Contents

1 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_4 as well as the export of P in form of sinking organic matter (POP). The concentration in the respective surface an deep water boxes is then 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 fraction of (F_{POP}), we express this model as coupled ordinary differential equations (ODE, or initial value problem): Note that the following equations do not display on github. Please use the pdf version

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

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

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

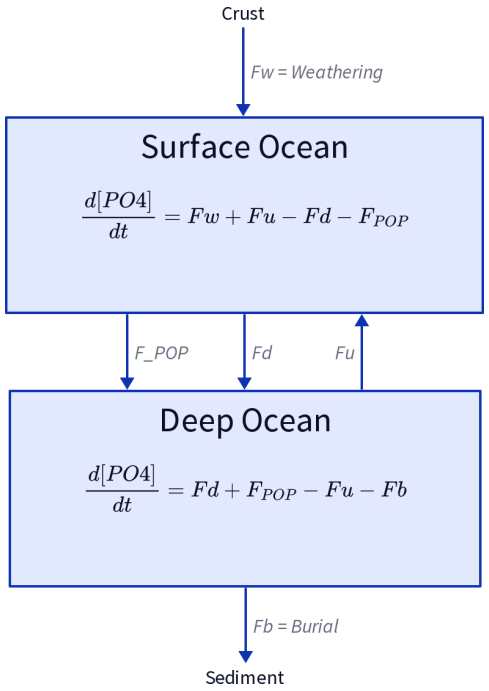

Figure 1: A two-box model of the marine P-cycle. F_w = weathering F_u = upwelling, F_d = downwelling, F_{POP} = particulate organic phosphor, F_b = burial.

```
5
6       :param C: list of initial concentrations mol/m*3
7       :param time: array of time points
8       :params V: lits of surface and deep ocean volume [m^3]
9       :param F_w: River (weathering) flux of PO4 mol/s
10      :param thx: thermohaline circulation in m*3/s
11      :returns dCdt: list of concentration changes mol/s
12      """
13
14      C_S = C_0[0]  # surface
15      C_D = C_0[1]  # deep
16      F_d = C_S * thx  # downwelling
17      F_u = C_D * thx  # upwelling
18      tau = 100 # residence time of P in surface waters [yrs]
19      F_POP = C_S * V[0] / tau  # export production
20      F_b = F_POP / 100  # burial
21
22      dCdt[0] = (F_w + F_u - F_d - F_POP) / V[0]
23      dCdt[1] = (F_d + F_POP - F_u - F_b) / V[1]
24
25      return dCdt
```

1.1 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.

Currently ESBMTK is only available via pip install as

```
1 import sys
2 !{sys.executable} -m pip install esbmtk
```

1.1.1 Defining the model geometry and initial conditions

In a first step one needs to define a model object that describes fundamental model parameters. The following code first loads the various esbmtk classes that will help with model construction, and then defines the model object. Note that units are automatically translated into model units. While convenient, there are some import 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 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.

```
# import classes from the esbmtk library
from esbmtk import (
    Model,  # the model class
    Reservoir,  # the reservoir class
    Connection,  # the connection class
    Source,  # the source class
    Sink,  # sink class
    Q_,  # Quantity operator
)

# define the basic model parameters
M = Model(
    name="M",  # model name
    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 the residence time τ. This variable is then used as input to calculate the scale for the primary production as `M.sb.volume / tau` which must fail since `M.sb.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 to relay on the automatic conversion. Note that it makes a difference for the mole to kilogram conversion whether ones 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 box
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 `Source` and `Reservoir` classes to create a source for the weathering flux, a sink for the burial flux, and instances of the surface and deep oceans 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
Source(
    name="weathering",
    species=M.PO4,
    register=M,  # i.e., the instance will be available as M.weathering
)
Sink(
    name="burial",
    species=M.PO4,
    register=M,  #
)

# reservoir definitions
Reservoir(
```

```
    name="sb",  # box name
    species=M.PO4,  # species in box
    register=M,  # this box will be available as M.sb
    volume="3E16 m**3",  # surface box volume
    concentration="0 umol/l",  # initial concentration
)
Reservoir(
    name="db",  # box name
    species=M.PO4,  # species in box
    register=M,  # this box will be available M.db
    volume="100E16 m**3",  # deeb box volume
    concentration="0 umol/l",  # initial concentration
)
```

1.1.2 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 `Connection` class. To connect the a weathering flux from the source object (M.w) to the surface ocean (M.sb) we declare a connection instance describing this relationship as follows:

```
Connection(
    source=M.weathering,  # source of flux
    sink=M.sb,  # target of flux
    rate=F_w,  # rate of flux
    id="river",  # connection id
)
```

Unless the=register= keyword is given, connections will be automatically registered withe 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 its 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.

```
Connection(  # thermohaline downwelling
    source=M.sb,  # source of flux
    sink=M.db,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="downwelling_PO4",
    # ref_reservoirs=M.sb, defaults to the source instance
)
Connection(  # thermohaline upwelling
    source=M.db,  # source of flux
    sink=M.sb,  # target of flux
    ctype="scale_with_concentration",
    scale=thc,
    id="upwelling_PO4",
)
```

There are several ways to define the biological export production, e.g., as function of the upwelling PO_4, or as function of the residence time of PO_4 in surface ocean. Here we follow Glover (2011), and use the residence time $\tau = 100$ years.

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

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 `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:

```
Connection(  #
    source=M.db,  # 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",
)
```

1.2 Working with the model instance

1.2.1 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.sb, M.db])
M.save_data()
```

1.2.2 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 save 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, that a repeated model run will not be initialized from the last known state, but rather starts from 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()
```

1.2.3 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.sb)  # will print the documentation for sb
dir(M.sb)  # will print all methods for sb
M.sb #  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.sb.c  # concentration
M.sb.m  # mass
M.sb.v  # volume
M.sb.d  # delta value (if used by model)
M.sb.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.