceanic sensible heat term (or it's diffusivity)
  - type: float
  - unit: -
  - value: any
- *factor_kwv*: A tuning factor applied to the thermal diffusivity of the watervapour term
  - type: float
  - unit: -
  - value: any
- *factor_kair*: A tuning factor applied to the thermal diffusivity of the atmospheric sensible heat term
  - type: float
  - unit: -
  - value: any

Returns The Sellers energy transfer flux $F_{transfer}$

Return type array(floats) (1D)

`lowEBMs.Packages.Functions.transfer.`**`watervapour_sel`**(*self*, *funcparam*)

The energy transfer flux through watervapour used in `transfer().sellers`.

It is based on the transport of watervapour to another latitudinal belt and it's condensation which releases energy. It is described through:

$$c_{wv} = \left(vq - K_{wv} \frac{\Delta q}{\Delta y} \right) \cdot \frac{\Delta p}{g}$$

with the meridional windspeed v provided by `earthsstem.meridionalwind_sel`, the specific saturation humidity q provided by `earthsystem().specific_saturation_humidity_sel` and the humidity difference *dq* provided by `earthsystem().humidity_difference`. Additional parameters are the thermal diffusivity of watervapour K_{wv}, the width of the latitudinal belts Δy, the tropospheric pressure depth Δp and the gravitational acceleration g.

For purposes of tuning, c_{wv} and K_{wv} are provided with the scaling factors *factor_wv* and *factor_kwv*.

Function-call arguments

Parameters **`funcparams`** (*dict*) –

- *K_wv*: The thermal diffusivity of the watervapour term
  - type: float
  - unit: $meter^2 \cdot second^{-1}$

  - value: 10^5 (imported by `Configuration.add_sellersparameters`)
- g: The gravitational acceleration
  - type: float
  - unit: $meter \cdot second^{-2}$
  - value: 9.81
- *eps*: Empirical constant of the saturation specific humidity
  - type: float
  - unit: -
  - value: 0.622
- p: The average sea level pressure
  - type: float
  - unit: $mbar$
  - value: 1000
- *e0*: The mean sea level saturation vapour pressure
  - type: float
  - unit: $mbar$
  - value: 17
- *L*: The latent heat of condensation
  - type: float
  - unit: $Joule \cdot gramm^{-1}$
  - value: $2.5 \cdot 10^3$
- *Rd*: The gas constant
  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 0.287
- *dy*: The width of an latitudinal belt
  - type: float
  - unit: $meter$
  - value: $1.11 \cdot 10^6$
- *dp*: The tropospheric pressure depth
  - type: float
  - unit: $mbar$
  - value: 700-900 (imported by `Configuration.add_sellersparameters`)
- *factor_wv*: A tuning factor applied to the watervapour term
  - type: float
  - unit: -

- value: any

- *factor_kwv*: A tuning factor applied to the thermal diffusivity of the watervapour term
  - type: float
  - unit: -
  - value: any

Returns The watervapour energy transfer flux c_{wv}

Return type array(floats) (1D)

lowEBMs.Packages.Functions.transfer.**sensibleheat_air_sel**(*self*, *funcparam*)

The energy transfer flux through atmospheric sensible heat used in `transfer().sellers`.

It is based on the heat transport through wind and convection to another latitudinal belt. It is described through:

$$C_{air} = \left(vT - K_h \frac{\Delta T}{\Delta y} \right) \cdot \frac{c_p}{g} \Delta p$$

with the meridional windspeed v provided by `earthsstem.meridionalwind_sel`, and the temperature difference ΔT provided by `earthsystem().tempdif`. Additional parameters are the temperature T, the thermal diffusivity of air K_h, the width of the latitudinal belts Δy, the tropospheric pressure depth Δp, the specific heat capacity of air c_p and the gravitational acceleration g.

For purposes of tuning, C_{air} and K_h are provided with the scaling factors *factor_air* and *factor_kair*.

Function-call arguments

Parameters **funcparams** (*dict*) –

- *K_h*: The thermal diffusivity of the atmospheric sensible heat term
  - type: float
  - unit: $meter^2 \cdot second^{-1}$
  - value: 10^6 (imported by `Configuration.add_sellersparameters`)
- *g*: The gravitational acceleration
  - type: float
  - unit: $meter \cdot second^{-2}$
  - value: 9.81
- *dy*: The width of an latitudinal belt
  - type: float
  - unit: $meter$
  - value: $1.11 \cdot 10^6$
- *dp*: The tropospheric pressure depth
  - type: float
  - unit: $mbar$
  - value: 700-900 (imported by `Configuration.add_sellersparameters`)
- *cp*: The specific heat capacity of air at constant pressure
  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$

  - value: 1.004
- *factor_air*: A tuning factor applied to the atmospheric sensible heat term
  - type: float
  - unit: -
  - value: any
- *factor_kair*: A tuning factor applied to the thermal diffusivity of the atmospheric sensible heat term
  - type: float
  - unit: -
  - value: any

Returns The atmospheric sensible heat energy transfer flux C_{air}

Return type array(floats) (1D)

`lowEBMs.Packages.Functions.transfer.`**`sensibleheat_ocean_sel`**(*self*, *funcparam*)

The energy transfer flux through oceanic sensible heat used in `transfer().sellers`.

It is based on the heat transport through oceanic convection to another latitudinal belt. It is described through:

$$F_{oc} = -K_o l_{cover} \Delta z \frac{\Delta T}{\Delta y} \cdot C_{p,w}$$

ho_{w}

with the temperature difference ΔT provided by `earthsystem().tempdif`. Additional parameters are the thermal diffusivity of the ocean K_o, the width of the latitudinal belts Δy, the average ocean depth Δz, the proportion of ocean cover l_{cover}, the specific heat capacity of water $c_{p,w}$ and the densitiy of water ρ_w.

For purposes of tuning, a scaling factors *factor_oc* is provided.

Function-call arguments

param dict funcparams

- *K_o*: The thermal diffusivity of the oceanic sensible heat term
  - type: float
  - unit: $meter^2 \cdot second^{-1}$
  - value: 10^2 (imported by `Configuration.add_sellersparameters`)
- *dz*: The average zonal ocean depth
  - type: float
  - unit: $meter$
  - value: 1000-4000 (imported by `Configuration.add_sellersparameters`)
- *l_cover*: The proportion of ocean covered surface
  - type: float
  - unit: -

  - value: 0.5
- *cp_w*: The specific heat capacity of sea water
  - type: float
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 4182
- *dens_w*: The density of water
  - type: float
  - unit: $gramm \cdot meter^{-3}$
  - value: $0.997 \cdot 10^6$
- *factor_oc*: A tuning factor applied to the oceanic sensible heat term (or it's diffusivity)
  - type: float
  - unit: -
  - value: any

returns The oceanic sensible heat energy transfer flux F_{oc}

rtype array(floats) (1D)

forcing

class lowEBMs.Packages.Functions.**forcing**

Bases: object

Class defining radiative forcing terms

Radiative forcing essentially can be comprehended as manipulation of the solar insolation which defines the amount of energy available in the system. This class defines terms which change the amount of energy in the system which can be derived from properties of the system (e.g. atmospheric CO2-concentrations). One has to be cautious when operating with these forcing because they are only used to mimic a radiative forcing detached from the systems propagation and therefore don't represent interactions of the property which causes the forcing (e.g. the carbon cycle is not considered, only measured or projected trends).

random	The random forcing mimics randomly occuring radiative forcing events.
predefined	The predefined forcing imports data containing external radiative forcings.
co2_myhre	The co2_myhre forcing calculates a radiative forcing from imported atmospheric CO2 conenctration data.

lowEBMs.Packages.Functions.forcing.**random**(*self*, *funcparam*)

The random forcing mimics randomly occuring radiative forcing events.

The random forcing is mainly used to mimic volcanic eruptions which are based on the idea that dust clouds, appearing after volcanic eruptions, affect the radiative balance. The consequence of an eruption is a negative radiative forcing over a specific time which generally causes a decrease in temperature, depending on the time and strentgh the forcing acts.

With this module such events are randomly generated. The parameters which have to be provided will determine

a rough guess of frequency, strength and length of the events and many more. By setting a time of start and stop, this can be used to turn on/off the random radiative forcing for specifc times.

Function-call arguments

Parameters **funcparams** (*dict*) –

- *forcingnumber* the number of the radiative forcing term (relevant if multiple forcings are used)
  - type: int
  - unit: -
  - value: 0, 1,...
- *start*: The time when the forcing starts
  - type: float
  - unit: depending on *timeunit*
  - value: any
- *stop*: The time when the forcing stops
  - type: float
  - unit: depending on *timeunit*
  - value: any
- *steps*: The amount of steps (timeresolution) between start and stop
  - type: int
  - unit: -
  - value: any (preferably the same as `builtins.stepsize_of_integration`)
- *timeunit*: The unit of time for the forcing
  - type: string
  - unit: -
  - value: 'minute', 'hour', 'day', 'week', 'month', 'year' (if none, seconds are used)
- *strength*: The maximal radiative forcing which can be randomly generated
  - type: float
  - unit: -
  - value: any
- *frequency*: A classification how often an event occurs. Creates a window of frequency, from a minimal duration between two events towards a maximal from which the duration to the next event is randomly chosen.
  - type: string
  - unit: -
  - value: options
    - 'common': the next event is in the following 0-4 steps/total_steps
    - 'intermediate': the next event is in the following 4-12 steps/total_steps

        * 'rare': the next event is in the following 12-30 steps/total_steps
        * 'superrare': the next event is in the following 30-60 steps/total_steps
    - *behaviour*: The behaviour/shape of the radiative forcing
      - type: string
      - unit: -
      - value: options
        * 'step': radiative forcing acts as stepfunction with width of one step defined by *lifetime*
        * 'exponential': radiative forcing acts exponentially with a halflife defined by *lifetime*s
    - *lifetime*: The length on event appears (coupled to *behaviour*)
      - type: float
      - unit: depending on *timeunit*
      - value: any
    - *seed*: Indicates whether the random numbers are generated from a specific seed (for comparison)
      - type: int
      - unit: -
      - value: any (if None, every call is random)
    - *sign*: The sign of the resulting radiative forcing
      - type: string
      - unit: -
      - value: 'negative', 'positive'

  Returns A randomly generated radiative forcing

  Return type float

lowEBMs.Packages.Functions.forcing.**predefined**(*self*, *funcparam*)

The predefined forcing imports data containing external radiative forcings.

This module imports radiative forcing data given as change in energy ($Watt \cdot meter^{-2}$) and applies it to the model run.

Function-call arguments

Parameters **funcparams** (*dict*) –

- *forcingnumber* the number of the radiative forcing term (relevant if multiple forcings are used)
  - type: int
  - unit: -
  - value: 0, 1,...
- *datapath*: The path to the file (give full path or relative path!)
  - type: string
  - unit: -

  - value: example: '/insert/path/to/file'
- *name*: The name of the file which is used
  - type: string
  - unit: -
  - value: example: 'datafile.txt'
- *delimiter*: How the data is delimited in the file
  - type: string
  - unit: -
  - value: example: ','
- *header*: The number of header rows to exclude
  - type: int
  - unit: -
  - value: any
- *col_time*: The column where the time is stored
  - type: int
  - unit: -
  - value: any
- *col_forcing*: The column where the forcing in stored
  - type: int
  - unit: -
  - value: any
- *timeunit*: The unit of time which is used in the file to convert it to seconds
  - type: string
  - unit: -
  - value: 'minute', 'hour', 'day', 'week', 'month', 'year' (if none, seconds are used)
- *BP*: If the time is given as "Before present"
  - type: boolean
  - unit: -
  - value: True / False
- *time_start*: The time of the first entry (or the time when is should be started to apply it)
  - type: float
  - unit: depending *timeunit*
  - value: any
- *k*: Scaling factor
  - type: float
  - unit: -

- value: any

Returns The radiative forcing for a specific time imported from a data file

Return type float

lowEBMs.Packages.Functions.forcing.**co2_myhre**(*self*, *funcparam*)

The co2_myhre forcing calculates a radiative forcing from imported atmospheric CO2 conenctration data.

This module imports atmospheric CO2 concentrations from a data file and converts them to a change in energy F_{CO2} (W/m^2) after *Myhre (1998)*:

$$F_{CO2} = A \cdot ln(C/C_0)$$

With the atmospheric CO2 concentration C ($ppmv$), the preindustrial atmospheric CO2 concentration C_0 and an empricial constant A (5.35 W/m^2).

Function-call arguments

Parameters **funcparams** (*dict*) –

- *A* The empirical constant A
  - type: float
  - unit: $Watt \cdot meter^{-2}$
  - value: 5.35
- *C_0*: The preindustrial atmospheric CO2 concentration
  - type: float
  - unit: $ppmv$
  - value: 280
- *C02_base*: The CO2 concentration to use before the forcing starts and after it ends
  - type: float
  - unit: $ppmv$
  - value: any
- *datapath*: The path to the file (give full path or relative path!)
  - type: string
  - unit: -
  - value: example: '/insert/path/to/file'
- *name*: The name of the file which is used
  - type: string
  - unit: -
  - value: example: 'datafile.txt'
- *delimiter*: How the data is delimited in the file
  - type: string
  - unit: -
  - value: example: ','
- *header*: The number of header rows to exclude

  - type: int
  - unit: -
  - value: any
- *footer*: The number of footer rows to exclude
  - type: int
  - unit: -
  - value: any
- *col_time*: The column where the time is stored
  - type: int
  - unit: -
  - value: any
- *col_conc*: The column where the concentration in stored
  - type: int
  - unit: -
  - value: any
- *timeunit*: The unit of time which is used in the file to convert it to seconds
  - type: string
  - unit: -
  - value: 'minute', 'hour', 'day', 'week', 'month', 'year' (if none, seconds are used)
- *BP*: If the time is given as "Before present"
  - type: boolean
  - unit: -
  - value: True / False
- *time_start*: The time of the first entry (or the time when is should be started to apply it)
  - type: float
  - unit: depending *timeunit*
  - value: any

Returns The radiative forcing for a specific time calculated from atmospheric CO2-concentrations imported from a data file

Return type float

albedo

class lowEBMs.Packages.Functions.**albedo**

Bases: object

Class defining the albedo distributions

static	Function defining a static albedo value
static_bud	A static albedo distribution as used in *Budyko*.
dynamic_bud	A temperature dependant albedo distribution with three albedo regions.
smooth	A temperature dependant albedo distribution with tangens hyperbolicus transition.
dynamic_sel	A albedo distribution with linear temperature dependence.

Note: These are special functions which are used by `flux_down.insolation`. In the *configuration.ini* they have to be inserted in its [func]-section with the parameters used (see *albedo*).

lowEBMs.Packages.Functions.albedo.**static**(*self*, *alpha*)

Function defining a static albedo value

Function-call arguments

Parameters **alpha** (*float*) – the globally averaged albedo value

- type: float
- unit: -
- value: $0 \leq$ alpha ≤ 1

Returns The globally averaged albedo value

Return type float (0D)

lowEBMs.Packages.Functions.albedo.**static_bud**(*self*, *alpha_p*, *border_1*, *border_2*)

A static albedo distribution as used in *Budyko*.

The albedo distribution is described through three zones of albedo values.

Latitude of transition	Albedo value alhpa
< *border_1*	low albedo zone: alpha= *alpha_p*
> *border_1*	intermediate zone: alpha= *alpha_p* +0.18
> *border_2*	high albedo zone: alpha= *alpha_p* +0.3

Function-call arguments

Parameters **albedoparam** (*array*) – albedo distribution parameters *[alpha_p,border_1,border_2]*

- *alpha_p*: The low albedo zone value
  - type: float
  - unit: -
  - value: $0 \leq$ albedo ≤ 1 (standard 0.3)
- *border_1*: Latitude of low to intermediate albedo zone transition
  - type: float
  - unit: Unit of latitude (degree)
  - value: $0 \leq$ border_1 ≤ 90 (standard 60)

- *border_2*: Latitude of intermediate to high albedo zone transition
  - type: float
  - unit: Unit of latitude (degree)
  - value: $0 \leq$ border_2 ≤ 1 (standard 70)

Returns The latitudinal albedo distribution

Return type array(floats) (1D)

lowEBMs.Packages.Functions.albedo.**dynamic_bud**(*self*, *T_1*, *T_2*, *alpha_0*, *alpha_1*, *alpha_2*)

A temperature dependant albedo distribution with three albedo regions. Approach as used in *Budyko* but complemented with albedo transition depending on temperature.

The albedo distribution is described through three zones of albedo values.

Temperature of transition	Albedo value alpha
> *T_1*	low albedo zone: alpha= *alpha_0*
< *T_1* & > *T_2*	intermediate zone: alpha= *alpha_1*
< *T_2*	high albedo zone: alpha= *alpha_2*

Function-call arguments

Parameters **albedoparam** (*array*) – albedo distribution parameters *[T_1,T_2,alpha_0,alpha_1,alpha_2]*

- *T_1*: Temperature of low to intermediate albedo zone transition
  - type: float
  - unit: $Kelvin$
  - value: > 0 in Kelvin (standard 273.15)
- *T_2*: Temperature of intermediate to high albedo zone transition
  - type: float
  - unit: $Kelvin$
  - value: > 0 in kelvin (standard 263.15)
- *alpha_0*: The low albedo zone value
  - type: float
  - unit: -
  - value: $0 \leq$ alpha_0 ≤ 1 (standard 0.32)
- *alpha_1*: The intermediate albedo zone value
  - type: float
  - unit: -
  - value: $0 \leq$ alpha_1 ≤ 1 (standard 0.5)
- *alpha_2*: The high albedo zone value
  - type: float
  - unit: -

- value: $0 \leq$ alpha_2 ≤ 1 (standard 0.62)

Returns The latitudinal albedo distribution

Return type array(floats) (1D)

lowEBMs.Packages.Functions.albedo.**smooth**(*self*, *T_ref*, *alpha_f*, *alpha_i*, *steepness*)

A temperature dependant albedo distribution with tangens hyperbolicus transition. A common approach in climate modelling (for example see *North*)

The albedo of one latitude is defined by:

$$\alpha(\phi) = \alpha_i - \frac{1}{2}(\alpha_i - \alpha_f) \cdot (1 + tanh(\gamma \cdot (T(\phi) - T_{ref})))$$

with the albedo value $\alpha(\phi)$ and temperature $T(\phi)$ of latitude ϕ, an ice-covered/ice-free albedo value α_i/α_f, the reference temperature of transition T_{ref} and the steepness of the transition γ.

Function-call arguments

Parameters **albedoparam** (*array*) – albedo distribution parameters *[T_ref,alpha_f,alpha_i,steepness]*

- *T_ref*: Reference transition temperature from ice-free to ice-covered albedo
  - type: float
  - unit: $Kelvin$
  - value: > 0 in Kelvin (standard 273.15)
- *alpha_i*: The ice-covered albedo value
  - type: float
  - unit: -
  - value: $0 \leq$ alpha_i ≤ 1 (standard 0.7)
- *alpha_f*: The ice-free albedo value
  - type: float
  - unit: -
  - value: $0 \leq$ alpha_f ≤ 1 (standard 0.3)
- *steepness*: The steepness of albedo transition (γ)
  - type: float
  - unit: $Kelvin^{-1}$
  - value: $0 \leq$ steepness ≤ 1 (standard 0.3)

Returns The latitudinal albedo distribution

Return type float / array(floats) (0D / 1D)

lowEBMs.Packages.Functions.albedo.**dynamic_sel**(*self*, *Z*, *b*)

A albedo distribution with linear temperature dependence. Approach as used by *Sellers*.

The albedo of one latitude is defined by:

$$\begin{aligned} & T_g(\phi) = T(\phi) - 0.0065 \cdot Z(\phi) \\ If \quad & T_g(\phi) < 283.15: \quad \alpha(\phi) = b(\phi) - 0.009 \cdot T_g(\phi) \\ If \quad & T_g(\phi) > 283.15: \quad \alpha(\phi) = b(\phi) - 2.548 \end{aligned}$$

with the albedo value $\alpha(\phi)$ (maximum of 0.85) and temperature $T(\phi)$ of latitude ϕ, the altitude weighted temperature T_g with the zonal mean altitude $Z(\phi)$ and empirical constants $b(\phi)$.

Function-call arguments

Parameters **albedoparam** (*array*) – albedo distribution parameters *[Z,b]*

- *Z*: Zonal mean altitude (provided by `Configuration.add_sellersparameters`)
  - type: array(float)
  - unit: $Kelvin \cdot meter^{-1}$
  - value: > 0
- *b*: Empirical constant to estimate the albedo (provided by `Configuration.add_sellersparameters`)
  - type: float
  - unit: -
  - value: > 0

Returns The latitudinal albedo distribution

Return type array(floats) (1D)

earthsystem

class lowEBMs.Packages.Functions.**earthsystem**

Bases: `object`

Class defining earthsystem properties

globalmean_temperature	The GMT calculated from the ZMT with a gridspcific areaweighting.
insolation	
solarradiation	
solarradiation_orbital	The solar insolation over the latitudes Q with changing orbital parameters.
specific_saturation_humidity_sel	The specific saturation humidity of a latitudinal belt.
saturation_pressure	The saturation pressure of a latitudinal belt.
humidity_difference	The humidity difference between latitudinal belts.
temperature_difference_latitudes	The temperature difference between latitudinal belts.
length_latitudes	The length (circumference) of the latitudinal circles.
area_latitudes	The area of the latitudinal belts.

lowEBMs.Packages.Functions.earthsystem.**globalmean_temperature**(*self*)

The GMT calculated from the ZMT with a gridspcific areaweighting.

The GMT, T_{global}, is given by:

$$T_{global} = \int_{\phi_s}^{\phi_n} T(\phi) \cdot cos(\phi) d\phi$$

with the ZMT, $T(\phi)$, of latitude ϕ, and the borders of the grid ϕ_s/ϕ_n.

Function-call arguments

Returns The global mean temperature in Kelvin

Return type float

lowEBMs.Packages.Functions.earthsystem.**solarradiation**(*self*, *convfactor*, *timeunit*, *orbitalyear*, *Q*)

lowEBMs.Packages.Functions.earthsystem.**solarradiation_orbital**(*self*, *convfactor*, *orbitalyear*, *unit*)

The solar insolation over the latitudes Q with changing orbital parameters.

The functionality of this module is in its main features the same as `earthsystem().solarradiation` with the addition that the orbital parameters are imported from `climlab.solar.orbital` and updated continously if `Vars.t` passes to the next century (can only be updated in kiloyears).

Function-call arguments

Parameters

- **convfactor** (*float*) – Conversionfactor if another unit is desired
  - unit: depending on the conversion
  - value: any
- **orbitalyear** (*string*) – Indicates for which year the orbitalparameters are chosen and updated from
  - unit: $kyear$
  - value: -5000 to 0
- **unit** (*string*) – Indicates which unit of time is used in the simulation
  - unit: -
  - value: various options
    - 'year': Returns the solarinsolation at time t which is given in unit years
    - None: Use the value given in the **Configuration.ini**

Returns The solar insolation over latitudes with update of orbital parameters

Return type float / array(float) (0D / 1D)

lowEBMs.Packages.Functions.earthsystem.**meridionalwind_sel**(*self*, *a*, *re*)

The meridional wind v between latitudinal belts.

This function is part of the `transfer().sellers` module which calculates the meridional windspeed depending on a latitudes temperature. It is given by:

$$v = -a \cdot (\Delta T \pm |\overline{\Delta T}|)$$

with $+$ north of 5°N and $-$ south of 5°N, the temperature difference between latitudes ΔT provided by `earthsystem().temperature_difference_latitudes`, empirical constants a and the area weighted mean temperature difference $|\overline{\Delta T}|$.

The required parameters are directly parsed from the `transfer().sellers` module, for details see *here*.

Function-call arguments

Parameters

- **a** (*float*) – Empirical constants estimating the windspeed of a latitudinal belt
  - unit: $meter \cdot second^{-1} \cdot Kelvin^{-1}$
  - value: (imported by `Configuration.add_sellersparameters`)

- **re** (*float*) – The earth's radius
  - unit: meter
  - value: $6.371 \cdot 10^6$

Returns The meridional windspeed

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**specific_saturation_humidity_sel**(*self*, *e0*, *eps*, *L*, *Rd*, *p*)

The specific saturation humidity of a latitudinal belt.

This function is part of the `transfer().sellers` module which calculates provides the required properties for `transfer().watervapour_sel`. It is given by:

$$q = \frac{\epsilon \cdot e}{p}$$

with an empirical constant ϵ, the average sea level pressure p and the saturation pressure e from `earthsystem().saturation_pressure`.

The required parameters are directly parsed from the `transfer().sellers` module, for details see *here*.

Function-call arguments

Parameters

- **e0** (*float*) – The mean sea level saturation vapour pressure
  - unit: $mbar$
  - value: 17
- **eps** (*float*) – Empirical constant of the saturation specific humidity
  - unit: -
  - value: 0.622
- **L** (*float*) – The latent heat of condensation
  - unit: $Joule \cdot gramm^{-1}$
  - value: $2.5 \cdot 10^3$
- **Rd** (*float*) – The gas constant
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 0.287
- **p** (*float*) – The average sea level pressure
  - unit: $mbar$
  - value: 1000

Returns The specific saturation humidity

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**saturation_pressure**(*self*, *e0*, *eps*, *L*, *Rd*)

The saturation pressure of a latitudinal belt.

This function is part of the `transfer().sellers` module which calculates provides the required properties for `earthsystem().humidity_difference` and `earthsystem().specific_saturation_humidity_sel`. It is given by:

$$e = e_0 \left(1 - 0.5 \frac{\epsilon L \Delta T}{R_d T^2}\right)$$

with the temperature difference between latitudes ΔT provided by `earthsystem().temperature_difference_latitudes`, the empirical constant ϵ, the gas constant R_d, the latent heat of condensation L, the mean sea level saturation vapour pressure e_0 and the temperature of the southern latitudinal belt T.

The required parameters are directly parsed from the `earthsystem().specific_saturation_humidity_sel` module, for details see *here*.

Function-call arguments

Parameters

- **e0** (*float*) – The mean sea level saturation vapour pressure
  - unit: $mbar$
  - value: 17
- **eps** (*float*) – Empirical constant of the saturation specific humidity
  - unit: -
  - value: 0.622
- **L** (*float*) – The latent heat of condensation
  - unit: $Joule \cdot gramm^{-1}$
  - value: $2.5 \cdot 10^3$
- **Rd** (*float*) – The gas constant
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 0.287

Returns The saturation pressure

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**humidity_difference**(*self*, *e0*, *eps*, *L*, *Rd*, *p*)

The humidity difference between latitudinal belts.

This function is part of the `transfer().sellers` module which calculates provides the required properties for `transfer().watervapour_sel`. It is given by:

$$\Delta q = \frac{e \epsilon^2 L \Delta T}{p R_d T^2}$$

with the temperature difference between latitudes ΔT provided by `earthsystem().temperature_difference_latitudes`, the empirical constant ϵ, the average sea level pressure p, the gas constant R_d, the latent heat of condensation L, the mean sea level saturation vapour pressure e_0 and the temperature of the southern latitudinal belt T.

The required parameters are directly parsed from the `transfer().sellers` module, for details see *here*.

Function-call arguments

Parameters

- **e0** (*float*) – The mean sea level saturation vapour pressure
  - unit: $mbar$
  - value: 17
- **eps** (*float*) – Empirical constant of the saturation specific humidity
  - unit: -
  - value: 0.622
- **L** (*float*) – The latent heat of condensation
  - unit: $Joule \cdot gramm^{-1}$
  - value: $2.5 \cdot 10^3$
- **Rd** (*float*) – The gas constant
  - unit: $Joule \cdot gramm^{-1} \cdot Kelvin^{-1}$
  - value: 0.287
- **p** (*float*) – The average sea level pressure
  - unit: $mbar$
  - value: 1000

Returns The humidity difference between two latitudinal belts

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**temperature_difference_latitudes**(*self*)

The temperature difference between latitudinal belts.

It is given by

$$\Delta T = T_{north} - T_{south}$$

with the temperature of the southern and northern latitudinal belt T_{south}, T_{south}.

Function-call arguments

Returns The temperature difference between the latitudinal belts over the latitudes

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**length_latitudes**(*self*, *re*)

The length (circumference) of the latitudinal circles.

It is given by

$$l = 2\pi \cdot r \cdot cos(\phi)$$

with the earths radius r and the degree of latitude ϕ.

Function-call arguments

Parameters **re** (*float*) – The earth's radius

- unit: meter
- value: $6.371 \cdot 10^6$

Returns The length of the latitudinal circles

Return type array(float) (1D)

lowEBMs.Packages.Functions.earthsystem.**area_latitudes**(*self*, *re*)

The area of the latitudinal belts.

It is given by

$$A = \pi r^2 \left([sin(90 - \phi_s)^2 + (1 - cos(90 - \phi_s))^2] - [sin(90 - \phi_n)^2 + (1 + cos(90 - \phi_n))^2] \right)$$

with the earths radius r and the degree of northern and southern latitudinal circle ϕ_n, ϕ_s.

Function-call arguments

Parameters **re** (*float*) – The earth's radius

- unit: meter
- value: $6.371 \cdot 10^6$

Returns The area of the latitudinal belts

Return type array(float) (1D)

tools

lna

nal

cosd

sind

plotmeanstd

datasetaverage

interpolator

SteadyStateConditionGlobal

BPtimeplot

lowEBMs.Packages.Configuration

Package with functions which configure the model setup.

importer	Reads a **configuration.ini-file** and creates the model run setup in a dictionary.
dict_to_list	Converts dictionaries returned from `Configuration.importer` into a list with the same structure.

Continued on next page

Table 10 – continued from previous page

parameterimporter	A function purpose-built to import 1-dimensional parameters for the sellers-type functions.
parameterinterpolator	An interpolation method fitting a polynomial of degree 10 to the parameter distributions.
parameterinterpolatorstepwise	An interpolation method stepwise fitting and averaging a polynomial of degree 2 to the parameter distribution.
add_sellersparameters	Overwrites the model setup with one-dimensional sellers parameters.
import_parallelparameter	Imports information from a .ini-file to create a setup of parallelized simulations.
allocate_parallelparameter	Transforms parameters for parallelization from tuple *[start,end]* to list *[start,...,end]* of length *number_of_cycles*.
write_parallelparameter	Overwrites the single run model setup with a parallelized model setup.

lowEBMs.Packages.Configuration.**importer**(*filename*, **args*, ***kwargs*)

Reads a **configuration.ini-file** and creates the model run setup in a dictionary. It is one of the coremodules of this project, mostly called as first step, because it gathers all information about the model setup and summarizes it.

Note: The file from which the information is imported has to have a specific structure, please read Input first to see how the **configuration.ini-files** are created.

The specificaton of the path to the filedirectory is optional. If none is given some standard diretories will be tried (python sys.paths and relative paths like '../', '../Config/',..)

Function-call arguments

Parameters

- **filename** (*string*) – The name of the **configuration.ini-file**
  - type: string
  - value: example: 'Configuration.ini'
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **configuration.ini-file** is located.
    - type: string
    - value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns configdic: Dictionary of model setup parameters distributed over several subdictionaries

Return type Dictionary

lowEBMs.Packages.Configuration.**dict_to_list**(*dic*)

Converts dictionaries returned from `Configuration.importer` into a list with the same structure. This allows calling the content by index not keyword. It works for a maximum of 3 dimensions of dictionaries (dictionary inside a dictionary).

Function-call arguments

Parameters **dic** (*dict*) – The dictionary to convert

Returns List with same structure as input dictionary

Return type List

lowEBMs.Packages.Configuration.**parameterimporter**(*filename*, **args*, ***kwargs*)

A function purpose-built to import 1-dimensional parameters for the sellers-type functions. The standard parameters (Sellers 1969) are written into a **.ini-file** and will be extracted to override the 0-dimensional parameters with 1-dimensional ones.

Important: This function is inbound into `Configuration.parameterinterpolater` or `Configuration.parameterinterpolaterstepwise` which interpolate the parameters to the gridresolution. **To import, interpolate and overwrite these parameter use** `Configuration.add_sellersparameters`.

Parameters which are imported 1-dimensionally:

- *b*: Empirical constant to estimate the albedo
- *Z*: Zonal mean altitude
- *Q*: Solar insolation
- *dp*: The tropospheric pressure depth
- *dz*: The average zonal ocean depth
- *Kh*: The thermal diffusivity of the atmospheric sensible heat term
- *Kwv*: The thermal diffusivity of the watervapour term
- *Ko*: The thermal diffusivity of the oceanic sensible heat term
- *a*: Empricial constant to calculate the meridional windspeed

The parameters are divided into two types, one defined on a latitudinal circle (gridlines) and one defined on a latitudinal belt (center point between two latitudinal circles/gridlines)

Note: The standard parameters from Sellers (1969) are already provided with this project in 'lowEBMs/Tutorials/Config/Data/'. By specifying no path (**path=None**) they can directly be used (advised since the parameters are structured in a special way).

Function-call arguments

Parameters

- **filename** (*string*) – The name of the **parameter.ini-file**
  - type: string
  - value: standard: 'SellersParameterization.ini'
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **parameter.ini-file** is located.
    - type: string
    - value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns circlecomb, beltcomb: List of parameters defined on a latitudinal circle, and latitudinal belt

Return type List, List

lowEBMs.Packages.Configuration.**parameterinterpolator**(*filename*, **args*, ***kwargs*)

An interpolation method fitting a polynomial of degree 10 to the parameter distributions. This creates parameter distributions suitable for the gridresolution (necessary if a higher resolution than 10° is used.

This function includes the function `Configuration.parameterimporter` and takes the same arguments.

Function-call arguments

Parameters

- **filename** (*string*) – The name of the **parameter.ini-file**
  - type: string
  - value: standard: 'SellersParameterization.ini'
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **parameter.ini-file** is located.
    - type: string
    - value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns newcircle, newbelt: List of interpolated parameters defined on a latitudinal circle, and latitudinal belt

Return type List, List

lowEBMs.Packages.Configuration.**parameterinterpolatorstepwise**(*filename*, **args*, ***kwargs*)

An interpolation method stepwise fitting and averaging a polynomial of degree 2 to the parameter distribution.

The interpolation method is more advanced compared to `Configuration.parameterinterpolator`. For each point (over the latitudes) a polynomial fit of degree 2 is made over the point plus the neighbouring points and estimates for the new gridresolution between these neighbouring points are stored. This is done for every point of the original parameters (except the endpoints). Because the interpolations overlap, the values are averaged to obtain a best estimate from multiple interpolations.

This function includes the function `Configuration.parameterimporter` and takes the same arguments.

Function-call arguments

Parameters

- **filename** (*string*) – The name of the **parameter.ini-file**
  - type: string
  - value: standard: 'SellersParameterization.ini'
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **parameter.ini-file** is located.
    - type: string
    - value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns newcircle, newbelt: List of interpolated parameters defined on a latitudinal circle, and latitudinal belt

Return type List, List

lowEBMs.Packages.Configuration.**add_sellersparameters**(*config*, *importer*, *file*, *transfernumber*, *downwardnumber*, *solar*, *albedo*, **args*, ***kwargs*)

Overwrites the model setup with one-dimensional sellers parameters. It takes a model configuration with 0D sellers parameters, the filename of new parameters and a method of interpolation.

This function uses either the method `Configuration.parameterinterpolator` or `Configuration.parameterinterpolatorstepwise` which both use the import function `Configuration.parameterimporter`, therefore it requires their attributes too.

Function-call arguments

Parameters

- **config** (*dict*) – The original config dictionary to overwrite
  - type: dictionary
  - value: created by `Configuration.importer`
- **importer** (*function*) – The name of the interpolator method
  - type: functionname
  - value: **parameterinterpolator** or **parameterinterpolatorstepwise**
- **file** (*string*) – The name of the **parameter.ini-file**
  - type: string
  - value: standard: 'SellersParameterization.ini'
- **transfernumber** (*integer*) – The [func] header-number in the **configuration.ini-file** which describes the transfer flux
  - type: integer
  - value: any
- **incomingnumber** (*integer*) – The [func] header-number in the **configuration.ini-file** which describes the downward flux
  - type: integer
  - value: any
- **solar** (*boolean*) – Indicates whether the insolation by Sellers is used
  - type: boolean
  - value: True / False
- **albedo** (*boolean*) – Indicates whether the albedo parameters by Sellers are used
  - type: boolean
  - value: True / False
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **parameter.ini-file** is located.

* type: string
* value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns configuration, parameters

Return type Dictionary, List

lowEBMs.Packages.Configuration.**import_parallelparameter**(*parallelconfig_filename*, **args*, ***kwargs*)

Imports information from a .ini-file to create a setup of parallelized simulations. This shall allow time-efficient creation of ensemble run, focused to run simulations with various parameters to gain best-fit parameters.

Function-call arguments

Parameters

- **parallelconfig_filename** (*string*) – The name of the **Parallelization.ini-file** for parallelization
  - type: string
  - value: standard: 'Parallelization.ini'
- **args** –
- **kwargs** – Optional Keyword arguments:
  - *path*: The directory path where the **Parallelization.ini-file** is located.
    * type: string
    * value: **full path** ('/home/user/dir0/dir1/filedir/') or **relative path** ('../../filedir/')

Returns raw parallelization setup

Return type Dictionary

lowEBMs.Packages.Configuration.**allocate_parallelparameter**(*parameter_raw*)

Transforms parameters for parallelization from tuple *[start,end]* to list *[start,...,end]* of length *number_of_cycles*. This shall create lists of parameters to be tested in parallelized simulations.

Function-call arguments

Parameters **parameter_raw** (*dict*) – A dictionary with parameters to allocate. The values of the parameters should have the form [start,end] to create *[start,...,end]* of length *number_of_cycles*

- type: dictionary
- value: as returned by **configuration.import_parallelparameter**

Returns allocated parallelization setup, parallelization information

Return type Dictionary, Dictionary

lowEBMs.Packages.Configuration.**write_parallelparameter**(*config*, *parameter*, *parametersetup*)

Overwrites the single run model setup with a parallelized model setup. This function uses the allocated parallelization setup as returned by `Configuration.allocate_parallelparameter`. Depending on the number of paramters **n** to parallelize, a **n-dimensional** matrix is created. This matrix will be transformed to a **one-dimensional** list by placing one row after another.

Function-call arguments

Parameters

- **config** (*dict*) – The original config dictionary to overwrite

- type: dictionary
- value: created by `Configuration.importer`

- **parameter** (*dict*) – A dictionary with allocated parameters for parallelization. First element returned by `Configuration.allocate_parallelparameter`.
  - type: dictionary
  - value: as returned by `Configuration.allocate_parallelparameter`
- **parametersetup** (*dict*) – A dictionary with information about parallelization. Second element returned by `Configuration.allocate_parallelparameter`.
  - type: dictionary
  - value: as returned by `Configuration.allocate_parallelparameter`

Returns config: Updated dictionary of model setup parameters for parallelization

Return type Dictionary

lowEBMs.Packages.Variables

Package which defines a large set of variables and functions to process them.

The variables defined are divided into three types:

*Running variables: they store information which is overwritten in each following iteration step *Static variables: they are non-changing system properties *Storage variables: these are lists filled with system properties during a model run

The centre piece of this package is the class `Variables.Vars`:

Vars	`Variables.Vars` defines any variable desired to store and access from another module's functions.

All variables defined in `Variable.Vars` can be read and written with:

```
from lowEBMs.Packages.Variable import Vars

Vars.x          #returns the current value of variable x in Vars
Vars.x = y      #variable x in Vars is permanently set to value y
```

Functions to process variables before a simulation run are, for single simulations

variable_importer	Executes all relevant functions to import variables for a single simulation run.
builtin_importer	Adds the most important variables to the python-builtin functions which are globally accessible.
initial_importer	Calculates the initial conditions of the *primary variables* from the `initials`-section.
output_importer	

and for parallelized ensemble simulations

`variable_importer_parallelized`

Continued on next page

Table 13 – continued from previous page

builtin_importer_parallelized
initial_importer_parallelized
output_importer_parallelized

Important: Variables.variable_importer and executes the in the list following processing functions which has to be executed before a simulation can be run for more information see How to use). For parallelized simulations this can be swapped to Variables.variable_importer_parallelized.

Functions to process variables during or after a simulation run are:

reset	Resets the given variable to the initial value specified in Vars.__init__.
datareset	Resets the *primary variables* to their initial values.

All modules defined in lowEBMs.Packages.Variables are:

Variables class

lowEBMs.Packages.Variables.**Vars**()

Variables.Vars defines any variable desired to store and access from another module's functions.

There are three different types of variables defined.

Running variables:

Static variables:

Lat	The latitudes of the gridpoints (or ZMT)
Lat2	The latitudes of the centres between gridpoints (or centered ZMT)
solar	The distribution of solar insolation
orbtable	The lookup-table for orbital parameters (from climlab)
area	The area of a latitudinal belt
bounds	The boundary latitudes used to calculate **Area**
latlength	The circumference of a latitudinal circle
External_time_start	The simulation time when the external forcing sets in
CO2_time_start	The simulation time when the CO2 forcing sets in
start_time	The real clock time when the simulation was started

Storage variables:

cL	The sellers watervapour energy transfer
C	The sellers atmospheric sensible heat energy transfer
F	The sellers oceanic sensible heat energy transfer
P	The total energy transfer , P=cL+C+F (non-weighted, one direction)
Transfer	The total sellers energy transfer for a latitudinal belt
BudTransfer	The budyko energy transfer for a latitudinal belt
alpha	The alpha value distribution
Rdown	The downward radiative energy flux
Rup	The upward radiative energy flux
ExternalOutput	List of radiative forcings
CO2Output	The CO2 radiative forcing
ExternalInput	List of the raw input to calculate the radiative forcing
CO2Input	The raw CO2 input

Single Simulation Functions

`variable_importer`
`builtin_importer`
`initial_importer`
`output_importer`

`lowEBMs.Packages.Variables.`**`variable_importer`**(*config*, *initialZMT=True*, *control=False*, *parallel=False*, *parallel_config=0*, *accuracy=0.001*, *accuracy_number=1000*)

Executes all relevant functions to import variables for a single simulation run. From the *configuration* dictionary, returned by `Configuration.importer`, the relevant information is extracted and the specific importer functions are executed in the following order:

$$buliltin\_importer \quad \rightarrow \quad initial\_importer \quad \rightarrow \quad output\_importer$$

Note: When doing this manually, maintain the order!

Function-call arguments

Parameters **`config`** (*dict*) – The configuration dictionary returned by `Configuration.importer`

Returns No return

`lowEBMs.Packages.Variables.`**`builtin_importer`**(*rk4input*, *control=False*, *parallel=False*, *parallel_config=0*, *accuracy=0.001*, *accuracy_number=1000*)

Adds the most important variables to the python-builtin functions which are globally accessible. This enables calling and writing variables globally and across different files.

Variables added to the builtin-functions are all arguments of the `[rk4input]`-section from the *configuration* dictionary, returned by `Configuration.importer`, and three additional ones.

Important: Variables from the `[rk4input]`-section are added with their key given in the *configuration.ini-file* and can be called by the same one later.

Here all added variables ([rk4input]-variables + additional ones):

Function-call arguments

Parameters **rk4input** (*dict*) – The [rk4input]-section from the configuration dictionary returned by Configuration.importer

Returns No return

lowEBMs.Packages.Variables.**initial_importer**(*initials*, *initialZMT=True*, *control=False*, *parallel=False*)

Calculates the initial conditions of the *primary variables* from the initials-section.

The initial conditions are directly written to their entry in Variable.Vars.

Function-call arguments

Parameters **initials** (*dict*) – The [initials]-section from the configuration dictionary returned by Configuration.importer

Returns No return

lowEBMs.Packages.Variables.**output_importer**(*functiondict*)

Ensemble Simulation Functions

variable_importer_parallelized
builtin_importer_parallelized
initial_importer_parallelized
output_importer_parallelized

Reset Functions

reset
datareset

lowEBMs.Packages.Variables.**reset**(*x*)

Resets the given variable to the initial value specified in Vars.__init__.

Function-call arguments

Parameters **x** (*float/list*) – The variable which shall be reset to the initial value

Returns No return

lowEBMs.Packages.Variables.**datareset**()

Resets the *primary variables* to their initial values. The *primary variables* are variables defined under the‘‘[initials]‘‘-section in the *configuration.ini-file*. These are:

1.8 To-Do

1.9 References

- IPCC. Climate Change 2013 - The Physical Science Basis. Cambridge University Press, Cambridge, 2013. ISBN 9781107415324. doi: 10.1017/CBO9781107415324. URL http://ebooks.cambridge.org/ref/id/CBO9781107415324.
- M. I. Budyko, G. Observatory, and M. Spasskaja. The effect of solar radiation variations on the climate of the Earth. Tellus XXI (1969), 7, 1968. doi: 10.1111/j.2153-3490.1969.tb00466.x
- W. D. Sellers. A Global Climatic Model Based on the Energy Balance of the Earth-Atmosphere System. Journal of Applied Meteorology, 8(3):392–400, 1969. ISSN 0021-8952. doi: 10.1175/1520-0450(1969)008<0392:AGCMBO>2.0.CO;2.
- C. D. Keeling, S. C. Piper, R. B. Bacastow, M. Wahlen, T. P. Whorf, M. Heimann, and H. A. Meijer, Exchanges of atmospheric CO2 and 13CO2 with the terrestrial biosphere and oceans from 1978 to 2000. I. Global aspects, SIO Reference Series, No. 01-06, Scripps Institution of Oceanography, San Diego, 88 pages, 2001.
- G. R. North and K. Kwang-Yul: Energy Balance Climate Models. Wiley-VCH Verlag GmbH & Co. KGaA, 2017.
- G. Myhre, E. J. Highwood, K. P. Shine and Frode Stordal: New estimates of radiative forcing due to well mixed greenhouse gases. Geophysical Research Letters, 25(14):2715–2718, 1998.
- M. Toohey, B. Stevens, H. Schmidt and C. Timmreck: Easy Volcanic Aerosol (EVA v1.0): an idealized forcing generator for climate simulations. Geosci. Model Dev., 2016. doi: 10.5194/gmd-9-4049-2016

1.10 Contact

Benjamin Schmiedel: benny.schmiedel@gmail.com

Python Module Index

l

Index

O

P

R

S

T

V

W