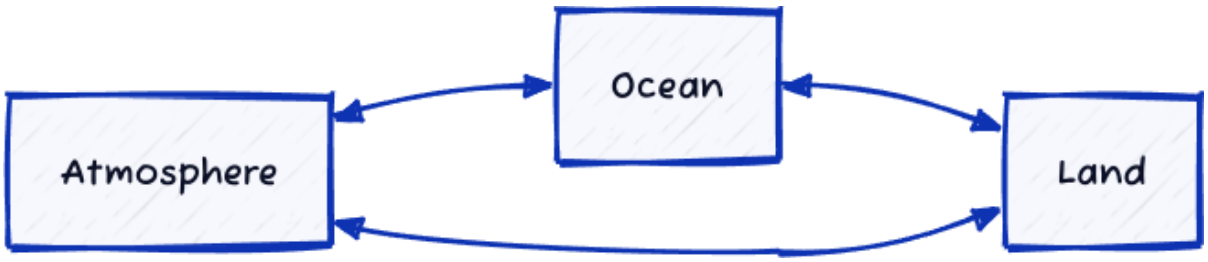
esbmtk Documentation

Release 0.12.0.25.post1.dev28+g1fd8a95

uliw

May 15, 2024

CONTENTS



The Earth Science Box Modeling Toolkit (ESBMTK) is a python library that provides a an object oriented approach to development of Harvardton-Bear type models. The ESBMTK classes allow to describe models in a declarative way where the model definition serves also as the model documentation.

ESBMTK provides abstractions for a variety of processes, e.g., gas-exchange across the air-sea interface, or marine carbonate chemistry and isotope calculations. It's modular nature allows to easily extend or change existing models.

Originally envisioned as a teaching tool, it is currently being used in several research projects. The library is under constant development, but the basic API is stable.

CHAPTER

ONE

CONTENTS

1.1 Introduction

1.1.1 Installation

Conda

Assuming you install into a new virtual environment the following should install the ESBMTK framework

```
conda create --name foo
conda activate foo
conda install esbmtk
```

To access the examples, please take a look at https://github.com/uliw/esbmtk

pip & github

If you work with pip, simply install with `python -m pip install esbmtk`, or download the code from https://github.com/uliw/esbmtk

1.1.2 A simple example

A simple model of the marine P-cycle would consider the delivery of P from weathering, the burial of P in the sediments, the thermohaline transport of dissolved PO_4as well as the export of P in the form of sinking organic matter (POP). The concentration in the respective surface and deep water boxes is then the sum of the respective fluxes (see Fig. 1). The model parameters are taken from Glover 2011, Modeling Methods in the Marine Sciences.

If we define equations that control the export of particulate P (F_{POP}) as a fraction of the upwelling P (F_u), and the burial of P (F_b) as a fraction of (F_{POP}), we express this model as coupled ordinary differential equations (ODE, or initial value problem):

$$\frac{d[PO_4]_S}{dt} = \frac{F_w + F_u - F_d - F_{POP}}{V_S}$$

and for the deep ocean,

$$\frac{d[PO_4]_D}{dt} = \frac{F_{POP} + F_d - F_u - F_b}{V_D}$$

which is easily encoded as a Python function

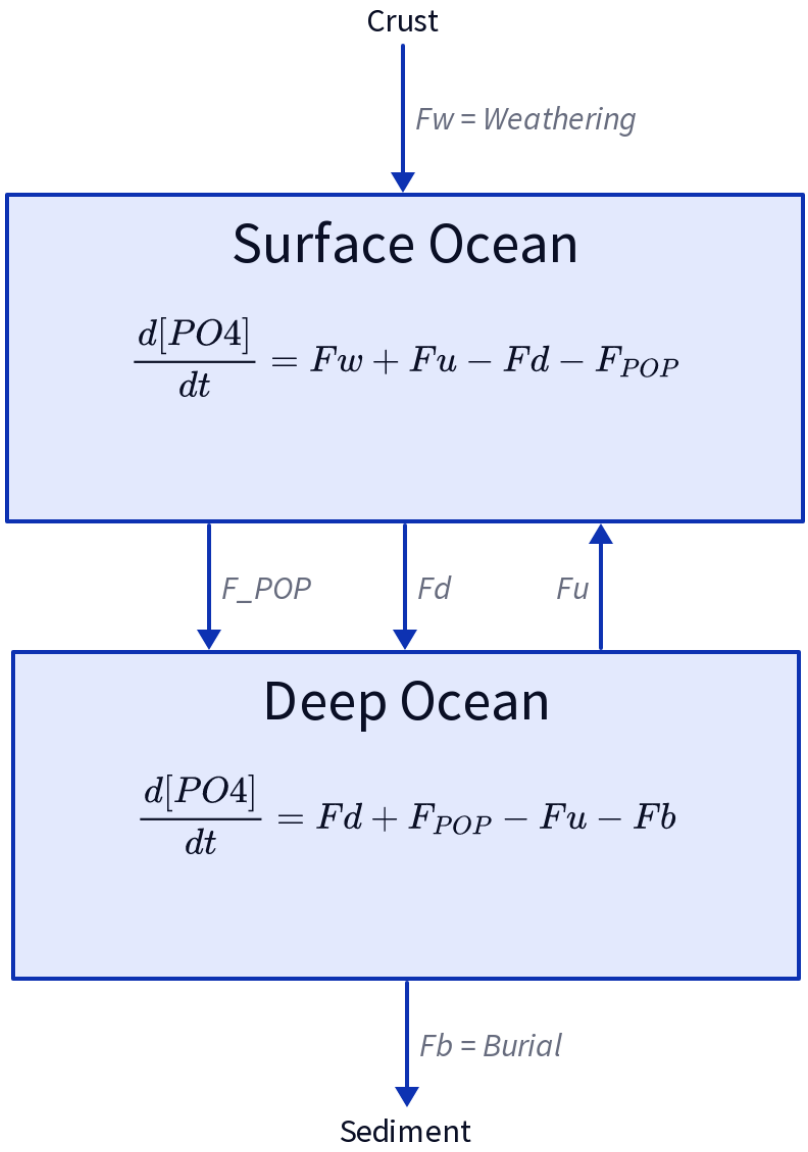


Fig. 1: A two-box model of the marine P-cycle. F_w= weathering F_u= upwelling, F_d= downwelling, F_{POP}= particulate organic phosphor, F_b= burial.

```
def dCdt(t, C_0, V, F_w, thx):
    """ Calculate the change in concentration as
    a function of time. After Glover 2011, Modeling
    Methods for Marine Science.

    :param C: list of initial concentrations mol/m*3
    :param time: array of time points
    :params V: lits of surface and deep ocean volume [m^3]
    :param F_w: River (weathering) flux of PO4 mol/s
    :param thx: thermohaline circulation in m*3/s
    :returns dCdt: list of concentration changes mol/s
    """

    C_S = C_0[0]  # surface
    C_D = C_0[1]  # deep
    F_d = C_S * thx  # downwelling
    F_u = C_D * thx  # upwelling
    tau = 100 # residence time of P in surface waters [yrs]
    F_POP = C_S * V[0] / tau  # export production
    F_b = F_POP / 100  # burial

    dCdt[0] = (F_w + F_u - F_d - F_POP) / V[0]
    dCdt[1] = (F_d + F_POP - F_u - F_b) / V[1]

    return dCdt
```

1.1.3 Implementing the P-cycle with ESBMTK

While ESBMTK provides abstractions to efficiently define complex models, the following section will use the basic ESBMTK classes to define the above model. While quite verbose, it demonstrates the design philosophy behind ESBMTK. More complex approaches are described further down.

Foundational Concepts

ESBMTK uses a hierarchically structured object-oriented approach to describe a model.
The topmost object is the model object that describes fundamental properties like run time, time step, elements and species information. All other objects derive from the model object. Reservoir objects define properties like volume or geometry, pressure and temperature, whereas species objects store initial conditions and concentration versus time data. Species Property objects store names and labels, and Element Property objects store e.g., isotopic reference ratios etc.

```
Model
    ├── Reservoir_1
    │   ├── Species_1
    │   │   └── SpeciesProperties
    │   │       └── ElementProperties
    │   └── Species_2
    │       └── SpeciesProperties
    │           └── ElementProperties
    └── Reservoir_2
        ├── Species_1
```

(continues on next page)

(continued from previous page)

```
        │   └── SpeciesProperties
        │       └── ElementProperties
        └── Species_2
            └── SpeciesProperties
                └── ElementProperties
```

The relationship between two reservoirs is specified by a connection properties object that specifies which reservoir is the upstream source, and which is the downstream sink. It also specifies the type of connection, e.g., to scale the flux between from upstream to downstream by the respective species concentrations.

```
Model
   └── ConnectionProperties
       ├── Species2Species_1
       │   ├── Sink
       │   │   └── Reservoir
       │   │       └── Species_1
       │   ├── Source
       │   │   └── Reservoir
       │   │       └── Species_1
       │   └── Type
       │       └── ProcessProperties
       └── Species2Species_2
           ├── Sink
           │   └── Reservoir
           │       └── Species_2
           ├── Source
           │   └── Reservoir
           │       └── Species_2
           └── Type
               └── ProcessProperties
```

The model geometry is then parsed to build a suitable equation system which is passed to an ODE solver library which returns the results once integration has finished. Since Python objects are persistent, the object hierarchy is open to introspection using the regular Python syntax.

Defining the model geometry and initial conditions

The below code examples are available at https://github.com/uliw/esbmtk-examples In the first step, one needs to define a model object that describes fundamental model parameters. The following code first loads the following ESBMTK classes that will help with model construction:

- *esbmtk.esbmtk.Model()*
- esbmtk.esbmtk.Reservoir()
- *esbmtk.connections.ConnectionProperties()* class
- esbmtk.esbmtk.SourceProperties() class
- esbmtk.esbmtk.SinkProperties() class
- and Q_ which belongs to the pint library.

```
# import classes from the esbmtk library
from esbmtk import (
```

(continues on next page)

(continued from previous page)

```
    Model,  # the model class
    Reservoir,  # the reservoir class
    ConnectionProperties,  # the connection class
    SourceProperties,  # the source class
    SinkProperties,  # sink class
    Q_,  # Quantity operator
)
```

Next we use the `Model` class to create a model instance that defines basic model properties. Note that units are automatically translated into model units. While convenient, there are some important caveats: Internally, the model uses 'year' as the time unit, mol as the mass unit, and liter as the volume unit. You can change this by setting these values to e.g., 'mol' and 'kg', however, some functions assume that their input values are in 'mol/l' rather than mol/m**3 or 'kg/s'. Ideally, this would be caught by ESBMTK, but at present, this is not guaranteed. So your mileage may vary if you fiddle with these settings. Note: Using mol/kg e.g., for seawater, will be discussed below.

```
# define the basic model parameters
M = Model(
    stop="3 Myr",  # end time of model
    timestep="1 kyr",  # upper limit of time step
    element=["Phosphor"],  # list of element definitions
)
```

Next, we need to declare some boundary conditions. Most ESBMTK classes will be able to accept input in the form of strings that also contain units (e.g., `"30 Gmol/a"`). Internally these strings are parsed and converted into the model base units. This works most of the time, but not always. In the below example, we define the residence time τ. This variable is then used as input to calculate the scale for the primary production as `M.S_b.volume / tau` which must fail since `M.S_b.volume` is a numeric value and `tau` is a string.

```
# try the following
tau = "100 years"
tau * 12
```

To avoid this we have to manually parse the string into a quantity. This is done with the quantity operator `Q_` Note that `Q_` is not part of ESBMTk but imported from the `pint` library.

```
# now try this
from esbmtk import Q_
tau = Q_("100 years")
tau * 12
```

Most ESBMTK classes accept quantities, strings that represent quantities as well as numerical values. Weathering and burial fluxes are often defined in `mol/year`, whereas ocean models use `kg/year`. ESBMTK provides a method (`set_flux()`) that will automatically convert the input into the correct units. In this example, it is not necessary since the flux and the model both use `mol`. It is however good practice to rely on the automatic conversion. Note that it makes a difference for the mol to kilogram conversion whether one uses `M.P` or `M.PO4` as the reference species!

```
# boundary conditions
F_w =  M.set_flux("45 Gmol", "year", M.P) # P @280 ppm (Filipelli 2002)
tau = Q_("100 year")  # PO4 residence time in surface boxq
F_b = 0.01  # About 1% of the exported P is buried in the deep ocean
thc = "20*Sv"  # Thermohaline circulation in Sverdrup
```

To set up the model geometry, we first use the *esbmtk.esbmtk.Source()* and *esbmtk.esbmtk.Species()* classes to create a source for the weathering flux, a sink for the burial flux, and instances of the surface and deep ocean boxes.

Since we loaded the element definitions for phosphor in the model definition above, we can directly refer to the "PO4" species in the reservoir definition.

```
# Source definitions
SourceProperties(
    name="weathering",
    species=[M.PO4],
)
SinkProperties(
    name="burial",
    species=[M.PO4],
)

# reservoir definitions
Reservoir(
    name="S_b",  # box name
    volume="3E16 m**3",  # surface box volume
    concentration={M.PO4: "0 umol/l"},  # initial concentration
)
Reservoir(
    name="D_b",  # box name
    volume="100E16 m**3",  # deeb box volume
    concentration={M.PO4: "0 umol/l"},  # initial concentration
)
```

Model processes

For many models, processes can mapped as the transfer of mass from one box to the next. Within the ESBMTK framework, this is accomplished through the *esbmtk.connections.Species2Species()* class. To connect the weathering flux from the source object (M.w) to the surface ocean ($M.S_b$) we declare a connection instance describing this relationship as follows:

```
ConnectionProperties(
    source=M.weathering,  # source of flux
    sink=M.S_b,  # target of flux
    rate=F_w,  # rate of flux
    id="river",  # connection id
    ctype="regular",
)
```

Unless the `register` keyword is given, connections will be automatically registered with the parent of the source, i.e., the model `M`. Unless explicitly given through the `name` keyword, connection names will be automatically constructed from the names of the source and sink instances. However, it is a good habit to provide the `id` keyword to keep connections separate in cases where two reservoir instances share more than one connection. The list of all connection instances can be obtained from the model object (see below).

To map the process of thermohaline circulation, we connect the surface and deep ocean boxes using a connection type that scales the mass transfer as a function of the concentration in a given reservoir (`ctype ="scale_with_concentration"`). The concentration data is taken from the reference reservoir which defaults to the source reservoir. As such, in most cases, the `ref_reservoirs` keyword can be omitted. The `scale` keyword can be a string or a numerical value. If it is provided as a string ESBMTK will map the value into model units. Note that the connection class does not require the `name` keyword. Rather the name is derived from the source and sink reservoir instances. Since reservoir instances can have more than one connection (i.e., surface to deep via downwelling, and surface to deep via primary production), it is required to set the `id` keyword.

```
ConnectionProperties(  # thermohaline downwelling
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="downwelling_PO4",
)
ConnectionProperties(  # thermohaline upwelling
    source=M.D_b,  # source of flux
    sink=M.S_b,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="upwelling_PO4",
)
```

There are several ways to define biological export production, e.g., as a function of the upwelling PO_4, or as a function of the residence time of PO_4in the surface ocean. Here we follow Glover (2011) and use the residence time τ = 100 years. Note that the below code species explicitly specifies the species that is affected by this process.

```
ConnectionProperties(  #
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
    ctype="scale_with_concentration",
    scale=M.S_b.volume / tau,
    id="primary_production",
    species=[M.PO4],  # apply this only to PO4
)
```

We require one more connection to describe the burial of P in the sediment. We describe this flux as a fraction of the primary export productivity. To create the connection we can either recalculate the export productivity or use the previously calculated flux. We can query the export productivity using the `id_string` of the above connection with the *esbmtk.esbmtk.Model.flux_summary()* method of the model instance:

```
M.flux_summary(filter_by="primary_production", return_list=True)[0]
```

The `flux_summary()` method will return a list of matching fluxes but since there is only one match, we can simply use the first result, and use it to define the phosphor burial as a consequence of export production in the following way:

```
ConnectionProperties(  #
    source=M.D_b,  # source of flux
    sink=M.burial,  # target of flux
    ctype="scale_with_flux",
    ref_flux=M.flux_summary(filter_by="primary_production",return_list=True)[0],
    scale=F_b,
    id="burial",
    species=[M.PO4],
)
```

Running the above code (see the file `po4_1.py` at https://github.com/uliw/ESBMTK-Examples) and results in the following graph:

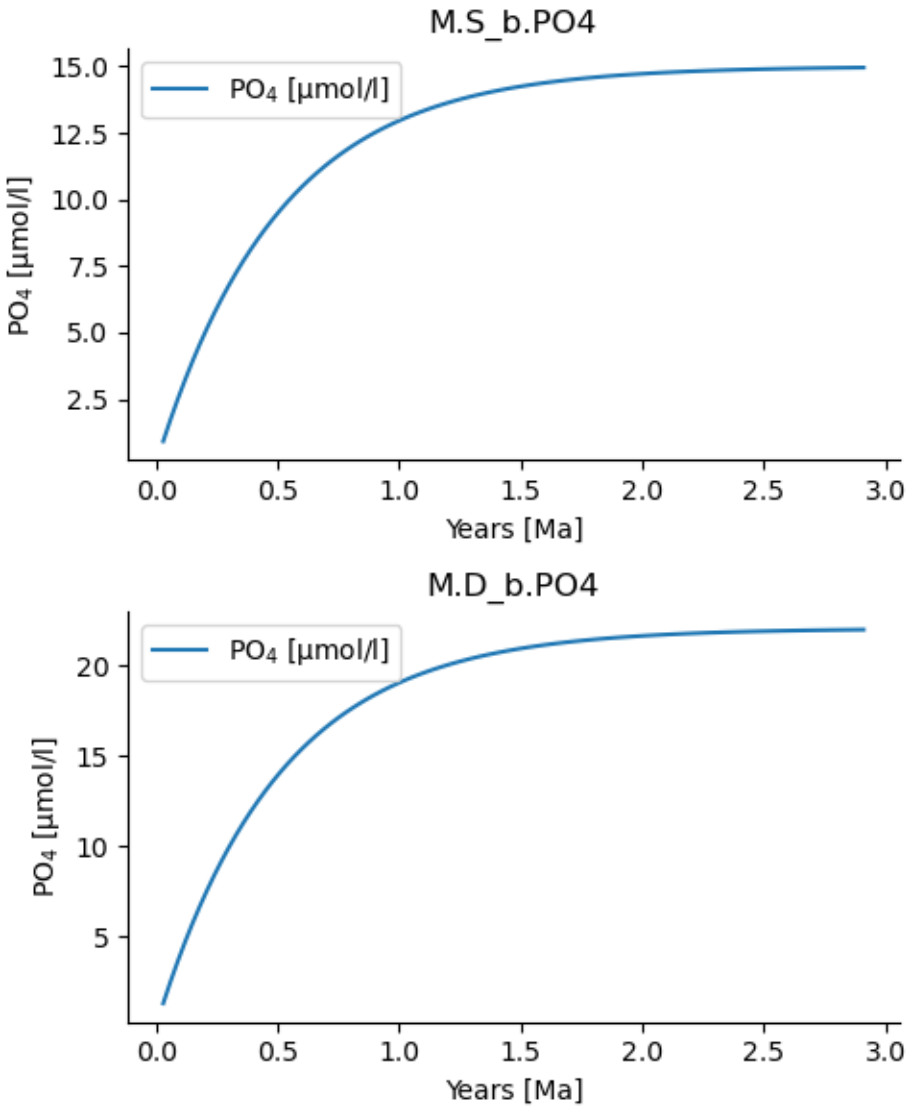


Fig. 2: Example output from `po4_1.png`

1.1.4 Working with the model instance

Running the model, visualizing and saving the results

To run the model, use the `run()` method of the model instance, and plot the results with the `plot()` method. This method accepts a list of ESBMTK instances, that will be plotted in a common window. Without further arguments, the plot will also be saved as a pdf file where `filename` defaults to the name of the model instance. The `save_data()` method will create (or recreate) the `data` directory which will then be populated by csv-files.

```
M.run()
M.plot([M.S_b.PO4, M.D_b.PO4], fn="po4_1.png")
M.save_data()
```

Saving/restoring the model state

Many models require a spin-up phase. Once the model is in equilibrium, you can save the save the state with the `save_state()` method.

```
M.run()
M.save_state()
```

Restarting the model from a saved state requires that you first initialize the model geometry (i.e., declare all the connections etc), and then read the previously saved model state.

```
....
....
M.read_state()
M.run()
```

Towards this end, note that a repeated model run will not be initialized from the last known state, but rather starts from a blank state.

```
.....
.....
M.run()
```

To restart a model from the last known state, the above would need to be written as

```
.....
.....
M.run()
M.save_state()
M.read_state()
M.run()
```

Introspection and data access

All ESBMTK instances and instance methods support the usual python methods to show the documentation, and inspect object properties.

```
help(M.S_b)  # will print the documentation for sb
dir(M.S_b)  # will print all methods for sb
M.S_b #  when issued in an interactive session, this will echo
# the arguments used to create the instance
```

The concentration data for a given reservoir is stored in the following instance variables:

```
M.S_b.c  # concentration
M.S_b.m  # mass
M.S_b.v  # volume
M.S_b.d  # delta value (if used by model)
M.S_b.l  # the concentration of the light isotope (if used)
```

The model time axis is available as `M.time` and the model supports the `connection_summary()` and `flux_summary` methods to query the respective `connection` and `flux` objects.

1.2 Adding Complexity

1.2.1 Model forcing

ESBMTK realizes model forcing through the *esbmtk.extended_classes.Signal()* class. Once defined, a signal instance can be associated with a *esbmtk.connections.Species2Species()* instance that will then act on the associated connection. This class provides the following keywords to create a signal:

- `square()`, `pyramid()`, `bell()` These are defined by specifying the signal start time (relative to the model time), its size (as mass) and duration, or as duration and magnitude (see the example below)
- `filename()` a string pointing to a CSV file that specifies the following columns: `Time [yr]`, `Rate/Scale [units]`, `delta value [dimensionless]` The class will attempt to convert the data into the correct model units. This process is however not very robust.

The default is to add the signal to a given connection. It is however also possible to use the signal data as a scaling factor. Signals are cumulative, i.e., complex signals are created by adding one signal to another (i.e., Snew = S1 + S2). Using the P-cycle model from the previous chapter (see `po4_1.py`) we can add a signal by first defining a signal instance, and then associating the instance with a weathering connection instance (this model is available as `po4_2.p4` see https://github.com/uliw/ESBMTK-Examples)

```
from esbmtk import Signal

Signal(
    name="CR",  # Signal name
    species=M.PO4,  # SpeciesProperties
    start="1 Myrs",
    shape="pyramid",
    duration="1 Myrs",
    mass="45 Pmol",
)
```

(continues on next page)

(continued from previous page)

```
ConnectionProperties(
    source=M.weathering,  # source of flux
    sink=M.S_b,  # target of flux
    rate=F_w,  # rate of flux
    id="river",  # connection id
    signal=M.CR,
    species=[M.PO4],
    ctype="regular",
)
M.run()
M.plot([M.S_b.PO4, M.D_b.PO4, M.CR], fn="po4_2.png")
M.save_data()
```

This will result in the following output:

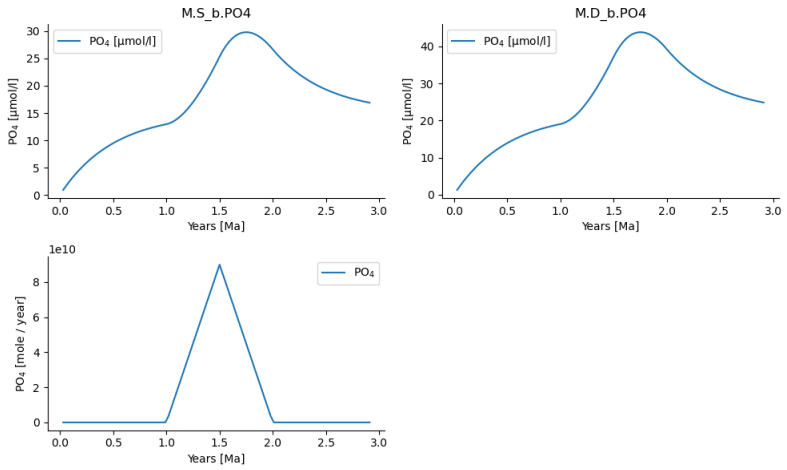


Fig. 3: Example output for the CR signal above. See po4_2.py in the examples directory.

1.2.2 Working with multiple species

The basic building blocks introduced so far, are sufficient to create a single species model. Adding further species, is straightforward. First one needs to import the species definitions. They than can be simply used by extending the dictionaries and lists used in the previous example. Using the previous example of a simple P-cycle model, we now express the P-cycling as a function of photosynthetic organic matter (OM) production and remineralization. First, we import the new classes and we additionally load the species definitions for carbon (this code is available as po4_3.p4 see https://github.com/uliw/ESBMTK-Examples).

```
from esbmtk import (
    Model,
    Reservoir,  # the reservoir class
    ConnectionProperties,  # the connection class
    SourceProperties,  # the source class
    SinkProperties,  # sink class
    data_summaries,
    Q_,
)
```

(continues on next page)

(continued from previous page)

```
M = Model(
    stop="6 Myr",  # end time of model
    timestep="1 kyr",  # upper limit of time step
    element=["Phosphor", "Carbon"],  # list of species definitions
)

# boundary conditions
F_w_PO4 =  M.set_flux("45 Gmol", "year", M.PO4) # P @280 ppm (Filipelli 2002)
tau = Q_("100 year")  # PO4 residence time in surface boxq
F_b = 0.01  # About 1% of the exported P is buried in the deep ocean
thc = "20*Sv"  # Thermohaline circulation in Sverdrup
Redfield = 106 # C:P

SourceProperties(
    name="weathering",
    species=[M.PO4, M.DIC],
)
SinkProperties(
    name="burial",
    species=[M.PO4, M.DIC],
)
Reservoir(
    name="S_b",
    volume="3E16 m**3",  # surface box volume
    concentration={M.DIC: "0 umol/l", M.PO4: "0 umol/l"},
)
Reservoir(
    name="D_b",
    volume="100E16 m**3",  # deeb box volume
    concentration={M.DIC: "0 umol/l", M.PO4: "0 umol/l"},
)
```

The `esbmtk.connections.ConnectionProperties.()` class definition is equally straightforward, and the following expression will apply the thermohaline downwelling to all species in the `M.S_b` group.

```
ConnectionProperties(  # thermohaline downwelling
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="thc_up",
)
ConnectionProperties(  # thermohaline upwelling
    source=M.D_b,  # source of flux
    sink=M.S_b,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="thc_down",
)
```

It is also possible, to specify individual rates or scales using a dictionary, as in this example that sets two different weathering fluxes:

```
ConnectionProperties(
    source=M.weathering,  # source of flux
    sink=M.S_b,  # target of flux
    rate={M.DIC: F_w_PO4 * Redfield, M.PO4: F_w_PO4},  # rate of flux
    ctype="regular",
    id="weathering",  # connection id
)
```

The following code defines primary production and its effects on DIC in the surface and deep box. The example is a bit contrived but demonstrates the principle. Note the use of the `ref_reservoirs` keyword and `Redfield` ratio

```
# P-uptake by photosynthesis
ConnectionProperties(  #
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
    ctype="scale_with_concentration",
    scale=M.S_b.volume / tau,
    id="primary_production",
    species=[M.PO4],  # apply this only to PO4
)
# OM Primary production as a function of P-concentration
ConnectionProperties(  #
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
    ref_reservoirs=M.S_b.PO4,
    ctype="scale_with_concentration",
    scale=Redfield * M.S_b.volume / tau,
    species=[M.DIC],
    id="OM_production",
)
# P burial
ConnectionProperties(  #
    source=M.D_b,  # source of flux
    sink=M.burial,  # target of flux
    ctype="scale_with_flux",
    ref_flux=M.flux_summary(filter_by="primary_production",return_list=True)[0],
    scale={M.PO4: F_b, M.DIC: F_b * Redfield},
    id="burial",
)
```

One can now proceed to define the particulate phosphate transport as a function of organic matter export

```
M.run()
pl = data_summaries(
    M,  # model instance
    [M.DIC, M.PO4],  # SpeciesProperties list
    [M.S_b, M.D_b],  # Reservoir list
)
M.plot(pl, fn="po4_3.png")
```

which results in the below plot. The full code is available in the examples directory as `po4_2.py`

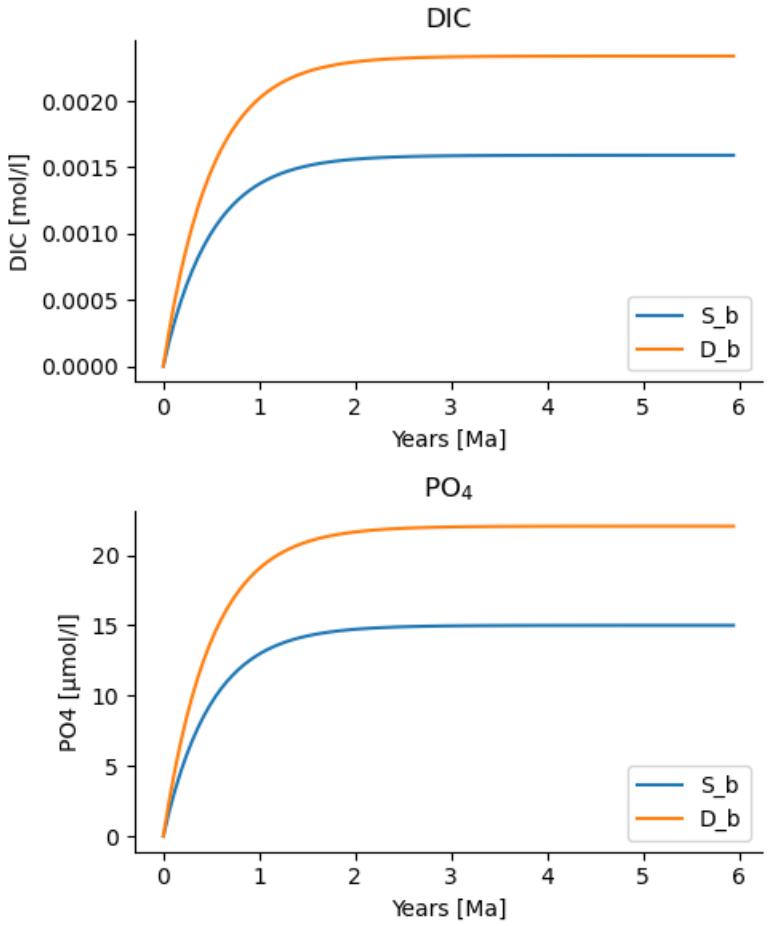


Fig. 4: Output of `po4_3.py` demonstrating the use of the `data_summaries()` function

1.2.3 Adding isotopes

Let's assume that the weathering flux of carbon has $\delta^{13}C$ value of 0 mUr, that photosynthesis fractionates by -28 mUr, and that organic matter burial does not fractionate . These changes require the following changes to the previous model code (the full code is available in the examples directory as `po4_4` at https://github.com/uliw/ESBMTK-Examples):

1. Isotope ratios require non-zero concentrations to avoid a division by zero,
2. You need to specify the initial isotope ratio for each reservoir
3. Sources and Sinks require a flag for each Species that uses isotopes
4. You need to indicate for each reservoir that `DIC` requires isotope calculations
5. You need to specify the isotope ratio of the weathering flux
6. You need to specify the fractionation factor during photosynthesis
7. You need to specify the fractionation factor during burial

```
SourceProperties(
    name="weathering",
    species=[M.PO4, M.DIC],
    isotopes={M.DIC: True},
)
SinkProperties(
    name="burial",
    species=[M.PO4, M.DIC],
    isotopes={M.DIC: True},
)
Reservoir(
    name="S_b",
    volume="3E16 m**3",  # surface box volume
    concentration={M.DIC: "2 umol/l", M.PO4: "0 umol/l"},
    isotopes={M.DIC: True},
    delta={M.DIC: 0},
)
Reservoir(
    name="D_b",
    volume="100E16 m**3",  # deeb box volume
    concentration={M.DIC: "2 umol/l", M.PO4: "0 umol/l"},
    isotopes={M.DIC: True},
    delta={M.DIC: 0},
)
# 4 weathering flux
ConnectionProperties(
    source=M.weathering,  # source of flux
    sink=M.S_b,  # target of flux
    rate={M.DIC: F_w_PO4 * Redfield, M.PO4: F_w_PO4},  # rate of flux
    ctype="regular",
    id="weathering",  # connection id
    delta={M.DIC: 0},
)
# 5 photosynthesis
ConnectionProperties(  #
    source=M.S_b,  # source of flux
    sink=M.D_b,  # target of flux
```

(continues on next page)

(continued from previous page)

```
    ref_reservoirs=M.S_b.PO4,
    ctype="scale_with_concentration",
    scale=Redfield * M.S_b.volume / tau,
    species=[M.DIC],
    id="OM_production",
    alpha=-28,  # mUr
)
# Burial
ConnectionProperties(  #
    source=M.D_b,  # source of flux
    sink=M.burial,  # target of flux
    ctype="scale_with_flux",
    ref_flux=M.flux_summary(filter_by="primary_production",return_list=True)[0],
    scale={M.PO4: F_b, M.DIC: F_b * Redfield},
    id="burial",
    alpha={M.DIC: 0},
)
```

Running the previous model with these additional 7 lines, results in the following graph. Note that the run-time has been reduced to 500 years so that the graph does not just show the steady state and that the P-data is not shown.

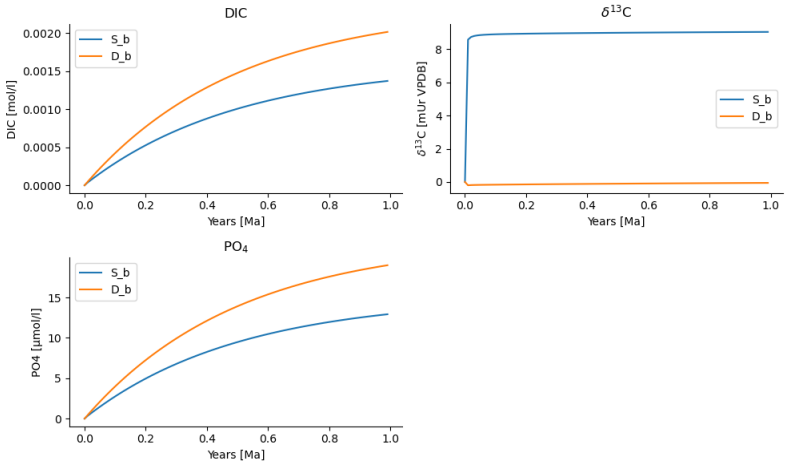


Fig. 5: Output of `po4_4.py` Note that the run-time has been reduced to 1000 years, so that the graph does not just show the steady state. The upper box shows the gradual increase in DIC concentrations and the lower shows the corresponding isotope ratios. The system will achieve isotopic equilibrium within approximately 2000 years.

1.2.4 Using many boxes

Using the ESBMTK classes introduced so far is sufficient to build complex models. However, it is easy to leverage Python syntax to create a few utility functions that help in reducing overly verbose code. The ESBMTK library comes with a few routines that help in this regard. However, they are not part of the core API, are not (yet) well documented and have not seen much testing. The following provides a brief introduction, but it may be useful to study the code for the Boudreau 2010 and LOSCAR-type models in the example directory. All of these make heavy use of the Python dictionary class.

For this function to work correctly, box names need to be specified following this template `Area_depth`, e.g., `A_sb` for the Atlantic surface water box, or `A_ib` for the Atlantic intermediate water box. The actual names, do not matter, but the

underscore is used to differentiate between ocean area and depth interval. The following code uses two dictionaries to specify the species and initial conditions for a multi-box model. Both dictionaries are then used as input for a function that creates the actual instances. Note that the meaning and syntax for the geometry list and seawater parameters are explained in the next chapter.

```
# ud = upper depth datum, ld = lower depth datum, ap = area percentage
# T = Temperature (C), P = Pressure (bar), S = Salinity in PSU
"""
box_parameters = {  # name: [[ud, ld ap], T, P, S]
        # Atlantic Ocean
        "M.A_sb": {"g": [0, -100, A_ap], "T": 20, "P": 5, "S": 34.7},
        "M.A_ib": {"g": [-100, -1000, A_ap], "T": 10, "P": 100, "S": 34.7},
        "M.A_db": {"g": [-1000, -6000, A_ap], "T": 2, "P": 240, "S": 34.7},
        # Indian Ocean
        "I_sb": {"g": [0, -100, I_ap], "T": 20, "P": 5},
        "I_ib": {"g": [-100, -1000, I_ap], "T": 10, "P": 100, "S": 34.7},
        "I_db": {"g": [-1000, -6000, I_ap], "T": 2, "P": 240, "S": 34.7},
        # Pacific Ocean
        "P_sb": {"g": [0, -100, P_ap], "T": 20, "P": 5, "S": 34.7},
        "P_ib": {"g": [-100, -1000, P_ap], "T": 10, "P": 100, "S": 34.7},
        "P_db": {"g": [-1000, -6000, P_ap], "T": 2, "P": 240, "S": 34.7},
        # High latitude box
        "H_sb": {"g": [0, -250, H_ap], "T": 2, "P": 10, "S": 34.7},
        # Weathering sources
        "Fw": {"ty": "Source", "sp": [M.DIC, M.TA, M.PO4]},
        # Burial Sinks
        "Fb": {"ty": "Sink", "sp": [M.DIC, M.TA, M.PO4]},
    }

initial_conditions= {
        # species: [concentration, Isotopes, delta value]
        M.PO4: [Q_("2.1 * umol/kg") * 1.024, False, 0],
        M.DIC: [Q_("2.21 mmol/kg") * 1.024, True, 2],
        M.TA: [Q_("2.31 mmol/kg") * 1.024, False, 0],
        M.O2: [Q_("200 umol/kg") * 1.024, False, 0],
    }

create_reservoirs(box_names, initial_conditions, M)
```

similarly, we can leverage Python dictionaries to set up the transport matrix. The dictionary key must use the following template: `boxname_to_boxname@id` where the `id` is used similarly to the connection id in the `Species2Species` and `ConnectionProperties` classes. So to specify thermohaline upwelling from the Atlantic deep water to the Atlantic intermediate water you would use `A_db_to_A_ib@thc` as the dictionary key, followed by the rate. The following examples define the thermohaline transport in a LOSCAR-type model:

```
# Conveyor belt
thc = Q_("20*Sv")
ta = 0.2  # upwelling coefficient Atlantic ocean
ti = 0.2  # upwelling coefficient Indian ocean

# Specify the mixing and upwelling terms as dictionary
thx_dict = {  # Conveyor belt
    "H_sb_to_A_db@thc": thc * M.H_sb.swc.density / 1e3,
    # Upwelling
```

(continues on next page)

(continued from previous page)

```
    "A_db_to_A_ib@thc": ta * thc * M.A_db.swc.density / 1e3,
    "I_db_to_I_ib@thc": ti * thc * M.I_db.swc.density / 1e3,
    "P_db_to_P_ib@thc": (1 - ta - ti) * thc * M.P_db.swc.density / 1e3,
    "A_ib_to_H_sb@thc": thc * M.A_ib.swc.density / 1e3,
    # Advection
    "A_db_to_I_db@adv": (1 - ta) * thc * M.A_db.swc.density / 1e3,
    "I_db_to_P_db@adv": (1 - ta - ti) * thc * M.I_db.swc.density / 1e3,
    "P_ib_to_I_ib@adv": (1 - ta - ti) * thc * M.P_ib.swc.density / 1e3,
    "I_ib_to_A_ib@adv": (1 - ta) * thc * M.I_ib.swc.density / 1e3,
}
```

to create the actual connections we need to:

1. Assemble a list of all species that are affected by thermohaline circulation
2. Specify the connection type that describes thermohaline transport, i.e., `scale_by_concentration`
3. Combine #1 & #2 into a dictionary that can be used by the `create_bulk_connections()` function to instantiate the necessary connections.

```
species_names = list(ic.keys())  # get species list
connection_type = {"ty": "scale_with_concentration", "sp": sl}
connection_dictionary = build_ct_dict(thx_dict, species_names)
create_bulk_connections(connection_dictionary, M, mt="1:1")
```

In the following example, we build the `connection_dictinary` in a more explicit way to define primary production as a function of P upwelling: The first line finds all the upwelling fluxes, and we can then use them as an argument in the `connection_dictionary` definition:

```
# get all upwelling P fluxes except for the high latitude box
pfluxes = M.flux_summary(filter_by="PO4_mix_up", exclude="H_", return_list=True)

# define export productivity in the high latitude box
PO4_ex = Q_(f"{1.8 * M.H_sb.area/M.PC_ratio} mol/a")

c_dict = {  # Surface box to ib, about 78% is remineralized in the ib
    ("A_sb_to_A_ib@POM_P", "I_sb_to_I_ib@POM_P", "P_sb_to_P_ib@POM_P"): {
        "ty": "scale_with_flux",
        "sc": M.PUE * M.ib_remin,
        "re": pfluxes,
        "sp": M.PO4,
    },  # surface box to deep box
    ("A_sb_to_A_db@POM_P", "I_sb_to_I_db@POM_P", "P_sb_to_P_db@POM_P"): {
        "ty": "scale_with_flux",
        "sc": M.PUE * M.db_remin,
        "re": pfluxes,
        "sp": M.PO4,
    },  # high latitude box to deep ocean boxes POM_P
    ("H_sb_to_A_db@POM_P", "H_sb_to_I_db@POM_P", "H_sb_to_P_db@POM_P"): {
        # here we use a fixed rate following Zeebe's Loscar model
        "ra": [
            PO4_ex * 0.3,
            PO4_ex * 0.3,
            PO4_ex * 0.4,
```

(continues on next page)

(continued from previous page)

```
        ],
        "sp": M.PO4,
        "ty": "Regular",
    },
}
create_bulk_connections(c_dict, M, mt="1:1")
```

In the last example, we use the `gen_dict_entries` function to extract a list of connection keys that can be used in the `connection_dictionary` . The following code specifies to find all connection keys that match the particulate organic phosphor fluxes (`POM_P`) defined in the code above, and to replace them with a connection key that uses `POM_DIC` as id-string. The function returns a list of fluxes and matching keys that can be used to specify new connections. See also `boudreau2010.py` which uses a less complex setup (https://github.com/uliw/ESBMTK-Examples).

```
keys_POM_DIC, ref_fluxes = gen_dict_entries(M, ref_id="POM_P", target_id="POM_DIC")

c_dict = {
    keys_POM_DIC: {
        "re": ref_fluxes,
        "sp": M.DIC,
        "ty": "scale_with_flux",
        "sc": M.PC_ratio,
        "al": M.OM_frac,
    }
}
create_bulk_connections(c_dict, M, mt="1:1")
```

1.3 Seawater and Carbon Chemistry

ESBMTK provides several classes that abstract the handling of basin geometry, seawater chemistry and air-sea gas exchange.

1.3.1 Hypsography

For many modeling tasks, it is important to know a globally averaged hypsometric curve. ESBMTK will automatically create a suitable hypsography instance if a *esbmtk.esbmtk.Species()* or *esbmtk.extended_classes.Reservoir()* instance is specified with the geometry keyword as in the following example where the first list item denotes the upper depth datum, the second list item, the lower depth datum, and the last list item denotes the fraction of the total ocean area if the upper boundary would be at sea level.

```
Reservoir(
    name="S_b",  # Name of reservoir group
    geometry=[-200, -800, 1],  # upper, lower, fraction
    concentration="0 mmol/kg",
    species=M.DIC,
    register=M,
)
print(f"M.S_b.area = {M.S_b.area:.2e}") # surface area at upper depth datum
print(f"M.S_b.sed_area = {M.S_b.sed_area:.2e}") # surface between upper and lower datum
print(f"M.S_b.volume = {M.S_b.volume:.2e}") # total volume
```

This will register 3 new instance variables, and also create a hypsometry instance at the model level that provides access to the following methods:

```
#return the ocean area at a given depth in m**2
print(f"M.hyp.area(0) = {M.hyp.area(0):.2e}")

# return the area between 2 depth datums in m**2
print(f"M.hyp.area_dz(0, -200) = {M.hyp.area_dz(0, -200):.2e}")

# return the volume between 2 depth datums in m**3
print(f"M.hyp.volume(0,-200) = {M.hyp.volume(0,-200):.2e}")

# return the total surface area of earth in m**2
print(f"M.hyp.sa = {M.hyp.sa:.2e}")
```

The hypsometric data is based on the Scripps' SRTM15+V2.5.5 grid (Tozer et al., 2019, https://doi.org/10.1029/2019EA000658), which was down-sampled to a 5-minute grid before processing the hypsometry.

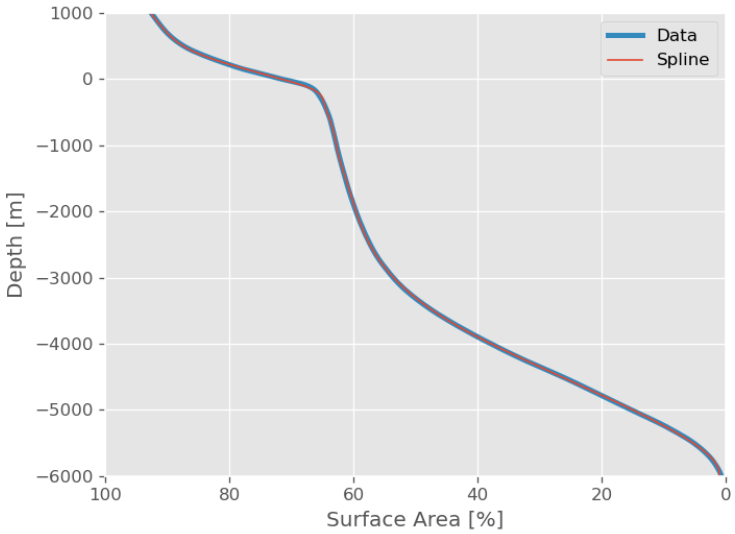


Fig. 6: Comparison between spline fit, and the actual data.

1.3.2 Seawater

ESBMTK provides a *esbmtk.seawater.SeawaterConstants()* class that will be automatically instantiated when a *esbmtk.extended_classes.Reservoir()* instance definition includes the `seawater_parameters` keyword. This keyword expects a dictionary that specifies temperature, salinity, and pressure for a given `Reservoirgroup`. The class methods and instance variables are accessible via the `swc` instance.

```
Reservoir(
    name="S_b",  # box name
    geometry=[-200, -800, 1],  # upper, lower, fraction
    concentration={M.DIC: "2220 umol/kg", M.TA: "2300 umol/kg"},
    seawater_parameters={
        "T": 25,  # Deg celsius
```

(continues on next page)

(continued from previous page)

```
        "P": 0,  # Bar
        "S": 35,  # PSU
    },
    register=M,
)
# Acess the sewater_parameters with the swc instance
print(f"M.S_b.density = {M.S_b.swc.density:.2e}")
```

Apart from density, this class will provide access to a host of instance parameters, e.g., equilibrium constants - see *esbmtk.seawater.SeawaterConstants.update_parameters()* for the currently defined names. Most of these values are computed by `pyCO2SYS` (https://doi.org/10.5194/gmd-15-15-2022). Using `pyCO2SYS` provides access to a variety of parametrizations for the respective equilibrium constants, various pH scales, as well as different methods to calculate buffer factors. Unless explicitly specified in the model definition, ESBMTK uses the defaults set by pyCO2SYS. Note that when using the seawater class, the model concentration unit must be set to `mol/kg` as in the following example:

```
M = Model(
    stop="6 Myr",  # end time of model
    timestep="1 kyr",  # upper limit of time step
    element=["Carbon"],  # list of element definitions
    concentration_unit="mol/kg",
    opt_k_carbonic=13,  # Use Millero 2006
    opt_pH_scale=1,  # 1:total, 3:free scale
    opt_buffers_mode=2, # carbonate, borate water alkalinity only
)
```

Caveats

- Seawater Parameters are only computed once when the `Reservoir` is instantiated, to provide an initial steady state. Subsequent changes to seawater chemistry or physical parameters do not affect the initial state.
- The `swc` instance provides a `show()` method listing most values. However, that list may not be comprehensive.
- See the pyCO2SYS documentation for a list of parameters and options https://pyco2sys.readthedocs.io/en/latest/
- The code example `seawater_example.py` in the examples directory

1.3.3 Carbon Chemistry

pH

Unless explicitly requested (see above), pH will be reported on the total scale. The hydrogen ion concentration ($[H^+]$) is computed by pyCO2SYS based on the initial DIC and total alkalinity (TA) concentrations. Subsequent hydrogen concentration calculations use the iterative approach of Follows et al. 2005 (https://doi.org/10.1016/j.ocemod.2005.05.004).

Provided that the model has terms for DIC and TA, pH calculations for a given *esbmtk.extended_classes.Reservoir()* instance are added using the `esbmtk.bio_pump_functions0.carbonate_chemistry.add_carbonate_system_1()` function:

```
box_names = [A_sb, I_sb, P_sb, H_sb]  # list of Reservoir handles
add_carbonate_system_1(box_names)
```

This will create Species *esbmtk.esbmtk.Species()* instances for `Hplus` and `CO2aq`. After running the model, the resulting concentration data is available in the usual manner:

```
A_sb.Hplus.c
A_sb.CO2aq.c
```

The remaining carbonate species are calculated during post-processing (see the *esbmtk.post_processing.carbonate_system_1_pp()* function) and are available as

```
A_sb.pH
A_sb.HCO3
A_sb.CO3
A_sb.Omega
```

Notes:

- The resulting concentration data depends on the choice of equilibrium constants and how they are calculated (see the `opt_k_carbonic`, `opt_buffers_mode` keywords above).
- The data from post-processing is currently available as *esbmtk.extended_classes.VectorData()* instance, rather than as *esbmtk.esbmtk.Species()* instance.
- Species that use carbonate system 2 (see below), do not need to use carbonate system 1
- ESBMTK will print a warning message of the pH changes by more than 0.01 units per time step. However, this is only a crude measure, since the solver also uses interpolation between integration steps. So this may not catch all possible scenarios.

Carbonate burial and dissolution

Carbonate burial and dissolution use the parametrization proposed by Boudreau et al. 2010 (https://doi.org/10.1029/2009gb003654). The current ESBMTK implementation has the following shortcomings:

- It only considers Calcium dissolution/burial (although it would be easy to add Aragonite)
- Results will only be correct as long as the depth of the saturation horizon remains below the upper depth datum of the deep-water box. Future versions will address this limitation.

The following figure provides an overview of the parametrizations and variables used by the `esbmtk.bio_pump_functions0.carbonate_chemistry.carbonate_system_2()` and `esbmtk.bio_pump_functions0.carbonate_chemistry.add_carbonate_system_2()` functions.

Provided a given model has data for DIC & TA, and that the carbonate export flux is known, `carbonate_system_2` can be added to a Reservoir instance in the following way:

```
surface_boxes: list = [M.L_b]
deep_boxes: list = [M.D_b]
export_fluxes: list = M.flux_summary(filter_by="PIC_DIC L_b", return_list=True)

add_carbonate_system_2(
        r_db=deep_boxes,  # list of reservoir groups
        r_sb=surface_boxes,  # list of reservoir groups
        carbonate_export_fluxes=export_fluxes,  # list of export fluxes
        z0=-200,  # depth of shelf
        alpha=alpha,  # dissolution coefficient, typically around 0.6
    )
```

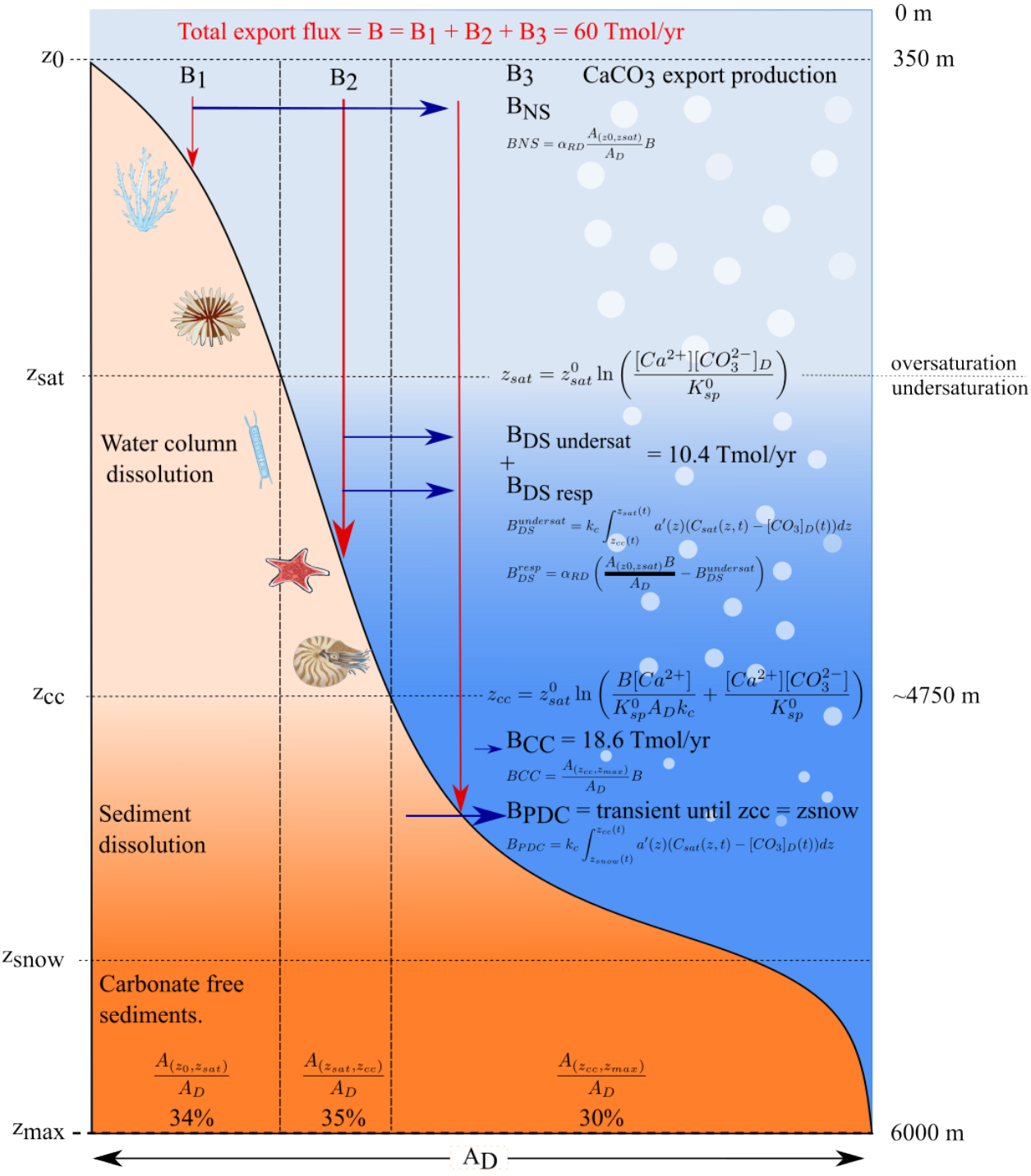


Fig. 7: Overview of the parametrizations and variables used by the `esbmtk.bio_pump_functions0.carbonate_chemistry.carbonate_system_2()` and `esbmtk.bio_pump_functions0.carbonate_chemistry.add_carbonate_system_2()` functions. Image Credit: Tina Tsan & Mahruk Niazi

Notes:

- boxes and fluxes are lists, since in some models there is more than one surface box (e.g., models that resolve individual ocean basins)
- ESBMTK only considers the sediment area to 6000 mbsl. The area contributed by the elevations below 6000 mbsl is negligible, and this constrain simplifies the hypsographic fit.
- The total sediment area of a given `Reservoir` is known provided the box-geometry was specified correctly.
- The `esbmtk.bio_pump_functions0.carbonate_chemistry.carbonate_system_2()` function only returns $[H^+]$ and the dissolution flux for given box. It does not return the burial flux.
- Please study the actual model implementations provided in the examples folder.

Post-Processing

As with `carbonate_system_1` the remaining carbonate species are not part of the equation system, rather they are calculated once a solution has been found. Since the solver does not store the carbonate export fluxes, one first has to calculate the relevant fluxes from the concentration data in the model solution. This is however model dependent (i.e., export productivity as a function of residence time, or as a function of upwelling flux), and as such post-processing of `carbonate_system_2` is not done automatically, but has to be initiated manually, e.g., like this:

```
# get CaCO3_export in mol/year
CaCO3_export = M.CaCO3_export.to(f"{M.f_unit}").magnitude
carbonate_system_2_pp(
    M.D_b,  # Reservoir
    CaCO3_export,  # CaCO3 export flux
    200,  # z0
    6000,  # zmax
)
```

This will compute all carbonate species similar to `carbonate_system_1_pp`, and in addition calculate:

```
M.D_b.Fburial  # CaCO3 burial flux mol/year
M.D_b.Fdiss  # CaCO3 dissolution flux mol/year
M.D_b.zsat  # Saturation depth in mbsl
M.D_b.zcc  # CCD depth in mbsl
M.D_b.zsnow  # Snowline depth in mbsl
```

see the *esbmtk.post_processing.carbonate_system_2_pp()* function for details.

1.3.4 Gas Exchange

ESBMTK implements gas exchange across the Air-Sea interface as a *esbmtk.connections.Species2Species()* instance, between a *esbmtk.extended_classes.GasReservoir()* and a *esbmtk.esbmtk.Species()* instance. In the following example, we first declare a `Gasreservoir` and then connect it with a regular surface box. Note that the CO_2gas transfer calculation requires that the respective surface reservoir carries the `CO2aq` tracer as calculated by the `esbmtk.bio_pump_functions0.carbonate_chemistry_carbonate_system_1.()` function since the gas-transfer depends on the dissolved CO_2rather than on the DIC concentration.

```
GasReservoir(
    name="CO2_At",
    species=M.CO2,
    reservoir_mass="1.833E20 mol",
```

(continues on next page)

(continued from previous page)

```
    species_ppm="280 ppm",
    register=M,
)

Species2Species(  # Example for CO2
    source=M.CO2_At,  # GasReservoir
    sink=M.L_b.DIC,  # Reservoir
    species=M.CO2,
    ref_species=M.H_b.CO2aq,
    solubility=M.H_b.swc.SA_co2,
    area=M.L_b.area,  # surface area
    id="L_b_GEX",  # connection id
    piston_velocity="4.8 m/d",
    water_vapor_pressure=M.H_b.swc.p_H2O,
    register=M,
    ctype="gasexchange",
)
```

Defining gas transfer for O2 uses the same approach, but note the use of the `solubility` and `ref_species` keywords. At present, ESBMTK only carries the solubility constants for $CO_2$and O_2.

```
Species2Species(  # Example for O2
    source=M.O2_At,  # GasReservoir
    sink=M.L_b.O2,  # Reservoir
    species=M.O2,
    ref_species=M.L_b.O2,
    solubility=M._b.swc.SA_o2,
    area=M._b.area,
    piston_velocity="4.8 m/d",
    water_vapor_pressure=M.L_b.swc.p_H2O,
    id=f"O2_gas_exchange_L_b",
    register=M,
    ctype="gasexchange",
)
```

1.3.5 pCO_2Dependent Weathering

ESBMTK defines a simple power law function to calculate pCO_2dependent weathering fluxes (see e.g., Walker and Hays, 1981, https://doi.org/10.1029/jc086ic10p09776):

$$f = A \times f_0 \times \left(\frac{pCO_2}{p_0CO_2} \right)^c$$

where A denotes the area, f_0 the weathering flux at p_0CO_2, $pCO_2$the CO_2partial pressure at a given time t, p_0CO_2 the reference partial pressure of $CO_2$and c a constant. See the *esbmtk.processes.weathering()* function for details. Within the context of ESBMTK, weathering fluxes are just another connection type:

```
Species2Species(  # CaCO3 weathering
    source=M.Fw.DIC,  # source of flux
    sink=M.L_b.DIC,
    reservoir_ref=M.CO2_At,  # pCO2
    ctype="weathering",
```

(continues on next page)

(continued from previous page)

```
    id="wca",
    scale=1,  # optional, defaults to 1
    ex=0.2,  # exponent c
    pco2_0="280 ppm",  # reference pCO2
    rate="12 Tmol/a",  # rate at pco2_0
    register=M,
)
```

1.4 Extending ESBMTK

1.4.1 The ElementProperties and SpeciesProperties Classes

ESBMTK uses the *esbmtk.esbmtk.SpeciesProperties()* and *esbmtk.esbmtk.ElementProperties()* class primarily to control plot labeling. Each `SpeciesProperties` instance is a child of an `ElementProperties` instance. Within the model hierarchy, one would access e.g., `DIC` as `M.Carbon.DIC` . However, this results in a lot of redundant code, so the `SpeciesProperties` instances are also registered with the `Model` instance.

```
from esbmtk import Model

M = Model(stop="6 Myr", timestep="1 kyr", element=["Carbon", "Oxygen"])
# Access using complete hirarchy
print(M.Carbon.DIC)
# Access using shorthand
print(M.DIC)
```

The distinction between ElementProperties and SpeciesProperties exists to group information that is common to all species of a given element. The current entry for Oxygen reads, e.g., like this

```
def Oxygen(model: Model) -> None:
    """Common Properties of Oxygen

    Parameters
    ----------
    model : Model
        Model instance

    """
    eh = ElementProperties(
        name="Oxygen",
        model=model,  # model handle
        mass_unit="mol",  # base mass unit
        li_label="$^{16$}O",  # Name of light isotope
        hi_label="$^{18}$)",  # Name of heavy isotope
        d_label=r"$\delta^{18}$)",  # Name of isotope delta
        d_scale="mUr VSMOV",  #
        r=2005.201e-6,  # https://nucleus.iaea.org/rpst/documents/vsmow_slap.pdf
        register=model,
    )
```

and the associated species definitions are:

```
SpeciesProperties(name="O", element=eh, display_as="O", register=eh)
SpeciesProperties(name="O2", element=eh, display_as=r"O$_{2}$", register=eh)
SpeciesProperties(name="OH", element=eh, display_as=r"OH$^{-}$", register=eh)
```

Note that the variable `eh` is used to associate the `SpeciesProperties` instance with the `ElementProperties` instance. Upon startup, ESBMTK loads all predefined species definitions for each element named in the `element_list` keyword and registers them with the model instance. See the file `species_definitions.py` in the source-code for the currently defined elements and species (https://github.com/uliw/esbmtk/blob/master/src/esbmtk/species_definitions.py)

To see a list of all known species for a given element use the `list_species` method of the ElementProperties instance

```
M.Oxygen.list_species()
```

Modifying/Extending an existing SpeciesProperties/ElementProperties definition

Modifying and existing definition is done after the model has been loaded, but before running the solver. The following two lines, show, e.g, how to change the isotope scale of Oxygen from mUR to permil, and how to set the plot concentration unit of O2 to μ mol:

```
M.Oxygen.d_scale="\u2030"
M.Oxygen.O2.scale_to="umol"
```

see the *esbmtk.esbmtk.SpeciesProperties()* and *esbmtk.esbmtk.ElementProperties()* definitions for a full list of implemented properties.

Adding custom SpeciesProperties definitions

To add a new species follow the examples in the `species_definitions.py` source code file. Provided you loaded `Oxygen` in the model definition, defining a new species instance for dissolved oxygen would look like this

```
from esbmtk import SpeciesProperties
SpeciesProperties(
    name="O2aq",
    element=M.Oxygen,
    display_as=r"[O$_{2}$]$_{aq}$",
)
M.O2aq = M.Oxygen.O2aq  # register shorthand with model
print(M.O2aq)
```

Adding a new ElementProperties and its species

In this example, I use Boron to demonstrate how to add a new element and its respective species. Note, however, that Boron is already part of ESBMTK, for this example it is simply not loaded.

```
from esbmtk import Model, ElementProperties, SpeciesProperties

M = Model(stop="6 Myr", timestep="1 kyr")

ElementProperties(
    name="Boron",
```

(continues on next page)

(continued from previous page)

```
    model=M,  # model handle
    mass_unit="mmol",  # base mass unit
    li_label=r"$^{11$}B",  # Name of light isotope
    hi_label=r"$^{10$}B",  # Name of heavy isotope
    d_label=r"$\delta{11}B",  # Name of isotope delta
    d_scale="mUr SRM951",  # Isotope scale.
    r=0.26888,  # isotopic abundance ratio for species
    register=M,
)

SpeciesProperties(name="B", element=M.Boron, display_as="B")
SpeciesProperties(name="BOH", element=M.Boron, display_as="BOH")
SpeciesProperties(name="BOH3", element=M.Boron, display_as=r"B(OH)$_{3}$")
SpeciesProperties(name="BOH4", element=M.Boron, display_as=r"B(OH)$_{4}^{-}$")

# register the species shorthands with the model.
for sp in M.Boron.lsp:
    setattr(M, sp.name, sp)

# verify the sucess
print(M.BOH3)
```

Note that in the above example, we leverage that `ElementProperties` instances keep track of their species in the `lsp` variable. Provided that none of the species was defined previously, we can thus simply loop over the list of species to register them with the model.

1.4.2 Adding custom functions to ESBMTK

ESBMTK has some rudimentary support to add custom functions. This is currently not very user-friendly, and a better interface may become available in the future. Adding a custom function to ESBMTK requires the following considerations:

- ESBMTK must be able to import the function so that it can be used in the equation system
- ESBMTK must have a way to assign the correct input & output variables to the function call
- Since we only declare a function and not a complete connection object, it is up to the user code to make sure that function parameters like scale factors (see below) are in the correct units, and of type `Number` (rather than string or quantity). Likewise, it is up to the user-provided code to ensure that the returned values have the correct sign.
- The function signature of any custom function must adhere to a format, where the first argument(s) are of type float, and the second argument is a tuple (which can be empty):

```
def custom(c0:float, t: tuple)  # valid
def custom(c0:float, c1:float, t: tuple) # valid
def custom(c0:float, c1:int, t: tuple) # invalid
```

The reason behind this rigid scheme has to do with memory management, but it is typically easy to adhere to them.

A worked example

Let's consider a simple case where we define a custom function `my_burial()` that returns a flux as a function of concentration. For this, we need a parameter that passes a concentration, and a parameter that passes a scaling factor. Since both are float, we could use this signature with an empty tuple

```
def my_burial(concentration: float, scale: float, t: tuple) -> float:
```

However, to demonstrate the use of a tuple to pass one or more parameters, I will pass the scaling factor as a tuple in the below example:

```
def my_burial(concentration: float, p: tuple) -> float:
    """Calculate a flux as a function of concentration

    Parameters
    ----------
    concentration : float
        substance concentration
    p : tuple
        where the first element is the scaling factor

    Returns
    -------
    float
        flux in model mass unit / time

    Notes: the scale information is passed as a tuple, so we need
    extract it from the tuple before using it

    f is a burial flux, so we need to return a negative number.
    """
    (scale,) = p

    f = concentration * scale

    return -f
```

ESBMTK needs to import this function into the code that builds the equation system, so this requires that we place this function into a module file (e.g., `my_functions.py`), and that we register this file and any custom functions with the model code. ESBMTK provides the `register_user_function()` function which is used like this

```
register_user_function(M, "my_functions", "my_burial")
```

Note that the last argument can also be a list of function names.

Next, we need to create code that maps the model variables required by `my_burial()` to the actual function call. Most of this work is done by the *esbmtk.extended_classes.ExternalCode()* class. In the following example, we wrap this task into a dedicated function, but this is not a hard requirement. I add this function to the `my_functions.py` file, but you can also keep it with the code that defines the model. Since we want to use this function to calculate a flux between two reservoirs (or a sink/source), we need to pass the source and sink reservoirs, as well as the species and the scale information, to `add_my_burial()`.

Notes on the below code:

- If `my_buria()` is defined in the same file as `add_my_burial()` there is no need to import `my_burial()`

- The `function_input_data` keyword requires the `Species` instance, not the array with the concentration values (i.e., `Species.c`). More than one argument can be given.
- The `return_values` keyword expects a dictionary. If the return value is a flux, the dictionary key must be preceded by `F_`. The key format must be `{Species.full_name}.{SpeciesProperties.name}`. The `id_string` must be unique within the model, and must not contain blanks or dots. If the return value is a Species, the dictionary entry reads like this `{f"R_{rg.full_name}.Hplus": rg.swc.hplus}`, where dictionary value is used to set the initial condition.
- In the last step, the `register_return_values` parses the return value dictionary and creates the necessary *esbmtk.esbmtk.Flux()* or *esbmtk.esbmtk.Species()* instances. This step may move to the init-section of the *esbmtk.extended_classes.ExternalCode()* class definition in a future version.

```
def add_my_burial(source, sink, species, scale) -> None:
    """This function initializes a user supplied function
    so that it can be used within the ESBMTK eco-system

    Parameters
    ----------
    source : Source | Species | Reservoir
        A source
    sink : Sink | Species | Reservoir
        A sink
    species : SpeciesProperties
        A model species
    scale : float
        A scaling factor

    """
    from esbmtk import ExternalCode, register_return_values

    p = (scale,)  # convert float into tuple
    ec = ExternalCode(
        name="mb",
        species=source.species,
        function=my_burial,
        fname="my_burial",
        function_input_data=[source],
        function_params=p,
        register=source,
        return_values=[
            {f"F_{sink.full_name}.{species.name}": "id_string"},
        ],
    )

    register_return_values(ec, source)
```

Once these functions are defined, we can use them in the model definition as follows

```
# register the new module and function with the model
register_user_function(M, "my_functions", "my_burial")

# import the add_my_burial into this script file
from my_functions import add_my_burial
```

(continues on next page)

(continued from previous page)

```
# add the my_burial_function to the model objects.
add_my_burial(
    M.D_b,  # Source
    M.burial,  # Sink
    M.PO4,  # SpeciesProperties
    M.D_b.volume.magnitude / 2000.0,  # Scale
)
```

Note that `M.D_b.volume.magnitude` is not a number but a quantity. So one needs to query the numerical value with `.magnitude` or add code to `add_my_burial` to query the type of the input arguments and convert as necessary.

The file `user_defined_functions.py` in the `examples` directory shows a working example.

1.4.3 Debugging custom function integration

The current custom function integration interface is not very user-friendly and often requires investigating the actual `equations.py` file. In the default operating mode, ESBMTK will recreate this file for each model run, so that print statements and breakpoints that have been placed in `equations.py` have no effect. Use the `parse_model` keyword in the model instance to keep the edited `equations.py` for the next run:

```
M = Model(
    stop="1000 yr",  # end time of model
    timestep="1 yr",  # upper limit of time step
    element=["Phosphor"],  # list of element definitions
    parse_model=False,  # do not overwrite equations.py
)
```

> The document assumes you are using a source repository service that promotes a contribution model similar to GitHub's fork and pull request workflow. While this is true for the majority of services (like GitHub, GitLab, BitBucket), it might not be the case for private repositories (e.g., when using Gerrit).
>
> Also notice that the code examples might refer to GitHub URLs or the text might use GitHub specific terminology (e.g., *Pull Request* instead of *Merge Request*).
>
> Please make sure to check the document having these assumptions in mind and update things accordingly. Especially if your project is open source. The text should be very similar to this template, but there are a few extra contents that you might decide to also include, like mentioning labels of your issue tracker or automated releases.

1.5 Contributing

Welcome to `esbmtk` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp[1].

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

[1] Even though, these resources focus on open source projects and communities, the general ideas behind collaborating with other developers to collectively create software are general and can be applied to all sorts of environments, including private companies and proprietary code bases.

1.5.1 Issue Reports

If you experience bugs or general issues with `esbmtk`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

1.5.2 Documentation Improvements

You can help improve `esbmtk` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`esbmtk` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

Tip: Please notice that the GitHub web interface provides a quick way of propose changes in `esbmtk`'s files. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to the `docs` folder in the source repository, find which file you would like to propose changes and click in the little pencil icon at the top, to open GitHub's code editor. Once you finish editing the file, please write a message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

1.5.3 Code Contributions

Please see the code documentation at https://esbmtk.readthedocs.io/en/latest/

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n esbmtk python=3 six virtualenv pytest pytest-cov
conda activate esbmtk
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/esbmtk.git
   cd esbmtk
   ```

4. You should run:

   ```
   pip install -U pip setuptools -e .
   ```

   to be able to import the package under development in the Python REPL.

Implement your changes

1. Create a branch to hold your changes:

   ```
   git checkout -b my-feature
   ```

   and start making changes. Never work on the main branch!
2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.
3. Add yourself to the list of contributors in `AUTHORS.rst`.
4. When you're done editing, do:

   ```
   git add <MODIFIED FILES>
   git commit
   ```

   to record your changes in git.

5. Please check that your changes don't break any unit tests with:

```
tox
```

(after having installed tox with `pip install tox` or `pipx`).

You can also use tox to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

```
git push -u origin my-feature
```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

   Find more detailed information in creating a PR. You might also want to open the PR as a draft first and mark it as ready for review after the feedbacks from the continuous integration (CI) system or any required fixes.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.
2. Sometimes tox misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with tox, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

```
tox -e docs
```

Try running:

```
tox -r -e docs
```

3. Make sure to have a reliable tox installation that uses the correct Python version (e.g., 3.7+). When in doubt you can run:

```
tox --version
# OR
which tox
```

If you have trouble and are seeing weird errors upon running tox, you can also try to create a dedicated virtual environment with a tox binary freshly installed. For example:

```
virtualenv .venv
source .venv/bin/activate
.venv/bin/pip install tox
.venv/bin/tox -e all
```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

1.5.4 Maintainer tasks

Releases

If instead you are using a different/private package index, please update the instructions accordingly.

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `esbmtk`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.
6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

1.6 Contributors

- uliw <https://github.com/uliw>
- Tina Tsan <https://github.com/tinatsan>
- rubentium <https://github.com/rubentium>
- Mahrukh-Niazi <https://github.com/Mahrukh-Niazi>

Author
Uli Wortmann

Contents

1.7 1 Changelog

- May 10th v 0.13.0.x fixes an error in the solubility calculation for oxygen and carbon dioxide. This will change the steady state results for pCO2 in existing models by about 4 ppm. This version renamed many classes. Existing models may need some editing.
  - Element -> ElementProperties
  - Species -> SpeciesProperties
  - Reservoir -> Species
  - ReservoirGroup -> Reservoir
  - ConnectionGroup -> ConnectionProperties
  - Connection -> Connect
  - SourceGroup -> SourceProperties
  - SinkGroup -> SinkProperties

  Since the respective ConnectionProperty, SourceProperty and SinkProperty objects which correlate with the former ConnectionGroup, SourceGroup SinkGroup classes. As such, existing code must be changed from

```
Sink(
name="burial",
species=M.PO4,
register=M,  #
)
```

  to

```
SinkProperties(
name="burial",
species=[M.PO4],
)
```

  and

```
Connection(  #
source=M.S_b,  # source of flux
sink=M.D_b,  # target of flux
ctype="scale_with_concentration",
scale=M.S_b.volume / tau,
id="primary_production",
)
```

to

```
ConnectionProperties(  #
source=M.S_b,  # source of flux
sink=M.D_b,  # target of flux
ctype="scale_with_concentration",
scale=M.S_b.volume / tau,
id="primary_production",
species=[M.PO4],  # apply this only to PO4
)
```

- May 1st, v 0.12.0.28 ESBMTK can now be installed via conda. Various documentation updates
- Dec. v 0.12.0.x This is a breaking change that requires the following updates to the model definition.
  - Models that use isotope calculations need to ensure that sources and sink also specify the isotope keyword.
  - Weathering and Gas-exchange have now become connection properties, see the examples in the online documentation
  - Models that used carbonate$_{system1pp}$() no longer need to call this specifically, as this function is now called automatically
- Oct. 12th, 2023 v 0.11.0.2 This is a breaking change. Added support to specify box area and volume explicitly, rather than as a function of hypsography. This is likely to affect existing geoemtry definitions since the (area/total area) parameter has changed meaning The area fraction is now calcualted automatically, and unless you split the model in specific basins the last parameter in the geometry list should always be 1 (i.e., [0, -350, 1]).

  Equilibrium constants are now calculated by pyCO2SYS. This facilitates a wide selection of parametrizations via the `opt_k_carbonic` and `opt_pH_scale` keywords in the Model definition. Options and defaults are the same as for pyCO2SYS.
- Oct. 30th, 2023 v 0.10.0.11 This is a breaking change. Remineralization and photosynthesis must be implemented via functions, rather than transport connections. CS1 and CS2 are retired, and replaced by photosynthesis, organic-matter remineralization and carbonate-dissolution functions. I've started writing a user guide, see https://esbmtk.readthedocs.io/en/latest/ESBMTK-Tutorial.html

So far, only the very basics are covered. More to come!

- July 28th, 2023, v 0.9.0.1 The ODEPACk backend is now fully functional, and the basic API is more or less stable.
- Nov. 11th2022, v 0.9.0.0 Moved to odepack based backend. Removed now defunct code. The odepack backend does not yet support isotope calculations.
- 0.8.0.0
  - Cleanup of naming scheme which is now strictly hierarchical.
  - Bulk connection dictionaries now have to be specified as `source_to_sink` instead of `source2sink`.
  - The connection naming scheme has been revamped. Please see `esbmtk.connect.__set_name__()` documentation for details.
  - Model concentration units must now match 'mole/liter' or 'mol/kg'. Concentrations can still be specified as `mmol/l` or `mmol/kg`, but model output will be in mole/liter or kg. At present, the model does not provide for the automatic conversion of mol/l to mol/kg. Thus you must specify units in a consistent way.
  - The SeawaterConstants class now always returns values as mol/kg solution. Caveat Emptor.
  - The SeawaterConstants class no longer accepts the 'model' keyword
  - All of his will break existing models.
  - Models assume by default that they deal with ideal water, i.e., where the density equals one. To work with seawater, you must set `ideal_water=False`. In that case, you should also set the `concentration_unit` keyword to `'mol/kg'` (solution).
  - Several classes now require the "register" keyword. You may need to fix your code accordingly
- The flux and connection summary methods can be filtered by more than one keyword. Provide a filter string in the following format `"keyword_1 keyword_2` and it will only return results that match both keywords.
- Removed the dependency on the nptyping and number libraries

- 0.7.3.9 Moved to setuptools build system. Lost of code fixes wrt isotope calculations, minor fixes in the carbonate module.
- March 2nd0.7.3.4 `Flux_summary` now supports an `exclude` keyword. Hot fixed an error in the gas exchange code, which affected the total mass of atmosphere calculations. For the time being, the mass of the atmosphere is treated as constant.
- 0.7.3.0 Flux data is no longer kept by default. This results in huge memory savings. esbmtk now requires python 3.9 or higher, and also depends on `os` and `psutil`. the scale with flux process now uses the `ref_flux` keyword instead of `ref_reservoirs`. Models must adapt their scripts accordingly. esbmtk objects no longer provide delta values by default. Rather they need to be calculated in the post-processing step via `M.get_delta_values()`. The `f_0` keyword in the weathering connection is now called `rate`. Using the old keyword will result in a unit error.
- January 8th0.7.2.2 Fixed several isotope calculation regressions. Added 31 Unit tests.

1.8 esbmtk

1.8.1 esbmtk package

Submodules

esbmtk.carbonate_chemistry module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020-2021 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

`esbmtk.carbonate_chemistry.`**`NDArrayFloat`**

First we define the actual function, carbonate_system_1_ode(). In the second step we create a wrapper init_carbonate_system_1() that defines how to integrate this function into esbmtk. In the third step we create a functiom that uses init_carbonate_system_1() to associates cs1 instances with the respective reservoirs.

The process for cs2 is analogous

Type
Carbonate System 1 setup requires 3 steps

alias of `ndarray`[`Any`, `dtype`[`float64`]]

`esbmtk.carbonate_chemistry.`**`add_carbonate_system_1`**(*rgs: list*)

Creates a new carbonate system virtual reservoir for each reservoir in rgs. Note that rgs must be a list of reservoir groups.

Required keywords:
rgs: list = [] of Reservoir Group objects

These new virtual reservoirs are registered to their respective Species as 'cs'.

The respective data fields are available as rgs.r.cs.xxx where xxx stands for a given key key in the vr_datafields dictionary (i.e., H, CA, etc.)

esbmtk.carbonate_chemistry.**add_carbonate_system_2**(***kwargs*) → None

Creates a new carbonate system virtual reservoir which will compute carbon species, saturation, compensation, and snowline depth, and compute the associated carbonate burial fluxes

Required keywords:

r_sb: list of Reservoir objects in the surface layer r_db: list of Reservoir objects in the deep layer carbonate_export_fluxes: list of flux objects which must match the list of Reservoir objects. zsat_min = depth of the upper boundary of the deep box z0 = upper depth limit for carbonate burial calculations typically zsat_min

Optional Parameters:

zsat = initial saturation depth (m) zcc = initial carbon compensation depth (m) zsnow = initial snowline depth (m) zsat0 = characteristic depth (m) Ksp0 = solubility product of calcite at air-water interface (mol^2/kg^2) kc = heterogeneous rate constant/mass transfer coefficient for calcite dissolution (kg m^-2 yr^-1) Ca2 = calcium ion concentration (mol/kg) pc = characteristic pressure (atm) pg = seawater density multiplied by gravity due to acceleration (atm/m) I = dissolvable CaCO3 inventory co3 = CO3 concentration (mol/kg) Ksp = solubility product of calcite at in situ sea water conditions (mol^2/kg^2)

esbmtk.carbonate_chemistry.**carbonate_system_1**(*dic*, *ta*, *hplus_0*, *co2aq_0*, *p*) → tuple

Calculates and returns the H+ and carbonate alkalinity concentrations

for the given reservoirgroup

Parameters

- **dic** – float with the dic concentration
- **ta** – float with the ta concentration
- **hplus_0** – float with the H+ concentration
- **co2aq_0** – float with the [CO2]aq concentration
- **p** – tuple with the parameter list

Returns

dCdt_Hplus, dCdt_co2aq

LIMITATIONS: - Assumes all concentrations are in mol/kg - Assumes your Model is in mol/kg ! Otherwise, DIC and TA updating will not be correct.

Calculations are based off equations from: Boudreau et al., 2010, https://doi.org/10.1029/2009GB003654 Follows, 2006, doi:10.1016/j.ocemod.2005.05.004

esbmtk.carbonate_chemistry.**carbonate_system_2**(*CaCO3_export: float*, *dic_t_db: float | tuple*, *ta_db: float*, *dic_t_sb: float | tuple*, *hplus_0: float*, *zsnow: float*, *p*) → tuple

Calculates and returns the fraction of the carbonate rain that is dissolved an returned back into the ocean. This functions returns:

DIC_burial, DIC_burial_l, Hplus, zsnow

LIMITATIONS: - Assumes all concentrations are in mol/kg - Assumes your Model is in mol/kg

Calculations are based off equations from: Boudreau et al., 2010, https://doi.org/10.1029/2009GB003654

esbmtk.carbonate_chemistry.**get_hplus**(*dic*, *ta*, *h0*, *boron*, *K1*, *K2*, *KW*, *KB*) → float

Calculate H+ concentration based on a previous estimate [H+]. After Follows et al. 2006, doi:10.1016/j.ocemod.2005.05.004

Parameters

- **dic** – DIC in mol/kg
- **ta** – TA in mol/kg
- **h0** – initial guess for H+ mol/kg
- **boron** – boron concentration
- **K1** – Ksp1
- **K2** – Ksp2
- **KW** – K_water
- **KB** – K_boron

Returns H
new H+ concentration in mol/kg

esbmtk.carbonate_chemistry.**get_pco2**(*SW*) → float

Calculate the concentration of pCO2

esbmtk.carbonate_chemistry.**init_carbonate_system_1**(*rg:* Reservoir)

Creates a new carbonate system virtual reservoir for each reservoir in rgs. Note that rgs must be a list of reservoir groups.

Required keywords:
rgs: list = [] of Reservoir Group objects

These new virtual reservoirs are registered to their respective Species as 'cs'.

The respective data fields are available as rgs.r.cs.xxx where xxx stands for a given key key in the vr_datafields dictionary (i.e., H, CA, etc.)

esbmtk.carbonate_chemistry.**init_carbonate_system_2**(*export_flux:* Flux, *r_sb:* Reservoir, *r_db:* Reservoir, *kwargs: dict*)

Initialize a carbonate system 2 instance. Note that the current implmentation assumes that the export flux is the total export flux over surface area of the mixed layer, i.e., the sediment area between z0 and zmax

Parameters

- **export_flux** (Flux) – CaCO3 export flux from the surface box
- **r_sb** (Reservoir) – Reservoir instance of the surface box
- **r_db** (*box*) – Reservoir instance of the deep box
- **kwargs** (*dict*) – dictionary of keyword value pairs

esbmtk.carbonate_chemistry.**phc**(*m: float*) → float

the reservoir class accepts a plot transform. here we use this to display the H+ concentrations as pH. After import, you can use it with like this in the reservoir definition

plot_transform_c=phc,

esbmtk.connections module

esbmtk.connections

Classes which handle the connections and fluxes between esbmtk objects like Species, Sources, and Sinks.

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

class esbmtk.connections.**ConnectionProperties**(***kwargs*)

Bases: *esbmtkBase*

Connect reservoir/sink/source groups when at least one of the arguments is a reservoirs_group object. This method will create regular connections for each matching species.

Use the connection.update() method to fine tune connections after creation

Example:

```
ConnectionProperties(source =  upstream reservoir / upstream reservoir group
   sink = downstrean reservoir / downstream reservoirs_group
   delta = defaults to zero and has to be set manually
   alpha =  defaults to zero and has to be set manually
   rate = shared between all connections
   ref_reservoirs = shared between all connections
   ref_flux = shared between all connections
   species = list, optional, if present, only these species will be connected
   ctype = needs to be set for all connections. Use "Regular"
           unless you require a specific connection type
   pl = [list]) process list. optional, shared between all connections
   id = optional identifier, passed on to individual connection
   plot = "yes/no" # defaults to yes, shared between all connections
)

ConnectionProperties(
          source=OM_Weathering,
          sink=Ocean,
          rate={DIC: f"{OM_w} Tmol/yr" ,
                ALK: f"{0} Tmol/yr"},
          ctype = {DIC: "Regular",
                   ALK: "Regular"},
        )
```

add_connections(***kwargs*) → None

Add connections to the connection group

info() → None

List all connections in this group

exception esbmtk.connections.**KeywordError**(*message*)

Bases: Exception

exception esbmtk.connections.**ScaleFluxError**(*message*)

Bases: Exception

class esbmtk.connections.**Species2Species**(***kwargs*)

Bases: *esbmtkBase*

Two reservoirs connect to each other via at least one flux. This
module creates the connecting flux and creates a connector object which stores all connection properties.

For simple connections, the type flux type is derived implcitly from the specified parameters. For complex connections, the flux type must be set explicitly. See the examples below:

Parameters:

- source: An object handle for a Source or Species
- sink: An object handle for a Sink or Species
- rate: A quantity (e.g., "1 mol/s"), optional
- delta: The isotope ratio, optional
- ref_reservoirs: Species or flux reference
- alpha: A fractionation factor, optional
- id: A string wich will become part of the object name, it will override automatic name creation
- signal: An object handle of signal, optional
- ctype: connection type, see below
- bypass :str optional defaults to "None" see scale with flux

The connection name is derived automatically, see the documentation of __set_name__() for details

Connect Types:

Basic Connects (the advanced ones are below):

- **If both =rate= and =delta= are given, the flux is treated as a**
  fixed flux with a given isotope ratio. This is usually the case for most source objects (they can still be affected by a signal, see above), but makes little sense for reservoirs and sinks.
- If both the =rate= and =alpha= are given, the flux rate is fixed (subject to any signals), but the isotopic ratio of the output flux depends on the isotopic ratio of the upstream reservoir plus any isotopic fractionation specified by =alpha=. This is typically the case for fluxes which include an isotopic fractionation (i.e., pyrite burial). This combination is not particularly useful for source objects.
- If the connection specifies only =delta= the flux is treated as a variable flux which is computed in such a way that the reservoir maintains steady state with respect to it's mass.
- If the connection specifies only =rate= the flux is treated as a fixed flux which is computed in such a way that the reservoir maintains steady state with respect to it's isotope ratio.

Connecting a Source to a Species

Unless you use a Signal, a source typically provides a steady stream with a given isotope ratio (if used)

Example:

```
Species2Species(source =  Source,
        sink = downstrean reservoir,
        rate = "1 mol/s",
        delta = optional,
        signal = optional, see the signal documentation
        )
```

Connecting a Species to Sink or another Species

Here we can distinguish between cases where we use fixed flux, or a flux that reacts to in some way to the upstream reservoir (see the Species to Species section for a more complete treatment):

Fixed outflux, with no isotope fractionation

Example:

```
Species2Species(source =  upstream reservoir,
      sink = Sink,
      rate = "1 mol/s",
      )
```

Fixed outflux, with isotope fractionation

Example:

```
Species2Species(source =  upstream reservoir,
      sink = Sink,
      alpha = -28,
      rate = "1 mol/s",
      )
```

Advanced Connects

You can aditionally define connection properties via the ctype keyword. This requires additional keyword parameters. The following values are recognized

ctype = “scale_with_flux”

This will scale a flux relative to another flux:

Example:

```
Species2Species(source =  upstream reservoir,
        sink = downstream reservoir,
        ctype = "scale_with_flux",
        ref_flux = flux handle,
```

(continues on next page)

(continued from previous page)

```
        scale = 1, #
        )
```

ctype = "scale_with_concentration"

This will scale a flux relative to the mass or concentration of a reservoir

Example:

```
Species2Species(source =  upstream reservoir,
       sink = downstream reservoir,
       ctype = "scale_with_concentration",
       ref_reservoirs = reservoir handle,
       scale = 1, # scaling factor
       )
```

Useful methods in this class

The following methods might prove useful:

- info() will provide a short description of the connection objects.
- list_processes() which will list all the processes which are associated with this connection.
- update() which allows you to update connection properties after the connection has been created

property alpha: float | int

property delta: float | int

get_species(*r1*, *r2*) → None

In most cases the species is set by r2. However, if we have backward fluxes the species depends on the r2

info(**kwargs) → None

Show an overview of the object properties. Optional arguments are index :int = 0 this will show data at the given index indent :int = 0 indentation

property rate: float | int

update(**kwargs)

Update connection properties. This will delete existing processes and fluxes, replace existing key-value pairs in the self.kwargs dict, and then re-initialize the connection.

exception esbmtk.connections.**Species2SpeciesError**(*message*)

Bases: Exception

esbmtk.esbmtk module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

class esbmtk.esbmtk.**ElementProperties**(***kwargs*)

Bases: *esbmtkBase*

Each model, can have one or more elements. This class sets element specific properties

Example:

```
ElementProperties(name      = "S "           # the element name
        model     = Test_model     # the model handle
        mass_unit =  "mol",        # base mass unit
        li_label  =  "$^{32$S",    # Label of light isotope
        hi_label  =  "$^{34}S",    # Label of heavy isotope
        d_label   =  r"$\delta^{34}$S",  # Label for delta value
        d_scale   =  "VCDT",       # Isotope scale
        r         = 0.044162589,   # isotopic abundance ratio for element
        reference = "https:/// or citation",
      )
```

list_species() → None

List all species which are predefined for this element

class esbmtk.esbmtk.**Flux**(***kwargs: dict[str, any]*)

Bases: *esbmtkBase*

A class which defines a flux object. Flux objects contain information which links them to an species, describe things like the mass and time unit, and store data of the total flux rate at any given time step. Similarly, they store the flux of the light and heavy isotope flux, as well as the delta of the flux. This is typically handled through the Species2Species object. If you set it up manually

Example:

```
Flux = (name = "Name" # optional, defaults to _F
     species = species_handle,
     delta = any number,
     rate  = "12 mol/s" # must be a string
     display_precision = number, optional, inherited from Model
```

)

You can access the flux data as

- Name.m # mass
- Name.d # delta

- Name.c # same as Name.m since flux has no concentration

info(***kwargs*) → None

Show an overview of the object properties. Optional arguments are:

Parameters

- **index** – int = 0 this will show data at the given index
- **indent** – int = 0 indentation

exception esbmtk.esbmtk.**FluxError**(*message*)

Bases: Exception

class esbmtk.esbmtk.**Model**(***kwargs: dict[any, any]*)

Bases: *esbmtkBase*

This class is used to specify a new model. See the __init__() method for a detailed explanation of the parameters

The user facing methods of the model class are

- Model_Name.info()
- Model_Name.save_data()
- Model_Name.plot([sb.DIC, sb.TA]) plot any object in the list
- Model_Name.save_state() Save the model state
- Model_name.read_state() Initialize with a previous model state
- Model_Name.run()
- Model_Name.list_species()
- Model_name.flux_summary()
- Model_Name.connection_summary()

clear()

delete all model objects

connection_summary(***kwargs: dict*) → None

Show a summary of all connections

Optional parameters:

Parameters

- **filter_by** – str = "" # filter on connection id. If more than one word is provided, all words must match
- **return_list** – bool if set, return a list object instead of printing to the terminal

flux_summary(***kwargs: dict*)

Show a summary of all model fluxes

Optional parameters:

Parameters

- **filter_by** – str = "" # filter on flux name or part of flux name words separated by blanks act as additional conditions, i.e., all words must occur in a given name
- **return_list** – bool = False, # if True return a list of fluxes matching the filter_by string.

- **exclude** – str = "" # exclude all results matching this string

  Example:

```
names = M.flux_summary(filter_by="POP A_sb", return_list=True)
```

get_delta_values()

Calculate masses and isotope ratios in the usual delta notation

info(***kwargs*) → None

Show an overview of the object properties. Optional arguments are (name/default/explanation)

Parameters

- **index** – int = 0 # this will show data at the given index
- **indent** – int = 0 # print indentation

list_species()

List all defined species.

merge_temp_results()

Replace the datafields which were used for an individual iteration with the data we saved from the previous iterations

ode_solver(*kwargs*)

Use the ode solver

plot(*pl: list = None*, ***kwargs*) → None

Plot all objects specified in pl

Parameters
pl – a list of ESBMTK instance (e.g., reservoirs)

optional keywords: fn = filename, defaults to the Model name

Example:

```
M.plot([sb.PO4, sb.DIC], fn='test.pdf')
```

will plot sb.PO4 and sb.DIC and save the plot as 'test.pdf'

post_process_data(*results*) → None

Map solver results back into esbmtk structures

Parameters
results – numpy arrays with solver results

read_data(*directory='./data'*) → None

Save the model results to a CSV file. Each reservoir will have their own CSV file

read_state(*directory='state'*)

This will initialize the model with the result of a previous model run. For this to work, you will need issue a Model.save_state() command at then end of a model run. This will create the necessary data files to initialize a subsequent model run.

restart()

Restart the model with result of the last run. This is useful for long runs which otherwise would used to much memory

run(***kwargs*) → None

Loop over the time vector, and for each time step, calculate the fluxes for each reservoir

save_data(*directory='./data'*) → None

Save the model results to a CSV file. Each reservoir will have their own CSV file

Calling save_data() without any arguments, will create (or recreate) the data directory in the current working directory which will then be populated by csv-files

Parameters
directory – a string with the directory name. It defaults to 'data'

save_state(*directory='state'*) → None

Save model state. Similar to save data, but only saves the last 10 time-steps

sub_sample_data()

Subsample the data. No need to save 100k lines of data You need to do this _after_ saving the state, but before plotting and saving the data

test_d_pH(*rg:* Species, *time: ndarray[Any, dtype[float64]]*) → None

Test if the change in pH exceeds more than 0.01 units per time step. Note that this is only a crude test, since the solver interpolates between intergration steps. So this may not catch all problems.

Parameters

- **rg** (Reservoir) – Reservoir instance
- **time** (*: NDArrayFloat*) – time vector as returned by the solver

exception esbmtk.esbmtk.**ModelError**(*message*)

Bases: Exception

exception esbmtk.esbmtk.**ReservoirError**(*message*)

Bases: Exception

exception esbmtk.esbmtk.**ScaleError**(*message*)

Bases: Exception

class esbmtk.esbmtk.**Sink**(***kwargs*)

Bases: *SourceSink*

This is a meta class to setup a Source/Sink objects. These are not actual reservoirs, but we stil need to have them as objects Example:

```
Sink(name = "Pyrite",
    species = SO4,
    display_precision = number, optional, inherited from Model
    delta = number or str. optional defaults to "None"
    register = Model handle
)
```

class esbmtk.esbmtk.**Source**(***kwargs*)

Bases: *SourceSink*

This is a meta class to setup a Source/Sink objects. These are not actual reservoirs, but we stil need to have them as objects Example:

```
Ssource(name = "weathering",
    species = SO4,
    display_precision = number, optional, inherited from Model
    delta = number or str. optional defaults to "None"
    register = Model handle
)
```

class esbmtk.esbmtk.**SourceSink**(***kwargs*)

Bases: *esbmtkBase*

This is a meta class to setup a Source/Sink objects. These are not actual reservoirs, but we stil need to have them as objects Example:

```
Sink(name = "Pyrite",
    species = SO4,
    display_precision = number, optional, inherited from Model
    delta = number or str. optional defaults to "None"
    register = Model handle
)
```

property delta

class esbmtk.esbmtk.**Species**(***kwargs*)

Bases: *SpeciesBase*

This object holds reservoir specific information.

Example:

```
Species(name = "foo",      # Name of reservoir
          species = S,          # SpeciesProperties handle
          delta = 20,           # initial delta - optional (defaults  to 0)
          mass/concentration = "1 unit"  # species concentration or mass
          volume/geometry = "1E5 l",      # reservoir volume (m^3)
          plot = "yes"/"no", defaults to yes
          plot_transform_c = a function reference, optional (see below)
          legend_left = str, optional, useful for plot transform
          display_precision = number, optional, inherited from Model
          register = Model instance
          isotopes = True/False otherwise use Model.m_type
          seawater_parameters= dict, optional
          )
```

You must either give mass or concentration. The result will always be displayed as concentration though.

You must provide either the volume or the geometry keyword. In the latter case provide a list where the first entry is the upper depth datum, the second entry is the lower depth datum, and the third entry is the total ocean area. E.g., to specify the upper 200 meters of the entire ocean, you would write:

geometry=[0,-200,3.6e14]

the corresponding ocean volume will then be calculated by the calc_volume method in this case the following instance variables will also be set:

self.volume in model units (usually liter) self.are:a surface area in m^2 at the upper bounding surface self.sed_area: area of seafloor which is intercepted by this box. self.area_fraction: area of seafloor which is intercepted by this relative to the total ocean floor area

It is also possible to specify volume and area explicitly. In this case provide a dictionary like this:

```
geometry = {"area": "1e14 m**2", # surface area
            "volume": "3e16 m**3", # box volume
           }
```

Adding seawater_properties:

If this optional parameter is specified, a SeaWaterConstants instance will be registered for this Species as Species.swc See the SeaWaterConstants class for details how to specify the parameters, e.g.:

```
seawater_parameters = {"temperature": 2,
                       "pressure": 240,
                       "salinity" : 35,
                      }
```

Using a transform function:

In some cases, it is useful to transform the reservoir concentration data before plotting it. A good example is the H+ concentration in water which is better displayed as pH. We can do this by specifying a function to convert the reservoir concentration into pH units:

```
.. code-block:: python
```

def phc(c :float) -> float:
    # Calculate concentration as pH. c can be a number or numpy array import numpy as np pH :float = -np.log10(c) return pH

this function can then be added to a reservoir as:

hplus.plot_transform_c = phc

You can modify the left legend to suit the transform via the legend_left keyword

Note, at present the plot_transform_c function will only take one argument, which always defaults to the reservoir concentration. The function must return a single argument which will be interpreted as the transformed reservoir concentration.

Accesing Species Data:

You can access the reservoir data as:

- Name.m # mass
- Name.d # delta
- Name.c # concentration

Useful methods include:

- Name.write_data() # save data to file
- Name.info() # info Species

property concentration: *float*

property delta: *float*

property mass: *float*

class esbmtk.esbmtk.**SpeciesBase**(***kwargs*)

Bases: *esbmtkBase*

Base class for all Species objects

get_plot_format()

Return concentrat data in plot units

info(***kwargs*) → None

Show an overview of the object properties. Optional arguments are

Parameters

- **index** – int = 0 # this will show data at the given index
- **indent** – int = 0 # print indentation

class esbmtk.esbmtk.**SpeciesProperties**(***kwargs*)

Bases: *esbmtkBase*

Each model, can have one or more species. This class sets species specific properties

Example:

```
SpeciesProperties(name = "SO4",
        element = S,
```

)

Defaults:

```
self.defaults: dict[any, any] = {
    "name": ["None", (str)],
    "element": ["None", (ElementProperties, str)],
    "display_as": [kwargs["name"], (str)],
    "m_weight": [0, (int, float, str)],
    "register": ["None", (Model, ElementProperties, Species, GasReservoir)],
    "parent": ["None", (Model, ElementProperties, Species, GasReservoir)],
    "flux_only": [False, (bool)],
    "logdata": [False, (bool)],
    "scale_to": ["None", (str)],
    "stype": ["concentration", (str)],
}
```

Required keywords: “name”, “element”

esbmtk.esbmtk_base module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

This module defines some shared methods

exception esbmtk.esbmtk_base.**FluxSpecificationError**(*message*)

Bases: Exception

exception esbmtk.esbmtk_base.**InputError**(*message*)

Bases: Exception

exception esbmtk.esbmtk_base.**KeywordError**(*message*)

Bases: Exception

exception esbmtk.esbmtk_base.**MissingKeywordError**(*message*)

Bases: Exception

exception esbmtk.esbmtk_base.**SpeciesPropertiesMolweightError**(*message*)

Bases: Exception

class esbmtk.esbmtk_base.**esbmtkBase**

Bases: *input_parsing*

The esbmtk base class template. This class handles keyword arguments, name registration and other common tasks

Useful methods in this class:

define required keywords in lrk dict:
self.lrk: list = [“name”]

define allowed type per keyword in lkk dict:

self.defaults: dict[str, list[any, tuple]] = {
“name”: [“None”, (str)], “model”: [“None”,(str, Model)], “salinity”: [35, (int, float)], # int or float }

parse and register all keywords with the instance self.__initialize_keyword_variables__(kwargs)

register the instance self.__register_name_new__ ()

ensure_q(*arg*)

Test that a given input argument is a quantity. If not convert into quantity

help() → None

Show all keywords, their fdefault values and allowed types.

info(**kwargs*) → None

Show an overview of the object properties. Optional arguments are

indent :int = 0 indentation

set_flux(*mass: str*, *time: str*, *substance:* SpeciesProperties)

set_flux converts() a flux rate that is specified as rate, time, substance so that it matches the correct model units (i.e., kg/s or mol/s)

Example:

```
M.set_flux("12 Tmol", "year", M.C)
```

if model mass units are in mol, no change will be made if model mass units are in kg, the above will return kg C/a (and vice verso)

Parameters

- **mass** – e.g., "12 Tmol"
- **time** – e.g., "year"
- **substance** – e.g., SpeciesProperties Instance e.g., M.PO4

Returns
mol/year or g/year

Raises
FluxSpecificationError

Raises
SpeciesPropertiesMolweightError

class esbmtk.esbmtk_base.**input_parsing**

Bases: object

Provides various routines to parse and process keyword arguments. All derived classes need to declare the allowed keyword arguments, their defualt values and the type in the following format:

defaults = {"key": [value, (allowed instances)]

the recommended sequence is to first set default values via __register_variable_names__()

__update_dict_entries__(defaults,kwargs) will compare the provided kwargs against this data, and upon successful parsing update the default dict with the new values

esbmtk.extended_classes module

class esbmtk.extended_classes.**DataField**(***kwargs: dict[str, any]*)

Bases: *esbmtkBase*

DataField: Datafields can be used to plot data which is computed after the model finishes in the overview plot windows. Therefore, datafields will plot in the same window as the reservoir they are associated with. Datafields must share the same x-axis is the model, and can have up to two y axis.

Example:

```
DataField(name = "Name"
          register = Model handle,
          y1_data = NDArrayFloat or list of arrays
          y1_label = Data label(s)
          y1_legend = Y-Axis Label
          y1_type = "plot", | "scatter"
          y2_data = NDArrayFloat    # optional
          y2_legend = Y-Axis label # optional
```

(continues on next page)

(continued from previous page)

```
        y2_label = Data legend(s) # optional
        y2_type = "plot", | "scatter"
        common_y_scale = "no",  #optional, default "no"
        display_precision = number, optional, inherited from Model
        )
```

Note that Datafield data is not mapped to model units. Care must be taken that the data units match the model units.

The instance provides the following data

Name.x = X-axis = model X-axis Name.y1_data Name.y1_label Name.y1_legend

Similarly for y2

You can specify more than one data set, and be explicit about color and linestyle choices.

Example:

```
DataField(
        name="df_pH",
        x1_data=[M.time, M.time, M.time, M.ef_hplus_l.x, M.ef_hplus_h.x, M.ef_hplus_
↪d.x],
        y1_data=[
        -np.log10(M.L_b.Hplus.c),
        -np.log10(M.H_b.Hplus.c),
        -np.log10(M.D_b.Hplus.c),
        -np.log10(M.ef_hplus_l.y),
        -np.log10(M.ef_hplus_h.y),
        -np.log10(M.ef_hplus_d.y),
        ],
        y1_label="Low latitude, High latitude, Deep box, d_L, d_H, d_D".split(", "),
        y1_color="C0 C1 C2 C0 C1 C2".split(" "),
        y1_style="solid solid solid dotted dotted dotted".split(" "),
        y1_legend="pH",
        register=M,
        )
```

exception esbmtk.extended_classes.**DataFieldError**(*message*)

Bases: Exception

exception esbmtk.extended_classes.**ESBMTKFunctionError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**ExternalCode**(***kwargs*)

Bases: *SpeciesNoSet*

This class can be used to implement user provided functions. The data inside a VR_no_set instance will only change in response to a user provided function but will otherwise remain unaffected. That is, it is up to the user provided function to manage changes in reponse to external fluxes. A VR_no_set is declared in the following way:

```
ExternalCode(
            name="cs",      # instance name
            species=CO2,    # species, must be given
            # the vr_data_fields contains any data that is referenced inside the
```

(continues on next page)

(continued from previous page)

```
            # function, rather than passed as argument, and all data that is
            # explicitly referenced by the model
            vr_datafields :dict ={"Hplus": self.swc.hplus,
                                  "Beta": 0.0},
            function=calc_carbonates, # function reference, see below
            fname = function name as string
            function_input_data="DIC TA",
            # Note that parameters must be individual float values
            function_params:tuple(float)
            # list of return values
            return_values={  # these must be known speces definitions
                         "Hplus": rg.swc.hplus,
                          "zsnow": float(abs(kwargs["zsnow"])),
                          },
            register=rh # reservoir_handle to register with.
        )

the dict keys of vr_datafields will be used to create alias
names which can be used to access the respective variable
```

The general template for a user defined function is a follows:

```
def calc_carbonates(i: int, input_data: list, vr_data: list, params: list) -> None:
    # i = index of current timestep
    # input_data = list of np.arrays, typically data from other Species
    # vr_data = list of np.arrays created during instance creation (i.e. the vr␣
↪data)
    # params = list of float values (at least one!)
    pass
 return
```

Note that this function should not return any values, and that all input fields must have at least one entry!

append(***kwargs*) → None

This method allows to update GenericFunction parameters after the VirtualSpecies has been initialized. This is most useful when parameters have to reference other virtual reservoirs which do not yet exist, e.g., when two virtual reservoirs have a circular reference.

Example:

```
VR.update(a1=new_parameter, a2=new_parameter)
```

create_alialises() → None

Register alialises for each vr_datafield

update_parameter_count()

class esbmtk.extended_classes.**ExternalData**(***kwargs: dict[str, str]*)

Bases: *esbmtkBase*

Instances of this class hold external X/Y data which can be associated with a reservoir.

Example:

```
ExternalData(name       = "Name"
             filename   = "filename",
             legend     = "label",
             offset     = "0 yrs",
             reservoir  = reservoir_handle,
             scale      = scaling factor, optional
             display_precision = number, optional, inherited from Model
             convert_to = optional, see below
            )
```

The data must exist as CSV file, where the first column contains the X-values, and the second column contains the Y-values.

The x-values must be time and specify the time units in the header between square brackets They will be mapped into the model time units.

The y-values can be any data, but the user must take care that they match the model units defined in the model instance. So your data file mujst look like this

Time [years], Data [units], Data [units] 1, 12 2, 13

By convention, the secon column should contaain the same type of data as the reservoir (i.e., a concentration), whereas the third column contain isotope delta values. Columns with no data should be left empty (and have no header!) The optional scale argument, will only affect the Y-col data, not the isotope data

The column headers are only used for the time or concentration data conversion, and are ignored by the default plotting methods, but they are available as self.xh,yh

The file must exist in the local working directory.

the convert_to keyword can be used to force a specific conversion. The default is to convert into the model concentration units.

- name.plot()

Data:

- name.x
- name.y
- name.df = dataframe as read from csv file

plot() → None

Plot the data and save a pdf

Example:

```
ExternalData.plot()
```

exception esbmtk.extended_classes.**ExternalDataError**(*message*)

Bases: Exception

exception esbmtk.extended_classes.**FluxError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**GasReservoir**(***kwargs*)

Bases: *SpeciesBase*

This object holds reservoir specific information similar to the Species class

Example:

```
Species(name = "foo",     # Name of reservoir
          species = CO2,    # SpeciesProperties handle
          delta = 20,       # initial delta - optional (defaults  to 0)
          reservoir_mass = quantity # total mass of all gases
                           defaults to 1.833E20 mol
          species_ppm =  number # concentration in ppm
          plot = "yes"/"no", defaults to yes
          plot_transform_c = a function reference, optional (see below)
          legend_left = str, optional, useful for plot transform
          display_precision = number, optional, inherited from Model
          register = optional, use to register with Reservoir Group
          isotopes = True/False otherwise use Model.m_type
          )
```

Accesing Species Data:

You can access the reservoir data as:

- Name.m # species mass
- Name.l # mass of light isotope
- Name.d # species delta (only avaible after M.get_delta_values()
- Name.c # partial pressure
- Name.v # total gas mass

Useful methods include:

- Name.write_data() # save data to file
- Name.info() # info Species

exception esbmtk.extended_classes.**GasResrvoirError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**Reservoir**(***kwargs*)

Bases: *esbmtkBase*

This class allows the creation of a group of reservoirs which share a common volume, and potentially connections. E.g., if we have twoy reservoir groups with the same reservoirs, and we connect them with a flux, this flux will apply to all reservoirs in this group.

A typical examples might be ocean water which comprises several species. A reservoir group like ShallowOcean will then contain sub-reservoirs like DIC in the form of ShallowOcean.DIC

Example:

```
Reservoir(name = "ShallowOcean",         # Name of reservoir group
            volume/geometry = "1E5 l",       # see below
            delta   = {DIC:0, TA:0, PO4:0]  # dict of delta values
            mass/concentration = {DIC:"1 unit", TA: "1 unit"}
            plot = {DIC:"yes", TA:"yes"}  defaults to yes
            isotopes = {DIC: True/False} see Species class for details
            seawater_parameters = dict, optional, see below
```

(continues on next page)

(continued from previous page)

```
            register= model handle, required
        )
```

Notes: The subreservoirs are derived from the keys in the concentration or mass
dictionary. Toward this end, the keys must be valid species handles and – not species names – !

Connecting two reservoir groups requires that the names in both group match, or that you specify a dictionary which delineates the matching.

Most parameters are passed on to the Species class. See the reservoir class documentation for details

The geometry keyword specifies the upper depth interval, the lower depth interval, and the fraction of the total ocean area inhabited by the reservoir

If the geometry parameter is supplied, the following instance variables will be computed:

- self.volume: in model units (usually liter)
- self.area: surface area in m^2 at the upper bounding surface
- self.sed_area: area of seafloor which is intercepted by this box.
- self.area_fraction: area of seafloor which is intercepted by this relative to the total ocean floor area

seawater_parameters:

If this optional parameter is specified, a SeaWaterConstants instance will be registered for this Species as Species.swc See the SeaWaterConstants class for details how to specify the parameters, e.g.:

```
seawater_parameters = {"temperature": 2,
                       "pressure": 240,
                       "salinity" : 35,
                       },
```

exception esbmtk.extended_classes.**ReservoirError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**Signal**(***kwargs*)

Bases: *esbmtkBase*

This class will create a signal which is described by its startime (relative to the model time), it's size (as mass) and duration, or as duration and magnitude. Furthermore, we can presribe the signal shape (square, pyramid, bell, file)and whether the signal will repeat. You can also specify whether the event will affect the delta value.

The default is to add the signal to a given connection. It is however also possible to use the signal data as a scaling factor.

Example:

```
Signal(name = "Name",
       species = SpeciesProperties handle,
       start = "0 yrs",      # optional
       duration = "0 yrs",   #
       delta = 0,            # optional
       stype = "addition"    # optional, currently the only type
       shape = "square/pyramid/bell/filename"
```

(continues on next page)

(continued from previous page)

```
      mass/magnitude/filename  # give one
      offset = '0 yrs',     #
      scale = 1, optional,  #
      offset = option #
      reservoir = r-handle # optional, see below
      source = s-handle optional, see below
      display_precision = number, optional, inherited from Model
      register,
     )
```

Signals are cumulative, i.e., complex signals ar created by adding one signal to another (i.e., Snew = S1 + S2)

The optional scaling argument will only affect the y-column data of external data files

Signals are registered with a flux during flux creation, i.e., they are passed on the process list when calling the connector object.

if the filename argument is used, you can provide a filename which contains the data to be used in csv format. The data will be interpolated to the model domain, and added to the already existing data. The external data need to be in the following format

Time, Rate, delta value 0, 10, 12

i.e., the first row needs to be a header line

All time data in the csv file will be treated as realative time (i.e., the start time will be mapped to zero). Use the offset keyword to shift the external signal data in the time domain.

Last but not least, you can provide an optional reservoir name. In this case, the signal will create a source as (signal_name_source) and the connection to the specified reservoir. If you build a complex signal do this as the last step. If you additionally provide a source name the connection will be made between the provided source (this can be useful if you use source groups).

This class has the following methods

Signal.repeat() Signal.plot() Signal.info()

repeat(*start*, *stop*, *offset*, *times*) → None

This method creates a new signal by repeating an existing signal. Example:

```
new_signal = signal.repeat(start,   # start time of signal slice to be repeated
                           stop,    # end time of signal slice to be repeated
                           offset,  # offset between repetitions
                           times,   # number of time to repeat the slice
                           )
```

exception esbmtk.extended_classes.**SignalError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**SinkProperties**(***kwargs*)

Bases: *SourceSinkProperties*

This is just a wrapper to setup a Sink object Example:

```
SinkProperties(name = "Burial",
     species = [SO42, H2S],
     delta = {"SO4": 10}
     )
```

class esbmtk.extended_classes.**SourceProperties**(***kwargs*)

Bases: *SourceSinkProperties*

This is just a wrapper to setup a Source object Example:

```
SourceProperties(name = "weathering",
        species = [SO42, H2S],
        delta = {"SO4": 10}
        )
```

class esbmtk.extended_classes.**SourceSinkProperties**(***kwargs*)

Bases: *esbmtkBase*

This is a meta class to setup Source/Sink Groups. These are not actual reservoirs, but we stil need to have them as objects Example:

```
SinkProperties(name = "Pyrite",
     species = [SO42, H2S],
     )
```

where the first argument is a string, and the second is a reservoir handle

exception esbmtk.extended_classes.**SourceSinkPropertiesError**(*message*)

Bases: Exception

class esbmtk.extended_classes.**SpeciesNoSet**(***kwargs*)

Bases: *SpeciesBase*

This class is similar to a regular reservoir, but we make no assumptions about the type of data contained. I.e., all data will be left alone

class esbmtk.extended_classes.**VectorData**(***kwargs: dict[str, any]*)

Bases: *esbmtkBase*

get_plot_format()

Return concentrat data in plot units

class esbmtk.extended_classes.**VirtualSpecies**(***kwargs*)

Bases: *Species*

A virtual reservoir. Unlike regular reservoirs, the mass of a virtual reservoir depends entirely on the return value of a function.

Example:

```
VirtualSpecies(name="foo",
            volume="10 liter",
            concentration="1 mmol",
            species=  ,
            function=bar,
            a1 to a3 =  to 3optional function arguments,
            display_precision = number, optional, inherited from Model,
            )
```

the concentration argument will be used to initialize the reservoir and to determine the display units.

The function definition follows the GenericFunction class. which takes a generic function and up to 6 optional function arguments, and will replace the mass value(s) of the given reservoirs with whatever the function calculates. This is particularly useful e.g., to calculate the pH of a given reservoir as function of e.g., Alkalinity and

DIC. :param - name = name of process: :param : :param - act_on = name of a reservoir this process will act upon: :param - function = a function reference: :param - a1 to a3 function arguments:

The function must return a list of numbers which correspond to the data which describe a reservoir i.e., mass, light isotope, heavy isotope, delta, and concentration

In order to use this function we need first declare a function we plan to use with the generic function process. This function needs to follow this template:

```
def my_func(i, a1, a2, a3) -> tuple:
    #
    # i = index of the current timestep
    # a1 to a3 =  optional function parameter. These must be present,
    # even if your function will not use it See above for details

    # calc some stuff and return it as

    return [m, l, h, d, c] # where m= mass, and l & h are the respective
                           # isotopes. d denotes the delta value and
                           # c the concentration
                           # Use dummy value as necessary.
```

This class provides an update method to resolve cases where e.g., two virtual reservoirs have a circular reference. See the documentation of update().

update(***kwargs*) → None

This method allows to update GenericFunction parameters after the VirtualSpecies has been initialized. This is most useful when parameters have to reference other virtual reservoirs which do not yet exist, e.g., when two virtual reservoirs have a circular reference.

Example:

```
VR.update(a1=new_parameter, a2=new_parameter)
```

class esbmtk.extended_classes.**VirtualSpeciesNoSet**(***kwargs*)

Bases: *ExternalCode*

Alias to ensure backwards compatibility

esbmtk.ode_backend module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

esbmtk.ode_backend.**check_isotope_effects**(*f_m: str*, *c:* Species2Species, *icl: dict*, *ind3: str*, *ind2: str*) → str

Test if the connection involves any isotope effects

Parameters

- **f_m** – string with the flux name
- **c** – connection object
- **icl** – dict of reservoirs that have actual fluxes
- **ind2** – indent 2 times
- **ind3** – indent 3 times

Returns eq
equation string

esbmtk.ode_backend.**check_signal_2**(*ex: str*, *exl: str*, *c:* Species2Species)

Test if connection is affected by a signal

Parameters

- **ex** – equation string
- **c** – connection object

Returns
(modified) equation string

esbmtk.ode_backend.**get_flux**(*flux:* Flux, *M:* Model, *R: list[float]*, *icl: dict*)

Create formula expressions that calcultes the flux F. Return the equation expression as string

Parameters

- **flux** – The flux object for which we create the equation
- **M** – The current model object
- **R** – The list of initial conditions for each reservoir
- **icl** – dict of reservoirs that have actual fluxes

Returns
A tuple where the first string is the equation for the total flux, and the second string is the equation for the flux of the light isotope

esbmtk.ode_backend.**get_ic**(*r:* Species, *icl: dict*, *isotopes=False*) → str

Get initial condition in a reservoir. If the reservoir is icl, return index expression into R.c. If the reservoir is not in the index, return the Species concentration a t=0

In both cases return these a string

If isotopes == True, return the pointer to the light isotope concentration

Parameters

- **r** – A reservoir handle
- **icl** – icl = dict[Species, list[int, int]] where reservoir indicates the reservoir handle, and the list contains the index into the reservoir data. list[0] = concentration list[1] concentration of the light isotope.

Raises
ValueError – get_ic: can't find {r.full_name} in list of initial conditions

Returns
the string s which is the full_name of the reservoir concentration or isotope concentration

esbmtk.ode_backend.**get_initial_conditions**(*M:* Model, *rtol: float*, *atol_d: float = 1e-07*) → tuple[list, dict, list, list, NDArrayFloat]

Get list of initial conditions. This list needs to match the number of equations.

Parameters

- **Model** – The model handle
- **rtol** – relative tolerance for BDF solver.
- **atol_d** – default value for atol if c = 0

Returns
R = list of initial conditions as floats

Returns
icl = dict[Species, list[int, int]] where reservoir indicates the reservoir handle, and the list contains the index into the reservoir data. list[0] = concentration list[1] concentration of the light isotope.

Returns
cpl = list of reservoirs that use function to evaluate reservoir data

Returns
ipl = list of static reservoirs that serve as input

Returns

rtol = array of tolerence values for ode solver

We need to consider 3 types of reservoirs:

1) Species that change as a result of physical fluxes i.e. r.lof > 0. These require a flux statements and a reservoir equation.

2) Species that do not have active fluxes but are computed as a tracer, i.e.. HCO3. These only require a reservoir equation

3) Species that do not change but are used as input. Those should not happen in a well formed model, but we cannot exclude the possibility. In this case, there is no flux equation, and we state that dR/dt = 0

get_ic() will look up the index position of the reservoir_handle on icl, and then use this index to retrieve the correspinding value in R

Isotopes are handled by adding a second entry

esbmtk.ode_backend.**get_regular_flux_eq**(*flux:* Flux, *c:* Species2Species, *icl: dict*, *ind2*, *ind3*) → tuple

Create a string containing the equation for a regular (aka fixed rate) connection

Parameters

- **flux** – flux instance
- **c** – connection object
- **icl** – dict of reservoirs that have actual fluxes
- **ind2** – indent 2 times
- **ind3** – indent 3 times

Returns
two strings, where the first describes the equation for the total flux, and the second describes the rate for the light isotope

esbmtk.ode_backend.**get_scale_with_concentration_eq**(*flux:* Flux, *c:* Species2Species, *cfn: str*, *icl: dict*, *ind2: str*, *ind3: str*)

Create equation string defining a flux that scales with the concentration in the upstream reservoir

Example: M1_CG_D_b_to_L_b_TA_thc__F = M1.CG_D_b_to_L_b.TA_thc.scale * R[5]

Parameters

- **flux** – Flux object
- **c** – connection instance
- **cfn** – full name of the connection instance
- **icl** – dict[Species, list[int, int]] where reservoir indicates the reservoir handle, and the list contains the index into the reservoir data. list[0] = concentration list[1] concentration of the light isotope.

Returns

two strings with the respective equations for the change in the total reservoir concentration and the concentration of the light isotope

esbmtk.ode_backend.**get_scale_with_flux_eq**(*flux:* Flux, *c:* Species2Species, *cfn: str*, *icl: dict*, *ind2: str*, *ind3: str*)

Equation defining a flux that scales with strength of another flux. If isotopes are used, use the isotope ratio of the upstream reservoir.

Parameters

- **flux** – Flux object
- **c** – connection instance
- **cfn** – full name of the connection instance
- **icl** – dict[Species, list[int, int]] where reservoir indicates the reservoir handle, and the list contains the index into the reservoir data. list[0] = concentration list[1] concentration of the light isotope.

Returns

two strings with the respective equations for the change in the total reservoir concentration and the concentration of the light isotope

esbmtk.ode_backend.**parse_esbmtk_input_data_types**(*d: any*, *r:* Species, *ind: str*, *icl: dict*) → str

Parse esbmtk data types that are provided as arguments to external function objects, and convert them into a suitable string format that can be used in the ode equation file

esbmtk.ode_backend.**parse_function_params**(*params*, *ind*) → str

Parse function_parameters and convert them into a suitable string format that can be used in the ode equation file

esbmtk.ode_backend.**write_ef**(*eqs*, *ef:* Species | *ExternalFunction*, *icl: dict*, *rel: str*, *ind2: str*, *ind3: str*, *gpt: tuple*) → str

Write external function call code

Parameters

- **eqs** – equation file handle
- **ef** – external_function handle
- **icl** – dict of reservoirs that have actual fluxes
- **rel** – string with reservoir names returned by setup_ode

- **ind2** – indent 2 times
- **ind3** – indent 3 times
- **gpt** – tuple with global paramaters

Returns
rel: modied string of reservoir names

esbmtk.ode_backend.**write_equations_2**(*M:* Model, *R: list[float]*, *icl: dict*, *cpl: list*, *ipl: list*) → tuple

Write file that contains the ode-equations for the Model Returns the list R that contains the initial condition for each reservoir

Parameters

- **Model** – Model handle
- **R** – list of floats with the initial conditions for each reservoir
- **icl** – dict of reservoirs that have actual fluxes
- **cpl** – list of reservoirs that have no fluxes but are computed based on other reservoirs
- **ipl** – list of reservoir that do not change in concentration

esbmtk.ode_backend.**write_reservoir_equations**(*eqs*, *M:* Model, *rel: str*, *ind2: str*, *ind3: str*) → str

Loop over reservoirs and their fluxes to build the reservoir equation

Parameters

- **eqs** – equation file handle
- **rel** – string with reservoir names used in return function. Note that these are the reervoir names as used by the equations and not the reservoir names used by esbmtk. E.g., M1.R1.O2 will be M1_R1_O2,
- **ind2** – string with indentation offset
- **ind3** – string with indentation offset

Returns
rel = updated list of reservoirs names

esbmtk.ode_backend.**write_reservoir_equations_with_isotopes**(*eqs*, *M:* Model, *rel: str*, *ind2: str*, *ind3: str*) → str

Loop over reservoirs and their fluxes to build the reservoir equation

esbmtk.post_processing module

esbmtk.post_processing.**carbonate_system_1_pp**(*box_names: SpeciesGroup*) → None

Calculates and returns various carbonate species based on previously calculated Hplus, TA, and DIC concentrations.

LIMITATIONS: - Assumes all concentrations are in mol/kg - Assumes your Model is in mol/kg ! Otherwise, DIC and TA updating will not be correct.

Calculations are based off equations from: Boudreau et al., 2010, https://doi.org/10.1029/2009GB003654 Follows, 2006, doi:10.1016/j.ocemod.2005.05.004

Parameters
rg – A reservoirgroup object with initialized carbonate system

esbmtk.post_processing.**carbonate_system_2_pp**(*bn:* Reservoir | *list*, *export_fluxes:* *float* | *list*, *zsat_min:* *float* = *200*, *zmax:* *float* = *6000*) → None

Calculates and returns the fraction of the carbonate rain that is dissolved an returned back into the ocean.

Parameters

- **rg** – Reservoir, e.g., M.D_b
- **export** – export flux in mol/year
- **zsat_min** – depth of mixed layer
- **zmax** – depth of lookup table

returns:

DIC_burial, DIC_burial_l, Hplus, zsnow

Additionally, it calculates the following critical depth intervals:

zsat: top of lysocline zcc: carbonate compensation depth

LIMITATIONS: - Assumes all concentrations are in mol/kg - Assumes your Model is in mol/kg ! Otherwise, DIC and TA updating will not be correct.

Calculations are based off equations from: Boudreau et al., 2010, https://doi.org/10.1029/2009GB003654 Follows, 2006, doi:10.1016/j.ocemod.2005.05.004

esbmtk.post_processing.**gas_exchange_fluxes**(*liquid_reservoir:* Species, *gas_reservoir:* GasReservoir, *pv:* *str*)

Calculate gas exchange fluxes for a given reservoir

Parameters

- **liquid_reservoir** – Species handle
- **gas_reservoir** – Species handle
- **pv** – piston velocity as string e.g., "4.8 m/d"

Returns

esbmtk.processes module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020-2021 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

esbmtk.processes.**gas_exchange**(*gas_c:* *float* | *tuple*, *liquid_c:* *float* | *tuple*, *gas_aq:* *float*, *p:* *tuple*) → float | tuple

Calculate the gas exchange flux across the air sea interface for co2 including isotope effects.

Parameters

- **gas_c** (*float* | *tuple*) – gas concentration in atmosphere

- **liquid_c** (*float | tuple*) – reference species in liquid phase, e.g., DIC
- **gas_aq** (*float*) – dissolved gas concentration, e.g., CO2aq
- **p** (*tuple*) – parameters, see init_gas_exchange

Returns

- *float | tuple* – gas flux across the air/sea interface
- *Note that the sink delta is co2aq as returned by the carbonate VR*
- *this equation is for mmol but esbmtk uses mol, so we need to*
- *multiply by 1E3*
- *The Total flux across interface dpends on the difference in either*
- *concentration or pressure the atmospheric pressure is known, as gas_c, and*
- *we can calculate the equilibrium pressure that corresponds to the dissolved*
- *gas in the water as [CO2]aq/beta.*
- *Conversely, we can convert the the pCO2 into the amount of dissolved CO2 =*
- *pCO2 * beta*
- *The h/c ratio in HCO3 estimated via h/c in DIC. Zeebe writes C12/C13 ratio*
- *but that does not work. the C13/C ratio results however in -8 permil*
- *offset, which is closer to observations*

esbmtk.processes.**init_gas_exchange**(*c:* Species2Species)

Create an ExternalCode instance for gas exchange reactions

Parameters

c (Species2Species) – connection instance

esbmtk.processes.**init_weathering**(*c:* Species2Species, *pco2: float*, *pco2_0: float | str | Q_*, *area_fraction: float*, *ex: float*, *f0: float | str | Q_*)

Creates a new external code instance

Parameters

- **c** – Species2Species
- **pco2** – float current pco2
- **pco2_0** – float reference pco2
- **ex** – exponent

Area_fraction

float area/total area

F0

flux at pco2_0

esbmtk.processes.**weathering**(*c_pco2: float | list[float]*, *p: tuple*) → float | tuple

Calculate weathering as a function of pCO2

Parameters

- **c_pco2** (*float | list[float]*) – current pCO2 concentration

- **p** (*tuple*) – a tuple with the following entries: pco2_0 = reference pCO2 area_fraction = fraction of total surface area ex = exponent used in the equation f0 = flux at the reference value isotopes = True/False

Returns

- *float | tuple* – a float or list value for the weathering flux
- *Explanation*
- ———
- *If the model uses isotopes, the function expects the concentration*
- *values for hthe total mass and the light isotope as a list, and*
- *will simiraly return the flux as a list of total flux and flux of*
- *the light isotope.*
- *The flux itself is calculated as* – F_w = area_fraction * f0 * (pco2/pco2_0)**ex

esbmtk.sealevel module

esbmtk.sealevel

Classes which provide access to hypsometric data

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

esbmtk.sealevel.**get_box_geometry_parameters**(*box*, *fraction=1*) → None

Calculate box volume and area from the data in box.

Parameters
box – list or dict with the geometry parameters

Fraction
0 to 1 to specify a fractional part (i.e., Atlantic)

If box is a list the first entry is the upper depth datum, the second entry is the lower depth datum, and the third entry is the total ocean area. E.g., to specify the upper 200 meters of the entire ocean, you would write:

geometry=[0,-200,3.6e14]

the corresponding ocean volume will then be calculated by the calc_volume method in this case the following instance variables will also be set:

- self.volume in model units (usually liter)
- self.are:a surface area in m^2 at the upper bounding surface
- self.sed_area: area of seafloor which is intercepted by this box.
- self.area_fraction: area of seafloor which is intercepted by this relative to the total ocean floor area

It is also possible to specify volume and area explicitly. In this case provide a dictionary like this:

```
box = {"area": "1e14 m**2", # surface area in m**2
       "volume": "3e16 m**3", # box volume in m**3
       "ta": "4e16 m**2", # reference area
      }
```

class esbmtk.sealevel.**hypsometry**(***kwargs*)

Bases: *esbmtkBase*

A class to provide hypsometric data for the depth interval between -6000 to 1000 meter (relative to sealevel)

Invoke as:
hyspometry(name="hyp")

area(*elevation: int*) → float

Calculate the ocean area at a given depth

Parameters
elevation (*int*) – Elevation datum in meters

Returns
area in m^2

Return type
float

area_dz(*u: float*, *l: float*) → float

calculate the area between two elevation datums

Parameters

- **u** (*float*) – upper elevation datum in meters (relative to sealevel)
- **l** (*float*) – lower elevation datum relative to sealevel

Returns
area in m^2

Return type
float

Raises
ValueError – if elevation datums are outside the defined interval

get_lookup_table_area() → ndarray[Any, dtype[float64]]

Return the area values between 0 and max_depth as 1-D array

get_lookup_table_area_dz() → ndarray[Any, dtype[float64]]

Return the are_dz values between 0 and max_depth as 1-D array

read_data() → None

Read the hypsometry data from a pickle file. If the pickle file is missing, create it from the csv data

save the hypsometry data as a numpy array with elevation, area, and area_dz in self.hypdata

show_data()

Provide a diagnostic graph that shows the hypsometric data use by ESBMTK

volume(*u: float*, *l: float*) → float

Calculate the area between two elevation datums

Parameters

- **u** (*float*) – upper elevation datum in meters (relative to sealevel)
- **l** (*float*) – lower elevation datum relative to sealevel

Returns
volume in m^3

Return type
float

Raises
ValueError – if elevation datums are outside the defined interval

esbmtk.seawater module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020-2021 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

class esbmtk.seawater.**SeawaterConstants**(***kwargs: dict[str, str]*)

Bases: *esbmtkBase*

Provide basic seawater properties as a function of T, P and Salinity. Since we cannot know if TA and DIC have already been specified, creating the instance uses standard seawater composition. Updating/Setting TA & DIC does not recalculate these values after initialization, unless you explicitly call the update_parameters() method.

Example:

```
Seawater(name="SW",
         register=M # model handle
         temperature = optional in C, defaults to 25,
         salinity  = optional in psu, defaults to 35,
         pressure = optional, defaults to 0 bars = 1atm,
         pH = 8.1, # optional
        )
```

Results are always in mol/kg

Acess the values “dic”, “ta”, “ca”, “co2”, “hco3”, “co3”, “boron”, “boh4”, “boh3”, “oh”, “ca2”, “so4”,”hplus”, as SW.co3 etc.

This method also provides “K0”, “K1”, “K2”, “KW”, “KB”, “Ksp”, “Ksp0”, “KS”, “KF” and their corresponding pK values, as well as the density for the given (P/T/S conditions)

useful methods:

SW.show() will list values

After initialization this class provides access to each value the following way

instance_name.variable_name

Since this class is just a frontend to PyCO2SYS, it is easy to add parameters that are supported in PyCO2SYS. See the update_parameter() method.

calc_solubility_term(*S*, *T*, *A1*, *A2*, *A3*, *A4*, *B1*, *B2*, *B3*) → float

co2_solubility_constant() → None

Calculate the solubility of CO2 at a given temperature and salinity.

The value for K0 is taken from pyCO2sys which is in mol/kg-SW/atm esbmtk uses mol/(t atm). pyCO2sys follows Weiss, R. F., Marine Chemistry 2:203-215, 1974.

get_density(*S*, *TC*, *P*) → float

Calculate seawater density as function of temperature, salinity and pressure

Parameters

- **S** – salinity in PSU
- **TC** – temp in C
- **P** – pressure in bar

Returns rho
in kg/m**3

o2_solubility_constant() → None

Calculate the solubility of CO2 at a given temperature and salinity. Coefficients after Sarmiento and Gruber 2006 which includes corrections for non ideal gas behavior

Parameters Ai & Bi from Tab 3.2.2 in Sarmiento and Gruber 2006

The result is in mol/(1000kg atm)

show() → None

Printout constants. Units are mol/kg or (mol**2/kg for doubly charged ions

update_parameters(**kwargs: *dict*) → None

Update values if necessary

water_vapor_partial_pressure() → None

Calculate the water vapor partial pressure at sealevel (1 atm) as a function of temperature and salinity. Eq. Weiss and Price 1980 doi:10.1016/0304-4203(80)90024-9

Since we assume that we only use this expression at sealevel, we drop the pressure term

The result is in p/1atm (i.e., a percentage)

esbmtk.species_definitions module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

esbmtk.species_definitions.**Boron**(*model*)

esbmtk.species_definitions.**Carbon**(*model*)

Some often used definitions

esbmtk.species_definitions.**Hydrogen**(*model*)

esbmtk.species_definitions.**Nitrogen**(*model*)

esbmtk.species_definitions.**Oxygen**(*model: Model*) → None

Common Properties of Oxygen

Parameters

model (Model) – Model instance

esbmtk.species_definitions.**Phosphor**(*model*)

esbmtk.species_definitions.**Sulfur**(*model*)

esbmtk.species_definitions.**misc_variables**(*model*)

esbmtk.utility_functions module

esbmtk: A general purpose Earth Science box model toolkit Copyright (C), 2020 Ulrich G. Wortmann

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

exception esbmtk.utility_functions.**ScaleError**(*message*)

Bases: Exception

esbmtk.utility_functions.**add_to**(*l*, *e*)

add element e to list l, but check if the entry already exist. If so, throw exception. Otherwise add

esbmtk.utility_functions.**build_concentration_dicts**(*cd: dict*, *bg: dict*) → dict

Build a dict which can be used by create_reservoirs

Parameters

- **bg** – dict where the box_names are dict keys.
- **cd** – dictionary

with the following format:

```
cd = {
     # species: [concentration, isotopes]
     PO4: [Q_("2.1 * umol/liter"), False],
```

(continues on next page)

(continued from previous page)

```
    DIC: [Q_("2.1 mmol/liter"), False],
   }
```

This function returns a new dict in the following format

box_names: [concentrations, isotopes] d= {"bn": [{PO4: .., DIC: ..},{PO4:False, DIC:False}]}

esbmtk.utility_functions.**build_ct_dict**(*d: dict*, *p: dict*) → dict

build a connection dictionary from a dict containing connection keys, and a dict containing connection properties. This is most useful for connections which a characterized by a fixed rate but apply to many species. E.g., mixing fluxes in a complex model etc.

esbmtk.utility_functions.**calc_volumes**(*bg: dict*, *M: any*, *h: any*) → list

Calculate volume contained in a given depth interval bg is a dictionary in the following format:

```
bg={
    "hb": (0.1, 0, 200),
    "sb": (0.9, 0, 200),
   }
```

where the key must be a valid box name, the first entry of the list denoted the areal extent in percent, the second number is upper depth limit, and last number is the lower depth limit.

M must be a model handle h is the hypsometry handle

The function returns a list with the corresponding volumes

esbmtk.utility_functions.**check_for_quantity**(*quantity*, *unit*)

check if keyword is quantity or string an convert as necessary

Parameters

- **quantity** (*str | quantity | float | int*) – e.g., "12 m/s", or 12,
- **unit** (*str*) – desired unit for keyword, e.g., "m/s"

Returns
Returns a Quantity

Return type
Q_

Raises
ValueError – if keywword is neither number, str or quantity

esbmtk.utility_functions.**convert_to_lists**(*d: dict*, *l: int*) → dict

expand mixed dict entries (i.e. list and single value) such that they are all lists of equal length

esbmtk.utility_functions.**create_bulk_connections**(*ct: dict*, *M:* Model, *mt: int = '1:1'*) → dict

Create connections from a dictionary. The dict can have the following keys following format:

mt = mapping type. See below for explanation

na: names, tuple or str. If lists, all list elements share the same properties # sp: species list or species # ty: type, str # ra: rate, Quantity # sc: scale, Number # re: reference, optional # al: alpha, optional # de: delta, optional # bp: bypass, see scale_with_flux # si: signal # mx: True, optional defaults to False. If set, it will create forward and backward fluxes (i.e. mixing)

There are 6 different cases how to specify connections

Case 1 One connection, one set of parameters
ct1 = {"sb2hb": {"ty": "scale", 'ra'....}}

Case 2 One connection, one set of instructions, one subset with multiple parameters
This will be expanded to create connections for each species ct2 = {"sb2hb": {"ty": "scale", "sp": ["a", "b"]}}

Case 3 One connection complete set of multiple characters. Similar to case 2,
but now all parameters are given explicitly ct3 = {"sb2hb": {"ty": ["scale", "scale"], "sp": ["a", "b"]}}

Case 4 Multiple connections, one set of parameters. This will create
identical connection for "sb2hb" and "ib2db" ct4 = {("sb2hb", "ib2db"): {"ty": "scale", 'ra': ...}}

Case 5 Multiple connections, one subset of multiple set of parameters. This wil

create a connection for species 'a' in sb2hb and with species 'b' in ib2db
ct5 = {("sb2hb", "ib2db"): {"ty": "scale", "sp": ["a", "b"]}}

Case 6 Multiple connections, complete set of parameters of multiple parameters
Same as case 5, but now all parameters are specified explicitly ct6 = {("sb2hb", "ib2db"): {"ty": ["scale", "scale"], "sp": ["a", "b"]}}

The default interpretation for cases 5 and 6 is that each list entry corresponds to connection. However, sometimes we want to create multiple connections for multiple entries. In this case provide the mt='1:N' parameter which will create a connection for each species in each connection group. See the below example.

It is easy to shoot yourself in the foot. It is best to try the above first with some simple examples, e.g.,

from esbmtk import expand_dict ct2 = {"sb2hb": {"ty": "scale", "sp": ["a", "b"]}}

It is best to use the show_dict function to verify that your input dictionary produces the correct results!

esbmtk.utility_functions.**create_connection**(*n: str*, *p: dict*, *M:* Model) → None

called by create_bulk_connections in order to create a connection group It is assumed that all rates are in liter/year or mol per year. This may not be what you want or need.

Parameters

- **n** – a connection key. if the mix flag is given interpreted as mixing a connection between sb and db and thus create connections in both directions
- **p** – a dictionary holding the connection properties
- **M** – the model handle

esbmtk.utility_functions.**create_reservoirs**(*box_dict: dict*, *ic_dict: dict*, *M: any*) → dict

boxes are defined by area and depth interval here we use an ordered dictionary to define the box geometries. The next column is temperature in deg C, followed by pressure in bar the geometry is [upper depth datum, lower depth datum, area percentage]

Parameters
bn – dictionary with box parameters,

e.g.:

```
box_dict: dict = {  # name: [[geometry], T, P]
            "sb": {"g": [0, 200, 0.9], "T": 20, "P": 5},
            "ib": {"g": [200, 1200, 1], "T": 10, "P": 100},
           }
```

Parameters
ic – dictionary with species default values.

ic is used to set up initial conditions. Here we use shortcut and use the same conditions in each box. If you need box specific initial conditions use the output of build_concentration_dicts as starting point, e.g.,:

```
ic_dict: dict = { # species: concentration, Isotopes, delta, f_only
             PO4: [Q_("2.1 * umol/liter"), False, 0, False],
             DIC: [Q_("2.1 mmol/liter"), False, 0, False],
             ALK: [Q_("2.43 mmol/liter"), False, 0, False],
         }
```

Parameters
M – Model object handle

esbmtk.utility_functions.**data_summaries**(*M:* Model, *species_names: list*, *box_names: list*, *register_with='None'*) → list

Group results by species and Reservoirs

Parameters

- **M** – model instance
- **species_names** – list of species instances
- **box_names** – list of Reservoir instances
- **register_with** – defaults to M

Returns pl
a list of datafield instance to be plotted

esbmtk.utility_functions.**debug**(*func*)

Print the function signature and return value

esbmtk.utility_functions.**dict_alternatives**(*d: dict*, *e: str*, *a: str*) → any

The =dict_alternatives= function takes a dictionary =d=, an expression =e=, and an alternative expression =a=. It returns the value associated with either =a= or =e= in the dictionary =d=.

Parameters

- **d** – A dictionary.
- **e** – The first expression to check.
- **a** – The alternative expression to check.

Returns r
The value associated with either =a= or =e= in the dictionary =d=.

Raises
ValueError – If neither =a= nor =e= are found in the dictionary.

esbmtk.utility_functions.**expand_dict**(*d: dict*, *mt: str = '1:1'*) → int

Determine dict structure

in case we have mutiple connections with mutiple species, the default action is to map connections to species (t = '1:1'). If you rather want to create mutiple connections (one for each species) in each connection set t = '1:N'

esbmtk.utility_functions.**find_matching_fluxes**(*l: list*, *filter_by: str*, *exclude: str*) → list

Loop over all reservoir in l, and extract the names of all fluxes which match the filter string. Return the list of names (not objects!)

esbmtk.utility_functions.**find_matching_strings**(*s: str*, *fl: list[str]*) → bool

test if all elements of fl occur in s. Return True if yes, otherwise False

esbmtk.utility_functions.**gen_dict_entries**(*M:* Model, ***kwargs*)

Find all fluxes that contain the reference string, and create a new Species2Species instance that connects the flux matching ref_id, with a flux matching target_id. The function will a tuple containig the new connection keys that can be used by the create bulk_connection() function. The second return value is a list containing the reference fluxes.

The optional inverse parameter, can be used where in cases where the flux direction needs to be reversed, i.e., the returned key will not read sb_to_dbPOM, but db_to_sb@POM

Parameters

- **M** – Model or list
- **kwargs** – keyword dictionary, known keys are ref_id, and raget_id, inverse

Return f_list
List of fluxes that match ref_id

Return k_tuples
tuple of connection keys

esbmtk.utility_functions.**get_connection_keys**(*f_list: set*, *ref_id: str*, *target_id: str*, *inverse: bool*, *exclude: str*) → list[str]

extract connection keys from set of flux names, replace ref_id with target_id so that the key can be used in create_bulk_connnections()

Parameters

- **f_list** – a set with flux objects
- **ref_id** – string with the reference id
- **target_id** – string with the target_id
- **inverse** – Bool, optional, defaults to false

Return cnc_l
a list of connection keys (str)

The optional inverse parameter, can be used where in cases where the flux direction needs to be reversed, i.e., the returned key will not read sb2db@POM, but db2s@POM

esbmtk.utility_functions.**get_delta**(*l: ndarray[Any, dtype[float64]]*, *h: ndarray[Any, dtype[float64]]*, *r: float*) → ndarray[Any, dtype[float64]]

Calculate the delta from the mass of light and heavy isotope

Parameters

- **l** – light isotope mass/concentration
- **h** – heavy isotope mass/concentration
- **r** – reference ratio

:return : delta

esbmtk.utility_functions.**get_delta_from_concentration**(*c*, *l*, *r*)

Calculate the delta from the mass of light and heavy isotope

Parameters

- **c** – total mass/concentration

- **l** – light isotope mass/concentration
- **r** – reference ratio

esbmtk.utility_functions.**get_delta_h**(*R*) → float

Calculate the delta of a flux or reservoir

Parameters
R – Species or Flux handle

returns d as vector of delta values R.c = total concentration R.l = concentration of the light isotope

esbmtk.utility_functions.**get_imass**(*m: float*, *d: float*, *r: float*) → [<class 'float'>, <class 'float'>]

Calculate the isotope masses from bulk mass and delta value. Arguments are m = mass, d= delta value, r = abundance ratio species

esbmtk.utility_functions.**get_l_mass**(*m: float*, *d: float*, *r: float*) → float

Parameters

- **m** – mass or concentration
- **d** – delta value
- **r** – isotopic reference ratio

return mass or concentration of the light isotopeb

esbmtk.utility_functions.**get_longest_dict_entry**(*d: dict*) → int

Get length of each item in the connection dict

esbmtk.utility_functions.**get_name_only**(*o: any*) → any

Test if item is an esbmtk type. If yes, extract the name

esbmtk.utility_functions.**get_new_ratio_from_alpha**(*ref_mass: float*, *ref_l: float*, *a: float*) → [<class 'float'>, <class 'float'>]

Calculate the effect of the istope fractionation factor alpha on the ratio between the mass of the light isotope devided by the total mass

Note that alpha needs to be given as fractional value, i.e., 1.07 rather than 70 (i.e., (alpha-1) * 1000

esbmtk.utility_functions.**get_object_from_list**(*name: str*, *l: list*) → any

Match a name to a list of objects. Return the object

esbmtk.utility_functions.**get_object_handle**(*res: list*, *M:* Model)

Test if the key is a global reservoir handle or exists in the model namespace

Parameters

- **res** – list of strings, or reservoir handles
- **M** – Model handle

esbmtk.utility_functions.**get_plot_layout**(*obj*)

Simple function which selects a row, column layout based on the number of objects to display. The expected argument is a reservoir object which contains the list of fluxes in the reservoir

esbmtk.utility_functions.**get_reservoir_reference**(*k: str*, *M:* Model) → tuple

Get SpeciesProperties and Species handles

Parameters

- **k** (*str*) – with the initial flux name, e.g., M_F_A_db_DIC

- **M** (Model) – Model handle

Returns
Species2Species, SpeciesProperties

Return type
tuple

Raises
ValueError – If reservoir_name is not of type ConnectionProperties or Species2Species

esbmtk.utility_functions.**get_simple_list**(*l: list*) → list

return a list which only has the full name rather than all the object properties

esbmtk.utility_functions.**get_string_between_brackets**(*s: str*) → str

Parse string and extract substring between square brackets

esbmtk.utility_functions.**get_sub_key**(*d: dict*, *i: int*) → dict

take a dict which has where the value is a list, and return the key with the n-th value of that list

esbmtk.utility_functions.**get_typed_list**(*data: list*) → list

esbmtk.utility_functions.**insert_into_namespace**(*name, value, name_space={'NDArrayFloat': numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]], 'ScaleError': <class 'esbmtk.utility_functions.ScaleError'>, '__addmissingdefaults__': <function __addmissingdefaults__>, '__annotations__': {}, '__builtins__': {'ArithmeticError': <class 'ArithmeticError'>, 'AssertionError': <class 'AssertionError'>, 'AttributeError': <class 'AttributeError'>, 'BaseException': <class 'BaseException'>, 'BaseExceptionGroup': <class 'BaseExceptionGroup'>, 'BlockingIOError': <class 'BlockingIOError'>, 'BrokenPipeError': <class 'BrokenPipeError'>, 'BufferError': <class 'BufferError'>, 'BytesWarning': <class 'BytesWarning'>, 'ChildProcessError': <class 'ChildProcessError'>, 'ConnectionAbortedError': <class 'ConnectionAbortedError'>, 'ConnectionError': <class 'ConnectionError'>, 'ConnectionRefusedError': <class 'ConnectionRefusedError'>, 'ConnectionResetError': <class 'ConnectionResetError'>, 'DeprecationWarning': <class 'DeprecationWarning'>, 'EOFError': <class 'EOFError'>, 'Ellipsis': Ellipsis, 'EncodingWarning': <class 'EncodingWarning'>, 'EnvironmentError': <class 'OSError'>, 'Exception': <class 'Exception'>, 'ExceptionGroup': <class 'ExceptionGroup'>, 'False': False, 'FileExistsError': <class 'FileExistsError'>, 'FileNotFoundError': <class 'FileNotFoundError'>, 'FloatingPointError': <class 'FloatingPointError'>, 'FutureWarning': <class 'FutureWarning'>, 'GeneratorExit': <class 'GeneratorExit'>, 'IOError': <class 'OSError'>, 'ImportError': <class 'ImportError'>, 'ImportWarning': <class 'ImportWarning'>, 'IndentationError': <class 'IndentationError'>, 'IndexError': <class 'IndexError'>, 'InterruptedError': <class 'InterruptedError'>, 'IsADirectoryError': <class 'IsADirectoryError'>, 'KeyError': <class 'KeyError'>, 'KeyboardInterrupt': <class 'KeyboardInterrupt'>, 'LookupError': <class 'LookupError'>, 'MemoryError': <class 'MemoryError'>, 'ModuleNotFoundError': <class 'ModuleNotFoundError'>, 'NameError': <class 'NameError'>, 'None': None, 'NotADirectoryError': <class 'NotADirectoryError'>, 'NotImplemented': NotImplemented, 'NotImplementedError': <class 'NotImplementedError'>, 'OSError': <class 'OSError'>, 'OverflowError': <class 'OverflowError'>, 'PendingDeprecationWarning': <class 'PendingDeprecationWarning'>, 'PermissionError': <class 'PermissionError'>, 'ProcessLookupError': <class 'ProcessLookupError'>, 'RecursionError': <class 'RecursionError'>, 'ReferenceError': <class 'ReferenceError'>, 'ResourceWarning': <class 'ResourceWarning'>, 'RuntimeError': <class 'RuntimeError'>, 'RuntimeWarning': <class 'RuntimeWarning'>, 'StopAsyncIteration': <class 'StopAsyncIteration'>, 'StopIteration': <class 'StopIteration'>, 'SyntaxError': <class 'SyntaxError'>,*

esbmtk.utility_functions.**is_name_in_list**(*n: str*, *l: list*) → bool

Test if an object name is part of the object list

esbmtk.utility_functions.**list_fluxes**(*self*, *name*, *i*) → None

Echo all fluxes in the reservoir to the screen

esbmtk.utility_functions.**make_dict**(*keys: list*, *values: list*) → dict

Create a dictionary from a list and value, or from two lists

esbmtk.utility_functions.**map_units**(*obj: any*, *v: any*, **args*) → float

parse v to see if it is a string. if yes, map to quantity. parse v to see if it is a quantity, if yes, map to model units and extract magnitude, assign mangitude to return value if not, assign value to return value

Parameters

- **obj** – connection object
- **v** – input string/number/quantity

Args
list of model base units

Returns
number

Raises
ScaleError – if input cannot be mapped to a model unit

esbmtk.utility_functions.**phc**(*c: float*) → float

Calculate concentration as pH. c can be a number or numpy array

Parameters
c (*float*) – H+ concentration

Returns
pH value

Return type
float

esbmtk.utility_functions.**plot_geometry**(*noo: int*)

Define plot geometry based on number of objects to plot

esbmtk.utility_functions.**register_new_flux**(*rg*, *dict_key*, *dict_value*) → list

Register a new flux object with a Species2Species instance

Parameters

- **rg** (Species | Reservoir) – instance to register with
- **dict_key** (*str*) – E.g., "M.A_db.DIC"
- **dict_value** (*str*) – id value, e.g., "db_cs2"

Returns
list of Flux instances

Return type
list

esbmtk.utility_functions.**register_new_reservoir**(*r*, *sp*, *v*)

Register a new reservoir

esbmtk.utility_functions.**register_return_values**(*ef: ExternalFunction*, *rg*) → None

Register the return values of an external function instance

Parameters

- **ec** (*ExternalFunction*) – ExternalFunction Instance
- **rg** (Reservoir | Species) – The Resevoir or Reservoirgroup the external function is associated with

Raises

- **ValueError** – If the return value type is undefined
- **Check the return values of external function instances,** –
- **and create the necessary reservoirs, fluxes, or connections** –
- **if they are missing.** –
- **These fluxes are not associated with a connection Object** –
- **so we register the source/sink relationship with the** –
- **reservoir they belong to.** –
- **This fails for GasReservoir since they can have a 1:many** –
- **relatioship. The below is a terrible hack, it would be** –
- **better to express this with several connection** –
- **objects, rather than overloading the source attribute of the** –
- **GasReservoir class.** –

esbmtk.utility_functions.**register_user_function**(*M:* Model, *lib_name: str*, *func_name: str | list*) → None

Register user supplied library and function with the model

Parameters

- **M** (Model) – Model handle
- **lib_name** (*str*) – name of python file that contains the function
- **func_name** (*str | list*) – Name of one or more user supplied function(s)

esbmtk.utility_functions.**reverse_key**(*key: str*) → str

reverse a connection key e.g., sb2db@POM becomes db2sb@POM

esbmtk.utility_functions.**rmtree**(*f*) → None

Delete file, of file is directorym delete all files in

Parameters
f – pathlib path object

esbmtk.utility_functions.**set_y_limits**(*ax: Axes*, *obj: any*) → None

Prevent the display or arbitrarily small differences

esbmtk.utility_functions.**show_data**(*self*, ***kwargs*) → None

Print the 3 lines of the data starting with index

Optional arguments:

index :int = 0 starting index indent :int = 0 indentation

esbmtk.utility_functions.**show_dict**(*d: dict*, *mt: str = '1:1'*) → None

show dict entries in an organized manner

esbmtk.utility_functions.**sort_by_type**(*l: list*, *t: list*, *m: str*) → list

divide a list by type into new lists. This function will return a list and it is up to the calling code to unpack the list

l is list with various object types t is a list which contains the object types used for sorting m is a string for the error function

esbmtk.utility_functions.**split_key**(*k: str*, *M: any*) → any | str

split the string k with letters _to_, and test if optional id string is present

esbmtk.utility_functions.**summarize_results**(*M:* Model)

Summarize all model results at t_max into a hirarchical dictionary, where values are accessed in the following way:

results[basin_name][level_name][species_name]

e.g., result[“A”][“sb”][“O2”]

Author
Uli Wortmann

Contents

1.9 1 Source Code Availability

- GitHub: https://github.com/uliw/esbmtk
- pypi: https://pypi.org/project/esbmtk/
- conda-forge: https://github.com/conda-forge/esbmtk-feedstock/

1.10 Related Software

- pyCO2sys https://pyco2sys.readthedocs.io/en/latest/
- A Python based LOSCAR implementation https://github.com/Shihan150/iloscar

1.11 License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:
  - 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
  - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for

you to choose that version for the Library.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

F

G

S

T

U

V

W