xarray-simlab Documentation

Release 0.2.1+2.g1142cbb

xarray-simlab Developers

Nov 07, 2018

Getting Started

xarray-simlab is a Python library that provides both a generic framework for building computational models in a modular fashion and a `xarray` extension for setting and running simulations using the `xarray.Dataset` structure. It is designed for interactive and exploratory modeling.

CHAPTER 1

Documentation index

Getting Started

1.1 About xarray-simlab

xarray-simlab provides a framework for easily building custom computational models from a set of modular components (i.e., Python classes), called processes.

The framework handles issues that scientists who are developing models should not care too much about, like the model interface and the overall workflow management. Both are automatically determined from the succint, declarative-like interfaces of the model processes.

Notably via its xarray extension, xarray-simlab has already deep integration with the SciPy / PyData stack. Next versions will hopefully handle other technical issues like command line integration, interactive visualization and/or running many simulations in parallel, e.g., in the context of sensitivity analyses or inversion procedures.

1.1.1 Motivation

xarray-simlab is being developped with the idea of reducing the gap between the environments used for building and running computational models and the ones used for processing and analyzing simulation results. If the latter environments become more powerful and interactive, progress has still to be done for the former ones.

xarray-simlab also encourages building new models from re-usable sets of components in order to avoid reinventing the wheel. In many cases we want to customize existing models (e.g., adding a new feature or slightly modifying the behavior) instead of building new models from scratch. This modular framework allows to do that with minimal effort. By implementing models using a large number of small components that can be easily plugged in/out, we eliminate the need of hard-coding changes that we want to apply to a model, which often leads to over-complicated code and interface.

The design of this tool is thus mainly focused on both fast model development and easy, interactive model exploration. Ultimately, this would optimize the iterative back-and-forth process between ideas that we have on how to model a particular phenomenon and insights that we get from the exploration of model behavior.

1.1.2 Sources of inspiration

xarray-simlab leverages the great number of packages that are part of the Python scientific ecosystem. More specifically, the packages below have been great sources of inspiration for this project.

- xarray: xarray-simlab actually provides an xarray extension for setting and running models.
- attrs: a package that allows writing Python classes without boilerplate. xarray-simlab uses and extends attrs for writing processes as succinct Python classes.
- luigi: the concept of Luigi is to use Python classes as re-usable units that help building complex workflows. xarray-simlab's concept is similar, but here it is specific to computational (numerical) modeling.
- django (not really a scientific package): the way that model processes are designed in xarray-simlab has been initially inspired from Django's ORM (i.e., the `django.db.models` part).
- param: another source of inspiration for the interface of processes (more specifically the variables that it defines).
- climlab: another python package for process-oriented modeling, which uses the same approach although having a slightly different design/API, and which is applied to climate modeling.
- landlab: like climlab it provides a framework for building model components but it is here applied to landscape evolution modeling. It already has a great list of components ready to use.
- dask: represents fine-grained processing tasks as Directed Acyclic Graphs (DAGs). xarray-simlab models are DAGs too, where the nodes are interdepent processes. In this project we actually borrow some code from dask for resolving process dependencies and for model visualization.

1.2 Frequently Asked Questions

1.2.1 Does xarray-simlab provide built-in models?

No, xarray-simlab provides only the framework for creating, customizing and running computational models. It is intended to be a general-purpose tool. Domain specific models should be implemented in 3rd party packages. For example, xarray-topo provides xarray-simlab models and model components for simulating landscape evolution.

1.2.2 Can xarray-simlab be used with existing model implementations?

Yes, it should be easy to wrap existing model implementations using xarray-simlab. Even monolithic codes may leverage the xarray interface. However, as the framework works best at a fine grained level (i.e., with models built from many "small" components) it might be worth to refactor those monolithic implementations.

1.2.3 Does xarray-simlab allow fast model execution?

Yes, although it depends on how the model is implemented.

xarray-simlab is written in pure-Python and so is the outer (time) loop in simulations. The execution of Python code is slow compared to other languages, but for the outer loop only it wouldn't represent the main bottleneck of the overall model execution, especially when using an implicit time scheme. For inner (e.g., spatial) loops in each model processes, it might be better to have a numpy vectorized implementation, use tools like Cython or Numba or call wrapped code that is written in, e.g., C/C++ or Fortran (see for example f2py for wrapping Fortran code or pybind11 for wrapping C++11 code).

As with any other framework, xarray-simlab introduces an overhead compared to a simple, straightforward (but non-flexible) implementation of a model. The preliminary benchmarks that we have run show only a very small (almost free) overhead, though. This overhead is mainly introduced by the thin object-oriented layer that model components (i.e., Python classes) together form.

1.2.4 Does xarray-simlab support running model(s) in parallel?

There is currently no support for model execution in parallel but it is a top priority for the next releases!

Three levels of parallelism are possible:

- "inter-model" parallelism, i.e., execution of multiple model runs in parallel,
- "inter-process" parallelism, i.e., execution of multiple processes of a model in parallel,
- "intra-process" parallelism, i.e., parallel execution of some code written in one or more processes.

Note that the notion of process used above is different from multiprocessing: a process here corresponds to a component of a model (see *Modeling Framework* section).

The first level "inter-model" is an embarrassingly parallel problem. Next versions of xarray-simlab will allow to very easily run simulations in parallel (e.g., for sensitivity analyses).

It shouldn't be hard to add support for the second level "inter-process" given that processes in a model together form a directed acyclic graph. However, those processes usually perform most of their computation on shared data, which may significantly reduce the gain of parallel execution when using multiple OS processes or in distributed environments. Using multiple threads is limited by the CPython's GIL, unless it is released by the code executed in model processes.

The third level "intra-process" is more domain specific. Users are free to develop xarray-simlab compatible models with custom code (in processes) that is executed either sequentially or in parallel.

1.2.5 Is it possible to use xarray-simlab without xarray?

Although it sounds a bit odd given the name of this package, in principle it is possible. The implementation of the modeling framework is indeed completely decoupled from the xarray interface.

However, the xarray extension provided in this package aims to be the primary, full-featured interface for setting and running simulations from within Python.

The modeling framework itself doesn't have any built-in interface apart from a few helper functions for running specific stages of a simulation. Any other interface has to be built from scratch, but in many cases it wouldn't require a lot of effort. In the future, we plan to also provide an experimental interface for real-time, interactive simulation based on tools like ipywidgets, bokeh and/or holoviews.

1.2.6 Will xarray-simlab support Python 2.7.x?

No, unless there are very good reasons to do so. The main packages of the Python scientific ecosystem support Python 3.4 or later, and it seems that Python 2.x will not be maintained anymore past 2020 (see PEP 373). Although some tools easily allow supporting both Python 2 and 3 versions in a single code base, it still makes the code harder to maintain.

1.2.7 Which features are likely to be implemented in next xarray-simlab releases?

xarray-simlab is a very young project. Some ideas for future development can be found in the roadmap on the xarray-simlab's Github wiki.

1.3 Install xarray-simlab

1.3.1 Required dependencies

- Python 3.5 or later.
- attrs (18.1.0 or later)
- numpy
- xarray (0.10.0 or later)

1.3.2 Optional dependencies

For model visualization

- graphviz

1.3.3 Install using conda

xarray-simlab can be installed or updated using conda:

```
$ conda install xarray-simlab -c conda-forge
```

This installs xarray-simlab and all common dependencies, including numpy and xarray.

xarray-simlab conda package is maintained on the conda-forge channel.

1.3.4 Install using pip

You can also install xarray-simlab and its required dependencies using `pip`:

```
$ pip install xarray-simlab
```

1.3.5 Install from source

To install xarray-simlab from source, be sure you have the required dependencies (numpy and xarray) installed first. You might consider using conda to install them:

```
$ conda install attrs xarray numpy pip -c conda-forge
```

A good practice (especially for development purpose) is to install the packages in a separate environment, e.g. using conda:

```
$ conda create -n simlab_py36 python=3.6 attrs xarray numpy pip -c conda-forge
$ source activate simlab_py36
```

Then you can clone the xarray-simlab git repository and install it using `pip` locally:

```
$ git clone https://github.com/benbovy/xarray-simlab
$ cd xarray-simlab
$ pip install .
```

For development purpose, use the following command:

```
$ pip install -e .
```

1.3.6 Import xarray-simlab

To make sure that xarray-simlab is correctly installed, try to import it by running this line:

```
$ python -c "import xsimlab"
```

1.4 Examples

An example of simple advection is given in the user guide sections. Here below are more advanced examples of using xarray-simlab.

1.4.1 Landscape Evolution Modeling

A showcase of xarray-simlab in the context of landscape evolution modeling (an almost real world example).

```
In [1]: import numpy as np
        import xarray as xr
        import xsimlab as xs
```

Import and inspect a model

The model (i.e., the `xsimlab.Model` object) that we use here is provided by the xarray-topo package (**Note:** check the version of this package below, it may not correspond to the latest stable release).

```
In [2]: import xtopo
        print(xtopo.__version__)
```

```
v0.0.10+2.ga7d7728
```

```
In [3]: from xtopo.models.fastscape_base import fastscape_base_model
```

This model simulates the long-term evolution of topographic surface elevation (hereafter noted h) on a 2D regular grid. The local rate of elevation change, $\partial h/\partial t$, is determined by the balance between uplift (uniform in space and time) U and erosion E.

$$\frac{\partial h}{\partial t} = U - E$$

Total erosion E is the combined effect of the erosion of (bedrock) river channels, noted E_r, and erosion- transport on hillslopes, noted E_d

$$E = E_r + E_d$$

Erosion of river channels is given by the stream power law:

$$E_r = K_r A^m (\nabla h)^n$$

where A is the drainage area and K, m and n are parameters.

Erosion on hillslopes is given by a linear diffusion law:

$$E_d = K_d \nabla^2 h$$

We can see these parameters - as well as the initial elevation surface and the grid parameters - as model inputs in the `repr`.

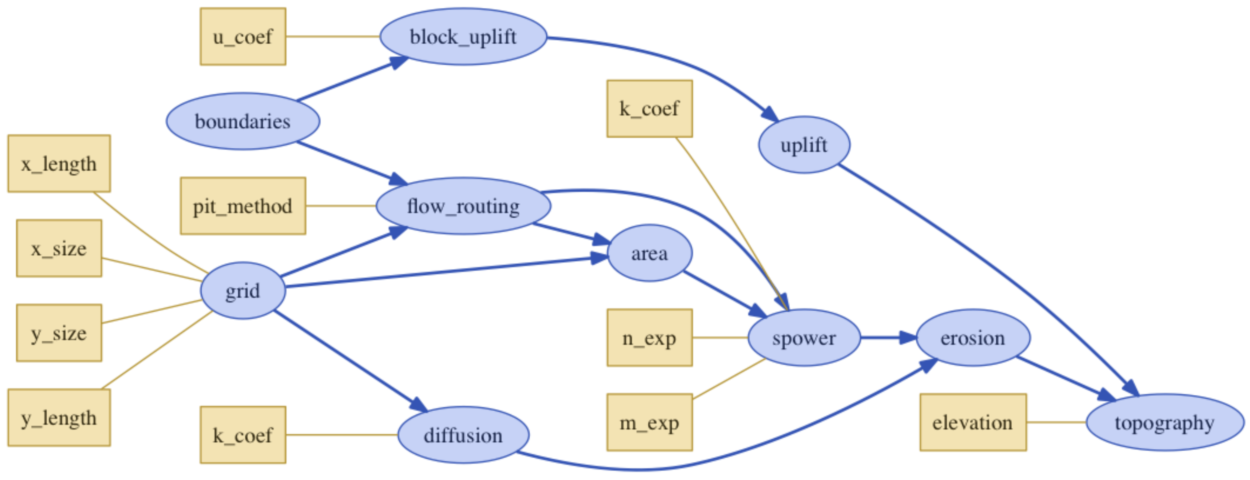
```
In [4]: fastscape_base_model
```

```
Out[4]: <xsimlab.Model (10 processes, 11 inputs)>
        grid
            x_length       [in] total grid length in x
            x_size         [in] nb. of nodes in x
            y_size         [in] nb. of nodes in y
            y_length       [in] total grid length in y
        boundaries
        block_uplift
            u_coef         [in] () or ('y', 'x') uplift rate
        flow_routing
            pit_method     [in]
        area
        spower
            n_exp          [in] stream-power slope exponent
            m_exp          [in] stream-power drainage area exponent
            k_coef         [in] stream-power constant
        diffusion
            k_coef         [in] diffusivity
        erosion
        uplift
        topography
            elevation   [inout] ('y', 'x') topographic elevation
```

To have a better picture of all processes (and inputs and/or variables) in the model, we can visualize it as a graph. Processes are in blue and inputs are in yellow. The order in the graph corresponds to the order in which the processes will be exectued during a simulation.

Note: the visualization requires graphviz and python-graphviz packages (both can be installed using conda and the conda-forge channel).

```
In [5]: fastscape_base_model.visualize(show_inputs=True)
```



More information can be shown for each process in the model, e.g., for the grid

```
In [6]: fastscape_base_model.grid
```

```
Out[6]: <Grid2D 'grid' (xsimlab process)>
        Variables:
            x_size         [in] nb. of nodes in x
            y_size         [in] nb. of nodes in y
            x_length       [in] total grid length in x
            y_length       [in] total grid length in y
            x_spacing     [out]
            y_spacing     [out]
            x             [out] ('x',)
            y             [out] ('y',)
        Simulation stages:
            initialize
```

Create a model setup

We create a simulation setup using the `create_setup` function.

```
In [7]: nx = 101
        ny = 101

        in_ds = xs.create_setup(
            model=fastscape_base_model,
            clocks={
                'time': np.linspace(0., 1e6, 101),
                'out': np.linspace(0., 1e6, 11)
            },
            master_clock='time',
            input_vars={
                'grid': {'x_size': nx, 'y_size': ny, 'x_length': 1e5, 'y_length' :1e5},
                'topography': {'elevation': (('y', 'x'), np.random.rand(ny, nx))},
                'flow_routing': {'pit_method': 'mst_linear'},
                'spower': {'k_coef': 7e-5, 'm_exp': 0.4, 'n_exp': 1},
                'diffusion': {'k_coef': 1.},
                'block_uplift': {'u_coef': 2e-3}
            },
            output_vars={
                'out': {'topography': 'elevation'},
                None: {'grid': ('x', 'y')}
        }
    )
```

Some explanation about the arguments of `create_setup` and the values given above:

- we specify the model we want to use, here `fastscape_base_model`,
- we specify values for clock coordinates (i.e., time coordinates),
- among these coordinates, we specify the master clock, i.e., the coordinate that will be used to set the time steps,
- we set values for model inputs, grouped by process in the model,
- we set the model variables for which we want to take snapshots during a simulation, grouped first by clock coordinate (`None` means that only one snapshot will be taken at the end of the simulation) and then by process.

Here above, we define a 'time' coordinate and another coordinate 'out' with much larger but aligned time steps (the values are in years). 'time' will be used for the simulation time steps and 'out' will be used to take just a few, evenly-spaced snapshots of variable 'elevation' in process 'topography'. We also want to save the x and y coordinates of the grid (values in meters), which are time-invariant.

The initial conditions consist here of a nearly flat topographic surface with small random perturbations.

`create_setup` returns a `xarray.Dataset` object that contains everything we need to run the simulation.

```
In [8]: in_ds

Out[8]: <xarray.Dataset>
        Dimensions:                    (out: 11, time: 101, x: 101, y: 101)
        Coordinates:
          * time                       (time) float64 0.0 1e+04 2e+04 3e+04 4e+04 ...
          * out                        (out) float64 0.0 1e+05 2e+05 3e+05 4e+05 ...
        Dimensions without coordinates: x, y
        Data variables:
            grid__x_size               int64 101
            grid__y_size               int64 101
            grid__x_length             float64 1e+05
            grid__y_length             float64 1e+05
            topography__elevation      (y, x) float64 0.7633 0.6732 0.9937 0.9121 ...
            flow_routing__pit_method   <U10 'mst_linear'
            spower__k_coef             float64 7e-05
            spower__m_exp              float64 0.4
            spower__n_exp              int64 1
            diffusion__k_coef          float64 1.0
            block_uplift__u_coef       float64 0.002
        Attributes:
            __xsimlab_output_vars__:  grid__x,grid__y
```

If present, the metadata (e.g., description, units, math_symbol...) associated to each input variable in the model are added as attributes in the dataset, e.g.,

```
In [9]: in_ds.spower__k_coef

Out[9]: <xarray.DataArray 'spower__k_coef' ()>
        array(7e-05)
        Attributes:
            description:  stream-power constant
```

Run the model

We run the model simply by calling `in_ds.xsimlab.run()`, which returns a new Dataset with both the inputs and the outputs.

```
In [10]: out_ds = in_ds.xsimlab.run(model=fastscape_base_model)

         out_ds

Out[10]: <xarray.Dataset>
         Dimensions:                      (out: 11, time: 101, x: 101, y: 101)
         Coordinates:
           * time                         (time) float64 0.0 1e+04 2e+04 3e+04 4e+04 ...
           * out                          (out) float64 0.0 1e+05 2e+05 3e+05 4e+05 ...
         Dimensions without coordinates: x, y
         Data variables:
             grid__x_size                 int64 101
             grid__y_size                 int64 101
             grid__x_length               float64 1e+05
             grid__y_length               float64 1e+05
             topography__elevation        (out, y, x) float64 0.7633 0.6732 0.9937 ...
             flow_routing__pit_method     <U10 'mst_linear'
             spower__k_coef               float64 7e-05
             spower__m_exp                float64 0.4
             spower__n_exp                int64 1
             diffusion__k_coef            float64 1.0
             block_uplift__u_coef         float64 0.002
             grid__x                      (x) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
             grid__y                      (y) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
```

Note in `out_ds` the `topography__elevation` variable which has now an additional `out` dimension and also the new variables `grid__x` and `grid__y`.

Analyse, plot and save the results

The simulation input and output data is already in a format that allows us using all the nice features of xarray to further analyse, process, plot and/or write to disk (e.g., in a netCDF file) the data.

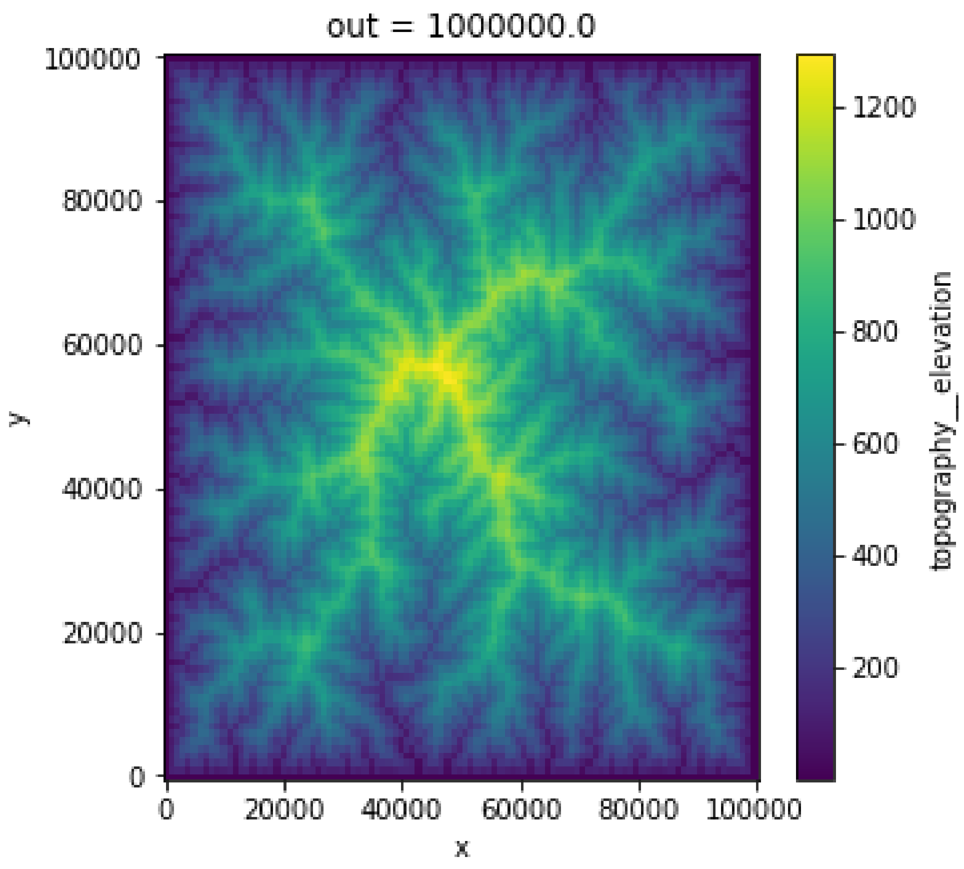
In this case for example, before doing any further processing it is more convenient to set x and y coordinates as coordinates of the output `Dataset` instead of data variables, using the `set_index` method. We can easily chain this method with `xsimlab.run` as it both return Dataset objects:

```
In [11]: out_ds = (in_ds
                   .xsimlab.run(model=fastscape_base_model)
                   .set_index(x='grid__x', y='grid__y'))

         out_ds

Out[11]: <xarray.Dataset>
         Dimensions:                      (out: 11, time: 101, x: 101, y: 101)
         Coordinates:
           * time                         (time) float64 0.0 1e+04 2e+04 3e+04 4e+04 ...
           * out                          (out) float64 0.0 1e+05 2e+05 3e+05 4e+05 ...
           * x                            (x) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
           * y                            (y) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
         Data variables:
             grid__x_size                 int64 101
             grid__y_size                 int64 101
             grid__x_length               float64 1e+05
             grid__y_length               float64 1e+05
             topography__elevation        (out, y, x) float64 0.7633 0.6732 0.9937 ...
             flow_routing__pit_method     <U10 'mst_linear'
             spower__k_coef               float64 7e-05
             spower__m_exp                float64 0.4
```

```
    spower__n_exp             int64 1
    diffusion__k_coef         float64 1.0
    block_uplift__u_coef      float64 0.002
```

It is then easier to plot the simulation outputs, e.g., here below the elevation values at the end of the simulation:

```
In [12]: %matplotlib inline

         xr.plot.pcolormesh(out_ds.isel(out=-1).topography__elevation,
                            size=5, aspect=1);
```



xarray datasets can be used with Holoview, a plotting package that is really helpful for quickly and interactively exploring multi-dimensional data. (it can be installed using conda).

```
In [13]: import holoviews as hv

         hv.notebook_extension('matplotlib')
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

We can for example see below how the relief is created during the simulation (snapshots are taken every 100000 years and elevation values are in meters).

Note: There may be issues with rendering Holoview interactive visualizations if you are on xarray-simlab's documentation. Fortunately you should be able to see this page as a notebook properly rendered on nbviewer.org.

```
In [14]: %%opts Image style(interpolation='bilinear', cmap='viridis') plot[colorbar=True]
         hv_ds = hv.Dataset(out_ds.topography__elevation)
         hv_ds.to(hv.Image, ['x', 'y'])
```

```
Out[14]: :HoloMap   [out]
            :Image   [x,y]   (topography__elevation)
```

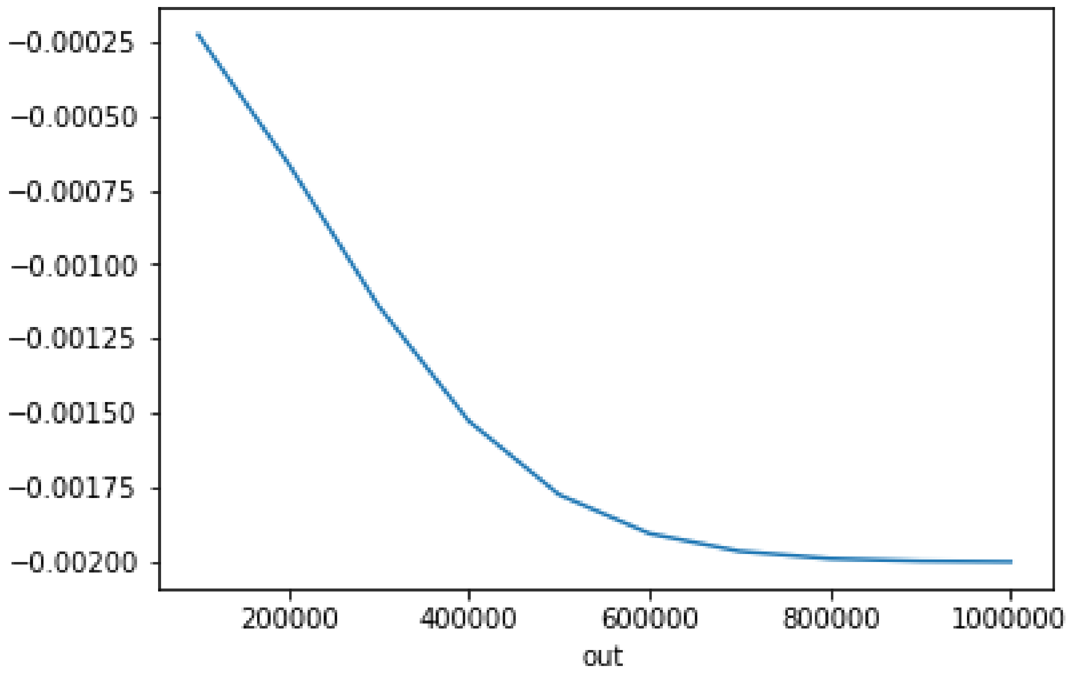
Additionally, We can compute derived quantities without much effort. Here below we calculate the surface denudation rates (in m/yr) averaged over each time steps of the output `out` dimension.

```
In [15]: def denudation_rate(ds, time_dim='out'):
             topo = ds.topography__elevation
             dt = ds[time_dim].diff(time_dim)
             den_rate = topo.diff(time_dim) / dt - ds.block_uplift__u_coef
             return den_rate
```

```
In [16]: den_rate = denudation_rate(out_ds)
```

We further compute and plot the spatially averaged denudation rate.

```
In [17]: den_rate.mean(('x', 'y')).plot();
```

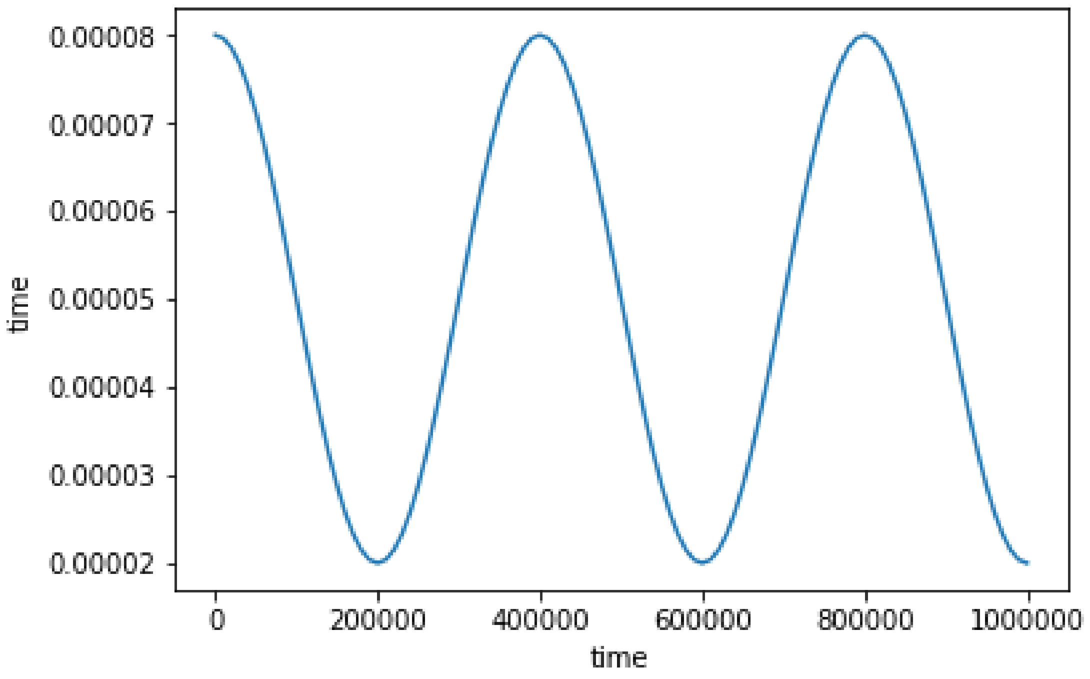


Run the model with time-varying parameter values

Instead of providing constant, scalar values for model inputs, it is possible to provide arrays which have the same dimension as the one used for the "master clock" (the `time` dimension in this case).

As an example, we try below a sinusoidal variation for the K parameter of the stream power law, with a period of 400000 years.

```
In [18]: da_k_time = 5e-5 + 3e-5 * np.cos((2 * np.pi / 4e5) * in_ds.time)

         da_k_time.plot();
```



We then re-use the simulation setup created above and only update the parameters of the stream-power process with the new values for K (using `Dataset.xsimlab.update_vars`).

Note the `time` dimension of the `spower__k_coef` variable in the new returned Dataset.

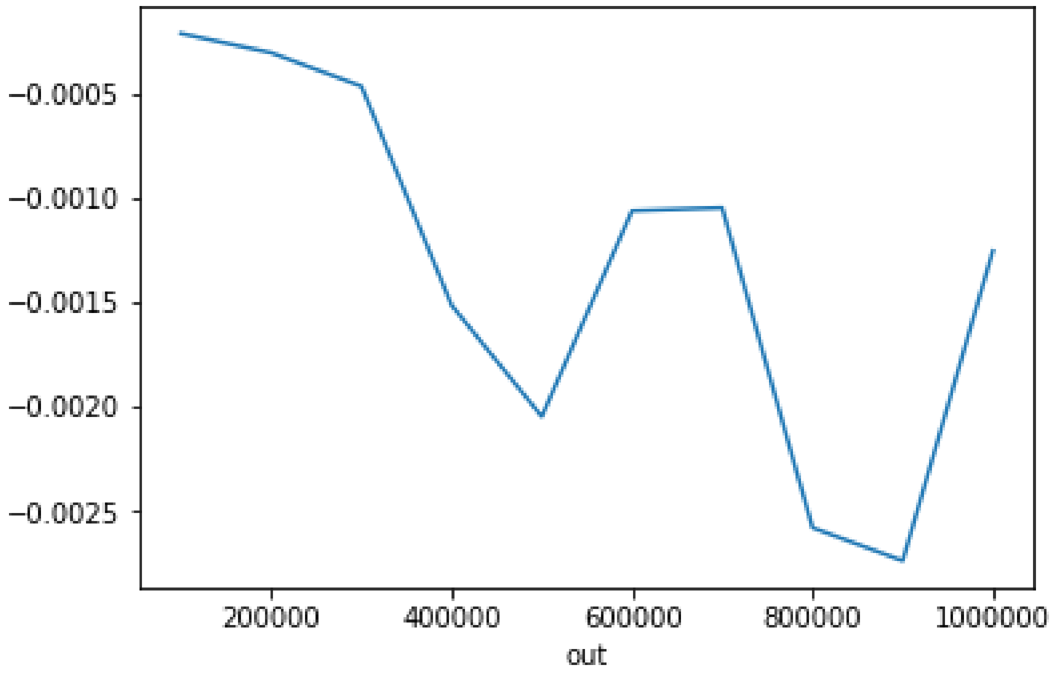
```
In [19]: in_ds_kt = in_ds.xsimlab.update_vars(
             model=fastscape_base_model,
             input_vars={'spower': {'k_coef': da_k_time, 'm_exp': 0.4, 'n_exp': 1}}
         )

         in_ds_kt
```

```
Out[19]: <xarray.Dataset>
         Dimensions:                  (out: 11, time: 101, x: 101, y: 101)
         Coordinates:
           * time                     (time) float64 0.0 1e+04 2e+04 3e+04 4e+04 ...
           * out                      (out) float64 0.0 1e+05 2e+05 3e+05 4e+05 ...
         Dimensions without coordinates: x, y
         Data variables:
             grid__x_size             int64 101
             grid__y_size             int64 101
             grid__x_length           float64 1e+05
             grid__y_length           float64 1e+05
             topography__elevation    (y, x) float64 0.7633 0.6732 0.9937 0.9121 ...
             flow_routing__pit_method <U10 'mst_linear'
             spower__k_coef           (time) float64 8e-05 7.963e-05 7.853e-05 ...
             spower__m_exp            float64 0.4
             spower__n_exp            int64 1
             diffusion__k_coef        float64 1.0
             block_uplift__u_coef     float64 0.002
         Attributes:
             __xsimlab_output_vars__:  grid__x,grid__y
```

We then run the model, unstack the spatial dimensions, compute the denudation rates and plot the spatial averages, here again by easily chaining xarray and xarray-simlab methods on the input Dataset.

If we compare the results with those from the previous run, we clearly see the impact of the time-varying K parameter values on the denudation rates.

```
In [20]: den_rate_kt = (in_ds_kt
                        .xsimlab.run(model=fastscape_base_model)
                        .set_index(x='grid__x', y='grid__y')
                        .pipe(denudation_rate))

         den_rate_kt.mean(('x', 'y')).plot();
```



Run and combine different model setups

Here is an brief example of running the model multiple times for different fixed values of K and then concatenate the results into a single dataset. In next versions of xarray-simlab, this process will be even simpler.

```
In [21]: def run_model(k_value):
             print('run k=%f' % k_value)

             ivars = {'spower': {'k_coef': k_value, 'm_exp': 0.4, 'n_exp': 1}}

             out_ds = (in_ds
                       .xsimlab.update_vars(model=fastscape_base_model,
                                            input_vars=ivars)
                       .xsimlab.run(model=fastscape_base_model)
                       .set_index(x='grid__x', y='grid__y'))

             return out_ds


         out_ds_multi = xr.concat(
             [run_model(k) for k in (5e-5, 6e-5, 7e-5)],
             dim='spower__k_coef', data_vars='different'
         )
```

```
run k=0.000050
run k=0.000060
run k=0.000070
```

Note the additional `spower__k_coef` dimension, which has its own coordinate with labels corresponding to the different K values.

```
In [22]: out_ds_multi
```

```
Out[22]: <xarray.Dataset>
         Dimensions:                      (out: 11, spower__k_coef: 3, time: 101, x: 101, y: 101)
         Coordinates:
           * time                         (time) float64 0.0 1e+04 2e+04 3e+04 4e+04 ...
           * out                          (out) float64 0.0 1e+05 2e+05 3e+05 4e+05 ...
           * x                            (x) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
           * y                            (y) float64 0.0 1e+03 2e+03 3e+03 4e+03 5e+03 ...
           * spower__k_coef               (spower__k_coef) float64 5e-05 6e-05 7e-05
         Data variables:
             grid__x_size                 int64 101
             grid__y_size                 int64 101
             grid__x_length               float64 1e+05
             grid__y_length               float64 1e+05
             flow_routing__pit_method     <U10 'mst_linear'
             spower__m_exp                float64 0.4
             spower__n_exp                int64 1
             diffusion__k_coef            float64 1.0
             block_uplift__u_coef         float64 0.002
             topography__elevation        (spower__k_coef, out, y, x) float64 0.7633 ...
```

This new dimension also appears in the Holoview figure

```
In [23]: %%opts Image style(interpolation='bilinear', cmap='viridis') plot[colorbar=True]
         hv_ds = hv.Dataset(out_ds_multi.topography__elevation)
         hv_ds.to(hv.Image, ['x', 'y'])
```

```
Out[23]: :HoloMap   [out,spower__k_coef]
            :Image   [x,y]   (topography__elevation)
```

Create an alternative version of the model

xarray-simlab makes it easy to create alternative versions of a model. In the example below, instead of using constant block uplift, we set a linear uplift function along the x dimension. The first step is to create a new process, i.e., a Python class decorated by `xsimlab.process`.

```
In [24]: from xtopo.models.fastscape_base import Grid2D, ClosedBoundaryFaces


         @xs.process
         class VariableUplift(object):
             """Compute spatially variable uplift as a linear function of x."""

             x_coef = xs.variable(description='uplift function x coefficient')

             active_nodes = xs.foreign(ClosedBoundaryFaces, 'active_nodes')
             x = xs.foreign(Grid2D, 'x')

             uplift = xs.variable(intent='out', group='uplift')

             def initialize(self):
                 mask = self.active_nodes
```

```
        ny, nx = mask.shape

        u_rate = np.ones((ny, nx)) * self.x_coef * self.x[None, :]

        self._u_rate = np.zeros((ny, nx))
        self._u_rate[mask] = u_rate[mask]

    def run_step(self, dt):
        self.uplift = self._u_rate * dt
```

We then update the model that we used above with the new process (note the change in repr: the `uplift` process has now an `x_coef` input).

```
In [25]: alt_model = (fastscape_base_model.drop_processes('block_uplift')
                                          .update_processes({'uplift_func': VariableUplift}))

         alt_model

Out[25]: <xsimlab.Model (10 processes, 11 inputs)>
         grid
             x_length       [in] total grid length in x
             x_size         [in] nb. of nodes in x
             y_length       [in] total grid length in y
             y_size         [in] nb. of nodes in y
         boundaries
         flow_routing
             pit_method     [in]
         area
         spower
             n_exp          [in] stream-power slope exponent
             m_exp          [in] stream-power drainage area exponent
             k_coef         [in] stream-power constant
         diffusion
             k_coef         [in] diffusivity
         erosion
         uplift_func
             x_coef         [in] uplift function x coefficient
         uplift
         topography
             elevation   [inout] ('y', 'x') topographic elevation
```

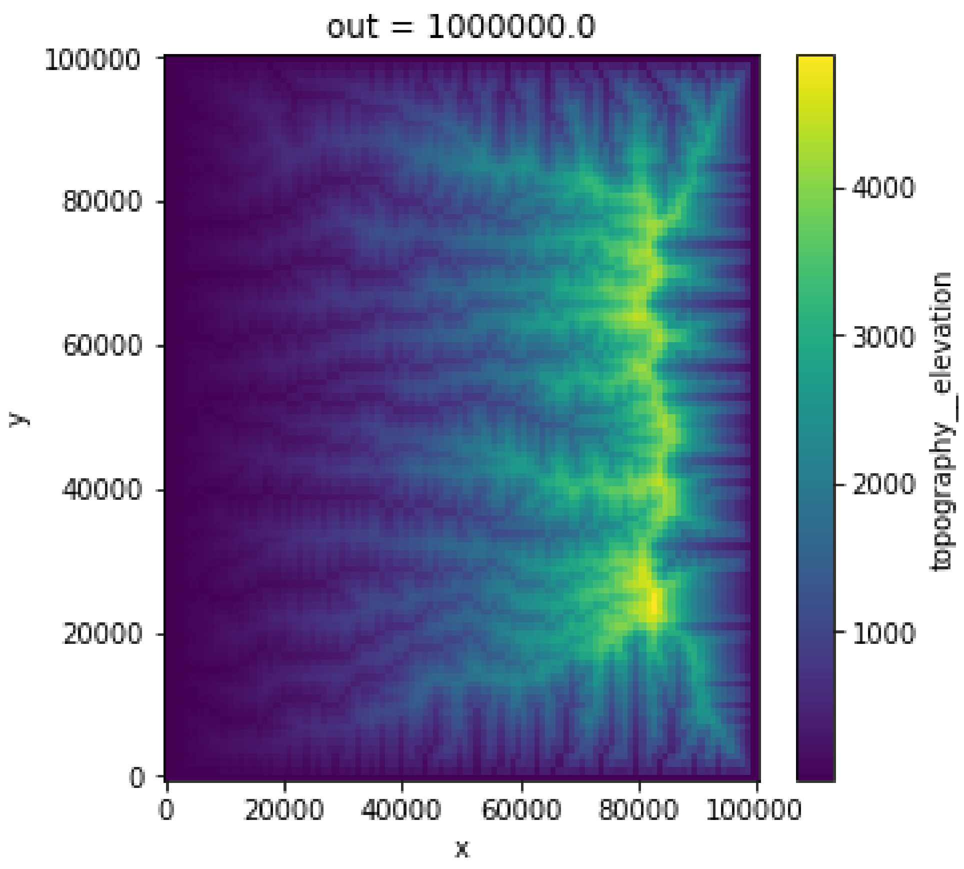
We then re-use our intial setup, remove from this setup everything that is not related to the new model (using `Dataset.xsimlab.filter_vars` which here drops the `uplift__u_coef` variable), update the setup with the new parameter and then run the model.

Note that in some cases it is convenient to use the `with` statement with a `Model` object. For example we don't need to provide the `model` argument in `filter_vars`, `update_vars` and `run` methods below.

```
In [26]: with alt_model:
             out_ds_alt = (
                 in_ds
                 .xsimlab.filter_vars()
                 .xsimlab.update_vars(input_vars={'uplift_func': {'x_coef': 1e-7}})
                 .xsimlab.run()
                 .set_index(x='grid__x', y='grid__y')
             )
```

You can compare the results obtained here with the results obtained above.

```
In [27]: xr.plot.pcolormesh(out_ds_alt.isel(out=-1).topography__elevation,
                            size=5, aspect=1);
```



```
In [28]: %%opts Image style(interpolation='bilinear', cmap='viridis') plot[colorbar=True]
         hv_ds = hv.Dataset(out_ds_alt.topography__elevation)
         hv_ds.to(hv.Image, ['x', 'y'])
```

```
Out[28]: :HoloMap   [out]
            :Image   [x,y]   (topography__elevation)
```

```
In [ ]:
```

User Guide

- *Modeling Framework*
- *Create and Modify Models*
- *Inspect Models*
- *Setup and Run Models*

1.5 Modeling Framework

This section explains the design of the xarray-simlab modeling framework. It is useful mostly for users who want to create new models from scratch or customize existing models. Users who only want to run simulations from existing models may skip this section.

For more practical details on how using the API to create, inspect and run models, see the relevant sections of this user guide.

1.5.1 Main concepts

The xarray-simlab framework is built on a very few concepts that allow great flexibility in model customization:

- models
- processes
- variables

These are detailed here below.

1.5.2 Models

Models are instances of the `Model` class. They consist of ordered, immutable collections of processes. The ordering is inferred automatically from the given processes (see below).

The Model class also implements specific methods for:

- introspection,
- running simulations,
- easy creation of new Model objects from existing ones by dropping, adding or replacing one or more processes.

1.5.3 Processes

Processes are defined as Python classes that are decorated by `process()`. The role of a process is twofold:

- declare a given subset of the variables used in a model,
- define a specific set of instructions that use or compute values for these variables during a model run.

Conceptually, a process is a logical component of a computational model. It may for example represent a particular physical mechanism that is described in terms of one or more state variables (e.g., scalar or vector fields) and one or more operations – with or without parameters – that modify those state variables through time. Note that some processes may be time-independent or may even be used to declare variables without implementing any computation.

Note: xarray-simlab does not provide any built-in logic for tasks like generating computational meshes or setting boundary conditions, which should rather be implemented in 3rd-party libraries as processes. Even those tasks may be too specialized to justify including them in this framework, which aims to be as general as possible.

A process-ified class behaves mostly like any other regular Python class, i.e., there is a-priori nothing that prevents you from using the common object-oriented features as you like. The only difference is that you can here create classes in a very succinct way without boilerplate, i.e., you don’t need to implement dunder methods like `__init__` or `__repr__` as this is handled by the framework. In fact, this framework uses and extends the attrs package: `process()` is a wrapper around `attr.s()` and the functions used to create variables (see below) are thin wrappers around `attr.ib()`.

1.5.4 Variables

Variables are the most basic elements of a model. They are declared in processes as class attributes, using `variable()`. Declaring variables mainly consists of defining useful metadata such as:

- labeled dimensions (or no dimension for scalars),
- predefined meta-data attributes, e.g., a short description,

- user-defined meta-data attributes, e.g., units or math symbol,
- the intent for a variable, i.e., whether the process needs (`intent='in'`), updates (`intent='inout'`) or computes (`intent='out'`) a value for that variable.

Note: xarray-simlab does not distinguish between model parameters, input and output variables. All can be declared using *`variable()`*.

Foreign variables

Like different physical mechanisms involve some common state variables (e.g., temperature or pressure), different processes may operate on common variables.

In xarray-simlab, a variable is declared at a unique place, i.e., within one and only one process. Using common variables across processes is achieved by declaring *`foreign()`* variables. These are simply references to variables that are declared in other processes.

You can use foreign variables for almost any computation inside a process just like original variables. The only difference is that `intent='inout'` is not supported for a foreign variable, i.e., a process may either need or compute a value of a foreign variable but may not update it (otherwise it would not be possible to unambiguously determine process dependencies – see below). For the same reason, only one process in a model may compute a value of a variable (i.e., `intent='out'`).

The great advantage of declaring variables at unique places is that all their meta-data are defined once. However, a downside of this approach is that foreign variables may potentially add many hard-coded links between processes, which makes harder reusing these processes independently of each other.

Group variables

In some cases, using group variables may provide an elegant alternative to hard-coded links between processes.

The membership of variables to a group is defined via their `group` attribute. If you want to use in a separate process all the variables of a group, instead of explicitly declaring foreign variables you can declare a *`group()`* variable. The latter behaves like an iterable of foreign variables pointing to each of the variables that are members of the group, across the model.

Note that group variables only support `intent='in'`, i.e, group variables should only be used to get the values of multiple foreign variables of a same group.

Group variables are useful particularly in cases where you want to combine (aggregate) different processes that act on the same variable, e.g. in landscape evolution modeling combine the effect of different erosion processes on the evolution of the surface elevation. This way you can easily add or remove processes to/from a model and avoid missing or broken links between processes.

On-demand variables

On-demand variables are like regular variables, except that their value is not intended to be computed systematically, e.g., at the beginning or at each time step of a simulation, but instead only at a given few times (or not at all). These are declared using *`on_demand()`* and must implement in the same process-ified class a dedicated method – i.e., decorated with `@foo.compute` where `foo` is the name of the variable – that returns their value. They have always `intent='out'`.

On-demand variables are useful, e.g., for optional model diagnostics.

1.5.5 Simulation workflow

A model run is divided into four successive stages:

1. initialization
2. run step
3. finalize step
4. finalization

During a simulation, stages 1 and 4 are run only once while stages 2 and 3 are repeated for a given number of (time) steps.

Each process-ified class may provide its own computation instructions by implementing specific methods named `.initialize()`, `.run_step()`, `.finalize_step()` and `.finalize()` for each stage above, respectively. Note that this is entirely optional. For example, time-independent processes (e.g., for setting model grids) usually implement stage 1 only. In a few cases, the role of a process may even consist of just declaring some variables that are used elsewhere.

1.5.6 Get / set variable values inside a process

Once you have declared a variable as a class attribute in a process, you can further get and/or set its value like it was defined as a property of that class. For example, if you declare a variable `foo` you can just use `self.foo` to get/set its value inside one method of that class.

This is exactly what does the *`process()`* decorator: it takes all variables declared as class attributes and turns them into properties, which may be read-only depending on the `intent` set for the variables.

Basically, the getter (setter) methods of these properties read (write) values from (into) a simple key-value store (except for on-demand variables). Currently the store is fully in-memory but it could be easily replaced by an on-disk or a distributed store. The xarray-simlab's modeling framework can thus be viewed as a thin object-oriented layer built on top of an abstract key-value store.

1.5.7 Process dependencies and ordering

The order in which processes are executed during a simulation is critical. For example, if the role of a process is to compute a value for a given variable, then the execution of this process must happen before the execution of all other processes that use the same variable in their computation.

In a model, the processes and their dependencies together form the nodes and the edges of a Directed Acyclic Graph (DAG). The graph topology is fully determined by the `intent` set for each variable or foreign variable declared in each process. An ordering that is computationally consistent can then be obtained using topological sorting. This is done at Model object creation. The same ordering is used at every stage of a model run.

In principle, the DAG structure would also allow running the processes in parallel at every stage of a model run. This is not yet implemented, though.

1.5.8 Model inputs

In a model, inputs are variables that need a value to be set by the user before running a simulation.

Like process ordering, inputs are automatically retrieved at Model object creation by looking at the `intent` set for all variables and foreign variables in the model. A variable is a model input if it has `intent` set to `'in'` or `'inout'` and if it has no linked foreign variable with `intent='out'`.

1.6 Create and Modify Models

Like the previous *Modeling Framework* section, this section is useful mostly for users who want to create new models from scratch or customize existing models. Users who only want to run simulations from existing models may skip this section.

As a simple example, we will start here from a model which numerically solves the 1-d advection equation using the Lax method. The equation may be written as:

$$\frac{\partial u}{\partial t} + \nu \frac{\partial u}{\partial x} = 0$$

with $u(x, t)$ as the quantity of interest and where ν is the velocity. The discretized form of this equation may be written as:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \nu \frac{\Delta t}{2 \Delta x}(u_{i+1}^n - u_{i-1}^n)$$

Where Δx is the fixed spacing between the nodes i of a uniform grid and Δt is the step duration between times n and $n + 1$.

We could just implement this numerical model with a few lines of Python / Numpy code, e.g., here below assuming periodic boundary conditions and a Gaussian pulse as initial profile. We will show, however, that it is very easy to refactor this code for using it with xarray-simlab. We will also show that, while enabling useful features, the refactoring still results in a short amount of readable code.

```python
import numpy as np


# grid
spacing = 0.01
length = 1.5
x = np.arange(0, length, spacing)

# velocity
v = 1.

# time
start = 0.
end = 1.
step = 0.01

# initial gauss profile
loc = 0.3
scale = 0.1
u = np.exp(-1 / scale**2 * (x - loc)**2)
u0 = u.copy()

# time loop - Lax method
factor = (v * step) / (2 * spacing)

for t in np.arange(start, end, step):
    u_left = np.roll(u, 1)
    u_right = np.roll(u, -1)
    u1 = 0.5 * (u_right + u_left) - factor * (u_right - u_left)
    u = u1.copy()
```

1.6.1 Anatomy of a Process subclass

Let's first wrap the code above into a single class named `AdvectionLax1D` decorated by *process*. Next we'll explain in detail the content of this class.

```
import xsimlab as xs


@xs.process
class AdvectionLax1D(object):
    """Wrap 1-dimensional advection in a single Process."""

    spacing = xs.variable(description='grid spacing')
    length = xs.variable(description='grid total length')
    x = xs.variable(dims='x', intent='out')

    v = xs.variable(dims=[(), 'x'], description='velocity')

    loc = xs.variable(description='location of initial profile')
    scale = xs.variable(description='scale of initial profile')
    u = xs.variable(dims='x', intent='out', description='quantity u',
                    attrs={'units': 'm'})

    def initialize(self):
        self.x = np.arange(0, self.length, self.spacing)
        self.u = np.exp(-1 / self.scale**2 * (self.x - self.loc)**2)

    def run_step(self, dt):
        factor = (self.v * dt) / (2 * self.spacing)
        u_left = np.roll(self.u, 1)
        u_right = np.roll(self.u, -1)
        self.u1 = 0.5 * (u_right + u_left) - factor * (u_right - u_left)

    def finalize_step(self):
        self.u = self.u1
```

Process interface

`AdvectionLax1D` has some class attributes declared at the top, which together form the process' "public" interface, i.e., all the variables that we want to be publicly exposed by this process. Here we use *variable()* to add some metadata to each variable of the interface.

We first may specify the labels of the dimensions expected for each variable, which defaults to an empty tuple (i.e., a scalar value is expected). In this example, variables `spacing`, `length`, `loc` and `scale` are all scalars, whereas `x` and `u` are both arrays defined on the 1-dimensional x grid. Multiple choices can also be given as a list, like variable `v` which represents a velocity field that can be either constant (scalar) or variable (array) in space.

Note: All variable objects also implicitly allow a time dimension. See section *Setup and Run Models*.

Additionally, it is also possible to add a short `description` and/or custom metadata like units with the `attrs` argument.

Another important argument is `intent`, which specifies how the process deals with the value of the variable. By default, `intent='in'` means that the process just needs the value of the variable for its computation ; this value should either be computed elsewhere by another process or be provided by the user as model input. By contrast, variables `x` and `u` have `intent='out'`, which means that the process `AdvectionLax1D` itself initializes and computes a value for these two variables.

Process "runtime" methods

Beside its interface, the process `AdvectionLax1D` also implements methods that will be called during simulation runtime:

- `.initialize()` will be called once at the beginning of a simulation. Here it is used to set the x-coordinate values of the grid and the initial values of `u` along the grid (Gaussian pulse).
- `.run_step()` will be called at each time step iteration and have the current time step duration as required argument. This is where the Lax method is implemented.
- `.finalize_step()` will be called at each time step iteration too but after having called `run_step` for all other processes (if any). Its intended use is mainly to ensure that state variables like `u` are updated consistently and after having taken snapshots.

A fourth method `.finalize()` could also be implemented, but it is not needed in this case. This method is called once at the end of the simulation, e.g., for some clean-up.

Getting / setting variable values

For each variable declared as class attributes in `AdvectionLax1D` we can get their value (and/or set a value depending on their `intent`) elsewhere in the class like if it was defined as regular instance attributes, e.g., using `self.u` for variable `u`.

Note: In xarray-simlab it is safe to run multiple simulations concurrently: each simulation has its own process instances.

Beside variables declared in the process interface, nothing prevent us from using regular attributes in process classes if needed. For example, `self.u1` is set as a temporary internal state in `AdvectionLax1D` to wait for the "finalize step" stage before updating u.

1.6.2 Creating a Model instance

Creating a new *`Model`* instance is very easy. We just need to provide a dictionary with the process class(es) that we want to include in the model, e.g., with only the process created above:

```
model1 = xs.Model({'advect': AdvectionLax1D})
```

That's it! Now we have different tools already available to inspect the model (see section *Inspect Models*). We can also use that model with the xarray extension provided by xarray-simlab to create new setups, run the model, take snapshots for one or more variables on a given frequency, etc. (see section *Setup and Run Models*).

1.6.3 Fine-grained process refactoring

The model created above isn't very flexible. What if we want to change the initial conditions? Use a grid with variable spacing? Add another physical process impacting u such as a source or sink term? In all cases we would need to modify the class `AdvectionLax1D`.

This framework works best if we instead split the problem into small pieces, i.e., small process classes that we can easily combine and replace in models.

The `AdvectionLax1D` process may for example be refactored into 4 separate processes:

- `UniformGrid1D` : grid creation
- `ProfileU` : update u values along the grid at each time iteration
- `AdvectionLax` : perform advection at each time iteration
- `InitUGauss` : create initial u values along the grid.

UniformGrid1D

This process declares all grid-related variables and computes x-coordinate values.

```
@xs.process
class UniformGrid1D(object):
    """Create a 1-dimensional, equally spaced grid."""

    spacing = xs.variable(description='uniform spacing')
    length = xs.variable(description='total length')
    x = xs.variable(dims='x', intent='out')

    def initialize(self):
        self.x = np.arange(0, self.length, self.spacing)
```

Grid x-coordinate values only need to be set once at the beginning of the simulation ; there is no need to implement `.run_step()` here.

ProfileU

```
@xs.process
class ProfileU(object):
    """Compute the evolution of the profile of quantity `u`."""

    u_vars = xs.group('u_vars')
    u = xs.variable(dims='x', intent='inout', description='quantity u',
                    attrs={'units': 'm'})

    def run_step(self, *args):
        self._delta_u = sum((v for v in self.u_vars))

    def finalize_step(self):
        self.u += self._delta_u
```

`u_vars` is declared as a *`group()`* variable, i.e., an iterable of all variables declared elsewhere that belong the same group (‘u_vars’ in this case). In this example, it allows to further add one or more processes that will also affect the evolution of u in addition to advection (see below).

Note also `intent='inout'` set for `u`, which means that `ProfileU` updates the value of u but still needs an initial value from elsewhere.

AdvectionLax

```
@xs.process
class AdvectionLax(object):
    """Advection using finite difference (Lax method) on
    a fixed grid with periodic boundary conditions.

    """
    v = xs.variable(dims=[(), 'x'], description='velocity')
    grid_spacing = xs.foreign(UniformGrid1D, 'spacing')
```

(continues on next page)

(continued from previous page)

```
    u = xs.foreign(ProfileU, 'u')
    u_advected = xs.variable(dims='x', intent='out', group='u_vars')

    def run_step(self, dt):
        factor = self.v / (2 * self.grid_spacing)

        u_left = np.roll(self.u, 1)
        u_right = np.roll(self.u, -1)
        u_1 = 0.5 * (u_right + u_left) - factor * dt * (u_right - u_left)

        self.u_advected = u_1 - self.u
```

`u_advected` represents the effect of advection on the evolution of u and therefore belongs to the group ‘u_vars’.

Computing values of `u_advected` requires values of variables `spacing` and `u` that are already declared in the `UniformGrid1D` and `ProfileU` classes, respectively. Here we declare them as *`foreign()`* variables, which allows to handle them like if these were the original variables. For example, `self.grid_spacing` in this class will return the same value than `self.spacing` in `UniformGrid1D`.

InitUGauss

```
@xs.process
class InitUGauss(object):
    """Initialize `u` profile using a Gaussian pulse."""

    loc = xs.variable(description='location of initial pulse')
    scale = xs.variable(description='scale of initial pulse')
    x = xs.foreign(UniformGrid1D, 'x')
    u = xs.foreign(ProfileU, 'u', intent='out')

    def initialize(self):
        self.u = np.exp(-1 / self.scale**2 * (self.x - self.loc)**2)
```

A foreign variable can also be used to set values for variables that are declared in other processes, as for `u` here with `intent='out'`.

Refactored model

We now have all the building blocks to create a more flexible model:

```
model2 = xs.Model({'grid': UniformGrid1D,
                   'profile': ProfileU,
                   'init': InitUGauss,
                   'advect': AdvectionLax})
```

The order in which processes are given doesn’t matter (it is a dictionary). A computationally consistent order, as well as model inputs among all declared variables, are both automatically figured out when creating the Model instance.

In terms of computation and inputs, `model2` is equivalent to the `model1` instance created above ; it is just organized differently.

1.6.4 Update existing models

Between the two Model instances created so far, the advantage of `model2` over `model1` is that we can easily update the model – change its behavior and/or add many new features – without sacrificing readability or losing the ability to get back to the original, simple version.

Example: adding a source term at a specific location

For this we create a new process:

```
@xs.process
class SourcePoint(object):
    """Source point for quantity `u`.

    The location of the source point is adjusted to coincide with
    the nearest node the grid.

    """
    loc = xs.variable(description='source location')
    flux = xs.variable(description='source flux')
    x = xs.foreign(UniformGrid1D, 'x')
    u_source = xs.variable(dims='x', intent='out', group='u_vars')

    @property
    def nearest_node(self):
        idx = np.abs(self.x - self.loc).argmin()
        return idx

    @property
    def source_rate(self):
        src_array = np.zeros_like(self.x)
        src_array[self.nearest_node] = self.flux
        return src_array

    def run_step(self, dt):
        self.u_source = self.source_rate * dt
```

Some comments about this class:

- `u_source` belongs to the group 'u_vars' and therefore will be added to `u_advected` in `ProfileU` process.
- Methods and/or properties other than the reserved "runtime" methods may be added in a Process subclass, just like in any other Python class.
- Nearest node index and source rate array will be recomputed at each time iteration because variables `loc` and `flux` may both have a time dimension (variable source location and intensity), i.e., `self.loc` and `self.flux` may both change at each time iteration.

In this example we also want to start with a flat, zero u profile instead of a Gaussian pulse. We create another (minimal) process for that:

```
@xs.process
class InitUFlat(object):
    """Flat initial profile of `u`."""

    x = xs.foreign(UniformGrid1D, 'x')
    u = xs.foreign(ProfileU, 'u', intent='out')

    def initialize(self):
        self.u = np.zeros_like(self.x)
```

Using one command, we can then update the model with these new features:

```
model3 = model2.update_processes({'source': SourcePoint,
                                  'init': InitUFlat})
```

Compared to model2, this new model3 have a new process named 'source' and a replaced process 'init'.

Removing one or more processes

It is also possible to create new models by removing one or more processes from existing Model instances, e.g.,

```
model4 = model2.drop_processes('init')
```

In this latter case, users will have to provide initial values of u along the grid directly as an input array.

Note: Model instances are immutable, i.e., once created it is not possible to modify these instances by adding, updating or removing processes. Both methods .update_processes() and .drop_processes() always return new instances of Model.

1.7 Inspect Models

We can inspect xarray-simlab's *Model* objects in different ways. As an example we'll use here the object model2 which has been created in the previous section *Create and Modify Models* of this user guide.

```
In [1]: import xsimlab as xs
```

1.7.1 Inspect model inputs

Model *repr* already gives information about the number and names of processes and their variables that need an input value (if any):

```
In [2]: model2
Out[2]:
<xsimlab.Model (4 processes, 5 inputs)>
grid
    spacing     [in] uniform spacing
    length      [in] total length
init
    scale       [in] scale of initial pulse
    loc         [in] location of initial pulse
advect
    v           [in] () or ('x',) velocity
profile
```

For each input, a one-line summary is shown with the intent (either 'in' or 'inout') as well as the dimension labels for inputs that don't expect a scalar value only. If provided, a short description is also displayed in the summary.

The convenient property *input_vars* of Model returns all inputs as a list of 2-length tuples with process and variable names, respectively.

```
In [3]: model2.input_vars
Out[3]:
[('advect', 'v'),
 ('grid', 'spacing'),
 ('init', 'scale'),
 ('grid', 'length'),
 ('init', 'loc')]
```

input_vars_dict returns all inputs grouped by process, as a dictionary:

```
In [4]: model2.input_vars_dict
Out[4]: {'advect': ['v'], 'grid': ['spacing', 'length'], 'init': ['scale', 'loc']}
```

1.7.2 Inspect processes and variables

For deeper inspection, Model objects support both dict-like and attribute-like access to their processes, e.g.,

```
In [5]: model2['advect']
Out[5]:
<AdvectionLax 'advect' (xsimlab process)>
Variables:
    v                 [in] () or ('x',) velocity
    grid_spacing      [in] <--- grid.spacing
    u                 [in] <--- profile.u
    u_advected       [out] ('x',)
Simulation stages:
    run_step

In [6]: model2.grid
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
<UniformGrid1D 'grid' (xsimlab process)>
Variables:
    spacing     [in] uniform spacing
    length      [in] total length
    x          [out] ('x',)
Simulation stages:
    initialize
```

As shown here above, process *repr* includes:

- the name to the process class and the name of the process in the model (top line) ;
- a "Variables" section with all variables declared in the process (not only model inputs) including one-line summaries that depend on their type (i.e., `variable`, `foreign`, `group`, etc.) ;
- a "Simulation stages" section with the stages that are implemented in the process.

It is also possible to inspect a process class taken individually with *`process_info()`*:

```
In [7]: xs.process_info(ProfileU)
<ProfileU  (xsimlab process)>
Variables:
    u_vars      [in] <--- group 'u_vars'
    u        [inout] ('x',) quantity u
Simulation stages:
    run_step
    finalize_step
```

Similarly, *`variable_info()`* allows inspection at the variable level:

```
In [8]: xs.variable_info(ProfileU, 'u')
Quantity u

- type : variable
- intent : inout
- dims : (('x',),)
```

(continues on next page)

(continued from previous page)

```
- group : None
- attrs : {'units': 'm'}

In [9]: xs.variable_info(model2.profile, 'u_vars')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪of all variables that belong to group 'u_vars'

- type : group
- intent : in
- group : u_vars
```

Alternatively, we can look at the docstrings of auto-generated properties for each variable, e.g.,

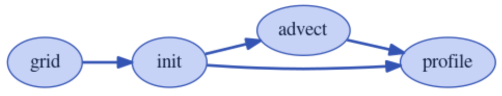
```
In [10]: ProfileU.u?
Type:        property
String form: <property object at 0x7f78083f95e8>
Docstring:
Quantity u

- type : variable
- intent : inout
- dims : (('x',),)
- group : None
- attrs : {'units': 'm'}
```

Like *input_vars* and *input_vars_dict*, Model properties *all_vars* and *all_vars_dict* are available for all model variables, not only inputs.

1.7.3 Visualize models as graphs

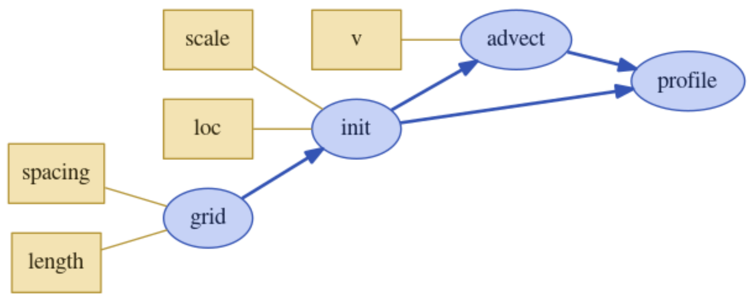
It is possible to visualize a model and its processes as a directed graph (note: this requires installing Graphviz and its Python bindings, which both can be found on conda-forge):

```
In [11]: model2.visualize();
```



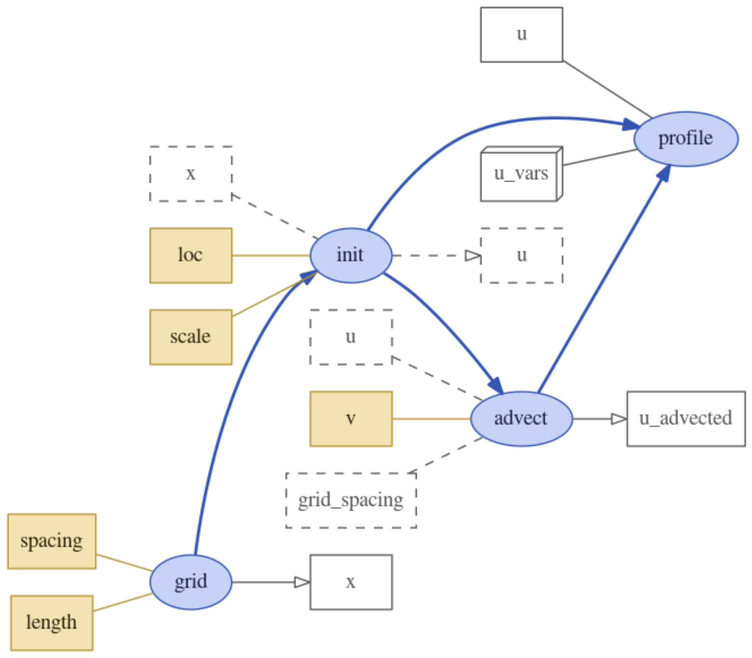
`show_inputs` option allows to show model input variables as yellow square nodes linked to their corresponding processes:

```
In [12]: model2.visualize(show_inputs=True);
```



`show_variables` option allows to show the other variables as white square nodes:

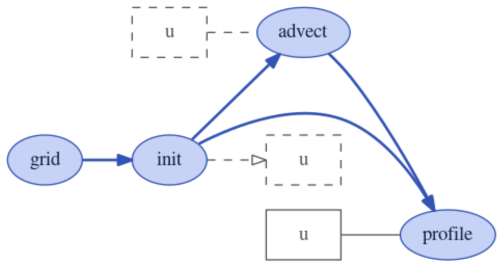
```
In [13]: model2.visualize(show_inputs=True, show_variables=True);
```



Nodes with solid border correspond to regular variables while nodes with dashed border correspond to foreign variables. 3d-box nodes correspond group variables. Variables connected to their process with an arrow have a value computed by the process itself (i.e., `intent='out'`).

A third option `show_only_variable` allows to show only one given variable and all its references in other processes, e.g.,

```
In [14]: model2.visualize(show_only_variable=('profile', 'u'));
```



Note that there is another function `dot_graph` available in module `xsimlab.dot` which produces similar graphs and which has a few more options.

1.8 Setup and Run Models

This section shows how to create new settings (either from scratch or from existing settings) and run simulations with `Model` instances, using the xarray extension provided by xarray-simlab. We'll use here the simple advection models that we have created in section *Create and Modify Models*.

The following imports are necessary for the examples below.

```
In [1]: import numpy as np

In [2]: import xsimlab as xs

In [3]: import matplotlib.pyplot as plt
```

Note: When the `xsimlab` package is imported, it registers a new namespace named `xsimlab` for `xarray.Dataset` objects. As shown below, this namespace is used to access all xarray-simlab methods that can be applied to those objects.

1.8.1 Create a new setup from scratch

In this example we use the `model2` Model instance:

```
In [4]: model2
Out[4]:
<xsimlab.Model (4 processes, 5 inputs)>
grid
    spacing     [in] uniform spacing
    length      [in] total length
init
    scale       [in] scale of initial pulse
    loc         [in] location of initial pulse
advect
    v           [in] () or ('x',) velocity
profile
```

The convenient *`create_setup()`* function can be used to create a new setup in a very declarative way:

```
In [5]: in_ds = xs.create_setup(
   ...:     model=model2,
   ...:     clocks={'time': np.linspace(0., 1., 101),
   ...:             'otime': [0, 0.5, 1]},
   ...:     master_clock='time',
   ...:     input_vars={'grid': {'length': 1.5, 'spacing': 0.01},
   ...:                 'init': {'loc': 0.3, 'scale': 0.1},
   ...:                 'advect': {'v': 1.}},
   ...:     output_vars={None: {'grid': 'x'},
   ...:                  'otime': {'profile': 'u'}}
   ...: )
   ...:
```

A setup consists in:

- one or more time dimensions ("clocks") and their given coordinate values ;
- one of these time dimensions, defined as master clock, which will be used to define the simulation time steps (the other time dimensions usually serve to take snapshots during a simulation on a different but synchronized clock) ;
- values given for input variables ;
- one or more variables for which we want to take snapshots on given clocks (time dimension) or just once at the end of the simulation (`None`).

In the example above, we set `time` as the master clock dimension and `otime` as another dimension for taking snapshots of u along the grid at three given times of the simulation (beginning, middle and end). The time-independent x-coordinate values of the grid will be saved as well.

`create_setup` returns all these settings packaged into a `xarray.Dataset` :

```
In [6]: in_ds
Out[6]:
<xarray.Dataset>
Dimensions:        (otime: 3, time: 101)
Coordinates:
  * time           (time) float64 0.0 0.01 0.02 0.03 0.04 ... 0.97 0.98 0.99 1.0
  * otime          (otime) float64 0.0 0.5 1.0
Data variables:
    grid__length   float64 1.5
    grid__spacing  float64 0.01
    init__loc      float64 0.3
    init__scale    float64 0.1
    advect__v      float64 1.0
Attributes:
    __xsimlab_output_vars__:  grid__x
```

If defined in the model, variable metadata such as description are also added in the dataset as attributes of the corresponding data variables, e.g.,

```
In [7]: in_ds.advect__v
Out[7]:
<xarray.DataArray 'advect__v' ()>
array(1.)
Attributes:
    description:  velocity
```

1.8.2 Run a simulation

A new simulation is run by simply calling the `xsimlab.run()` method from the input dataset created above. It returns a new dataset:

```
In [8]: out_ds = in_ds.xsimlab.run(model=model2)
```

The returned dataset contains all the variables of the input dataset. It also contains simulation outputs as new or updated data variables, e.g., `grid__x` and `profile__u` in this example:

```
In [9]: out_ds
Out[9]:
<xarray.Dataset>
Dimensions:        (otime: 3, time: 101, x: 150)
Coordinates:
  * time           (time) float64 0.0 0.01 0.02 0.03 0.04 ... 0.97 0.98 0.99 1.0
  * otime          (otime) float64 0.0 0.5 1.0
Dimensions without coordinates: x
Data variables:
    grid__length   float64 1.5
    grid__spacing  float64 0.01
    init__loc      float64 0.3
    init__scale    float64 0.1
    advect__v      float64 1.0
```

(continues on next page)

(continued from previous page)

```
    profile__u      (otime, x) float64 0.0001234 0.0002226 ... 0.03916 0.02705
    grid__x         (x) float64 0.0 0.01 0.02 0.03 0.04 ... 1.46 1.47 1.48 1.49
```

1.8.3 Post-processing and plotting

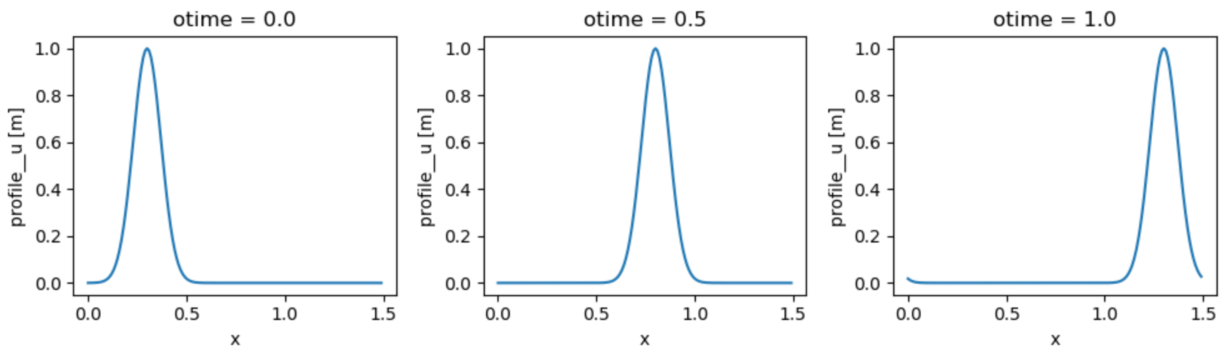
A great advantage of using xarray Datasets is that it is straightforward to include the simulation as part of a processing pipeline, i.e., by chaining `xsimlab.run()` with other methods that can also be applied on Dataset objects.

As an example, instead of a data variable `grid__x` it would be nicer to save the grid x values as a coordinate in the output dataset:

```
In [10]: out_ds = (in_ds.xsimlab.run(model=model2)
   ....:                 .set_index(x='grid__x'))
   ....:

In [11]: out_ds
Out[11]:
<xarray.Dataset>
Dimensions:         (otime: 3, time: 101, x: 150)
Coordinates:
  * time            (time) float64 0.0 0.01 0.02 0.03 0.04 ... 0.97 0.98 0.99 1.0
  * otime           (otime) float64 0.0 0.5 1.0
  * x               (x) float64 0.0 0.01 0.02 0.03 0.04 ... 1.46 1.47 1.48 1.49
Data variables:
    grid__length    float64 1.5
    grid__spacing   float64 0.01
    init__loc       float64 0.3
    init__scale     float64 0.1
    advect__v       float64 1.0
    profile__u      (otime, x) float64 0.0001234 0.0002226 ... 0.03916 0.02705
```

All convenient methods provided by xarray are directly accessible, e.g., for plotting snapshots:

```
In [12]: def plot_u(ds):
   ....:     fig, axes = plt.subplots(ncols=3, figsize=(10, 3))
   ....:     for t, ax in zip(ds.otime, axes):
   ....:         ds.profile__u.sel(otime=t).plot(ax=ax)
   ....:     fig.tight_layout()
   ....:     return fig
   ....:

In [13]: plot_u(out_ds);
```



1.8.4 Reuse existing settings

Update inputs

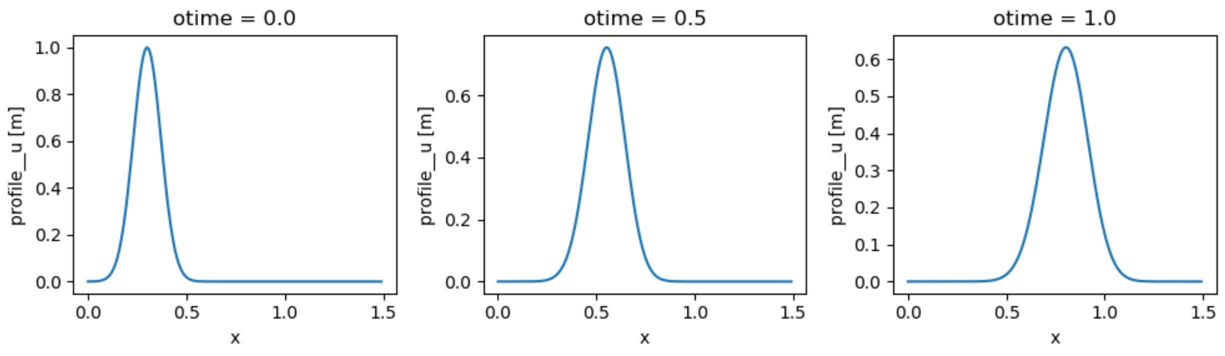
In the following example, we set and run another simulation in which we decrease the advection velocity down to 0.5. Instead of creating a new setup from scratch, we can reuse the one created previously and update only the value of velocity, thanks to `xsimlab.update_vars()`.

```
In [14]: in_vars = {('advect', 'v'): 0.5}

In [15]: with model2:
   ....:     out_ds2 = (in_ds.xsimlab.update_vars(input_vars=in_vars)
   ....:                     .xsimlab.run()
   ....:                     .set_index(x='grid__x'))
   ....:
```

Note: For convenience, a Model instance may be used in a context instead of providing it repeatedly as an argument of xarray-simlab's functions or methods in which it is required.

We plot the results to compare this simulation with the previous one (note the numerical dissipation as a side-effect of the Lax scheme, which is more visible here):

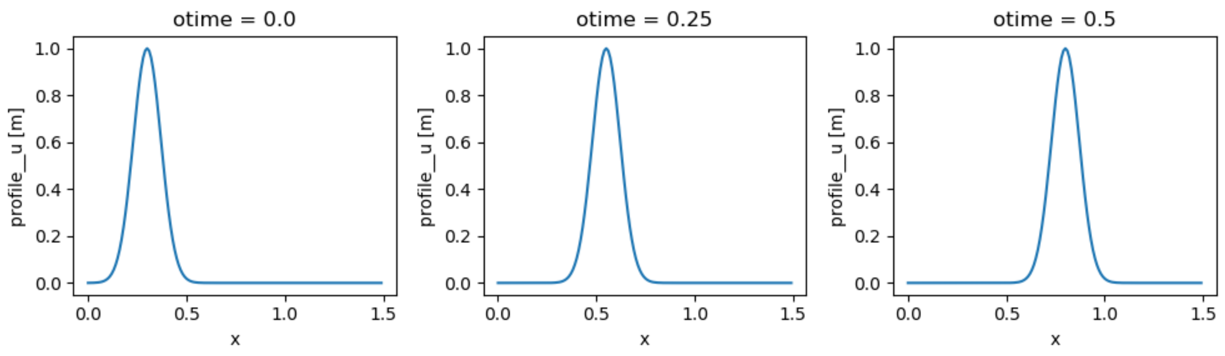
```
In [16]: plot_u(out_ds2);
```



Update time dimensions

`xsimlab.update_clocks()` allows to only update the time dimensions and/or their coordinates. Here below we set other values for the `otime` coordinate (which serves to take snapshots of u):

```
In [17]: clocks = {'otime': [0, 0.25, 0.5]}

In [18]: with model2:
   ....:     out_ds3 = (in_ds.xsimlab.update_clocks(clocks=clocks,
   ....:                                            master_clock='time')
   ....:                     .xsimlab.run()
   ....:                     .set_index(x='grid__x'))
   ....:

In [19]: plot_u(out_ds3);
```

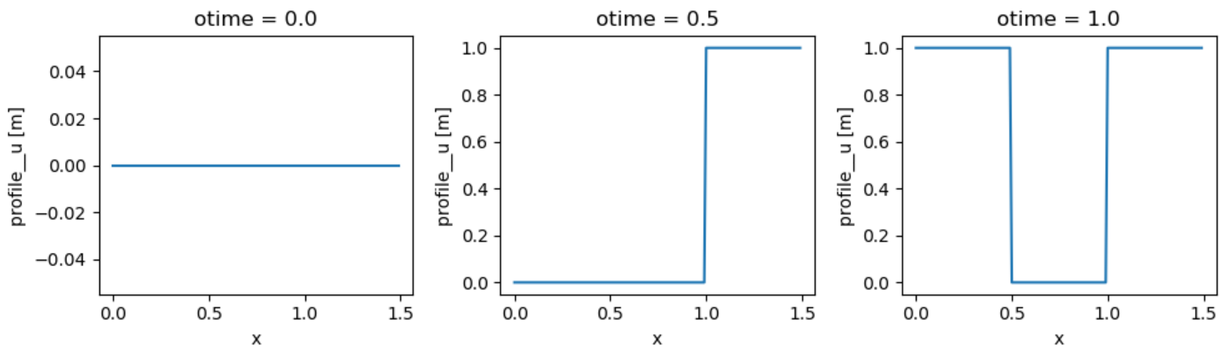


Use an alternative model

A model and its alternative versions often keep inputs in common. It this case too, it would make sense to create an input dataset from an existing dataset, e.g., by dropping data variables that are irrelevant (see *xsimlab.filter_vars()*) and by adding data variables for inputs that are present only in the alternative model.

Here is an example of simulation using `model3` (source point and flat initial profile for u) instead of `model2` :

```
In [20]: in_vars = {'source': {'loc': 1., 'flux': 100.}}

In [21]: with model3:
   ....:     out_ds4 = (in_ds.xsimlab.filter_vars()
   ....:                     .xsimlab.update_vars(input_vars=in_vars)
   ....:                     .xsimlab.run()
   ....:                     .set_index(x='grid__x'))
   ....:

In [22]: plot_u(out_ds4);
```



1.8.5 Time-varying input values

All model inputs accept arrays which have a dimension that corresponds to the master clock.

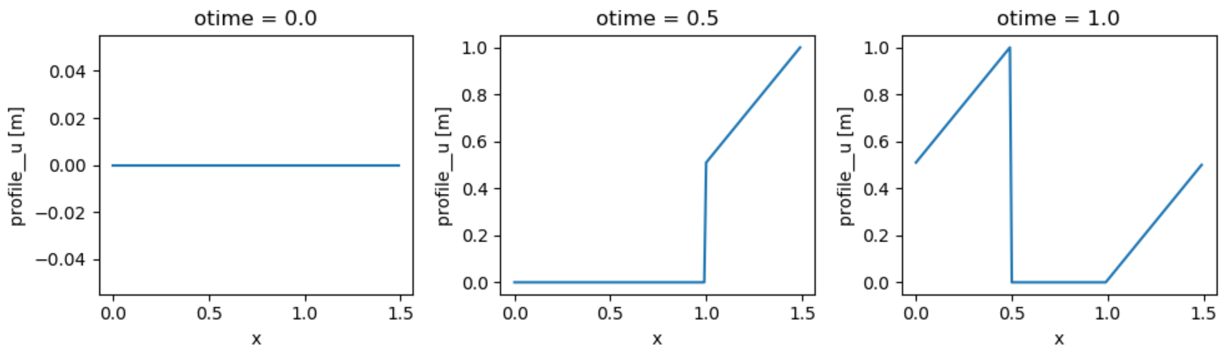
The example below is based on the last example above, but instead of being fixed, the flux of u at the source point decreases over time at a fixed rate:

```
In [23]: flux = 100. - 100. * in_ds.time
```

(continues on next page)

(continued from previous page)

```
In [24]: in_vars = {'source': {'loc': 1., 'flux': flux}}

In [25]: with model3:
   ....:     out_ds5 = (in_ds.xsimlab.filter_vars()
   ....:                      .xsimlab.update_vars(input_vars=in_vars)
   ....:                      .xsimlab.run()
   ....:                      .set_index(x='grid__x'))
   ....:

In [26]: plot_u(out_ds5);
```



Help & Reference

- *API Reference*
- *Release Notes*
- *Citation*

1.9 API Reference

This page provides an auto-generated summary of xarray-simlab's API. For more details and examples, refer to the relevant sections in the main part of the documentation.

1.9.1 Top-level functions

create_setup([model, clocks, master_clock, . . .])	Create a specific setup for model runs.

xsimlab.create_setup

xsimlab.**create_setup**(*model=None*, *clocks=None*, *master_clock=None*, *input_vars=None*, *output_vars=None*)

Create a specific setup for model runs.

This convenient function creates a new `xarray.Dataset` object with everything needed to run a model (i.e., input values, time steps, output variables to save at given times) as data variables, coordinates and attributes.

Parameters

- **model** (*xsimlab.Model* object, optional) – Create a simulation setup for this model. If None, tries to get model from context.

- **clocks** (*dict, optional*) – Used to create one or several clock coordinates. Dictionary values are anything that can be easily converted to `xarray.IndexVariable` objects (e.g., a 1-d `numpy.ndarray` or a `pandas.Index`).
- **master_clock** (*str or dict, optional*) – Name of the clock coordinate (dimension) to use as master clock. If not set, the name is inferred from `clocks` (only if one coordinate is given and if Dataset has no master clock defined yet). A dictionary can also be given with one of several of these keys:
  - `dim` : name of the master clock dimension/coordinate
  - `units` : units of all clock coordinate labels
  - `calendar` : a unique calendar for all (time) clock coordinates
- **input_vars** (*dict, optional*) – Dictionary with values given for model inputs. Entries of the dictionary may look like:
  - `'foo': {'bar': value, ...}` or
  - `('foo', 'bar'): value` or
  - `'foo__bar': value`

  where `foo` is the name of a existing process in the model and `bar` is the name of an (input) variable declared in that process.

  Values are anything that can be easily converted to `xarray.Variable` objects, e.g., single values, array-like, `(dims, data, attrs)` tuples or xarray objects.
- **output_vars** (*dict, optional*) – Dictionary with model variable names to save as simulation output, given per clock coordinate. Entries of the given dictionary may look like:
  - `'dim': {'foo': 'bar'}` or
  - `'dim': {'foo': ('bar', 'baz')}` or
  - `'dim': ('foo', 'bar')` or
  - `'dim': [('foo', 'bar'), ('foo', 'baz')]` or
  - `'dim': 'foo__bar'` or
  - `'dim': ['foo__bar', 'foo__baz']`

  where `foo` is the name of a existing process in the model and `bar`, `baz` are the names of variables declared in that process.

  If `'dim'` corresponds to the dimension of a clock coordinate, new output values will be saved at each time given by the coordinate labels. if None is given instead, only one value will be saved at the end of the simulation.

Returns **dataset** – A new Dataset object with model inputs as data variables or coordinates (depending on their given value) and clock coordinates. The names of the input variables also include the name of their process (i.e., 'foo__bar').

Return type `xarray.Dataset`

Notes

Output variable names are added in Dataset as specific attributes (global and/or clock coordinate attributes).

1.9.2 Dataset.xsimlab (xarray accessor)

This accessor extends `xarray.Dataset` with all the methods and properties listed below. Proper use of this accessor should be like:

```
>>> import xarray as xr          # first import xarray
>>> import xsimlab               # import xsimlab (the 'xsimlab' accessor is␣
↪registered)
>>> ds = xr.Dataset()            # create or load an xarray Dataset
>>> ds.xsimlab.<meth_or_prop>    # access to the methods and properties listed below
```

Properties

`Dataset.xsimlab.clock_coords`	Dictionary of `xarray.DataArray` objects corresponding to clock coordinates.
`Dataset.xsimlab.master_clock_dim`	Dimension used as master clock for model runs.
`Dataset.xsimlab.output_vars`	Returns a dictionary of clock dimension names (or None) as keys and output variable names - i.e.

xarray.Dataset.xsimlab.clock_coords

`Dataset.xsimlab.`**`clock_coords`**

Dictionary of `xarray.DataArray` objects corresponding to clock coordinates.

xarray.Dataset.xsimlab.master_clock_dim

`Dataset.xsimlab.`**`master_clock_dim`**

Dimension used as master clock for model runs. Returns None if no dimension is set as master clock.

See also:

`Dataset.xsimlab.update_clocks()`

xarray.Dataset.xsimlab.output_vars

`Dataset.xsimlab.`**`output_vars`**

Returns a dictionary of clock dimension names (or None) as keys and output variable names - i.e. lists of `('p_name', 'var_name')` tuples - as values.

Methods

`Dataset.xsimlab.update_clocks`([model, . . .])	Set or update clock coordinates.
`Dataset.xsimlab.update_vars`([model, . . .])	Update model input values and/or output variable names.
`Dataset.xsimlab.filter_vars`([model])	Filter Dataset content according to Model.
`Dataset.xsimlab.run`([model, safe_mode])	Run the model.

xarray.Dataset.xsimlab.update_clocks

`Dataset.xsimlab.`**`update_clocks`**(*model=None*, *clocks=None*, *master_clock=None*)

Set or update clock coordinates.

Also copy from the replaced coordinates any attribute that is specific to model output variables.

Parameters

- **model** (*xsimlab.Model* object, optional) – Reference model. If None, tries to get model from context.
- **clocks** (*dict, optional*) – Used to create one or several clock coordinates. Dictionary values are anything that can be easily converted to `xarray.IndexVariable` objects (e.g., a 1-d `numpy.ndarray` or a `pandas.Index`).
- **master_clock** (*str or dict, optional*) – Name of the clock coordinate (dimension) to use as master clock. If not set, the name is inferred from `clocks` (only if one coordinate is given and if Dataset has no master clock defined yet). A dictionary can also be given with one of several of these keys:
  - `dim` : name of the master clock dimension/coordinate
  - `units` : units of all clock coordinate labels
  - `calendar` : a unique calendar for all (time) clock coordinates

Returns **updated** – Another Dataset with new or replaced coordinates.

Return type Dataset

See also:

xsimlab.create_setup()

xarray.Dataset.xsimlab.update_vars

`Dataset.xsimlab.`**`update_vars`**`(`*model=None*, *input_vars=None*, *output_vars=None*`)`

Update model input values and/or output variable names.

More details about the values allowed for the parameters below can be found in the doc of *xsimlab.create_setup()*.

Parameters

- **model** (*xsimlab.Model* object, optional) – Reference model. If None, tries to get model from context.
- **input_vars** (*dict, optional*) – Model input values (may be grouped per process name, as dict of dicts).
- **output_vars** (*dict, optional*) – Model variables to save as simulation output, given per clock coordinate.

Returns **updated** – Another Dataset with new or replaced variables (inputs) and/or attributes (snaphots).

Return type Dataset

See also:

xsimlab.create_setup()

xarray.Dataset.xsimlab.filter_vars

`Dataset.xsimlab.`**`filter_vars`**`(`*model=None*`)`

Filter Dataset content according to Model.

Keep only data variables and coordinates that correspond to inputs of the model (keep clock coordinates too).

Also update xsimlab-specific attributes so that output variables given per clock only refer to processes and variables defined in the model.

Parameters **`model`** (*`xsimlab.Model`* object, optional) – Reference model. If None, tries to get model from context.

Returns **filtered** – Another Dataset with (maybe) dropped variables and updated attributes.

Return type Dataset

See also:

`Dataset.xsimlab.update_vars()`

xarray.Dataset.xsimlab.run

`Dataset.xsimlab.`**`run`**(*model=None*, *safe_mode=True*)

Run the model.

Parameters

- **`model`** (*`xsimlab.Model`* object, optional) – Reference model. If None, tries to get model from context.
- **`safe_mode`** (*`bool`*, *`optional`*) – If True (default), it is safe to run multiple simulations simultaneously. Generally safe mode shouldn't be disabled, except in a few cases (e.g., debugging).

Returns **output** – Another Dataset with both model inputs and outputs.

Return type Dataset

1.9.3 Model

Creating a model

`Model`(processes)	An immutable collection of process units that together form a computational model.

xsimlab.Model

`class` `xsimlab.`**`Model`**(*processes*)

An immutable collection of process units that together form a computational model.

This collection is ordered such that the computational flow is consistent with process inter-dependencies.

Ordering doesn't need to be explicitly provided ; it is dynamically computed using the processes interfaces.

Processes interfaces are also used for automatically retrieving the model inputs, i.e., all the variables that require setting a value before running the model.

`__init__`(*processes*)

Parameters **`processes`** (*`dict`*) – Dictionnary with process names as keys and classes (decorated with *`process()`*) as values.

Raises

- `TypeError` – If values in `processes` are not classes.
- `NoteAProcessClassError` – If values in `processes` are not classes decorated with *`process()`*.

Methods

`__init__`(processes)	**param processes** Dictionnary with process names as keys and classes (decorated with
`clone`()	Clone the Model, i.e., create a new Model instance with the same process classes but different instances.
`drop_processes`(keys)	Drop processe(s) from this model.
`finalize`()	Run the 'finalize' stage of a simulation.
`finalize_step`()	Run a single 'finalize_step' stage of a simulation.
`get`(k[,d])	
`get_context`()	Return the deepest context on the stack.
`get_contexts`()	
`initialize`()	Run the 'initialize' stage of a simulation.
`items`()	
`keys`()	
`run_step`(step)	Run a single 'run_step()' stage of a simulation.
`update_processes`(processes)	Add or replace processe(s) in this model.
`values`()	
`visualize`([show_only_variable, show_inputs, …])	Render the model as a graph using dot (require graphviz).

Attributes

`all_vars`	Returns all variables in the model as a list of `(process_name, var_name)` tuples (or an empty list).
`all_vars_dict`	Returns all variables in the model as a dictionary of lists of variable names grouped by process.
`contexts`	
`dependent_processes`	Returns a dictionary where keys are process names and values are lists of the names of dependent processes.
`input_vars`	Returns all variables that require setting a value before running the model.
`input_vars_dict`	Returns all variables that require setting a value before running the model.

Creating a new model from an existing one

`Model.clone`()	Clone the Model, i.e., create a new Model instance with the same process classes but different instances.

Continued on next page

Table 7 – continued from previous page

`Model.update_processes`(processes)	Add or replace processe(s) in this model.
`Model.drop_processes`(keys)	Drop processe(s) from this model.

xsimlab.Model.clone

`Model.clone()`

Clone the Model, i.e., create a new Model instance with the same process classes but different instances.

xsimlab.Model.update_processes

`Model.update_processes(`*processes*`)`

Add or replace processe(s) in this model.

Parameters **processes** (`dict`) – Dictionnary with process names as keys and process classes as values.

Returns **updated** – New Model instance with updated processes.

Return type *Model*

xsimlab.Model.drop_processes

`Model.drop_processes(`*keys*`)`

Drop processe(s) from this model.

Parameters **keys** (`str or list of str`) – Name(s) of the processes to drop.

Returns **dropped** – New Model instance with dropped processes.

Return type *Model*

Model introspection

`Model` implements an immutable mapping interface where keys are process names and values are objects of `Process` subclasses (attribute-style access is also supported).

`Model.all_vars`	Returns all variables in the model as a list of `(process_name, var_name)` tuples (or an empty list).
`Model.all_vars_dict`	Returns all variables in the model as a dictionary of lists of variable names grouped by process.
`Model.input_vars`	Returns all variables that require setting a value before running the model.
`Model.input_vars_dict`	Returns all variables that require setting a value before running the model.
`Model.dependent_processes`	Returns a dictionary where keys are process names and values are lists of the names of dependent processes.
`Model.visualize`([show_only_variable, . . .])	Render the model as a graph using dot (require graphviz).

xsimlab.Model.all_vars

`Model.`**`all_vars`**

Returns all variables in the model as a list of `(process_name, var_name)` tuples (or an empty list).

xsimlab.Model.all_vars_dict

`Model.`**`all_vars_dict`**

Returns all variables in the model as a dictionary of lists of variable names grouped by process.

xsimlab.Model.input_vars

`Model.`**`input_vars`**

Returns all variables that require setting a value before running the model.

A list of `(process_name, var_name)` tuples (or an empty list) is returned.

xsimlab.Model.input_vars_dict

`Model.`**`input_vars_dict`**

Returns all variables that require setting a value before running the model.

Unlike *`Model.input_vars`*, a dictionary of lists of variable names grouped by process is returned.

xsimlab.Model.dependent_processes

`Model.`**`dependent_processes`**

Returns a dictionary where keys are process names and values are lists of the names of dependent processes.

xsimlab.Model.visualize

`Model.`**`visualize`**(*show_only_variable=None*, *show_inputs=False*, *show_variables=False*)

Render the model as a graph using dot (require graphviz).

Parameters

- **`show_only_variable`** (*`tuple, optional`*) – Show only a variable (and all other variables sharing the same value) given as a tuple `(process_name, variable_name)`. Deactivated by default.
- **`show_inputs`** (*`bool, optional`*) – If True, show all input variables in the graph (default: False). Ignored if *show_only_variable* is not None.
- **`show_variables`** (*`bool, optional`*) – If True, show also the other variables (default: False). Ignored if `show_only_variable` is not None.

See also:

`dot.dot_graph()`

Running a model

In most cases, the methods listed below should not be used directly. For running simulations, it is preferable to use the `Dataset.xsimlab` accessor instead. These methods might be useful though, e.g., for debugging or for using `Model` objects with other interfaces.

`Model.initialize`()	Run the 'initialize' stage of a simulation.
`Model.run_step`(step)	Run a single 'run_step()' stage of a simulation.
`Model.finalize_step`()	Run a single 'finalize_step' stage of a simulation.
`Model.finalize`()	Run the 'finalize' stage of a simulation.

xsimlab.Model.initialize

`Model.`**`initialize`**`()`

Run the 'initialize' stage of a simulation.

xsimlab.Model.run_step

`Model.`**`run_step`**`(`*step*`)`

Run a single 'run_step()' stage of a simulation.

xsimlab.Model.finalize_step

`Model.`**`finalize_step`**`()`

Run a single 'finalize_step' stage of a simulation.

xsimlab.Model.finalize

`Model.`**`finalize`**`()`

Run the 'finalize' stage of a simulation.

1.9.4 Process

Creating a process

`process`([maybe_cls, autodoc])	A class decorator that adds everything needed to use the class as a process.

xsimlab.process

`xsimlab.`**`process`**`(`*maybe_cls=None*, *autodoc=False*`)`

A class decorator that adds everything needed to use the class as a process.

A process represents a logical unit in a computational model.

A process class usually implements:

- An interface as a set of variables defined as class attributes (see *`variable()`*, *`on_demand()`*, `foreign()` and `group()`). This decorator automatically adds properties to get/set values for these variables.

- One or more methods among `initialize()`, `run_step()`, `finalize_step()` and `finalize()`, which are called at different stages of a simulation and perform some computation based on the variables defined in the process interface.
- Decorated methods to compute, validate or set a default value for one or more variables.

Parameters

- **maybe_cls** (*class, optional*) – Allows to apply this decorator to a class either as `@process` or `@process(*args)`.
- **autodoc** (*bool, optional*) – If True, render the docstrings template and fill the corresponding sections with variable metadata (default: False).

Process introspection and variables

`process_info`(process[, buf])	Concise summary of process variables and simulation stages implemented.
`variable_info`(process, var_name[, buf])	Get detailed information about a variable.
`filter_variables`(process[, var_type, . . .])	Filter the variables declared in a process.

xsimlab.process_info

`xsimlab.`**`process_info`**(*process*, *buf=None*)

Concise summary of process variables and simulation stages implemented.

Equivalent to __repr__ of a process but accepts either an instance or a class.

Parameters

- **process** (*object or class*) – Process class or object.
- **buf** (*object, optional*) – Writable buffer (default: sys.stdout).

xsimlab.variable_info

`xsimlab.`**`variable_info`**(*process*, *var_name*, *buf=None*)

Get detailed information about a variable.

Parameters

- **process** (*object or class*) – Process class or object.
- **var_name** (*str*) – Variable name.
- **buf** (*object, optional*) – Writable buffer (default: sys.stdout).

xsimlab.filter_variables

`xsimlab.`**`filter_variables`**(*process*, *var_type=None*, *intent=None*, *group=None*, *func=None*)

Filter the variables declared in a process.

Parameters

- **process** (*object or class*) – Process class or object.
- **var_type** (*{'variable', 'on_demand', 'foreign', 'group'}, optional*) – Return only variables of a specified type.
- **intent** (*{'in', 'out', 'inout'}, optional*) – Return only input, output or input/output variables.
- **group** (*str, optional*) – Return only variables that belong to a given group.
- **func** (*callable, optional*) – A callable that takes a variable (i.e., a `attr.Attribute` object) as input and return True or False. Useful for more advanced filtering.

Returns attributes – A dictionary of variable names as keys and `attr.Attribute` objects as values.

Return type dict

1.9.5 Variable

`variable`([dims, intent, group, default, . . .])	Create a variable.
`foreign`(other_process_cls, var_name[, intent])	Create a reference to a variable that is defined in another process class.
`group`(name)	Create a special variable which value returns an iterable of values of variables in a model that all belong to the same group.
`on_demand`([dims, group, description, attrs])	Create a variable that is computed on demand.

xsimlab.variable

xsimlab.**variable**(*dims=()*, *intent='in'*, *group=None*, *default=NOTHING*, *validator=None*, *description=''*, *attrs=None*)

Create a variable.

Variables store useful metadata such as dimension labels, a short description, a default value, validators or custom, user-provided metadata.

Variables are the primitives of the modeling framework, they define the interface of each process in a model.

Variables should be declared exclusively as class attributes in process classes (i.e., classes decorated with `process()`).

Parameters

- **dims** (*str or tuple or list, optional*) – Dimension label(s) of the variable. An empty tuple corresponds to a scalar variable (default), a string or a 1-length tuple corresponds to a 1-d variable and a n-length tuple corresponds to a n-d variable. A list of str or tuple items may also be provided if the variable accepts different numbers of dimensions. This should not include a time dimension, which may always be added.
- **intent** (*{'in', 'out', 'inout'}, optional*) – Defines whether the variable is an input (i.e., the process needs the variable's value for its computation), an output (i.e., the process computes a value for the variable) or both an input/output (i.e., the process may update the value of the variable). (default: input).
- **group** (*str, optional*) – Variable group.
- **default** (*any, optional*) – Single default value for the variable, ignored when

`intent='out'` (default: NOTHING). A default value may also be set using a decorator.

- **`validator`** (*`callable or list of callable, optional`*) – Function that is called at simulation initialization (and possibly at other times too) to check the value given for the variable. The function must accept three arguments:
  - the process instance (access other variables)
  - the variable object (access metadata)
  - a passed value (check input).

  The function is expected to throw an exception in case of invalid value. If a `list` is passed, its items are treated as validators and must all pass. The validator can also be set using decorator notation.
- **`description`** (*`str, optional`*) – Short description of the variable.
- **`attrs`** (*`dict, optional`*) – Dictionnary of additional metadata (e.g., standard_name, units, math_symbol…).

xsimlab.foreign

`xsimlab.`**`foreign`**(*other_process_cls*, *var_name*, *intent='in'*)

Create a reference to a variable that is defined in another process class.

Parameters

- **`other_process_cls`** (*`class`*) – Class in which the variable is defined.
- **`var_name`** (*`str`*) – Name of the corresponding variable declared in *other_process_cls*.
- **`intent`** (*`{'in', 'out'}, optional`*) – Defines whether the foreign variable is an input (i.e., the process needs the variable's value for its computation), an output (i.e., the process computes a value for the variable). (default: input).

See also:

`variable()`

Notes

Unlike for *`variable()`*, `intent='inout'` is not supported here (i.e., the process may not update the value of a foreign variable) as it would result in ambiguous process ordering in a model.

xsimlab.group

`xsimlab.`**`group`**(*name*)

Create a special variable which value returns an iterable of values of variables in a model that all belong to the same group.

Access to the variable values is read-only (i.e., intent='in').

Good examples of using group variables are processes that aggregate (e.g., sum, product, mean) the values of variables that are defined in various other processes in a model.

Parameters **`group`** (*`str`*) – Name of the group.

See also:

`variable()`

xsimlab.on_demand

`xsimlab.on_demand(dims=(), group=None, description='', attrs=None)`

Create a variable that is computed on demand.

Instead of being computed systematically at every step of a simulation or at initialization, its value is only computed (or re-computed) each time when it is needed.

Like other variables, such variable should be declared in a process class. Additionally, it requires its own method to compute its value, which must be defined in the same class and decorated (e.g., using *@myvar.compute* if the name of the variable is *myvar*).

An on-demand variable is always an output variable (i.e., intent='out').

Its computation usually involves other variables, although this is not required.

These variables may be useful, e.g., for model diagnostics.

Parameters

- **dims** (*str or tuple or list, optional*) – Dimension label(s) of the variable. An empty tuple corresponds to a scalar variable (default), a string or a 1-length tuple corresponds to a 1-d variable and a n-length tuple corresponds to a n-d variable. A list of str or tuple items may also be provided if the variable accepts different numbers of dimensions. This should not include a time dimension, which may always be added.
- **group** (*str, optional*) –
- **description** (*str, optional*) – Short description of the variable.
- **attrs** (*dict, optional*) – Dictionnary of additional metadata (e.g., standard_name, units, math_symbol…).

See also:

`variable()`

1.10 Release Notes

1.10.1 v0.3.0 (Unreleased)

1.10.2 v0.2.1 (7 November 2018)

Bug fixes

- Fix an issue after a change in attrs 0.18.2 (GH47).

1.10.3 v0.2.0 (9 May 2018)

Highlights

This release includes a major refactoring of both the internals and the API on how processes and variables are defined and depends on each other in a model. xarray-simlab now uses and extends attrs (GH33).

Also, Python 3.4 support has been dropped. It may still work with that version but it is not actively tested anymore and it is not packaged with conda.

Breaking changes

As xarray-simlab is still at an early development stage and hasn't been adopted "in production" yet (to our knowledge), we haven't gone through any depreciation cycle, which by the way would have been almost impossible for such a major refactoring. The following breaking changes are effective now!

- `Variable`, `ForeignVariable` and `VariableGroup` classes have been replaced by `variable`, `foreign` and `group` factory functions (wrappers around `attr.ib`), respectively.
- `VariableList` has been removed and has not been replaced by anything equivalent.
- `DiagnosticVariable` has been replaced by `on_demand` and the `diagnostic` decorator has been replaced by the variable's `compute` decorator.
- The `provided` (`bool`) argument (variable constructors) has been replaced by `intent` (`{'in', 'out', 'inout'}`).
- The `allowed_dims` argument has been renamed to `dims` and is now optional (a scalar value is expected by default).
- The `validators` argument has been renamed to `validator` to be consistent with `attr.ib`.
- The `optional` argument has been removed. Variables that don't require an input value may be defined using a special validator function (see `attrs` documentation).
- Variable values are not anymore accessed using three different properties `state`, `rate` and `change` (e.g., `self.foo.state`). Instead, all variables accept a unique value, which one can get/set by simply using the variable name (e.g., `self.foo`). Now multiple variables have to be declared for holding different values.
- Process classes are now defined using the `process` decorator instead of inheriting from a `Process` base class.
- It is not needed anymore to explicitly define whether or not a process is time dependent (it is now deducted from the methods implemented in the process class).
- Using `class Meta` inside a process class to define some metadata is not used anymore.
- `Model.input_vars` now returns a list of `(process_name, variable_name)` tuples instead of a dict of dicts. `Model.input_vars_dict` has been added for convenience (i.e., to get input variables grouped by process as a dictionary).
- `Model.is_input` has been removed. Use `Model.input_vars` instead to check if a variable is a model input.
- `__repr__` has slightly changed for variables, processes and models. Process classes don't have an `.info()` method anymore, which has been replaced by the `process_info()` top-level function. Another helper function `variable_info()` has been added.
- In `Model.visualize()` and `xsimlab.dot.dot_graph()`, `show_variables=True` now shows all model variables including inputs. Items of group variables are not shown anymore as nodes.
- `Model.visualize()` and `xsimlab.dot.dot_graph()` now only accept tuples for `show_only_variable`.

- For simplicity, `Dataset.xsimlab.snapshot_vars` has been renamed to `output_vars`. The corresponding arguments in `create_setup` and `Dataset.xsimlab.update_vars` have been renamed accordingly.
- Values for all model inputs must be provided when creating or updating a setup using `create_setup` or `Dataset.xsimlab.update_vars`. this is a regression that will be fixed in the next releases.
- Argument values for generating clock data in `create_setup` and `Dataset.xsimlab.update_clocks` have changed and are now more consistent with how coordinates are set in xarray. Additionally, `auto_adjust` has been removed (an error is raised instead when clock coordinate labels are not synchronized).
- Scalar values from a input `xarray.Dataset` are now converted into scalars (instead of a 0-d numpy array) when setting input model variables during a simulation.

Enhancements

- The major refactoring in this release should reduce the overhead caused by the indirect access to variable values in process objects.
- Another benefit of the refactoring is that a process-decorated class may now inherit from other classes (possibly also process-decorated), which allows more flexibility in model customization.
- By creating read-only properties in specific cases (i.e., when `intent='in'`), the `process` decorator applied on a class adds some safeguards to prevent setting variable values where it is not intended.
- Some more sanity checks have been added when creating process classes.
- Simulation active and output data r/w access has been refactored internally so that it should be easy to later support alternative data storage backends (e.g., on-disk, distributed).
- Added `Model.dependent_processes` property (so far this was not in public API).
- Added `Model.all_vars` and `Model.all_vars_dict` properties that are similar to `Model.input_vars` and `Model.input_vars_dict` but return all variable names in the model.
- `input_vars` and `output_vars` arguments of `create_setup` and `Dataset.xsimlab.update_vars` now accepts different formats.
- It is now possible to update only some clocks with `Dataset.xsimlab.update_clocks` (previously all existing clock coordinates were dropped first).

Regressions (will be fixed in future releases)

- Although it is possible to set validators, converters and/or default values for variables (this is directly supported by `attrs`), these are not handled by xarray-simlab yet.
- Variables don't accept anymore a dimension that corresponds to their own name. This may be useful, e.g., for sensitivity analysis, but as the latter is not implemented yet this feature has been removed and will be added back in a next release.
- High-level API for generating clock coordinate data (i.e., `start`, `end`, `step` and `auto_adjust` arguments) is not supported anymore. This could be added back in a future release in a cleaner form.

1.10.4 v0.1.1 (20 November 2017)

Bug fixes

- Fix misinterpreted tuples passed as `allowed_dims` argument of `Variable` init (GH17).

- Better error message when a Model instance is expected but no object is found or a different object is provided (GH13).

1.10.5 v0.1.0 (8 October 2017)

Initial release.

1.11 Citation

If you want to use and cite xarray-simlab in a scientific publication, we provide citations and DOIs for specific versions via Zenodo. Click on the badge below to get citation information for the latest version of xarray-simlab. **For Contributors**

- *Contributor Guide*
- *Release Procedure*

1.12 Contributor Guide

xarray-simlab is an open-source project. Contributions are welcome, and they are greatly appreciated!

You can contribute in many ways, e.g., by reporting bugs, submitting feedbacks, contributing to the development of the code and/or the documentation, etc.

This page provides resources on how best to contribute.

1.12.1 Issues

The Github Issue Tracker is the right place for reporting bugs and for discussing about development ideas. Feel free to open a new issue if you have found a bug or if you have suggestions about new features or changes.

For now, as the project is still very young, it is also a good place for asking usage questions.

1.12.2 Development environment

If you wish to contribute to the development of the code and/or the documentation, here are a few steps for setting a development environment.

Fork the repository and download the code

To further be able to submit modifications, it is preferable to start by forking the xarray-simlab repository on GitHub (you need to have an account).

Then clone your fork locally:

```
$ git clone git@github.com:your_name_here/xarray-simlab.git
```

Alternatively, if you don't plan to submit any modification, you can clone the original xarray-simlab git repository:

```
$ git clone git@github.com:benbovy/xarray-simlab.git
```

Install

To install the dependencies, we recommend using the conda package manager with the conda-forge channel. For development purpose, you might consider installing the packages in a new conda environment:

```
$ conda create -n xarray-simlab_dev python=3.6 attrs numpy xarray -c conda-forge
$ source activate xarray-simlab_dev
```

Then install xarray-simlab locally using `pip`:

```
$ cd xarray-simlab
$ pip install -e .
```

Run tests

To make sure everything behaves as expected, you may want to run xarray-simlab's unit tests locally using the pytest package. You can first install it with conda:

```
$ conda install pytest -c conda-forge
```

Then you can run tests from the main xarray-simlab directory:

```
$ pytest xsimlab --verbose
```

1.12.3 Contributing to code

Below are some useful pieces of information in case you want to contribute to the code.

Local development

Once you have setup the development environment, the next step is to create a new git branch for local development:

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

Now you can make your changes locally.

Submit changes

Once you are done with the changes, you can commit your changes to git and push your branch to your xarray-simlab fork on GitHub:

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

(note: this operation may be repeated several times).

We you are ready, you can create a new pull request through the GitHub website (note that it is still possible to submit changes after your created a pull request).

Python versions

xarray-simlab supports Python versions 3.4 and higher. It is not compatible with Python versions 2.x. We don't plan to make it compatible with Python 2.7.x unless there are very good reasons to do so.

Test

xarray-simlab's uses unit tests extensively to make sure that every part of the code behaves as we expect. Test coverage is required for all code contributions.

Unit test are written using pytest style (i.e., mostly using the `assert` statement directly) in various files located in the `xsimlab/tests` folder. The file `conftest.py` defines some `Process` subclasses, `Model` objects and `xarray.Dataset` objects that can be used as fixtures for testing.

You can run tests locally from the main xarray-simlab directory:

```
$ pytest xsimlab --verbose
```

All tests are also executed automatically on the Travis.ci continuous integration platform on every push to every pull request on GitHub.

Docstrings

Everything (i.e., classes, methods, functions. . .) that is part of the public API should follow the numpydoc standard when possible.

Coding style

The xarray-simlab code mostly follows the style conventions defined in PEP8.

Source code checker

To check about any potential error or bad style in your code, you might want using a source code checker like flake8. You can install it in your development environment:

```
$ conda install flake8 -c conda-forge
```

What's new entry

Every significative code contribution should be listed in the *Release Notes* section of this documentation under the corresponding version.

1.12.4 Contributing to documentation

xarray-simlab uses Sphinx for documentation, hosted on http://readthedocs.org . Documentation is maintained in the RestructuredText markup language (`.rst` files) in `xarray-simlab/doc`.

To build the documentation locally, first install requirements (for example here in a separate conda environment):

```
$ conda env create -n xarray-simlab_doc -f doc/environment.yml
$ source activate xarray-simlab_doc
```

Then build documentation with `make`:

```
$ cd doc
$ make html
```

The resulting HTML files end up in the `build/html` directory.

You can now make edits to rst files and run `make html` again to update the affected pages.

1.13 Release Procedure

How to issue a xarray-simlab release in a few steps:

1. Ensure local master branch is synced to upstream:

   ```
   $ git pull upstream master
   ```

2. Check `whats_new.rst` and the docs. Make sure "Release Notes" is complete (check the date!) and if needed add a brief summary note describing the release at the top.

3. If you have any doubts, run the full test suite one final time!:

   ```
   $ pytest xsimlab -vv
   ```

5. On the master branch, commit the release in git:

   ```
   $ git commit -a -m 'release v0.X.Y'
   ```

6. Tag the release:

   ```
   $ git tag -a v0.X.Y -m '0.X.Y'
   ```

7. Push to GitHub:

   ```
   $ git push upstream master --tags
   ```

8. Publish the release on GitHub: go to the repository's URL, follow the `releases` link, click on the `Draft a new release` button, select the tag of this release, add a title (e.g., the tag name) and a description (e.g., the summary added in `whats_new.rst`).

8. Before build the package and upload to PyPI, make sure that you didn't make a local install using pip (maybe due to .egg-info conflict, this make cause issue with the packaged version on PyPI, which may be unusable and this is irreversible!). For steps 8 and 9 below, either you can switch to another local clone of the repository (clean and up-to-date!! repeat step 1 if needed), or first clean the repository from build/dist files and/or all git untracked and ignored files (if you don't mind losing them):

   ```
   $ rm -rf dist build */*.egg-info *.egg-info
   $ git clean -xfd
   ```

9. Build source and binary wheels for PyPI:

   ```
   $ python setup.py bdist_wheel sdist
   ```

10. Use twine to register and upload the release on pypi. You will need to be listed as a package owner at https://pypi.python.org/pypi/xarray-simlab for this to work. Be careful, this is irreversible!!:

```
$ twine upload dist/xarray-simlab-0.X.Y*
```

11. Update conda-forge. Clone https://github.com/conda-forge/xarray-simlab-feedstock and update the version number and sha256 in `recipe/meta.yaml` (check also dependencies). Submit a pull request (and merge it, once CI passes). Note: on macOS, you can calculate sha256 with:

```
$ shasum -a 256 xarray-simlab-0.X.Y.tar.gz
```

12. Add a section for the next release (v.X.(Y+1)) to `doc/whats-new.rst`.
13. Commit your changes and push to master again:

```
$ git commit -a -m 'Revert to dev version'
$ git push upstream master
```

CHAPTER 2

Get in touch

- You can report bugs, suggest features or view the source code on GitHub.

CHAPTER 3

License

3-clause (“Modified” or “New”) BSD license.

CHAPTER 4

Acknowledgment

This project is supported by the Earth Surface Process Modelling group of the GFZ Helmholtz Centre Potsdam.

CHAPTER 5

Citation

If you use xarray-simlab and would like to cite it in a scientific publication, we would certainly appreciate it (see *Citation* section).

Index