MicroBenthos Documentation

Release 0.15

Arjun Chennu

Dec 19, 2020

CONTENTS

MicroBenthos is a modeling framework (in python) useful for studies of benthic microbial systems, such as microbial mats, marine sediments, etc. It enables in silico experiments in microbial ecophysiology and biogeochemistry through computational simulations of geochemical processes, microbial metabolism and mass transport physics.

- Free software: MIT license
- Documentation: https://microbenthos.readthedocs.io
- Source code: https://github.com/achennu/microbenthos

CHAPTER

ONE

SIMULATION EXAMPLE

From youtube

CHAPTER

TWO

FEATURES

- **Create microbenthic systems:** Study dynamics of microbial and geochemical sedimentary processes by solving their partial differential equations. Easily include microbial populations, solar irradiance, anisotropic diffusion and more in your models.
- **Scientist-friendly:** Define constitutive chemical and biological processes simply as mathematical formulae. Data is saved in open archival format.
- **Visualize on the go:** Includes visualization of modelled parameters and results that can be viewed live, and exported to video. Progress information on simulation is also shown.
- **Stateful simulations:** Simulations can be interrupted and then resumed, allowing for interactive inspection and time-saving.
- **No programming required:** Systems and behavior can be defined through a structured text file.
- **Extensible:** New modalities of microbial systems can be created with simple programming around a defined structure.

2.1 Contents

2.1.1 Installation

Dependencies

The dependencies for *microbenthos* are:

- python3
- numpy
- scipy
- fipy
- sympy
- cerberus
- click
- pyyaml
- h5py (and libhdf)
- tqdm
- matplotlib >=2.1

- logutils

Install

Install MicroBenthos using either:

```
pip install microbenthos
```

Or using:

```
pip install git+https://github.com/achennu/microbenthos
```

Source install

The sources for MicroBenthos can be downloaded from the github repo.

You can either clone the public repository:

```
$ git clone git://github.com/achennu/microbenthos
```

Or download the tarball:

```
$ curl  -OL https://github.com/achennu/microbenthos/tarball/master
```

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

```
$ # If using conda...
$ conda env create -n microbenthos -f microbenthos/requirements.yml
$ conda env activate microbenthos
$ cd microbenthos
$ python setup.py install
```

Development install

MicroBenthos uses `pytest` to run automated unit testing. If you want to run the included tests, then install the test requirements:

```
$ pip install microbenthos[test]
```

To run the tests, change to the tests directory of the source tree.

```
$ cd microbenthos/tests
$ pytest .
```

MicroBenthos currently includes 250+ tests of its API entities.

MicroBenthos documentation is rendered using `sphinx`. To generate the documentation from the source tree, install the docs requirements and then run the build command.

```
$ pip install microbenthos[docs]

$ # change to the docs directory of microbenthos
$ cd microbenthos/docs
$ make html
```

2.1.2 Usage

MicroBenthos can be used to define a benthic microbial system, with multiple interacting entities, and study the spatial and temporal evolution of the entities.

The main user interface for using MicroBenthos is to define a model and simulation in a structured text file. The text file is structured in a format called YAML, which is a human-editable machine-friendly format. It takes a few minutes to learn yaml for regular use. The *Tutorials* provide guidance on defining and solving models of microbenthic systems. For details on the concept and mechanisms behind the software, head over to the *Design* section . For details on the commands available in MicroBenthos, see *Commands*.

Commands

MicroBenthos provides a command line interface for its operations under `microbenthos`.

```
Usage: microbenthos [OPTIONS] COMMAND [ARGS]...

  Console entry point for microbenthos

Options:
  -v, --verbosity                 Set verbosity of console logging
  --logger <TEXT INTEGER RANGE>...
                                  Set specified logger to loglevel (example:
                                  microbenthos.model 20)

  --version                       Show the version and exit.
  --help                          Show this message and exit.

Commands:
  export    Export options
  setup     Setup configuration
  simulate  Run simulation from definition file
```

To view the log messages on the console, use the `-v` flag. For increased verbosity of console log messages, use it multiple times, e.g. `-vv`, `-vvv`, etc.

The various subcommands available are:

Command: simulate

```
$ microbenthos simulate --help
Usage: microbenthos simulate [OPTIONS] MODEL_FILE

  Run simulation from definition file

Options:
  -o, --output-dir DIRECTORY      Output directory for simulation
  --output-same                   Output to same location as definition file
  -x, --exporter TEXT             Add an exporter to run. Form: -x <name>
                                  <export_type>
                                  ["<option1>=val,<option2>=val2"]

  -sTime, --simtime-total TEXT    Total simulation time. Example: "10h"
  -sLims, --simtime-lims TEXT     Simulation time step limits. Example: "0.1
                                  500"
```

(continues on next page)

(continued from previous page)

```
  -sSweeps, --max-sweeps INTEGER RANGE
                                  Max number of sweeps of equation in each
                                  timestep

  -sRes, --max-residual FLOAT     The max residual allowed for the time steps
  -sSolver, --fipy-solver TEXT    Solver type to use from fipy
  -sSnapshot, --snapshot-interval INTEGER RANGE
                                  Interval in seconds between simulation
                                  snapshots

  -O, --overwrite                 Overwrite file, if exists
  -c, --compression INTEGER RANGE
                                  Compression level for data (default: 9)
  --confirm / -Y, --no-confirm    Confirm before running simulation
  --progress INTEGER RANGE        Show progress bar (0 to disable)
  --progress-tag TEXT             Tag for progress bar
  --plot / --no-plot              Show graphical plot of model data
  --video / --no-video            Save video of simulation plot. This can slow
                                  things down.

  --frames / --no-frames          Save frames of simulation plot. This can
                                  slow things down.

  --budget                        Track variable budget over time and show in
                                  plot

  --resume INTEGER                Resume simulation by restoring from stored
                                  data at time index

  -eqns, --show-eqns              Show equations that will be solved
  --help                          Show this message and exit.
```

Note: By default, a progress bar exporter and model data exporter are added if not specified. The progress bar can be turned off using the `--no-progress` switch.

Command: export video

```
$ microbenthos export video --help
Usage: microbenthos export video [OPTIONS] DATAFILE

  Export video from model data

Options:
  -o, --outfile FILE        Name of the output file
  -O, --overwrite           Overwrite file, if exists
  --from-time TEXT          Clock time to start video ("5 h" or "0.25 d" or
                            "0.45 day")

  --till-time TEXT          Clock time to end video ("5 h" or "0.25 d" or
                            "0.45 day")

  --style TEXT              Plot style name from matplotlib
```

(continues on next page)

(continued from previous page)

```
  --writer TEXT             Animation writer class to use
  --figsize TEXT            Figure size in inches (example: "9.6x5.4")
  --dpi INTEGER RANGE       Dots per inch for figure export
  --show                    Show figure on screen during export
  --budget                  Show temporal budget error of variables
  --fps INTEGER RANGE       Frames per second for export (default: 10)
  --bitrate INTEGER RANGE   Bitrate for video encoding (default: 1400)
  --artist-tag TEXT         Artist tag in metadata
  --progress INTEGER RANGE  Position of progress bar
  --help                    Show this message and exit.
```

Note: The graphical exporter in MicroBenthos uses `matplotlib` to render the plots and animations. For video export, `ffmpeg` or one of the other animation backends for `matplotlib.animation` should be available.

Command: export model

```
$ microbenthos export model --help
Usage: microbenthos export model [OPTIONS] MODEL_FILE

  Load a model from a file and export it after validation.

Options:
  --key TEXT     Load this key from the input file
  -v, --verbose  Set this to see verbose output
  --help         Show this message and exit.
```

Command: setup completion

```
$ microbenthos setup completion --help
Usage: microbenthos setup completion [OPTIONS]

  Setup CLI completion for shell

Options:
  --shell TEXT  The shell to install completion  [required]
  --show-code   Show the installed code
  --help        Show this message and exit.
```

Tutorials

The tutorials reside in the `tutorials` folder of the MicroBenthos docs directory. Each of them contains a `definition_input.yml` file, which can be used with *Command: simulate*.

Defining a model

Model definitions are specified in a YAML format file, here called `definition_input.yml`. This file is parsed and validated by *`MicroBenthosSchemaValidator`*. Upon running with *Command: simulate*, the validated definition file is written out to `definition.yml`.

See the `definition file` for this tutorial.

Domain

Here we go through the basic steps to define a model. First we have to define a model domain for a microbial mat or sedimentary system. This is a diffusive system, i.e. the mass transport of chemical solutes is primarily through physical diffusion. We define the model under the `model` key, and specify the domain properties.

```
model:

    domain:
        cls: SedimentDBLDomain
        init_params:
            cell_size: !unit 50 mum
            sediment_length: !unit 10 mm
            dbl_length: !unit 2 mm
            porosity: 0.6
```

This specifies that the domain should be a *`SedimentDBLDomain`*, with each cell of size 50 micrometers. We specify that the sediment sub-domain is 10 mm long, and has a 2 mm thick diffusive boundary layer (DBL) on top of it. Additionally, we specify that the porosity of the sediment domain is 0.6.

Environment

The model `environment` is a container for various entities that live and interact within the domain. The environment can contain specifications of

- *`Irradiance`*
- *`ModelVariable`*
- *`Process`*

We can specify a solar irradiance within the `environment` with

```
    environment:

        irradiance:
            cls: Irradiance
            init_params:
                hours_total: !unit 4h
                day_fraction: 0.5

                channels:
                    - name: par
                      k0: !unit 15.3 1/cm
```

This will render an irradiance source with a diel period (or daylength) of 4 hours with 50% of the duration being illuminated. Essentially the irradiance source varies in a cosinusoidal fashion (to approximate solar irradiance) with the specified period. This intensity is at the surface. However, light propogates variably through a scattering medium

like sediments. We can specify one or more irradiance channels. We specify here that the photosynthetically active radiation `par`, usually means 400–700 nm, propagates in the sediment with an exponential attenuation coefficient of 15.3 per centimeter. As the model clock is incremented, the value of the surface irradiance varies with a zenith (noon time) value of 100 and dark value of 0. This "incident" irradiance propagates through the domain according to the channels specified.

Variables

Suppose we want to construct a model where one of the variables we are solving for is the distribution of oxygen. In the `environment` we define the variable

```
        oxy:
            cls: ModelVariable
            init_params:
                name: oxy
                create:
                    hasOld: true
                    value: !unit 0.0 mol/m**3

                constraints:
                    top: !unit 230 mumol/l
                    bottom: !unit 0 mol/l

                seed:
                    profile: linear
                    # params:
                    #     start: 0.1 mol/l
                    #     stop: 1e-25 mol/l
```

With this block, we have specified a `ModelVariable` called `oxy`. The `hasOld: true` is necessary to retain the values of the previous time-step during the simulation evolution. The value is set to `0.0 mol/m**3` here (See *using units*). For numerical approximation of the model equations, we have to specify the boundary conditions for the variables we want to solve for. We specify with `constraints` here the values at the top and bottom of the model domains. Optionally, we can specify to seed the initial values of the variable with a `linear` profile. Without `start` and `stop` conditions given, it will use the values from the `top` and `bottom` constraints. This is only to set the value initially, i.e. when the model clock is 0. The `name` entry can be any sympy compatible string, and is used in string representations of the equation. But the actual key `oxy` is used in references of the model path.

Note: Due to current limitations of the solving library `fipy`, the mesh of the domain is created in base SI units (meters), and so it is required for now to specify the variable's spatial units in meters. MicroBenthos attemps to cast the units to the domain mesh units internally, but it is better to specify it in the variable creation. All other parameters can be specified in other SI units and will be converted to the correct units internally.

Processes

We have only specified a numerical placeholder for the oxygen variable as `oxy`. In order for it to behave as a chemical solute diffusing through the model domain, we have to specify the process that will enable this.

```
        D_oxy:
            cls: Process
            init_params:
                expr:
                    formula: porosity * D0_oxy

                params:
                    D0_oxy: !unit 0.03 cm**2/h
```

Process definitions allow you to write a formula simply as a string, to represent a symbolic expression that relates various domain variables and parameters. Here we specify the diffusion coefficient of oxygen `D_oxy` which is the porosity multiplied by the free diffussion coefficient `D0_oxy`. The porosity is 0.6 in the sediment but 1.0 in the DBL, and this variation will be reflected in the effective diffusion coefficient for `oxy` at each cell in the model domain. This is a simple example, but much more complicated dependencies and variations can be specified symbolically.

Equations

Finally, we need to construct the actual equations we will want to solve. In this simplistic example, we have only defined one variable which should exhibit diffusive transport in the domain . So there will be only one equation for oxygen with only a diffusive and a transient term, as we have not included any other sources or responses to irradiance. That will come in subsequent tutorials, but let's proceed to build the single equation here.

```
    equations:
        oxyEqn:
            transient: [domain.oxy, 1]

            diffusion: [env.D_oxy, 1]
```

We specify an equation named `oxyEqn` which contains a transient term, pointing to the variable we want to solve for `domain.oxy` (this could also be `env.oxy`), and specify a diffusive term that takes `env.D_oxy` as its diffusion coefficient. The second terms is a multiplicative coefficient for each term in the final equation, here set to 1.

Run it

This creates the equation to solve

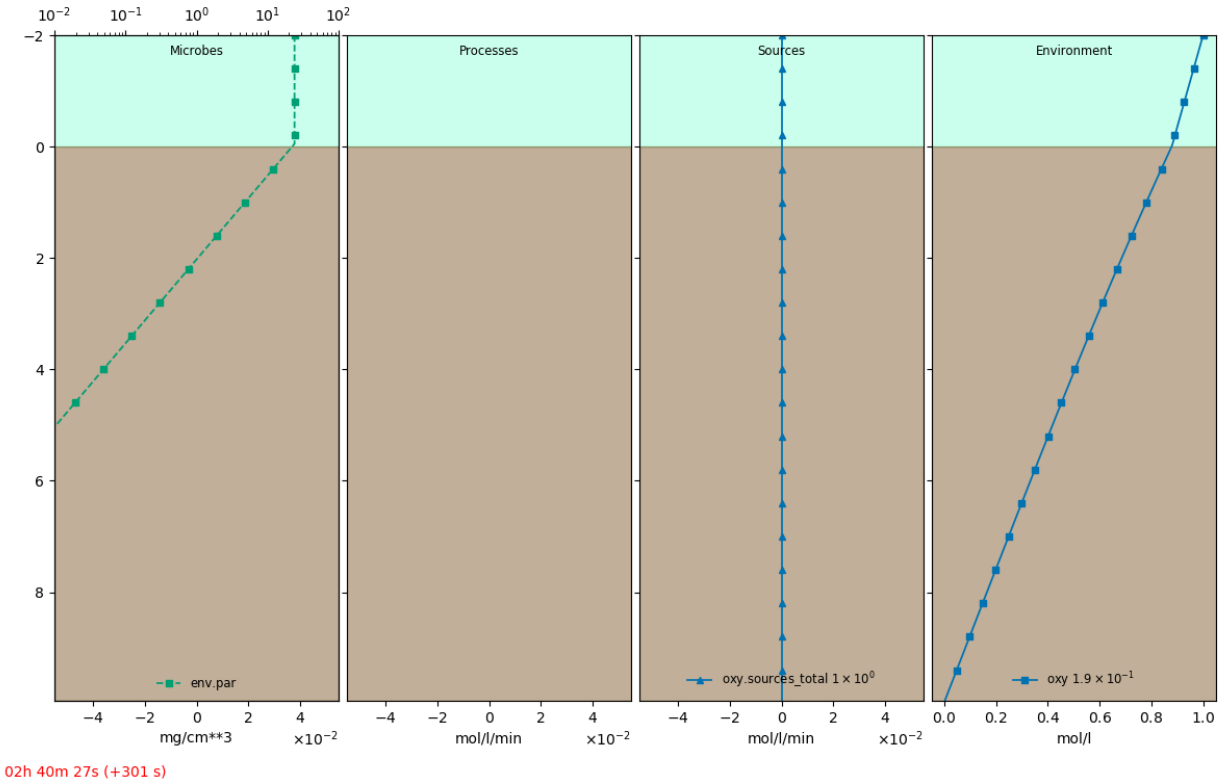
$$\frac{d}{dt}oxy = Doxy\frac{d^2}{dz^2}oxy$$

Running the model simulation with:

```
microbenthos -v simulate definition_input.yml --plot --show-eqns
```

should show the equation in the console and open up a graphical view of the model as it is simulated.

An extracted frame is shown below.



02h 40m 27s (+301 s)

Simulation run

Additionally, we can specify the parameters for the simulation evolution under the main `simulation` key.

```
simulation:

    simtime_total: !unit 8h
    # simtime_days: 2
    simtime_lims: [0.01, 300]
```

We specify with `simtime_total` that the total time for the simulation run is 8 hours. Alternatively, one can set `simtime_days` to a number which will set the `simtime_total` as the multiplication of the number with the `hours_total` of the irradiance source. MicroBenthos performs adaptive time stepping of the simulation, i.e. increasing the simulation time steps to larger values when possible and reducing it to smaller time-steps when the equations undergo a lot of change. The range of the time-steps, usually dependent on the problem being solved, can be set through `simtime_lims`, in seconds.

The full `definition file` is:

```
# start: model
model:

    domain:
        cls: SedimentDBLDomain
        init_params:
            cell_size: !unit 50 mum
            sediment_length: !unit 10 mm
```

(continues on next page)

(continued from previous page)

```
            dbl_length: !unit 2 mm
            porosity: 0.6

    # start: environment
    environment:

        irradiance:
            cls: Irradiance
            init_params:
                hours_total: !unit 4h
                day_fraction: 0.5

                channels:
                    - name: par
                      k0: !unit 15.3 1/cm

        # start: oxygen model variable
        oxy:
            cls: ModelVariable
            init_params:
                name: oxy
                create:
                    hasOld: true
                    value: !unit 0.0 mol/m**3

                constraints:
                    top: !unit 230 mumol/l
                    bottom: !unit 0 mol/l

                seed:
                    profile: linear
                    # params:
                    #     start: 0.1 mol/l
                    #     stop: 1e-25 mol/l
        # stop: oxygen model variable

        D_oxy:
            cls: Process
            init_params:
                expr:
                    formula: porosity * D0_oxy

                params:
                    D0_oxy: !unit 0.03 cm**2/h
        # stop: oxy diffusion
    # start: equations
    equations:
        oxyEqn:
            transient: [domain.oxy, 1]

            diffusion: [env.D_oxy, 1]
    # stop: equations

# start: simulation
simulation:

    simtime_total: !unit 8h
```

(continues on next page)

(continued from previous page)

```
    # simtime_days: 2
    simtime_lims: [0.01, 300]
# stop: simulation
```

Chemical reactions

In the previous tutorial *Defining a model*, we defined a microbenthic domain and a single variable to construct a single differential equation. We will now create chemical reactions between two solutes and study the dynamics and distributions of their variables by solving couple partial differential equations.

See the `definition file` for this tutorial.

Firstly, we define another diffusive solute hydrogen sulfide H_2S under `environment`.

```
        h2s:
            cls: ModelVariable
            init_params:
                name: h2s
                create:
                    hasOld: true
                    value: !unit 0.0 mol/m**3

                constraints:
                    top: !unit 10.0 mumol/l
                    bottom: !unit 1e-3 mol/l

                clip_min: 0.0

                seed:
                    profile: linear

        D_h2s:
            cls: Process
            init_params:
                expr:
                    formula: porosity * D0_h2s
                params:
                    D0_h2s: !unit 0.02 cm**2/h
```

Respiration

We will now specify reactions that involve the `oxy` and `h2s` variables, which will be cast as "source" terms in the differential equations. First we specify that the oxygen within the sediment is consumed through aerobic respiration in the `environment`.

```
        aero_respire:
            cls: Process
            init_params:
                expr:
                    formula: -Vmax * porosity * sed_mask * saturation(oxy, Km)
                params:
                    Vmax: !unit 1.0 mmol/l/h
                    Km: &aero_Km !unit 1e-5 mol/l
```

This specification states that the respiration process at a rate of `Vmax`. Since respiration does not occur within the sediment grains but within the porespaces, we multiply it by `porosity`, which is defined from the model domain. We want that the reaction occurs only in the sediment and not in the water column, so we use the variable `sed_mask`. This merely selects the region of the domain that is the sediment. Additionally, we specify that the rate of respiration has a saturation dependence on oxygen, that is the rate of the process slows down at high enough levels (parameterized by `Km`) of oxygen. What is the formulation for the `saturation(oxy, Km)` function? It can be specified under the `namespace` key of the process' `init_params`. Alternatively, if a formula is to be reused then in a `formulae` section of the `model` as follows.

```
121     formulae:
122
123         saturation:
124             vars: [x, Km]
125             expr: x / (Km + x)
```

Abiotic sulfide oxidation

Another process that occurs in sedimentary system is the abiotic oxidation of sulfide. That is oxygen reacts with hydrogen sulfide in a 2:1 stoichiometry. We can define this process also in the `environment`.

```
87          abio_sulfoxid:
88              cls: Process
89              init_params:
90                  expr:
91                      formula: porosity * k * oxy * oxy * h2s * sed_mask
92                  params:
93                      k: !unit -70.0 1/h/(mmol/l)**2
```

This reaction process therefore couples the equations of the two variables `oxy` and `h2s`, that so far had no shared process terms. The definition of this process should therefore appear in both equations.

The equations for this model will therefore be:

```
97      equations:
98
99          oxyEqn:
100
101             transient: [domain.oxy, 1]
102
103             diffusion: [env.D_oxy, 1]
104
105             sources:
106                 - [env.abio_sulfoxid, 2]
107
108                 - [env.aero_respire, 1]
109
110         h2sEqn:
111
112             transient: [domain.h2s, 1]
113
114             diffusion: [env.D_h2s, 1]
115
116             sources:
117                 - [env.abio_sulfoxid, 1]
```

Note that the stoichiometry of the sulfide oxidation process is represented by a coefficient of 2 in the oxygen equation,

indicating that for each H2S consumed two O2 are consumed by this process. The definition of the environmental process `abio_sulfoxid` enables us to easily represent the process in multiple equations.

Run it

This creates the equation to solve

$$\frac{d}{dt}oxy = Doxy\frac{d^2}{dz^2}oxy - \frac{Vmax \cdot oxy \cdot porosity \cdot sed\_mask}{Km + oxy} + 2 \cdot h2s \cdot k \cdot oxy^2 \cdot porosity \cdot sed\_mask$$

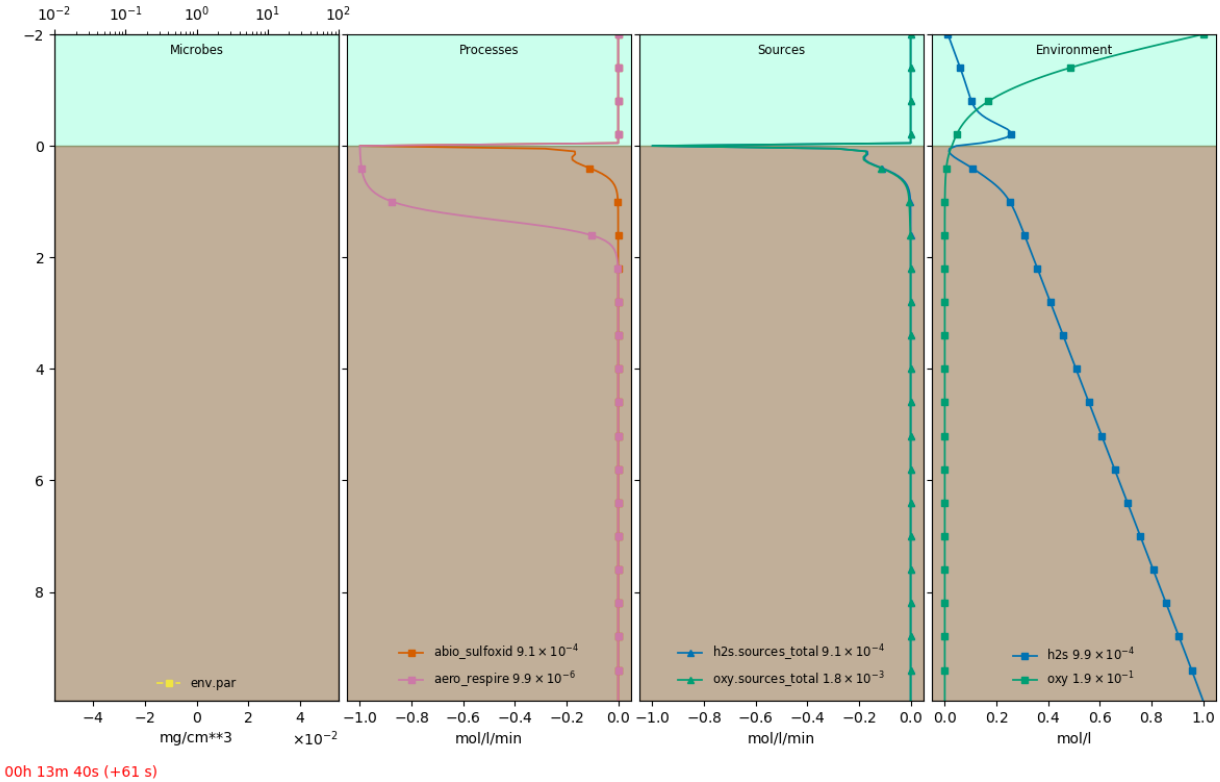
$$\frac{d}{dt}h2s = Dh2s\frac{d^2}{dz^2}h2s + h2s \cdot k \cdot oxy^2 \cdot porosity \cdot sed\_mask$$

Running the model simulation with:

```
microbenthos -v simulate definition_input.yml --plot --show-eqns
```

should show the equation in the console and open up a graphical view of the model as it is simulated.

An extracted frame is shown below.



The full `definition file` is:

```
model:

    domain:
        cls: SedimentDBLDomain
```

(continues on next page)

(continued from previous page)

```
        init_params:
            cell_size: !unit 50 mum
            sediment_length: !unit 10 mm
            dbl_length: !unit 2 mm
            porosity: 0.6


    environment:

        irradiance:
            cls: Irradiance
            init_params:
                hours_total: !unit 4h
                day_fraction: 0.5

                channels:
                    - name: par
                      k0: !unit 15.3 1/cm

        oxy:
            cls: ModelVariable
            init_params:
                name: oxy
                create:
                    hasOld: true
                    value: !unit 0.0 mol/m**3

                constraints:
                    top: !unit 230 mumol/l
                    bottom: !unit 0.0 mol/l

                seed:
                    profile: linear


        D_oxy:
            cls: Process
            init_params:
                expr:
                    formula: porosity * D0_oxy

                params:
                    D0_oxy: !unit 0.03 cm**2/h
        # start: h2s
        h2s:
            cls: ModelVariable
            init_params:
                name: h2s
                create:
                    hasOld: true
                    value: !unit 0.0 mol/m**3

                constraints:
                    top: !unit 10.0 mumol/l
                    bottom: !unit 1e-3 mol/l

                clip_min: 0.0
```

(continues on next page)

(continued from previous page)

```
            seed:
                profile: linear

    D_h2s:
        cls: Process
        init_params:
            expr:
                formula: porosity * D0_h2s
            params:
                D0_h2s: !unit 0.02 cm**2/h
    # stop: h2s

    # start: aero_respire
    aero_respire:
        cls: Process
        init_params:
            expr:
                formula: -Vmax * porosity * sed_mask * saturation(oxy, Km)
            params:
                Vmax: !unit 1.0 mmol/l/h
                Km: &aero_Km !unit 1e-5 mol/l
    # stop: aero_respire

    # start: abio_sulfoxid
    abio_sulfoxid:
        cls: Process
        init_params:
            expr:
                formula: porosity * k * oxy * oxy * h2s * sed_mask
            params:
                k: !unit -70.0 1/h/(mmol/l)**2
    # stop: abio_sulfoxid

# start: equations
equations:

    oxyEqn:

        transient: [domain.oxy, 1]

        diffusion: [env.D_oxy, 1]

        sources:
            - [env.abio_sulfoxid, 2]

            - [env.aero_respire, 1]

    h2sEqn:

        transient: [domain.h2s, 1]

        diffusion: [env.D_h2s, 1]

        sources:
            - [env.abio_sulfoxid, 1]
# stop: equations
```

(continues on next page)

(continued from previous page)

```
    # start: formulae
    formulae:

        saturation:
            vars: [x, Km]
            expr: x / (Km + x)
    # stop: formulae

simulation:
    simtime_total: !unit 8h
    simtime_lims: [0.1, 180]
    max_residual: 1e-13
```

Microbial groups

In the previous tutorial *Chemical reactions*, we defined a microbenthic domain with two reacting chemical solutes in the environment. We will now define specific microbial populations or groups which perform specific metabolic processes involving the solutes. This tutorial derives conceptually from de Wit et al (1995), but modifies it to focus on metabolic rates instead of biomass growth. The formulations of the constitutive relationships between metabolisms and environmental parameters are used from the paper, and the tutorial mainly indicates how these symbolic relationships can be used to construct and solve coupled partial differential equations.

See the `definition file` for this tutorial.

See also:

Rutger de Wit, Frank P. van den Ende, Hans van Gemerden; Mathematical simulation of the interactions among cyanobacteria, purple sulfur bacteria and chemotrophic sulfur bacteria in microbial mat communities, FEMS Microbiology Ecology, Volume 17, Issue 2, 1 June 1995, Pages 117–135 (DOI)

Cyanobacteria: biomass

We will define a distribution of cyanobacterial population in the microbenthic domain, which will perform

- respiration
- oxygenic photosynthesis
- sulfide-driven anoxygenic photosynthesis

Microbial groups are defined under the `microbes` key under `model`. Microbial groups contain `features`, which represent variables related to the population, the primary one being biomass. The definition of the `cyano` group could be specified as:

```
    microbes:
        cyano:
            init_params:
                name: cyano
                features:
                    biomass:
                        init_params:
                            name: biomass
                            create:
                                value: !unit 0 kg/m**3
```

(continues on next page)

(continued from previous page)

```
                seed:
                    profile: normal
                    params:
                        loc: !unit 1 mm
                        scale: !unit 2 mm
                        coeff: !unit 12 mg/cm**3
```

We have specified a (typically required) feature called `biomass` that has the units of kg/m^3, and is seeded with a normal distribution centered at 1 mm depth with a width of 2 mm, reaching a maximum value of $12 mg/cm^3$.

Light absorption

Phototrophs, such as cyanobacteria, absorb light energy to use in their photosynthetic reactions. The biomass-related depletion of the irradiance can be specified by modifying the irradiance channel `par` as follows.

```
            channels:
                - name: par
                  k0: !unit 15.3 1/cm
                  k_mods:
                    - [microbes.cyano.biomass, !unit 7687.5 cm**2/g]
```

This specifies that an additional attenuation based on the biomass should occur in the photosynthetically active radiation.

Oxygenic photosynthesis

Cyanobacteria perform oxygenic photosynthesis, i.e. they use the absorbed light energy to fix carbon into their biomass and release dioxygen as a side product. For the current case, we ignore any changes in biomass and consider the model dynamics for a short period of 4 hours. Oxygenic photosynthesis produces oxygen, and the rate of this production would depend on the available light and biomass at a specific depth in the microbial mat. So approximately, we consider the oxygenic photosynthesis rate as $Qmax \cdot biomass \cdot optimum(par, Ks, Ki)$, where the `optimum` function (unitless) describes the irradiance intensities which are optimal for the biomass. However, it is known that oxygenic photosynthesis is inhibited by the presence of sulfide and oxygen itself. These are represented through an `inhibition` function depending on a particular variable. So the formulae we need are:

```
    formulae:

        saturation:
            vars: [x, Km]
            expr: x / (Km + x)

        optimum:
            vars: [x, Ks, Ki]
            expr: x/(x + Ks)/(1 + x/Ki)

        inhibition:
            vars: [x, Kmax, Khalf]
            expr: (Kmax - x) / (2*Kmax - Khalf - x) * (x < Kmax)
```

Using these in the `model/formulae` section, we can now define the oxygenic photosynthesis process in the `cyano` group as:

```yaml
processes:

    oxyPS:
        cls: Process
        init_params:

            params:
                Ks: 1
                Ki: 10
                Qmax: !unit 8.4 mmol/g/h
                Kmax: !unit 0.35 mmol/l
                Khalf: !unit 0.3 mmol/l
                Kmax2: !unit 0.8 mmol/l
                Khalf2: !unit 0.7 mmol/l

            expr:
                formula: "Qmax * biomass * sed_mask * optimum(par, Ks, Ki)
                            * inhibition(h2s, Kmax, Khalf)
                            * inhibition(oxy, Kmax2, Khalf2)"
            implicit: false
```

Note here that the `biomass` variable is sourced from the `cyano` microbial group, which behaves as the domain. Variables not found in the group are then looked up from the group containing the `cyano` group, that is the model. So the model domain is used to source the `par` and `h2s` variables.

This formulation is straightforward to write, but pops up a mathematical problem. The inhibition function contains a $x < Kmax$ term, which makes the function only piecewise polynomial. Piecewise functions cannot be generally considered differentiable. This affects the casting of the process expression as an implicit source term (see fipy docs about implicit terms), since it tries to estimate the first derivative of the function with respect to the variable. So here we can simply turn off this behavior on a per-process level, by setting `implicit: false` in the initialization parameters.

However, for complex non-linear source terms it is advantageous to be able to specify the pieces of the piece-wise response in the equation terms. MicroBenthos allows you to therefore specify the inhibition response of oxygen to the oxygen concentration itself by the formulation below.

```yaml
            expr:
                formula:
                    base: "Qmax * biomass * sed_mask * optimum(par, Ks, Ki)
                             * inhibition(h2s, Kmax, Khalf)"

                    pieces:
                        - expr: (Kmax2 - oxy) / (2*Kmax2 - Khalf2 - oxy)
                          where: oxy < Kmax2
```

In this formulation, we replaced the `inhibition(oxy)` term from before, and defined a piecewise response of the process ourselves. The formulation of `inhibition` is taken as before, but without the `( x < Kmax )` term, and is instead set as a piece mask in `where`. Now, we do not need to set `implicit: false`, as each of the defined pieces in `expr` is differentiable and MicroBenthos will handle the symbolic math accordingly to cast the overall expression as an implicit source, if required.

Anoxygenic photosynthesis

Cyanobacteria also can use other electron donors than water, specifically, they can drive carbon fixation through sulfide-driven photosynthesis. This process uses H_2S instead of H_2O. We consider that this process is also driven by $Qmax \cdot biomass \cdot optimum(par, Ks, Ki)$, and further modulated by metabolic response functions. We consider that the cyanobacterial population responds to the sulfide as an `optimum` function. So we write the process for anoxygenic photosynthesis as follows.

```
    anoxyPS:
        cls: Process
        init_params:

            expr:
                formula: "Qmax * biomass * sed_mask * optimum(par, Ks, Ki)
                * optimum(h2s, Ksh2s, Kih2s)"

            params:
                Ks: 1
                Ki: 10
                Qmax: !unit -1.2 mmol/g/h
                Ksh2s: !unit 900 mumol/l
                Kih2s: !unit 3 mmol/l
```

Respiration

We can also include biomass-dependent respiration by the cyanobacteria itself. Similar to the sedimentary aerobic respiration case, the process rate saturates with respect to oxygen.

```
    respire:
        cls: Process
        init_params:
            expr:
                formula: Qmax * biomass * sed_mask * saturation(oxy, Km)
            params:
                Qmax: !unit -0.0002 mumol/g/h
                Km: !unit 1e-6 mol/l
```

We note in all the cases above, that each process essentially provides a local "namespace" to avoid clash of identically named parameters in other processes. Additionally, the microbial group provides a local store of variables, like the model domain, and passes forward the look up for missing variables to the model domain itself.

Run it

This creates the equations to solve

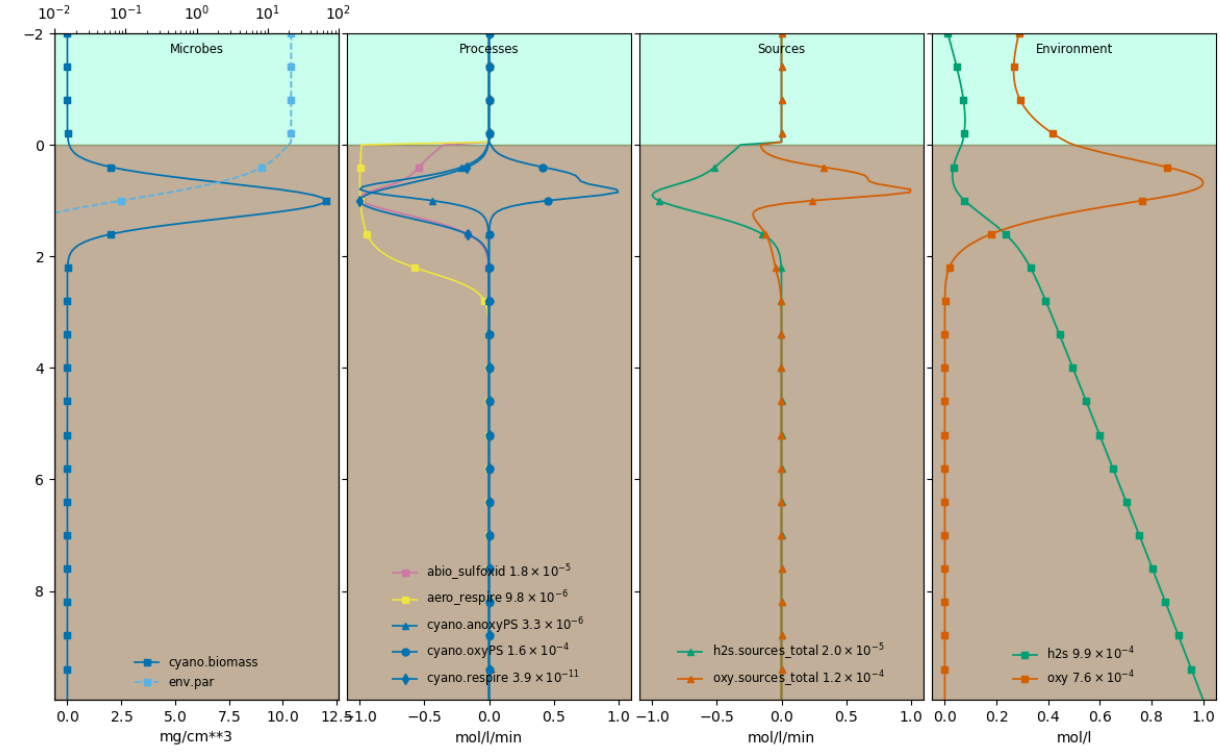
$$\mathrm{d}\frac{}{dtoxy=Doxy\cdot\frac{d^2}{dz^2}oxy+\frac{Qmax\cdot biomass\cdot oxy\cdot sed\_mask}{Km+oxy}-\frac{Vmax\cdot oxy\cdot porosity\cdot sed\_mask}{Km+oxy}+2\cdot h2s\cdot k\cdot oxy^2\cdot porosity\cdot sed\_mask+\frac{Qmax\cdot biomass\cdot par\cdot sed\_mask\cdot(Km}{\left(1+\frac{par}{Ki}\right)\cdot(Ks+par)(-Khalf+2Kmax-h}$$

Running the model simulation with:

```
microbenthos -v simulate definition_input.yml --plot --show-eqns
```

should show the equation in the console and open up a graphical view of the model as it is simulated.

An extracted frame is shown below.



The full `definition file` is:

```yaml
model:

    domain:
        cls: SedimentDBLDomain
        init_params:
            cell_size: !unit 50 mum
            sediment_length: !unit 10 mm
            dbl_length: !unit 2 mm
            porosity: 0.6


    environment:

        irradiance:
            cls: Irradiance
            init_params:
                hours_total: !unit 4h
```

(continues on next page)

(continued from previous page)

```
            day_fraction: 0.5

            # start: channels
            channels:
                - name: par
                  k0: !unit 15.3 1/cm
                  k_mods:
                    - [microbes.cyano.biomass, !unit 7687.5 cm**2/g]
            # stop: channels

    oxy:
        cls: ModelVariable
        init_params:
            name: oxy
            create:
                hasOld: true
                value: !unit 0.0 mol/m**3

            constraints:
                top: !unit 230.0 mumol/l
                bottom: !unit 0.0 mol/l

            seed:
                profile: linear

    D_oxy:
        cls: Process
        init_params:
            expr:
                formula: porosity * D0_oxy

            params:
                D0_oxy: !unit 0.03 cm**2/h

    h2s:
        cls: ModelVariable
        init_params:
            name: h2s
            create:
                hasOld: true
                value: !unit 0.0 mol/m**3

            constraints:
                top: !unit 10.0 mumol/l
                bottom: !unit 1e-3 mol/l

            seed:
                profile: linear

            clip_min: 0.0

    D_h2s:
        cls: Process
        init_params:
            expr:
                formula: porosity * D0_h2s
            params:
```

(continues on next page)

(continued from previous page)

```
                    D0_h2s: !unit 0.02 cm**2/h

        aero_respire:
            cls: Process
            init_params:
                expr:
                    formula: -Vmax * porosity * sed_mask * saturation(oxy, Km)
                params:
                    Vmax: !unit 1.0 mmol/l/h
                    Km: &aero_Km !unit 1e-5 mol/l

        abio_sulfoxid:
            cls: Process
            init_params:
                expr:
                    formula: porosity * sed_mask * k * oxy * oxy * h2s
                params:
                    k: !unit -70.0 1/h/(mmol/l)**2

    # start: microbes
    microbes:
        cyano:
            init_params:
                name: cyano
                features:
                    biomass:
                        init_params:
                            name: biomass
                            create:
                                value: !unit 0 kg/m**3
                            seed:
                                profile: normal
                                params:
                                    loc: !unit 1 mm
                                    scale: !unit 2 mm
                                    coeff: !unit 12 mg/cm**3
                                    # stop: microbes
                # start: oxyPS1
                processes:

                    oxyPS:
                        cls: Process
                        init_params:

                            params:
                                Ks: 1
                                Ki: 10
                                Qmax: !unit 8.4 mmol/g/h
                                Kmax: !unit 0.35 mmol/l
                                Khalf: !unit 0.3 mmol/l
                                Kmax2: !unit 0.8 mmol/l
                                Khalf2: !unit 0.7 mmol/l

                            expr:
                                formula: "Qmax * biomass * sed_mask * optimum(par, Ks,
↪ Ki)
                                          * inhibition(h2s, Kmax, Khalf)
```

(continues on next page)

(continued from previous page)

```
                                        * inhibition(oxy, Kmax2, Khalf2)"
                            implicit: false
                            # stop: oxyPS1
                            # start: oxyPS2
                            expr:
                                formula:
                                    base: "Qmax * biomass * sed_mask * optimum(par,␣
↪Ks, Ki)
                                           * inhibition(h2s, Kmax, Khalf)"

                                    pieces:
                                        - expr: (Kmax2 - oxy) / (2*Kmax2 - Khalf2 -␣
↪oxy)
                                          where: oxy < Kmax2
                            # stop: oxyPS2

                    # start: anoxyPS
                    anoxyPS:
                        cls: Process
                        init_params:

                            expr:
                                formula: "Qmax * biomass * sed_mask * optimum(par, Ks,
↪ Ki)
                                * optimum(h2s, Ksh2s, Kih2s)"

                            params:
                                Ks: 1
                                Ki: 10
                                Qmax: !unit -1.2 mmol/g/h
                                Ksh2s: !unit 900 mumol/l
                                Kih2s: !unit 3 mmol/l
                    # stop: anoxyPS

                    # start: respire
                    respire:
                        cls: Process
                        init_params:
                            expr:
                                formula: Qmax * biomass * sed_mask * saturation(oxy,␣
↪Km)
                            params:
                                Qmax: !unit -0.0002 mumol/g/h
                                Km: !unit 1e-6 mol/l
                    # stop: respire

    equations:

        oxyEqn:
            transient: [domain.oxy, 1]

            diffusion: [env.D_oxy, 1]

            sources:
                - [env.abio_sulfoxid, 2]

                - [env.aero_respire, 1]
```

(continues on next page)

(continued from previous page)

```
                - [microbes.cyano.processes.oxyPS, 1]

                - [microbes.cyano.processes.respire, 1]

        h2sEqn:

            transient: [domain.h2s, 1]

            diffusion: [env.D_h2s, 1]

            sources:
                - [env.abio_sulfoxid, 1]

                - [microbes.cyano.processes.anoxyPS, 1]

    # start: formulae
    formulae:

        saturation:
            vars: [x, Km]
            expr: x / (Km + x)

        optimum:
            vars: [x, Ks, Ki]
            expr: x/(x + Ks)/(1 + x/Ki)

        inhibition:
            vars: [x, Kmax, Khalf]
            expr: (Kmax - x) / (2*Kmax - Khalf - x) * (x < Kmax)
    # stop: formulae

simulation:
    simtime_total: !unit 8h
    simtime_lims: [0.1, 180]
    max_residual: 1e-13
```

Running simulations

The main command to run a simulation is `microbenthos simulate`. A few main cases are described here below, but see *Command: simulate* for detailed options.

The `simulate` command takes a definition file (such as `definition_input.yml`) as its main argument.

```
microbenthos simulate definition_input.yml
```

When run without arguments, a default *model_data* exporter is added to the simulation run. This would result in several files being created in the current working directory, one of them being a HDF5 data file, typically named *simulation_data.h5*. Alongside the data file, various run time information is written out: a log file, software versions, and the validated definition of model and simulation that was created from the supplied *definition.yml* file.

Note: The default name for the output definition file is *definition.yml*. So if your input file is called *definition.yml*, then it will get overwritten on running the command.

In order to see a mathematical representation of the equations being solved for the model simulation, use the `--show-eqns` option:

```
microbenthos simulate definition_input.yml --show-eqns
```

With visualization

The model parameters can be visualized during the simulation by invoking a graphical exporter using the `--plot` flag.

```
microbenthos simulate definition_input.yml --plot
```

This opens up a window with graphical plots of the model domain, the microbial groups, irradiance, rates of the various microbial and abiotic processes and the distribution of the analytes being solved for. Additionally, to save a video of the simulation run, use the `--video` flag:

```
microbenthos simulate definition_input.yml --video
```

To specify the output directory to be other than the current one, use the `-o` option:

```
microbenthos simulate definition_input.yml -o simrun23
```

Resume simulation

Previously run simulations can be resumed, by using the `--resume` flag. The flag takes an integer parameter, which is the index along the time dimension from which to resume . This follows python indexing semantics. Briefly, 0 refers to the first time point, and 1, 2, 3, ... to the subsequent time points. -1 refers to the last saved time point, -2 to the one before that, etc.

So, to resume from the last saved time point:

```
microbenthos simulate definition_input.yml --resume -1
```

Or from 10 time points before:

```
microbenthos simulate definition_input.yml --resume -10
```

Or to start from the 100th time point:

```
microbenthos simulate definition_input.yml --resume 100
```

Note that a supplied `--resume` directive takes precedence over `--overwrite`. Also, that `--resume 0` is equivalent to `--overwrite`.

Render video from simulation

To create an animated visualization from a saved model data file, use the `export video` command:

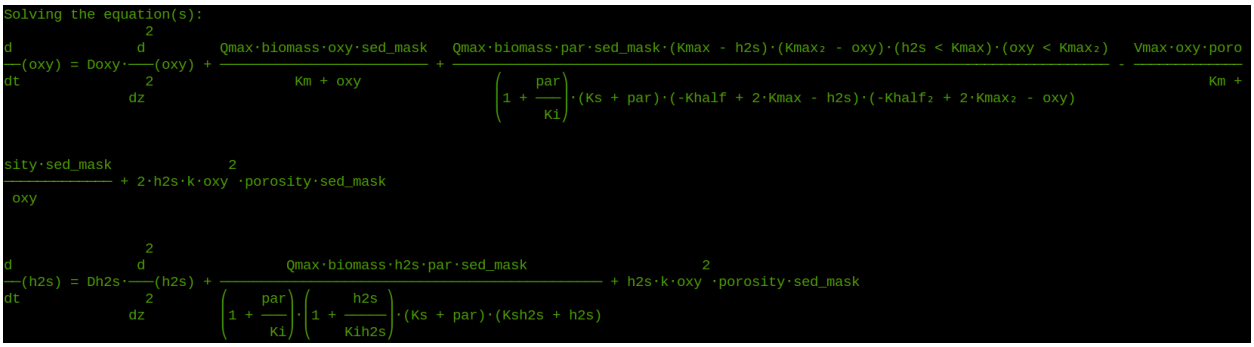
```
microbenthos export video simulation_data.h5 --show
```

The `--show` option creates the current frame being rendered to also be shown on the screen. See *Command: export video* for details. Note that some `matplotlib` backends (such as Tk) may show the canvas on the screen anyway.

2.1.3 Design

Concepts

The concept behind *microbenthos* models is to generate partial differential equations of various analytes/variables that constitute the relationships between the entities of the model. So the starting point for creating microbenthic models is to write down the constitutive relationships between chemical or metabolic processes and their dependencies on environmental conditions.



The special case of microbenthic systems emerges from the extremely steep gradients of most environmental parameters within a few millimeters, i.e. irradiance intensity, oxygen, sulfide, biomass, etc. Microbial mats constitute very dense, laminated populations of different microbial groups that occupy the niches created by the microenvironments and exploit them through adapted and specific metabolic processes.

Microbial mats are typically modelled as diffusive-reactive systems, i.e. the dominant mode of mass transport is physical diffusion of solutes. This diffusion occurs within the porespaces of the sediment matrix, which are considered to be occupied by microbial populations. The "reactive" aspect refers to the presence of a large number of "local" sources and sinks within the mat system. Indeed, a vast array of metabolic pathways can be found within a few millimeters of microbial mats, with one population thriving off the side-products of another populations main process. Furthermore, the various metabolisms have responses to multiple environmental parameters, responding to some with inhibition, to others with saturation and preferences for an optimum level. This could be with respect to light, or temperature, or oxygen or any other environmental parameters.

MicroBenthos recognizes that while the process of physical diffusion is simple to model, the larger challenge is to have a flexible way to experiment with *in silico* microbial mats, that can exhibit various types of metabolic responses. So MicroBenthos helps in converting constitutive relationships for metabolic processes, e.g. oxygenic photosynthesis is inhibited by oxygen and sulfide and has an optimal light level, into mathematical equations and compiling several of these into highly non-linear, coupled, partial differential equations. This is done through symbolic computation (using `sympy`) and abstractions at the level useful for microbial mats (sediment, diffusive boundary layer, microbial biomass, etc) and then iteratively solving the differential equations (using `fipy`) as the model clock evolves.

The *Tutorials* provide a hands-on path towards creating and studying your own *in silico* microbial mats which can include features such as:

- solar irradiance

- photopigment absorption
- microbial metabolisms (respiration, photosynthesis, etc)
- abiotic (geo)chemical processess
- biomass synthesis and growth
- ... and more to come

References

- Rutger de Wit, Frank P. van den Ende, Hans van Gemerden; Mathematical simulation of the interactions among cyanobacteria, purple sulfur bacteria and chemotrophic sulfur bacteria in microbial mat communities, FEMS Microbiology Ecology, Volume 17, Issue 2, 1 June 1995, Pages 117–135 (DOI)
- Hans van Gemerden; Microbial mats: A joint venture, Marine Geology, Volume 113, Issues 1-2, July 1993, Pages 3-25 (DOI)
- J. E. Guyer, D. Wheeler & J. A. Warren, FiPy: Partial Differential Equations with Python, Computing in Science & Engineering 11 (3) pp. 6-15 (2009), (DOI)

API

This is the complete class and function reference of *microbenthos*.

Contents:

microbenthos package

Subpackages

microbenthos.core package

Submodules

microbenthos.core.domain module

Module that defines the microbenthic domain: sediment + diffusive boundary layer

class microbenthos.core.domain.**SedimentDBLDomain**(*cell_size=0.1*, *sediment_length=10*, *dbl_length=1*, *porosity=0.6*)

Bases: object

Class that defines the model domain as a sediment column with a diffusive boundary layer.

__init__(*cell_size=0.1*, *sediment_length=10*, *dbl_length=1*, *porosity=0.6*)

Create a model domain that defines a sediment column and a diffusive boundary layer column on top of it. The mesh parameters should be supplied.

If the mesh dimensions are given as PhysicalField then they are converted into meters. If plain numbers (float) are given, then they are interpreted as being in units of mm. Finally, the parameters are all converted to base units (meters) for the creation of the mesh, so that the model equations and parameters can all work on a common dimension system.

Parameters

- **cell_size** (*float, PhysicalField*) – The size of a cell (default: 100 micron)
- **sediment_length** (*float*) – The length of the sediment column in mm (default: 10)
- **dbl_length** (*float*) – The length of the DBL in mm (default: 1)
- **porosity** (*float*) – The porosity value for the sediment column (default: 0.6)

VARS
Mapping of names to `CellVariable` instances available on the domain:

mesh
The fipy Mesh instance

cell_size
the mesh cell size as a PhysicalField

total_cells
sediment + DBL

Type total cells in the domain

sediment_length
the sediment subdomain length as a PhysicalField

DBL_length
the diffusive boundary layer subdomain length as a PhysicalField

idx_surface
The coordinate index for the sediment surface

distances
An array of the scaled cell distances of the mesh

depths
An array of the cell center coordinates, with the 0 set at the sediment surface

sediment_mask
A variable named "sed_mask" which is 1 in the sediment subdomain

create_mesh()
Create the `fipy` mesh for the domain using *cell_size* and *total_cells*.

The arrays *depths* and *distances* are created, which provide the coordinates and distances of the mesh cells.

create_var(*name*, *store=True*, ***kwargs*)
Create a variable on the mesh as a `CellVariable`.

If a *value* is not supplied, then it is set to 0.0. Before creating the cell variable, the value is multiplied with an array of ones of the shape of the domain mesh. This ensures that no single-valued options end up creating an improper variable. As a result, several types for *value* are valid.

Parameters

- **name** (*str*) – The name identifier for the variable
- **store** (*bool*) – If True, then the created variable is stored in *VARS*
- **value** (float, `numpy.ndarray`, PhysicalField) – value to set on the variable
- **unit** (*str*) – The physical units for the variable
- **hasOld** (*bool*) – Whether the variable maintains the older values, which is required for numerical solution through time discretization. This should be set to *True* for the variables of the equations that will be solved.

- ****kwargs** – passed to the call to `CellVariable.__init__()`

Returns The created variable

Raises

- **ValueError** – If *name* is not a string with len > 0
- **ValueError** – If value has a shape incompatible with the mesh
- **RuntimeError** – If domain variable with same name already exists & *store* = True

var_in_sediment(*vname*)
Convenience method to get the value of domain variable in the sediment

Parameters **vname** (*str*) – Name of the variable

Returns Slice of the variable in the sediment subdomain

var_in_DBL(*vname*)
Convenience method to get the value of domain variable in the DBL

Parameters **vname** (*str*) – Name of the variable

Returns Slice of the variable in the DBL subdomain

set_porosity(*porosity*)
Set the porosity for the sediment subdomain. The DBL porosity is set to 1.0. The supplied value is set in `sediment_porosity`.

Parameters **porosity** (*float*) – A value between 0.1 and 0.9

Returns The instance of the porosity variable

snapshot(*base=False*)
Returns a snapshot of the domain's state, with the following structure:

- **depths:**
  - **"data_static"**
    - (*depths*, *dict(unit="m")*)
- **distances:**
  - **"data_static"**
    - (*distances*, *dict(unit="m")*)
- **metadata**
  - *cell_size*
  - *sediment_length*
  - *sediment_cells*
  - *DBL_cells*
  - *DBL_length*
  - *total_cells*
  - *total_length*
  - *sediment_porosity*
  - *idx_surface*

Returns dict

microbenthos.core.entity module

class microbenthos.core.entity.**Entity**(*name=None*, *logger=None*)

Bases: object

A base class to represent a model entity.

It defines the interface for subclasses for:

- Creation of instances through *from_params()* and *from_dict()*
- Response to model clock through *on_time_updated()*
- (de-)serialization of state through *snapshot()* and *restore_from()*

name

The name of the entity

classmethod from_params(*cls*, *init_params*, *post_params=None*)

Create entity instance from the given parameters.

The *cls* path is a string that specificies a class definition to be imported, and initialized with the given parameters. As a result, the returned instance need not be that of the calling class.

Parameters

- **cls** (*str*) – The qualified *module.class_name*. If no *module* is in the string, then it is assumed to be "microbenthos".
- **init_params** (*dict*) – Params to supply to the __init__ of the target class
- **post_params** (*dict*) – Dictionary of params for *post_init()* if available

Returns Instance of the entity

Examples

```
>>> Entity.from_params(cls="Irradiance", ...)
>>> Entity.from_params(cls="microbenthos.Process", ...)
>>> Entity.from_params(cls="sympy.Lambda", ...)
```

classmethod from_dict(*cdict*)

Pre-processor for *from_params()* that extracts and sends the keys *"cls"*, *"init_params"* and *"post_params"*.

Parameters **cdict** (*dict*) – parameters for *from_params()*

Returns Instance of the entity created

Raises **KeyError** – If the *cls* key is missing in *cdict*

post_init(***kwargs*)

Hook to customize initialization of entity after construction by *from_params()*.

This must be overriden by subclasses to be useful.

Parameters ****kwargs** – arbitrary parameters for *post_init()*

Returns None

on_time_updated(*clocktime*)
Hook to respond to the model clock changing.

This must be overridden by subclasses to be useful.

Parameters **clocktime** (*float, PhysicalField*) – The model clock time.

snapshot()
Returns a snapshot of the entity's state which will be a node in the nested structure to be consumed by *save_snapshot()*.

Returns *dict* – a representation of the state of the entity

restore_from(*state*, *tidx*)
Restore the entity from a saved state

tidx is the time index. If it is *None*, then it is set to *slice(None, None)* and the entire time series is read out. Typically, *tidx = -1*, to read out the values of only the last time point.

The *state* dictionary must be of the structure as defined in *snapshot()*.

Raises **ValueError** – if the state restore does not succeed

class microbenthos.core.entity.**DomainEntity**(***kwargs*)
Bases: *microbenthos.core.entity.Entity*

An *Entity* that is aware of the model domain (see *domain*), and serves as the base class for MicrobialGroup, Irradiance, Variable, etc.

This class defines the interface to:

- register the entity with the model domain (see *set_domain()* and *domain*)
- check that entity *has_domain()*
- respond to event *on_domain_set()*
- *setup()* the entity for the simulation

property domain
The domain for the entity

Parameters **domain** – An instance of SedimentDBLDomain or similar

Raises **RuntimeError** – if domain is already set

set_domain(*domain*)
Silly method that just does self.domain = domain. Will likely be removed in a future version.

check_domain()
A stricter version of *has_domain*

Returns True – if *domain* is not None

Raises **RuntimeError** – if *has_domain* is False

property has_domain
Returns: True if *domain* is not None

on_domain_set()
Hook for when a domain is set

To be used by sub-classes to setup sub-entities

setup(***kwargs*)
Method to set up the mat entity once a domain is available

This may include logic to add any featuers it has also to the domain.

To be overridden by subclasses

property is_setup
A flag to indicate if an entity still needs setting up

Must be overriden by subclasses to be useful

microbenthos.core.variable module

class microbenthos.core.variable.**ModelVariable**(*create*, *constraints=None*, *seed=None*, *clip_min=None*, *clip_max=None*, ***kwargs*)

Bases: *microbenthos.core.entity.DomainEntity*

A class to represent a variable on the model domain.

This class serves as means for defining features such as environmental variables, chemical analytes or microbiological features (biomass). The class allows defining the variable but deferring the initialization of the associated `CellVariable` until the domain is available.

The interface defined allows to:

- set boundary condition to *constrain()* the values
- *seed()* the variable values
- create a *snapshot()* of the state and *restore_from()* it
- set *clip_min* and *clip_max* limits on the values

__init__(*create*, *constraints=None*, *seed=None*, *clip_min=None*, *clip_max=None*, ***kwargs*)
Configure the creation of a variable and its boundary conditions

Parameters

- **name** (*str*) – The name of the variable
- **create** (*dict*) – parameters for variable creation (see *create()*)
- **constraints** (*dict*) – Mapping of *location* to value of boundary condition through *constrain()*
- **seed** (*dict*) – parameters to seed initial value of variable

var
type : `fipy.CellVariable`

Object created on domain.mesh

create_params
type : dict

Validated dict of params for creation of variable

constraints
mapping of domain location to values for boundary conditions (see *constrain()*)

clip_min
the min value to which the variable array is clipped

clip_max
the max value to which the variable array is clipped

static check_create(**params*)
Check that the given *params* are valid for creating the variable once the domain is available

Parameters

- **name** (*str*) – a required identifier string
- **unit** (*str*) – a string like ("kg/m**3") that defines the physical units of the variable
- **value** (float, ndarray, PhysicalField) – the value to set for the variable. If a PhysicalField is supplied, then its (base) unit is used as *unit* and overrides any supplied *unit*.

Returns *dict* – the params dictionary to be used

Raises

- **ValueError** – if no *name* is supplied
- **ValueError** – if *unit* is not a valid input for PhysicalField

---

Note: Due to limitation in fipy (v3.1.3) that meshes do not accept arrays PhysicalField as inputs, the variables defined here are cast into base units since the domain mesh is created in meters.

---

static check_constraints(*constraints*)
Check that constraints is a mapping of location to value of boundary conditions. Recognized location values are: *"top", "bottom"*.

Raises

- **TypeError if constraints is not a mapping** –
- **ValueError if loc is not valid** –
- **ValueError if value is not single-valued** –

setup(**kwargs*)
Once a domain is available, *create()* the variable with the requested parameters and apply any constraints.

Constraints are specified as:

- "top" : domain.mesh.facesLeft
- "bottom" : domain.mesh.facesRight
- "dbl" : slice(0, domain.idx_surface)
- "sediment: : slice(domain.idx_surface, None)

Note: The constraints "dbl" and "sediment" are not yet tested to work with fipy equations. It is likely that this formulation may not work correctly.

Specifying both "top" and "dbl" or "bottom" and "sediment" does not currently raise an error, but instead warning messages are logged.

---

create(*value*, *unit=None*, *hasOld=False*, ***kwargs*)
Create a CellVariable by calling domain.create_var().

Parameters

- **value** (*int, float, array, PhysicalField*) – the value for the array

- **unit** (*str*) – physical units string for the variable. Is overridden if *value* is a *PhysicalField*.
- **hasOld** (*bool*) – flag to indicate that the variable should store the older value separately (see `fipy.CellVariable`).

Returns instance of the variable created

Raises

- **RuntimeError** – if a domain variable with *name* already exists, or no mesh exists on the domain.
- **ValueError** – if value.shape is not 1 or the domain shape

constrain(*loc*, *value*)

Constrain the variable at the given location to the given value

Parameters

- **loc** (*str*) – One of `("top", "bottom", "dbl", "sediment")`
- **value** (float, `PhysicalField`) – a numeric value for the constraint

Returns None

Raises

- **TypeError** – if the units for *value* are incompatible with *var* units
- **ValueError** – for improper values for *loc*
- **ValueError** – if value is not a 0-dimension array
- **RuntimeError** – if variable doesn't exist

seed(*profile*, ***kwargs*)

Seed the value of the variable based on the profile and parameters

Available profiles are:

- **"normal"**
  - normal distribution of values from `scipy.stats.norm`
  - *loc* and *scale* in units compatible with domain mesh
  - *coeff* to multiply the distribution with, in units compatible with that of *var*
  - the normal distribution is created to have unit height for different *loc* and *scale* values
- **"linear"**
  - uses `linspace()` to fill the first dimension of *var*
  - *start*: the start value given, else taken from constraint "top"
  - *stop*: the stop value given, else taken from constraint "bottom"
- **"lognormal"**
  - lognormal distributuion from `scipy.stats.lognorm`
  - *loc* and *scale* should be in units compatible with domain mesh
  - *shape* should be a float > 0
  - the distribution is normalized to have max value = 1

Parameters

- **profile** (*str*) – The type of profile to use
- ****kwargs** – Parmeters for the profile

Returns None

Raises **ValueError** – if incompatible units encountered

snapshot(*base=False*)

Returns a snapshot of the variable's state, with the following structure:

- **data:**
  - (*var*, *dict(unit=:meth:*.var.unit.name()`)
- **metadata**
  - constraint_<name>: constraint_value (in *constraints*)

Parameters **base** (*bool*) – Convert to base units?

Returns *dict* – the variable state

restore_from(*state*, *tidx*)

Restore the variable from a saved state

It sets the value of the *var* to that stored in the *state*.

Parameters

- **state** (*dict*) – the saved state
- **tidx** (*int, None*) – passed to *restore_var()*

The *state* dictionary must be of the structure:

- *data*: (array, dict(unit=str))

Raises **ValueError** – if the state restore does not succeed

microbenthos.core.expression module

class microbenthos.core.expression.**Expression**(*formula=None*, *name=None*, *namespace=None*, *derived=None*)

Bases: object

Representation of mathematical expressions as strings to be used for definition of processes in the model domain.

The class relies on sympy to parse the formula, but allows the definition of piecewise functions (in *add_piece()*) to maintain differentiability of each piece. Calling *expr()* returns the combined expression of all the pieces, and *diff()* returns the derivative with respect to a symbol of all the pieces.

Note: As sympify() uses eval() internally, the same caveats apply on processing unvalidated inputs as listed in the sympy docs.

__init__(*formula=None*, *name=None*, *namespace=None*, *derived=None*)

Create an expression with a symbolic representation

Parameters

- **formula** (*str, dict*) – Definition of a base and possibly piecewise function symbolically. This is processed by *parse_formula()*.
- **name** (*str*) – An identifier for the instance
- **namespace** (*None, dict*) – A mapping of symbolic names to expressions to be added to the _sympy_ns. The structure of the dictionary should be
  - **name (str)**
    - vars (list) : symbols in the *expr* to be passed to `symbols()`
    - expr (str) : the expression to be created with `Lambda`
- **derived** (*None, dict*) – A mapping of symbol to expressions to be added to the _sympy_ns used. This is useful when a complicated subexpression needs to be specified separately.

Examples

```
Expression(formula = 'x**2 + 3*y')
Expression(formula=dict(base="x**2 + 3*y"))
Expression(formula=dict(
                base="x**2 + 3*y",
                pieces = [
                        dict(where="y>0", expr= "1"),
                        dict(where="y<=0", expr= "0.25*x"),
                        ]))
Expression(formula="3x * myXvar",
           derived=dict(myXvar="sin(x)**(x-0.5/(2-x))"))
```

base
the base expression as a sympy Expr

parse_formula (*formula*)
Process the formula and return symbolic expressions for the base and any pieces.

Parameters **formula** (*str, dict*) – Input should be one of

- str: which is then only a *base* expression with no piece-wise definitions.
- dict: with the structure
  - "base": the base expression string
  - **"pieces": a list of dicts where each dict has the structure**
    - "where": str that expresses the condition for the functional space
    - "expr: The value of the function in that space

Returns *(base, pieces)* – the base expression and a list of piece-wise expressions

Raises **ValueError** – Improper input for *formula*

_sympify (*formula*)
Run the given formula expression through `sympify()` using the _sympy_ns namespace.

Parameters **formula** (*str*) – the expression as str

Returns expr (`sympy.core.expr.Expr`) – the symbolic expression

Raises **ValueError** – if `sympify(formula, locals=self._sympy_ns)` fails

add_piece(*expr*, *condition*)
Add the *expr* as a piecewise term where *condition* is valid

Parameters

- **expr** (Expr) – The sympy expression
- **condition** (int, Expr) – The condition where the expression is active

symbols()

Returns *set* – atoms in *expr()* which are of type Symbol

expr()
Evaluate the symbolic expression

Returns Expr – full expression including piece-wise definitions

diff(**args*)
Compute the differentiation of *expr()* as symbolic expression

Parameters ***args** – input to diff()

Returns Expr – full expression including piece-wise definitions

microbenthos.core.irradiance module

class microbenthos.core.irradiance.**Irradiance**(*hours_total=24*, *day_fraction=0.5*, *channels=None*, ***kwargs*)
Bases: *microbenthos.core.entity.DomainEntity*

Class that represents a source of irradiance in the model domain.

The irradiance is discretized into separate "channels" (see *IrradianceChannel*), representing a range of the light spectrum. This is useful to define channels such as PAR (photosynthetically active radiation) or NIR (near-infrared), etc.

The irradiance has a surface_level which is modulated as cos(time), to mimic the cosinusoidal variation of solar radiance during a diel period. The diel period is considered to run from midnight to midnight. The intensity in each channel is then represented as a fraction of the surface level (set at 100).

__init__(*hours_total=24*, *day_fraction=0.5*, *channels=None*, ***kwargs*)
Initialize an irradiance source in the model domain

Parameters

- **hours_total** (*int*, *float*, *PhysicalField*) – Number of hours in a diel period
- **day_fraction** (*float*) – Fraction (between 0 and 1) of diel period which is illuminated (default: 0.5)
- **channels** – See *create_channel()*
- ****kwargs** – passed to superclass

channels
the channels in the irradiance entity

hours_total
the number of hours in a diel period

day_fraction
fraction of diel period that is illuminated

hours_day
numer of hours in the illuminated fraction

zenith_time
the time within the diel period which is the zenith of radiation

zenith_level
the intensity level at the zenith time

surface_irrad
a `Variable` for the momentary radiance level at the surface

setup (***kwargs*)
With an available *model* instance, setup the defined *channels*.

property is_setup
Returns: bool: True if all the *channels* are setup

create_channel (*name*, *k0=0*, *k_mods=None*, *model=None*)
Add a channel with *IrradianceChannel*, such as PAR or NIR

Parameters

- **name** (*str*) – The channel name stored in *channels*
- **k0** (int, *PhysicalField*) – The base attenuation for this channel through the sediment
- **k_mods** (*list*) – `(var, coeff)` pairs to add attenuation sources to k0 for the channel
- **model** (*None, object*) – instance of the model, if available

Returns The created *IrradianceChannel* instance

on_time_updated (*clocktime*)
Update the surface irradiance according to the clock time

Parameters **clocktime** (`PhysicalField`) – The model clock time

snapshot (*base=False*)
Returns a snapshot of the Irradiance's state with the structure

- **"channels"**
  - "name" : *IrradianceChannel.snapshot()* of each channel
- **"metadata"**
  - *"hours_total"*: str(*hours_total*)
  - *"day_fraction"*: *day_fraction*
  - *"zenith_time"*: str(*zenith_time*)
  - *"zenith_level"*: *zenith_level*

Parameters **base** (*bool*) – Convert to base units?

Returns *dict* – the state dictionary

restore_from (*state*, *tidx*)
Restore state of each irradiance channel

class microbenthos.core.irradiance.**IrradianceChannel**(*name*, *k0=PhysicalField(0, '1/cm')*, *k_mods=None*, ***kwargs*)

Bases: *microbenthos.core.entity.DomainEntity*

Class that represents a single scalar irradiance channel in *Irradiance*

__init__(*name*, *k0=PhysicalField(0, '1/cm')*, *k_mods=None*, ***kwargs*)

A scalar irradiance channel.

This creates variables for the channel intensities, for the attenuation values.

Parameters

- **name** (*str*) – The channel name
- **k0** (*float, PhysicalField*) – The base attenuation for this channel through the sediment with units of (1/cm)
- **k_mods** (*None, list*) – `(var, coeff)` pairs that modify *k0* based on the value of the variable pointed at by *var* (example: *"microbes.cyano.biomass"*) and multiplied with a *coeff*. *coeff* must have the units such that *var * coeff* has the units of *k0*.

Raises **ValueError** – if the units of *k0* are not compatible with 1/cm

intensities

CellVariable to hold the intensities of the irradiance channel through the domain

k0

the base attenuation through the sediment

k_var

variable that represents the net attenuation

k_mods

list of modulations for the attenuation in *k0*

setup(***kwargs*)

Define attenuations when domain is available

This initializes the *intensities* of the channel through the domain.

Parameters **model** (*object*) – The model object lookups sources for *k_mods*. It should have a callable `get_object(path)()`

property k_name

Returns: str: name for the attenuation variable in the domain

define_attenuation()

Create the attenuation *k0* for the channel

add_attenuation_source(*var*, *coeff*, *model=None*)

Add an extra source of attenuation to this channel, for example through biomass that attenuates light intensity

Parameters

- **var** (*str*) – The name for the variable as attenuation source. *var* should be a string to a model variable (example `"microbes.cyano.biomass"`) in which case it is looked up from the *model*.
- **coeff** – The coefficient to multiply with. The term `var * coeff` should have dimensions of 1/length

- **model** (*object*) – The model object to perform object lookups on if necessary with `model.get_object(var)()`

Raises **ValueError** – If the units of *var* * *coeff* not compatible with 1/m

property is_setup
Returns: bool: True if all pending attenuation sources in *k_mods* have been added

property attenuation_profile
Calculates the attenuation profile for this channel

This returns the cumulative product of attenuation factors in each cell of the domain, allowing this to be multiplied by a surface value to get the irradiance intensity profile.

update_intensities(*surface_level*)
Update the *intensities* of the channel based on the surface level

Parameters **surface_level** – The variable indicating the surface intensity

Returns `numpy.ndarray` – The intensity profile through the domain

snapshot(*base=False*)
Returns a snapshot of the channel's state, with the structure

- **"attenuation"**
  - "data" : (*k_var*, dict(unit = `k_var.unit`))
  - **"metadata"**
    - "varname": str(coeff) of the sources in *k_mods*
- "intensity" : (*intensities*, dict(unit = `intensities.unit`))
- **"metadata"**
  - "k0" : str(*k0*)

Parameters **base** (*bool*) – Convert to base units?

Returns *dict* – state dictionary

restore_from(*state*, *tidx*)
Restore the *intensities* from the state

microbenthos.core.microbes module

class microbenthos.core.microbes.**MicrobialGroup**(*features=None*, *processes=None*, ***kwargs*)
Bases: *microbenthos.core.entity.DomainEntity*

Class to represent a category of microorganisms, such as cyanobacteria, sulfur bacteria, etc.

This class defines the interface to
- **define features such as *biomass*, distributed** through the domain
- setup *processes* which the microbes perform on domain entities

__init__(*features=None*, *processes=None*, ***kwargs*)
Initialize a microbial group

Parameters

- **features** (*dict*) – the definitions for *add_feature_from()*
- **processes** (*dict*) – the definitions for *add_process_from()*

VARS
container of the fipy Variables belonging to the microbes

processes
the processes of the microbes, which are instances of *Process*.

add_feature_from(*name*, ***params*)
Create a feature and add it it `features`

Parameters

- **name** (*str*) – name of the feature
- ****params** – parameters passed to *from_dict()*

add_process_from(*name*, ***params*)
Create a process and add it it *processes*

Parameters

- **name** (*str*) – name of the process
- ****params** – parameters passed to *from_dict()*

property biomass
The feature "biomass" stored in `features`. This is considered as an essential feature for a microbial group. However, no error is raised if not defined currently.

Returns `fipy.CellVariable` – biomass variable stored on the domain

on_domain_set()
Set up the domain on the microbial group.

Note: Features, which are handlers for domain variables, receive the domain instances. However processes receive *self* as the domain, so that variable lookup happens first locally on the instance and then passed on to the domain.

---

setup(***kwargs*)
If the domain is available, then setup all the features and processes.

Store any `fipy.CellVariable` created in `features` into *VARS*.

on_time_updated(*clocktime*)
When model clock updated, delegate to feature and process instances

snapshot(*base=False*)
Returns a snapshot of the state with the structure:

- **"metadata"**
  - "name": *name*
- **"features"**
  - "name" : *snapshot()* of `features`
- **"processes"**
  - "name" : *snapshot()* of *processes*

Returns *dict* – state of the microbial group

restore_from(*state*, *tidx*)
Simply delegate to `restore_from()` of the features and processes

microbenthos.core.process module

class microbenthos.core.process.**Process**(*expr*, *params=None*, *implicit=True*, *events=None*, ***kwargs*)
Bases: *microbenthos.core.entity.DomainEntity*

Class to represent a reaction occurring in the model domain.

It is used as an adapter to the symbolic formulation in `Expression` into the model equation terms. By calling *evaluate()* the symbolic expression, is cast into `fipy` terms which can be used as `fipy.SourceTerm` in the model. Additionally, if *implicit* is set, then the expression will attempted to be cast into a linear and implicit source term which can help the speed of numerical approximation.

__init__(*expr*, *params=None*, *implicit=True*, *events=None*, ***kwargs*)
Initialize the process

Parameters

- **expr** (dict, `Expression`) – input for *expr* of the process
- **params** (*dict*) – mapping of symbolic vars to numerical values uses in expression, typically to represent process parameters or constants.
- **implicit** (*bool*) – Whether to cast the equation as implicit source term (default: True)
- **events** (*None, dict*) – definitions for *add_event()*

params
the container (dict) for the parameters uses to *evaluate()* the process

implicit
flag which controls if expression will be cast into linearized and implicit source terms

events
container (dict) of *ProcessEvent*

property expr
The symbolic expression of the mathematical formulation

Parameters **e** (dict, `Expression`) – if a dict, then it is passed to `Expression.__init__()` with *name* = *self.name* to create the instance.

Returns `Expression`

evaluate(*expr*, *params=None*, *domain=None*)
Evaluate the given sympy expression on the supplied domain and param containers.

If *domain* is not given, then the `domain` is used. If *params* is not given, then *params* is used.

The *expr* atoms are inspected, and all symbols collected. Symbols that are not found in the *params* container, are set as variables to be sourced from the *domain* or *events* container.

The symbols from the expression are collected, the expression lambdified and then evaluated using the objects sourced from the containers and events.

Parameters

- **expr** (int, `Expr`) – The expression to evaluate
- **params** (*dict, None*) – The parameter container

- **domain** (*dict, None*) – The domain container for variables

Returns evaluated result typically one of (`fipy binOp`, `numpy.ndarray`)

as_source_for(*varname*, ***kwargs*)

Cast the *expr* as a source condition for the given variable name.

If *implicit* is True, then the expression will be differentiated (symbolically) with respect to variable *v* (from *varname*), and the source expression *S* will be attempted to be split as *S1 = dS/dv* and *S0 = S - S1 * v* (see fipy docs).

If *implicit* is False, then returns *(S,0)*. This also turns out to be the case when the expression *S* is linear with respect to the variable *v*.

Finally *S0* and *S1* are evaluated and returned.

Parameters

- **varname** (*str*) – The variable that the expression is a source for
- **coeff** (*int, float*) – Multiplier for the terms
- **kwargs** (*dict*) – Keyword arguments forwarded to *evaluate()*

Returns

tuple –

A *(varobj, S0, S1)* tuple, where *varobj* is the variable evaluated on the domain, and *S0* and *S1* are the evaluated source expressions on the domain. If *S1* is non-zero, then it indicates it should be cast as an implicit source.

as_term(***kwargs*)

Return the process as `fipy` term by calling *evaluate()*

repr_pretty()

Return a pretty representation of the *expr*.

See `pretty()`

snapshot(*base=False*)

Returns a snapshot of the state with the structure

- "metadata"
  - "expr" : str(*.expr.expr()*)
  - "param_names": tuple(*params*)
  - name: str(p) for p in *params*
- "data" : (*snapshot_var()* of *as_term()*)

Parameters **base** (*bool*) – Convert to base units?

Returns *dict* – the process state

restore_from(*state*, *tidx*)

Restore the process state. This is a no-op as the term of the process is symbolically defined through `fipy.binOp`.

add_event(*name*, ***definition*)

Add an event to this process through *ProcessEvent*

Parameters

- **name** (*str*) – Name for the event
- **definition** (*dict*) – definition for the event

Returns:

setup(***kwargs*)
Sets up any contained *events*

on_time_updated(*clocktime*)
Updates any contained *events*

class microbenthos.core.process.**ProcessEvent**(*expr*, ***kwargs*)
Bases: *microbenthos.core.entity.DomainEntity*

Class that represents a temporal event in the domain. Events are useful for tracking the last time at which a certain condition occurred in the domain.

It is represented as a relational expression between different domain variables. When this relation is True, then the time of the event is considered active at that domain depth location. The event time tracks the duration for which the event relation is True. This can be used to model inductive processes in microbial groups, where a certain amount of time is necessary after conditions are met for the metabolic activity to begin.

__init__(*expr*, ***kwargs*)
Create the process event by specifying a symbolic expression for the relation between domain variables.

Parameters **expr** (dict, Expression) – The relational expression between domain variables in symbolic form

Example

- "oxy >= OxyThreshold"
- "(oxy > oxyMin) & (light > lightMin)"

process
The process that this event belongs to

event_time
The depth distribution of the time when the event condition was reached

condition
The event condition expressed through domain variables

property expr
Similar to *Process.expr*

setup(*process*, *model*, ***kwargs*)
Setup the event time variable that contains the duration since when the event condition was reached at each depth location. The event condition is created by using *evaluate()*.

Parameters

- **process** (*Process*) – The process this event belongs to
- **model** (MicroBenthosModel) – The model instance

on_time_updated(*clock*)
The event_time must be reset, wherever *condition* evaluates to False. Wherever it evaluates to True, then increment the value by `(clock - prev_clock)`.

Module contents

microbenthos.dataview package

Submodules

microbenthos.dataview.base module

class microbenthos.dataview.base.**ModelData**(*store=None*)
Bases: object

Abstract Base Class that encapsulates the model data from a simulation, and provides a uniform interface to access elements in the nested hierarchy.

PATH_DEPTHS = '/domain/depths'

PATH_TIMES = '/time'

ENTRY_ENV = 'env'

ENTRY_EQUATIONS = 'equations'

ENTRY_MICROBES = 'microbes'

ENTRY_MICROBE_FEATURES = 'features'

ENTRY_IRRADIANCE = 'irradiance'

PATH_IRRADIANCE_CHANNELS = 'env/irradiance/channels'

times
numerical array of the model clock times

depths
numerical array of the domain depths

eqn_vars
Set of data paths for equation variables

eqn_processes
Set of data paths for equation process expressions

eqn_source_totals
Set of data paths for equation source totals

eqn_var_actual
Set of data paths for equation var actual density

eqn_var_expected
Set of data paths for equation var expected density

eqn_var_difference
Set of data paths var (expected - actual)

microbe_features
set of data paths for microbial features

irradiance_intensities
set of data paths for irradiance intensity channels

aliased_paths
mapping of aliased to real path

derived_paths
mapping of derived paths to its inputs and processor

property store
The backing data store

get_data(*path*, *tidx=None*)
Get the data at the given path and time index

The path is first checked if it is in the *aliased_paths* or *derived_paths*, and if not the data is looked for in the store.

Parameters

- **path** (*str*) – A dotted path to the data
- **tidx** (*int*) – The index for the *times*

Returns A `PhysicalField` of the data

abstract check_store(*obj*)
Check if the given store is of the right type

Returns *bool* – True if a valid store obj

abstract get_node(*path*)
Return the node at the given path

Parameters **path** (*str*) – A "/" separated path

Returns The node in the nested data store

Raises **KeyError** – if no such node exists

abstract read_data_from(*path*, *tidx*)
Data read out method to be implemented by subclasses

Returns a `PhysicalField` of the data

abstract read_metadata_from(*path*)
Reads out the metadata for a given node in the model data

Returns a `dict` of the metadata

update()

update_domain_info()
Read in the domain info and create the attributes *times* and *depths*.

update_equations()
Update the information about the equation variables and sources in *eqn_vars*, *eqn_source_totals* and *eqn_processes*.

add_aliased_path(*path*, *alias*)
Alias a given path

add_derived_data(*path*, *inputs*, *processor*)
Add a path entry for derived data

This method is used to add a data path which will provide data based on the *inputs* and a callable *processor*, as *processor(*inputs)*.

Parameters

- **path** (*str*) – The new derived path
- **inputs** (*tuple*) – Set of input paths for the calculation

- **processor** (*callable*) – The callable that performs the calculation

Returns *PhysicalField* – The output from the calculation

Raises

- **ValueError** – if *path* exists in store
- **TypeError** – if *processor* is not a callable

update_tracked()
Update the information about the tracked history of the equation variables & sources

update_microbes()
Update the microbial features from the model data into `microbes_features`

update_irradiance()
Update the set of data paths for irradiance intensities in *irradiance_intensities*.

microbenthos.dataview.hdfstore module

class microbenthos.dataview.hdfstore.**HDFModelData**(*store=None*)
Bases: *microbenthos.dataview.base.ModelData*

Class that encapsulates the model data stored in a HDF `Group`.

check_store(*obj*)
Check if the given store is of the right type

Returns *bool* – True if a valid store obj

get_node(*path*)
Return the node at the given path

Parameters **path** (*str*) – A "/" separated path

Returns `Group` or `Dataset` stored at *path*

Raises **KeyError** – if no such node exists

read_data_from(*path*, *tidx=None*)
Read out the data in *path* (a `Dataset`) into a `PhysicalField`

If *tidx* is *None*, then no slicing of the dataset is done

read_metadata_from(*path*)
Return a dict-like (`AttributeManager`) view of the metadata at *path*

microbenthos.dataview.plotter module

microbenthos.dataview.plotter.**fexp**(*number*)

microbenthos.dataview.plotter.**fman**(*number*)

microbenthos.dataview.plotter.**frepr**(*number*)

microbenthos.dataview.plotter.**flabel**(*number*)

class microbenthos.dataview.plotter.**ModelPlotter**(*model=None*, *style=None*, *figsize=None*, *dpi=None*, *unit_env='mol/l'*, *unit_microbes='mg/cm**3'*, *unit_sources='mol/l/min'*, *unit_process='mol/l/min'*, *track_budget=False*)

Bases: object

property model

_create_figure(***kwargs*)

Create the figure based on the model data structure.

The following vs-depth axes are created by default:

- Microbes (for microbial features & irradiance distribution)
- Environment (for eqn variable distribution)
- Sources (eqn source totals distribution)
- Processes (eqn process expression distribution)

If `track_budget` is True and corresponding data is available in model, then the *time_vars* axes is created.

Returns:

property axes_all

property axes_depth_all

property axes_time_all

setup_model()

_get_label(*path*)

Create the artist label from the path :param path: A "/" nested path :type path: str

Returns A string that can be used as the label

_init_artist_styles()

Create the styles for the artists

Prepare color cycle and line styles so that entities from the same group get the same color

create_artists()

Create the artists for the model plotter, and store it in `artists`

create_clock_artist()

create_line_artists(*data_paths*, *ax*)

update_legends(*axes=None*)

update_artists(*tidx*)

Update the data of the line artists for time point

Parameters **tidx** (*int*) – The time index

draw()

Draw the changes on to the canvas. This is meant to be called after each *update_artists()*

show(*block=False*)

close()

microbenthos.dataview.snapshot module

class microbenthos.dataview.snapshot.**SnapshotModelData**(*store=None*)
Bases: *microbenthos.dataview.base.ModelData*

Class that encapsulates the model data stored in snapshot `dict`

check_store(*obj*)
Check if the given store is of the right type

Returns *bool* – True if a valid store obj

get_node(*path*)
Return the node at the given path

Parameters **path** (*str*) – A "/" separated path

Returns The node in the nested data store

Raises **KeyError** – if no such node exists

read_metadata_from(*path*)
Reads out the metadata for a given node in the model data

Returns a `dict` of the metadata

read_data_from(*path*, *tidx=None*)
Get the data from the path in the snapshot

Parameters

- **path** (*str*) – Input used for *get_node()*
- **tidx** (*None*) – This is ignored, since a model snapshot only contains one time-point

Returns a `PhysicalField` of the data with units at the node

Module contents

microbenthos.exporters package

Submodules

microbenthos.exporters.exporter module

Base class definition for exporters

class microbenthos.exporters.exporter.**BaseExporter**(*name='exp'*, *logger=None*, ***kwargs*)
Bases: `object`

An abstract base class to define the interface for model state exporters to be used by the classes defined in *runners*.

is_eager = False

started
flag indicating if export has started

runner
the runner instance operating this exporter

setup(*runner*, *state*)
Set up the exporter

Parameters

- **runner** (*object*) – instance of the runner class operating it
- **state** (*dict*) – a model state snapshot

Returns:

property sim
The simulation object of the runner, if any

Returns None or `Simulation`

abstract prepare(*state*)
Prepare the exporter based on the model state

Parameters **state** (*dict*) – a model snapshot dict

close()
Close the exporter by calling *finish()* to clean up resources. Sets *started* to *False*.

finish()
Clean up resources. To be overridden by subclasses.

abstract process(*num*, *state*)
Process the model state to create export

Parameters

- **num** (*int*) – an index for the *state*
- **state** (*dict*) – a model snapshot

get_info()

Returns *dict* – basic metadata about self (exports, version and cls)

microbenthos.exporters.graphic module

class microbenthos.exporters.graphic.**GraphicExporter**(*show=False*, *write_video=False*, *video_dpi=100*, *video_filename='simulation.mp4'*, *track_budget=False*, *write_frames=False*, *frames_dpi=100*, *frames_folder='frames'*, ***kwargs*)
Bases: `microbenthos.exporters._output_dir_mixin.OutputDirMixin`, *microbenthos.exporters.exporter.BaseExporter*

An exporter for model snapshot data into a graphical representation.

This can write out videos (with `write_video` = True) and image frames (with `write_frames` = True). This uses `matplotlib` to render the plots.

property video_outpath

property frames_outdir

prepare(*state*)
Prepare the *ModelPlotter* that will generate the plots for graphical export

process(*num*, *state*)
Update the model plotter and grab or write frames

write_frame(*state*)
Save a frame into the output directory for the current state

finish()
Close the video writer and the model plotter.

microbenthos.exporters.model_data module

class microbenthos.exporters.model_data.**ModelDataExporter**(*overwrite=False*, *filename='simulation_data.h5'*, *compression=6*, ***kwargs*)
Bases: microbenthos.exporters._output_dir_mixin.OutputDirMixin, *microbenthos.exporters.exporter.BaseExporter*

A 'stateless' exporter for model snapshot data into HDF file. The exporter only keeps the output path, and reopens the file for each snapshot. This ensures that each snapshot is committed to disk, reducing risk of data corruption. It uses *save_snapshot()* internally.

property outpath

prepare(*state*)
Check that the output path can be created. Create the HDF file and add some metadata from the exporter.

If newly created, then the *state* is stored into the disk.

> **Warning:** If the output file already exists on disk, then it is just appended to. So, care must be taken by the caller that old files are removed, if so desired.

process(*num*, *state*)
Append the *state* to the HDF store.

microbenthos.exporters.progress module

class microbenthos.exporters.progress.**ProgressExporter**(*desc='evolution'*, *position=None*, ***kwargs*)
Bases: *microbenthos.exporters.exporter.BaseExporter*

An exporter that displays a progress bar of a running simulation.

is_eager = True

prepare(*state*)
Set up the progress bar

srepr(*v*, *prec=2*)

process(*num*, *state*)
Write the progress information to the progress bar

This includes info about residual, duration (dt), and number of sweeps in the timestep and the global progress through the model clock.

finish()
Clean up resources. To be overridden by subclasses.

Module contents

microbenthos.model package

Submodules

microbenthos.model.equation module

class microbenthos.model.equation.**ModelEquation**(*model*, *varpath*, *coeff=1*, *track_budget=False*)

Bases: object

Class that handles the creation of partial differential equations for a transient variable of the model

__init__(*model*, *varpath*, *coeff=1*, *track_budget=False*)
Initialize the model equation for a given variable

Parameters

- **model** (*MicroBenthosModel*) – model this belongs to
- **varpath** (*str*) – Model path of the equation variable (example: "env.oxygen")
- **coeff** (*int, float*) – the coefficient for the transient term

model
the model instance this equation belongs to

varpath
the path (str) to the equation variable

var
The equation variable object (Variable)

varname
the name of *var*

source_formulae
container (dict) of the formulae of the sources (sympy expressions)

source_exprs
container (dict) of the expressions of the sources (fipy variable binOp)

source_terms
container (dict) of the fipy equation terms (explicit & implicit) of the sources

source_coeffs
the coefficients in the equation for the sources

sources_total
The additive sum of the values in *source_exprs*

diffusion_def
type : (str, float)

The model path to the diffussion coeff and a numeric coefficient to multiply in the equation

obj
the fipy term that is used in the model full_eqn

finalized
flag to indicate if the equation has been finalized

Tracked
namedtuple definition for tracked fields: ('time_step', 'var_expected', 'var_actual', 'sources_change', 'transport_change')

tracked
a tuple of tracked values according to *Tracked*

finalize()
Call this to setup the equation. Once this is called, no more terms can be added.

This does:

```
self.obj = self.term_transient == sum(self.RHS_terms)
```

Raises **RuntimeError** – if no *term_transient* defined or no RHS terms defined

property term_transient
dv/dt

Type The transient term of the equation

_get_term_obj(*path*)
Get the model object at the path and return a usable fipy type

Parameters **path** (*str*) – dotted path in the model store

Returns Variable | `fipy.terms.binaryTerm._BinaryTerm`

_add_diffusion_term(*coeff*)
Add a linear diffusion term to the equation

Parameters **coeff** (*int, float, term*) – Coefficient for diffusion term

add_diffusion_term_from(*path*, *coeff*)
Add diffusion term from the object path

Parameters

- **path** (*str*) – Path to model store
- **coeff** (*int, float*) – Multiplier coefficient for object

Returns Object stored on model store

Raises **ValueError if object not found at path** –

property term_diffusion
The diffusion term for the equation

Returns Instance of `fipy.DiffusionTerm`

add_source_term_from(*path*, *coeff=1*)
Add a source term from the model path

Parameters

- **path** (*str*) – Path to model store

- **coeff** (*int, float*) – coeff for source term

Raises **ValueError if path does not point to an object** –

as_symbolic()
Return a symbolic version (sympy) of the equation

as_latex_string()
Return a latex string of the equation through sympy

as_pretty_string()
Return a pretty (unicode) string of the equation through sympy

property RHS_terms
The right hand side terms of the equation

Returns A list of terms, with the first one being *term_diffusion*, followed by the values in *source_terms*.

snapshot(*base=False*)
Return a state dictionary of the equation, with the structure

- **"sources"**
  - "metadata": source paths and coefficients
  - "data": the net rate of the combined sources
- **"diffusion"**
  - metadata: dict(*diffusion_def*)
- **"transient"**
  - **"metadata":**
    - *varpath*: transient term coeff
- **"tracked_budget" (if *track_budget* is *True*)**
  - var_expected: data: integrated density of variable from tracked changes
  - var_actual: data: integrated density of variable
  - time_step: data: the time step duration
  - sources_change: data: integrated rate of combined sources over the time step
  - transport_change: data: change in variable quantity due to mass transport

"metadata"

- "variable": *varpath*

Returns *dict* – A dictionary of the equation state

Raises **RuntimeError** – if the equation is not yet finalized

restore_from(*state*, *tidx*)
If "tracked_budget" is in the *state*, then set the values on the instance

sources_rate()
Estimate the rate of change of the variable quantity caused by source terms.

Returns *PhysicalField* – The integrated quantity of the sources

transport_rate()
Estimate the rate of change of the variable quantity caused by transport at the domain boundaries

Returns *PhysicalField* – The integrated quantity of the transport rate

var_quantity()
Calculate the integral quantity of the variable in the domain

Returns *PhysicalField* – depth integrated amount

update_tracked_budget(*dt*)
Update the tracked quantities for the variable, sources and transport

Parameters **dt** (*PhysicalField*) – the time step

Note: This is not a very accurate way to measure it, because the boundaries conditions keep the value at the domain boundaries constant, and so is not a true measure of the equation change in the time step. However, it should provide an order of magnitude metric for the accuracy of the numerical approximation in solving the equation.

---

property track_budget
Flag to indicate if the variable quantity should be tracked.

When this is set, the variable quantity in the domain is *tracked* through *update_tracked_budget()*.

Returns *bool* – Flag state

microbenthos.model.model module

class microbenthos.model.model.**MicroBenthosModel**(***kwargs*)
Bases: *microbenthos.utils.create.CreateMixin*

The theater where all the actors of microbenthos come together in a concerted play driven by the clock. This is the class that encapsulates the nested structure and function of the various entities, variables, microbial groups and binds them with the domain.

schema_key = 'model'

__init__(***kwargs*)
Initialize the model instance.

Parameters ****kwargs** – definition dictionary assumed to be validated by *MicroBenthosSchemaValidator*.

See also:

CreateMixin.create_from()

env
container (dict) of the environmental variables and processes

equations
container (dict) of the *ModelEquation* defined in the model

add_formula(*name*, *vars*, *expr*)
Add a formula to the sympy namespace of Expression

Parameters

- **name** (*str*) – Name of the formula

- **vars** (*str, list*) – Variables in the formula expression
- **expr** (*str*) – The expression to be parsed by sympy

Example

```
name = "optimum_response"
variables = "x Ks Ki"
expr = "x / (x + Ks) / (1 + x/Ki)"
```

property domain
The model domain, typically *SedimentDBLDomain*

create_entity_from(*defdict*)
Create a model entity from dictionary, and set it up with the model and domain.

See Also: *Entity.from_dict()*

Returns The entity created

_create_entity_into(*target*, *name*, *defdict*)
Create an entity from its definition dictionary and store it into the target dictionary

Parameters

- **target** (*str*) – Target dict such as `"env"`, `"microbes"`
- **name** (*str*) – The key for the dictionary
- **defdict** (*dict*) – Parameter definition of the entity

_setup(***definition*)
Set up the model instance from the *definition* dictionary, which is assumed to be validated by *MicroBenthosSchemaValidator*.

entities_setup()
Check that the model entities are setup fully, if not attempt it for each entity in *env* and :attr:.microbes`

property all_entities_setup
Flag that indicates if all entities have been setup

snapshot(*base=False*)
Create a snapshot of the model state.

This method recursively calls the *snapshot()* method of all contained entities, and compiles them into a nested dictionary. The dictionary has the structure of the model, as well as nodes with the numeric data and metadata. The state of the model can then be serialized, for example through *save_snapshot()*, or processed through various exporters (in *exporters*).

Parameters **base** (*bool*) – Whether the entities should be converted to base units?

Returns *dict* – model state snapshot

See also:

save_snapshot() for details about the nested structure of the state and how it is processed.

restore_from(*store*, *time_idx*)
Restore the model entities from the given store

Parameters

- **store** (Group) – The root of the model data store

- **time_idx** (*int*) – the index along the time series to restore. Uses
- **syntax** (*python*) –

:param : :param i.e first element is 0: :param second is 1: :param last element is -1: :param etc.:

> **Warning:** This is a potentially destructive operation! After checking that we *can_restore_from()* the given *store*, `truncate_model_data()` is called. This method modifies the data structure in the supplied store by truncating the datasets to the length of the time series as determined from *time_idx*. Only in the case of `time_idx=-1` it may not modify the data.

Raises

- **TypeError** – if the store data is not compatible with model
- **Exception** – as raised by *truncate_model_data()*.

See also:

check_compatibility() to see how the store is assessed to be compatible with the instantiated model.

truncate_model_data() for details on how the store is truncated.

can_restore_from(*store*)
Check if the model can be resumed from the given store

Parameters **store** (`hdf.Group`) – The root of the model data store

Returns True if it is compatible

add_equation(*name*, *transient*, *sources=None*, *diffusion=None*, *track_budget=False*)
Create a transient reaction-diffusion equation for the model.

The term definitions are provided as *(model_path, coeff)* pairs to be created for the transient term, diffusion term and source terms.

If all inputs are correct, it creates and finalizes a *ModelEquation* instance, stored in *equations*.

Parameters

- **name** (*str*) – Identifier for the equation
- **transient** (*tuple*) – Single definition for transient term
- **sources** (*list*) – A list of definitions for source terms
- **diffusion** (*tuple*) – Single definition for diffusion term
- **track_budget** (*bool*) – flag whether the variable budget should be
- **over time** (*tracked*) –

create_full_equation()
Create the full equation (`full_eqn`) of the model by coupling the individual *equations*.

get_object(*path*)
Get an object stored in the model

Parameters **path** (*str*) – The stored path for the object in the model

Returns The stored object if found

Raises **ValueError if no object found at given path** –

on_time_updated()
Callback function to update the time on all the stored entities

revert_vars()
Revert vars to the old settings. This is used when sweeping equations has to be rolled back

update_vars()
Update all stored variables which have an *hasOld* setting. This is used while sweeping for solutions.

update_equations(*dt*)
Update the *equations* for the time increment.

Parameters **dt** (*PhysicalField*) – the time step duration

class microbenthos.model.model.**ModelClock**(**args*, ***kwds*)
Bases: fipy.variables.variable.Variable

Subclass of fipy.Variable to implement hooks and serve as clock of the model.

property value
"Evaluate" the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print(a.value)
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.value
7
```

increment_time(*dt*)
Increment the clock

Parameters **dt** (*float, PhysicalField*) – Time step in seconds

set_time(*t*)
Set the clock time in hours :param t: Time in hours :type t: float, PhysicalField

property as_hms
Return a tuple of (hour, minute, second)

property as_hms_string
Return a string of hour, min, sec

microbenthos.model.resume module

Module to implement the resumption of a simulation run.

The assumption is that a model object is created, and a HDF data store is available.

microbenthos.model.resume.**check_compatibility**(*state*, *store*)
Check that the given model snapshot is compatible with the structure of the store. This checks that every path in the snapshot exists in the HDF store.

Parameters

- **state** (*dict*) – a model snapshot dictionary
- **store** (Group) – the root node of the stored model data

Returns True if the structures are compatible

Raises

- **ValueError** – if an incompatible data is returned
- **NotImplementedError** – for state arrays of dim > 1

microbenthos.model.resume.**truncate_model_data**(*store*, *time_idx*)

Truncates the model data in store till the *time_idx* along the time axis.

> **Warning:** This is a destructive operation on the provided *store*, if it is write-enabled. Use with caution, because it will resize the datasets in the store to the extent determined by *time_idx* and all the data beyond that will be lost. Only in the case of *time_idx = -1*, may there be no data loss as the resize will occur to the same size as the time vector.

Parameters

- **store** (hdf.Group) – root store of the model data (should be writable)
- **time_idx** (*int*) – An integer indicating that index of the time point to truncate

Returns *size (int)* – the size of the time dimension after truncation

microbenthos.model.saver module

Implements a data saver for model snapshots

microbenthos.model.saver.**save_snapshot**(*fpath*, *snapshot*, *compression=6*, *shuffle=True*)

Save a snapshot dictionary of the model to a HDF file

This method preserves the nested structure of the model snapshot. If the specified file path already exists, then the snapshot data is recursively appended to the nested data structure within.

The nested data structure has three keys with special meaning:

- **"metadata"** The dictionary under this key will be saved as HDF attributes of the group it is in.
- **"data"** The value here is expected to be a tuple *(data_array, meta_dict)*, and is saved as a HDF dataset with the attributes set from the *meta_dict*. This dataset will be appended to if future snapshots are appended to the file.
- **"data_static"** This is similar to the *data* case, except it is for data that does not change during model evolution. So a fixed-size dataset called "data" is created here.

Note: Due to the recursive traversal of the snapshot, no guarantees are made that the data saving is done atomically for the dictionary. As far as possible, the file access is brief and done under a closing context of File.

Also, nested snapshot dictionaries with infinite self references in nodes will be problematic.

Parameters

- **fpath** (*str*) – Path to the target HDF File
- **snapshot** (*dict*) – Nested snapshot dictionary
- **compression** (*int*) – The compression level 0-9 for the created Dataset (default: 6)
- **shuffle** (*bool*) – Whether to use the shuffle filter

Raises

- **TypeError** – if *snapshot* is not a suitable mapping type

- **ValueError** – if saving fails due to incompatible data types

microbenthos.model.saver.**_save_nested_dict**(*D*, *root*, ***kwargs*)

Recursively traverse the nested dictionary and save data and metadata into a mirrored hierarchy

Parameters

- **D** (*dict*) – A possibly nested dictionary with special keys
- **root** (Group) – Reference to a node within the state hierarchy

microbenthos.model.saver.**_save_data**(*root*, *data*, *meta*, *name='data'*, ***kwargs*)

Commit the data to the *root* node under the given *name*, and resize the target Dataset accordingly. The dataset is created, if it doesn't exist.

microbenthos.model.simulation module

Module to handle the simulation of microbenthos models

class microbenthos.model.simulation.**Simulation**(*simtime_total=6*, *simtime_days=None*, *simtime_lims=(0.1, 120)*, *snapshot_interval=60*, *fipy_solver='scipy'*, *max_sweeps=100*, *max_residual=1e-12*)

Bases: *microbenthos.utils.create.CreateMixin*

This class enables the process of repeatedly solving the model's equations for a (small) time step to a certain numerical accuracy, and then incrementing the model clock. During the evolution of the simulation, the state of the model as well as the simulation is yielded repeatedly.

Numerically approximating the solution to a set of partial differential equations requires that the solver system has a reasonable target accuracy ("residual") and enough attempts ("sweeps") to reach both a stable and accurate approximation for a time step. This class attempts to abstract out these optimizations for the user, by performing adaptive time-stepping. The user needs to specify a worst-case residual (max_residual), maximum number of sweeps per time-step (*max_sweeps*) and the range of time-step values to explore during evolution (*simtime_lims*). During the evolution of the simulation, the time-step is penalized if the max residual is overshot or max sweeps reached. If not, the reward is a bump up in the time-step duration, allowing for faster evolution of the simulation.

See also:

The scheme of simulation *evolution()*.

The adaptive scheme to update_time_step().

schema_key = 'simulation'

FIPY_SOLVERS = ('scipy', 'pyAMG', 'trilinos', 'pysparse')

__init__(*simtime_total=6*, *simtime_days=None*, *simtime_lims=(0.1, 120)*, *snapshot_interval=60*, *fipy_solver='scipy'*, *max_sweeps=100*, *max_residual=1e-12*)

Parameters

- **simtime_total** (*float, PhysicalField*) – The number of hours for the
- **to run** (*simulation*) –
- **simtime_days** (*float*) – The number of days (in terms of the model's irradiance cycle) the simulation should run for. Note that specifying this will override the given *simtime_total* when the *model* is supplied.
- **simtime_lims** (*float, PhysicalField*) – The minimum and maximum

- **for the** (*limits*) – *simtime_step* for adaptive time-stepping. This should be supplied as a pair of values, which are assumed to be in seconds and cast into PhysicalField internally. (default: 0.01, 240)
- **max_sweeps** (*int*) – Maximum number of sweeps to attempt per
- **(default** (*timestep*) –

  50)
- **max_residual** (*float*) – Maximum residual value for the solver at a
- **(default** – 1e-14)
- **snapshot_interval** (int, float, `PhysicalField`) – the
- **in seconds** (*duration*) – of the model clock between yielding snapshots of the model state for exporters (default: 60)
- **fipy_solver** (*str*) – Name of the fipy solver to use. One of `('scipy', 'pyAMG', 'trilinos','pysparse')` (default: "scipy")

simtime_days
Numer of days to simulate in terms of the model's irradiance source

property started
Returns: bool: Flag for if the sim evolution has started

property fipy_solver

property simtime_total
The number of hours of the model clock the simulation should be evolved for.

The supplied value must be larger than the time-steps allowed. Also, it may be over-ridden by supplying *simtime_days*.

Returns *PhysicalField* – duration in hours

property simtime_step
The current time-step duration. While setting, the supplied value will be clipped to within *simtime_lims*.

Returns *PhysicalField* – in seconds

property simtime_lims
The limits for the time-step duration allowed during evolution.

This parameter determines the degree to which the simulation evolution can be speeded up. In phases of the model evolution where the numerical solution is reached within a few sweeps, the clock would run at the max limit, whereas when a large number of sweeps are required, it would be penalized towards the min limit.

A high max value enables faster evolution, but can also lead to numerical inaccuracy (higher residual) or solution breakdown (numerical error) during *run_timestep()*. A small enough min value allows recovery, but turning back the clock to the previous time step and restarting with the min timestep and allowing subsequent relaxation.

Parameters **vals** (*float, PhysicalField*) – the (min, max) durations in seconds

Returns *lims (tuple)* – The (min, max) limits of *simtime_step* each as a `PhysicalField`

property max_sweeps
The maximum number of sweeps allowed for a timestep

Parameters **val** (*int*) – should be > 0

Returns int

property model

The model to run the simulation on. This is typically an instance of `MicroBenthosModel` or its subclasses. The interface it must provide is:

- a method `create_full_equation()`
- an attribute `full_eqn` created by above method, which is a `_BinaryTerm` that has a `sweep()` method.
- method `model.update_vars()` which is called before each

timestep

- method `model.clock.increment_time(dt)()` which is called

after each timestep

Additionally, if *simtime_days* is set, then setting the model will try to find the "`env.irradiance:` object and use its *hours_total* attribute to set the *simtime_total*.

Parameters **model** (`MicroBenthosModel`) – model instance

Returns `MicroBenthosModel`

Raises

- **RuntimeError** – if model has already been set
- **ValueError** – if modes interface does not match
- **ValueError** – if model `model.full_eqn` does not get created even after `model.create_full_equation()` is called.

_create_solver()

Create the fipy solver to be used

run_timestep()

Evolve the model through a single timestep

evolution()

Evolves the model clock through the time steps for the simulation, i.e. by calling *run_timestep()* and `model.clock.increment_time()` repeatedly while `model.clock()` `<= self.simtime_total`.

This is a generator that yields the step number, and the state of the evolution after each time step. If *snapshot_due()* is true, then also the model snapshot is included in the state.

Yields *(step, state)* tuple of step number and simulation state

Raises **RuntimeError** – if *started* is already True

get_state(*state=None*, *metrics=None*, ***kwargs*)

Get the state of the simulation evolution

Parameters

- **state** (*None, dict*) – If state is given (from `model.snapshot()`), then that is used. If None, then just the time info is created by using `model.clock`.
- **metrics** (*None, dict*) – a dict to get the simulation metrics from, else from *kwargs*

- ****kwargs** – parameters to build metrics dict. Currently the keys *"calc_time"*, *"residual"* and *"num_sweeps"* are used, if available.

Returns *dict* – the simulation state

snapshot_due()

Returns *bool* – If the current model clock time has exceeded `snapshot_interval` since the last snapshot time

update_simtime_step(*residual*, *num_sweeps*)

Update the *simtime_step* to be adaptive to the current residual and sweeps.

A multiplicative factor for the time-step is determined based on the number of sweeps and residual. If the *residual* is more than `max_residual`, then the time-step is quartered. If not, it is boosted by up to double, depending on the *num_sweeps* and *max_sweeps*. Once a new timestep is determined, it is limited to the time left in the model simulation.

Parameters

- **residual** (*float*) – the residual from the last equation step
- **num_sweeps** (*int*) – the number of sweeps from the last equation step

Module contents

microbenthos.runners package

Submodules

microbenthos.runners.simulate module

class microbenthos.runners.simulate.**SimulationRunner**(*output_dir=None*, *resume=False*, *confirm=False*, *overwrite=False*, *model=None*, *simulation=None*, *progress=False*, *progress_tag='evolution'*, *plot=False*, *video=False*, *frames=False*, *budget=False*, *exporters=None*, *show_eqns=False*)

Bases: `object`

Class that handles the pipeline of creating and running simulations of models, and generating outputs from it.

This class will:

- check that the model & simulation setup are complete
- check the output path
- create a log file
- export the model definition
- export the run setup
- setup the exporters
- run the simulation evolution
- clean up

property model

The model to run with the *simulation*. Typically an instance of `MicroBenthosModel`.

property simulation
The `Simulation` instance that will run with the *model*

add_exporter(*exptype*, *name=None*, ***kwargs*)
Add an exporter to the simulation run

Parameters

- **exptype** (*str*) – The type of exporter. This should match the `_exports_` on the class of the exporter. (See :class:`~microbenthos.exporters.exporter.BaseExporter)
- **name** (*str*) – The name to set for the exporter
- ****kwargs** – passed to the init of the exporter class.

Returns The name of the exporter created

_create_output_dir()
Create the output directory

Raises **OSError** – if `output_dir` is a file and not a dir

resume_existing_simulation(*data_outpath=None*)

setup_logfile(*mode='a'*)
Setup log file in the output directory

teardown_logfile()

save_definitions()
Save the model and simulation definition to the output directory

save_run_info()
Save the runner info to output_dir

get_info()
Return a dictionary of info about the runtime environment

check_simulation()

prepare_simulation()
Prepare the simulation by setting up the model

exporters_activated()
A context manager that starts and closes the exporters Returns:

get_data_exporters()

run()
Run the simulation with the stored model and simulation setup.

This performs a sequence of operations:

- shows equations if `show_eqns` is set
- Announces simulation settings in output
- Runs *check_simulation()*
- creates output directory
- sets up the logfile
- saves definitions to yaml outputs of simulation & model
- saves the runtime info (library, exporter versions etc)

- prepares the simulation
- activates the exporter context (see *exporters_activated()*)
- iterates over the `simulation.evolution()` and passes returned state to the exporters
- after that tears down the logfile

Raises **RuntimeError** – if no simulation exists

Module contents

microbenthos.utils package

Submodules

microbenthos.utils.create module

class microbenthos.utils.create.**CreateMixin**

Bases: object

A Mixin class that can create instances of classes defined in schema.yml, based on the *schema_key*. A variety of object types are handled in *create_from()*, and the validated input is stored as `definition_`

classmethod create_from(*obj*, ***kwargs*)

Instantiate class from the given object, and if possible, save the definition that was used to create the instance.

Parameters

- **obj** (object, dict, str, IOBase) –
  - object: if an instance of the cls, then it is returned directly
  - dict: if a mapping, then it is passed to *validate_dict()*
  - str: if a string, it is converted to stream-like and passed to `validate_yaml()`
  - io.IOBase: if a stream, it is passed to `validate_yaml()`
- ****kwargs** – passed to `validate_dict()` or `validate_yaml()` as necessary.

Variables **`definition_`** – the validated definition used to create the instance

Returns instance of the class this is subclassed by

microbenthos.utils.loader module

class microbenthos.utils.loader.**MicroBenthosSchemaValidator**(**args*, ***kwargs*)

Bases: cerberus.validator.Validator

A `cereberus` validator for schema.yml in MicroBenthos

logger = <Logger microbenthos.utils.loader (DEBUG)>

types_mapping = {'binary': TypeDefinition(name='binary', included_types=(<class 'byte

_check_with_importpath (*field*, *value*)
Validates if the value is a usable import path for an entity class

Valid examples are:

- pkg1.pkg2.mod1.class
- class_name

Invalid examples:

- .class_name

Parameters **value** – A string

Returns True if valid

_check_with_unit_name (*field*, *value*)
Checks that the string can be used as units

_validate_like_unit (*unit*, *field*, *value*)
Test that the given value has compatible units

Parameters

- **unit** – A string useful with `PhysicalField`
- **field** –
- **value** – An instance of a physical unit

Returns boolean if validated

The rule's arguments are validated against this schema: {'type': 'string'}

_check_with_sympy_symbol (*field*, *value*)
String that can be run through sympify and only has one variable symbol in it.

_check_with_model_path (*field*, *value*)
Validate that the value of the field is a model store path

Value should be of type:

- domain.oxy
- env.oxy.var
- microbes.cyano.processes.oxyPS

The rule's arguments are validated against this schema: {'type': 'string'}

checkers = ('importpath', 'model_path', 'sympify', 'sympy_symbol', 'unit_name')

coercers = ('float',)

default_setters = ()

normalization_rules = {'coerce': {'oneof': [{'type': 'callable'}, {'type': 'list',

rules = {'allof': {'logical': 'allof', 'type': 'list'}, 'allow_unknown': {'oneof':

validation_rules = {'allof': {'logical': 'allof', 'type': 'list'}, 'allow_unknown':

microbenthos.utils.loader.**validate_yaml** (*stream*, *key=None*, *schema=None*, *schema_stream=None*)

microbenthos.utils.loader.**validate_dict** (*inp_dict*, *key*, *schema=None*, *schema_stream=None*)

microbenthos.utils.loader.**get_schema**(*schema_stream=None*)
Returns the inbuilt model schema

microbenthos.utils.loader.**find_subclasses_recursive**(*baseclass*, *subclasses=None*)
Find subclasses recursively. *subclasses* should be a set into which to add the subclasses

microbenthos.utils.log module

microbenthos.utils.snapshotters module

microbenthos.utils.snapshotters.**snapshot_var**(*V*, *base=False*, *to_unit=None*)
Utility to express a variable array as its numeric value and corresponding units

Parameters

- **V** (*PhysicalField*, *Variable*, *CellVariable*, *binOp*, *np.ndarray*, *int*, *float*) – The variable
- **base** (*bool*) – Whether to express in base units
- **to_unit** (*str*) – Return array in these units. Ignored if *base* = *True*

Returns Tuple of (array, dict(unit=unit))

microbenthos.utils.snapshotters.**restore_var**(*input*, *tidx*)
This is the inverse operation of *snapshot_var()*. It takes the output of that function and returns a PhysicalField quantity

Returns PhysicalField

microbenthos.utils.yaml_setup module

Imports yaml from PyYaml and adds serialization options for fipy.PhysicalField.

microbenthos.utils.yaml_setup.**unit_constructor**(*loader*, *node*)

microbenthos.utils.yaml_setup.**unit_representer**(*dumper*, *data*)

Module contents

Submodules

microbenthos.cli module

The command line interface for *microbenthos*

Module contents

microbenthos.**setup_console_logging**(*name=None*, *level=20*)

microbenthos.**add_console_logging**(*verbose*, *start_level=40*)

2.1.4 Credits

Development Lead

- Arjun Chennu (MPI Bremen)

Consultation

- Judith M. Klatt (MPI Bremen)
- Jonathan E. Guyer (FiPy developer)

Contributors

None yet. Why not be the first?

2.1.5 Contributing

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

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/achennu/microbenthos/issues.

If you are reporting a bug, please include:

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

Fix Bugs

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

Implement Features

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

Write Documentation

MicroBenthos could always use more documentation, whether as part of the official MicroBenthos docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

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

If you are proposing a feature:

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

Get Started!

Ready to contribute? Here's how to set up *microbenthos* for local development. Note that this project uses the git-flow model for development.

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

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

3. Install your local copy into a virtual environment. See *Development install*.
4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
# or if using git flow
# git flow feature start my-feature
```

   Now you can make your changes locally. Make sure to add tests to verify your changes. All tests should be passing.

5. When you're done making changes, check that your changes pass flake8 and the tests.

   $ flake8 microbenthos tests $ py.test -v tests

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

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
# or if using git flow
git flow feature finish my-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

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

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

2.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z