hydrobricks

Release 0.4.11

Pascal Horton

Jul 06, 2023

CONTENTS:

Hydrobricks is a flexible hydrological modelling framework. Its core is written in C++ and it has a Python interface. More specifically, the processes, fluxes and solver are coded in the C++ core, while data preparation is done with Python. The objective is to use Python wherever possible, and in particular for data processing, and C++ where necessary, for performance reasons.

Hydrobricks comes with pre-build model structures, but it aims at allowing structure definition from the user through the Python API. The existing model structures can be found under the *models page*. The main components of the model are described under the *basics page*.

CHAPTER

ONE

GETTING STARTED

Hydrobricks is distributed through PyPi and can be installed using pip:

```
pip install hydrobricks
```

Some code examples are provided in the python/examples directory of the repo. The tests can also be a useful resource to understand the behaviour of some functions.

Here is a minimum example:

```
import hydrobricks as hb
import hydrobricks.models as models

# Model structure
socont = models.Socont(soil_storage_nb=2, surface_runoff="linear_storage",
                       record_all=False)

# Parameters
parameters = socont.generate_parameters()
parameters.set_values({'A': 458, 'a_snow': 1.8, 'k_slow_1': 0.9, 'k_slow_2': 0.8,
                       'k_quick': 1, 'percol': 9.8})

# Hydro units
hydro_units = hb.HydroUnits()
hydro_units.load_from_csv(
    'path/to/elevation_bands.csv', area_unit='m2', column_elevation='elevation',
    column_area='area')

# Meteo data
forcing = hb.Forcing(hydro_units)
forcing.load_from_csv(
    'path/to/meteo.csv', column_time='Date', time_format='%d/%m/%Y',
    content={'precipitation': 'precip(mm/day)', 'temperature': 'temp(C)',
             'pet': 'pet_sim(mm/day)'})
ref_elevation = 1250  # Reference altitude for the meteo data
forcing.spatialize_temperature(ref_elevation, -0.6)
forcing.spatialize_pet()
forcing.spatialize_precipitation(ref_elevation=ref_elevation, gradient=0.05,
                                 correction_factor=0.75)

# Obs data
obs = hb.Observations()
```

(continues on next page)

(continued from previous page)

```
obs.load_from_csv('path/to/discharge.csv', column_time='Date', time_format='%d/%m/%Y',
                  content={'discharge': 'Discharge (mm/d)'})

# Model setup
socont.setup(spatial_structure=hydro_units, output_path=str('path/to/outputs'),
             start_date='1981-01-01', end_date='2020-12-31')

# Initialize and run the model
socont.initialize_state_variables(parameters=parameters, forcing=forcing)
socont.run(parameters=parameters, forcing=forcing)

# Get outlet discharge time series
sim_ts = socont.get_outlet_discharge()

# Evaluate
obs_ts = obs.data_raw[0]
nse = socont.eval('nse', obs_ts)
kge_2012 = socont.eval('kge_2012', obs_ts)

print(f"nse = {nse:.3f}, kge_2012 = {kge_2012:.3f}")
```

CHAPTER

TWO

THE BASICS

2.1 Model structure

A model is composed of three main elements: bricks, processes, and fluxes. The bricks are any component that can contain water, such as a snowpack, a glacier, or a ground reservoir. They can contain one or more water containers. For example, the snowpack has a snow and a liquid water container. These bricks are assigned with processes that can extract water. Processes are for example snowmelt, ET, or outflow according some behaviour. The water extracted from the bricks by the processes are then transferred to fluxes, which deliver it to other bricks, the atmosphere, or the outlet.

For now, only pre-built structures are available. One can create a pre-built instance of a model by using the provided class (to be considered as the blueprint) with some options. The options and the existing models are detailed in the *models page*.

```
socont = models.Socont(soil_storage_nb=2)
```

2.2 Spatial structure

The catchment is discretized into sub units named hydro units. These hydro units can represent HRUs (hydrological response units), pixels, elevation bands, etc. Their properties are loaded from csv files containing at minimum data on each unit area and elevation (mean elevation of each hydro unit). Loading such a file can be done as follows:

```
hydro_units = hb.HydroUnits()
hydro_units.load_from_csv(
   'path/to/file.csv', area_unit='m2', column_elevation='elevation',
   column_area='area')
```

The default land cover is named `ground` and it has no specific behaviour. When there is more than one land cover, these can be specified. Each hydro unit is then assigned a fraction of the provided land covers For example, for a catchment with a pure ice glacier and a debris-covered glacier, one then needs to provide the area for each land cover type and for each hydro unit (more information in *the Python API*):

```
land_cover_names = ['ground', 'glacier_ice', 'glacier_debris']
land_cover_types = ['ground', 'glacier', 'glacier']

hydro_units = hb.HydroUnits(land_cover_types, land_cover_names)
hydro_units.load_from_csv(
   'path/to/file.csv', area_unit='km', column_elevation='Elevation',
   columns_areas={'ground': 'Area Non Glacier',
```

(continues on next page)

(continued from previous page)

```
                  'glacier_ice': 'Area Ice',
                  'glacier_debris': 'Area Debris'})
```

The csv file containing elevation bands data can look like the following example.

Listing 1: Example of a csv file containing elevation bands data.

```
Elevation, Area Non Glacier, Area Ice, Area Debris
3986, 2.408, 0, 0
4022, 2.516, 0, 0
4058, 2.341, 0, 0.003
4094, 2.351, 0, 0.006
4130, 2.597, 0, 0.01
4166, 2.726, 0, 0.006
4202, 2.687, 0, 0.061
4238, 2.947, 0, 0.065
4274, 2.924, 0.013, 0.06
4310, 2.785, 0.019, 0.058
4346, 2.578, 0.052, 0.176
4382, 2.598, 0.072, 0.369
4418, 2.427, 0.129, 0.384
4454, 2.433, 0.252, 0.333
4490, 2.210, 0.288, 0.266
4526, 2.136, 0.341, 0.363
4562, 1.654, 0.613, 0.275
```

2.3 Parameters

The parameters are managed as parameter sets in an object that is an instance of the `ParameterSet` class. It means that there is a single variable containing all the parameters for a model. Within it, different properties are defined for each parameter (more information in *the Python API*):

- **component**: the component to which it refers to (e.g., glacier, slow_reservoir)
- **name**: the detailed name of the parameter (e.g., degree_day_factor)
- **unit**: the parameter unit (e.g., mm/d/°C)
- **aliases**: aliases for the parameter name; this is the short version of the parameter name (e.g., a_snow)
- **value**: the value assigned to the parameter
- **min**: the minimum value the parameter can accept
- **max**: the maximum value the parameter can accept
- **default_value**: the parameter default value; only few parameters have default values, such as the melting temperature, and these are usually not necessary to calibrate
- **mandatory**: defines if the parameter value needs to be provided by the user (i.e. it has no default value)
- **prior**: prior distribution to use for the calibration. See *the calibration page*

2.3.1 Creating a parameter set

When using a pre-build model structure, the parameters for this structure can be generated using the `model.generate_parameters()` function. For example, the following code creates a definition of the Socont model structure and generates the parameter set for the given structure, accounting for the options, such as the number of soil storages. Within this parameter set, the basic attributes are defined, such as the name, aliases, units, min/max values, etc.

```
socont = models.Socont(soil_storage_nb=2)
parameters = socont.generate_parameters()
```

2.3.2 Assigning the parameter values

To set parameter values, the `set_values()` function of the parameter set can be used with a dictionary as argument. The dictionary can use the full parameter names (e.g. `snowpack:degree_day_factor` with no space), or one of the aliases (e.g., `a_snow`):

```
parameters.set_values({'A': 100, 'k_slow': 0.01, 'a_snow': 5})
```

2.3.3 Parameter constraints

Some constraints can be added between parameters. Some of these are built-in when the parameter set is generated and are described in the respective model description. For example, in GSM-Socont, the degree day for the snow must be inferior to the one for the ice (`a_snow < a_ice`).

Constraints between parameters can be added by the user as follows:

```
parameters.define_constraint('k_slow_2', '<', 'k_slow_1')
```

The supported operators are: > (or `gt`), >= (or `ge`), < (or `lt`), <= (or `le`).

On the contrary, pre-defined constraints can be removed:

```
parameters.remove_constraint('a_snow', '<', 'a_ice')
```

2.3.4 Parameter ranges

The parameters are usually generated with a pre-defined range. This range is used to ensure that a provided value falls within the authorized range and to define the boundaries for the calibration algorithm. The pre-defined ranges can be changed as follows:

```
parameters.change_range('a_snow', 2, 5)
```

2.3.5 Adding data-related parameters

Data-related parameters target for example the spatialisation of the forcing data. As these are not model-dependent, but data-dependent, they are not pre-defined by the model and need to be added ba the user:

```
parameters.add_data_parameter('precip_corr_factor', 1, min_value=0.7, max_value=1.3)
parameters.add_data_parameter('precip_gradient', 0.05, min_value=0, max_value=0.2)
parameters.add_data_parameter('temp_gradients', -0.6, min_value=-1, max_value=0)
```

For the meaning of these parameters and the spatialisation procedures implemented in hydrobricks, refer to the section on *forcing data*.

It is also possible, for certain parameters, to define monthly values and ranges:

```
parameters.add_data_parameter(
    'temp_gradients',
    [-0.6, -0.6, -0.6, -0.6, -0.7, -0.7, -0.8, -0.8, -0.8, -0.7, -0.7, -0.6],
    min_value=[-0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8],
    max_value=[-0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3, -0.3])
```

2.4 Forcing data

The meteorological data is handled by the `Forcing` class. It handles the spatialization of the weather data to create per-unit time series. Therefore, when creating an instance of this class, the hydro units must be provided:

```
forcing = hb.Forcing(hydro_units)
```

The data, for example station time series, can the be loaded from csv files. Multiple files can be loaded successively, or a single file can contain different variables (as different columns). One needs to specify which column contains the dates, their format, and which column header represent what kind of variable. For example (more information in *the Python API*):

```
forcing.load_from_csv(
    'path/to/forcing.csv', column_time='Date', time_format='%d/%m/%Y',
    content={'precipitation': 'precip(mm/day)', 'temperature': 'temp(C)',
             'pet': 'pet_sim(mm/day)'})
```

A csv file containing forcing data can look like the following example:

Listing 2: Example of a csv file containing forcing data.

```
Date,precip(mm/day),temp(C),sunshine_dur(h),pet_sim(mm/day)
01/01/1981,8.24,-0.98,0.42,0.58
02/01/1981,4.02,-3.35,0.08,0
03/01/1981,22.27,0.96,0.44,0.95
04/01/1981,28.85,-2.11,0.08,0
05/01/1981,8.89,-5.62,0.07,0.06
06/01/1981,17.49,-4.72,0.09,0
07/01/1981,8.26,-8.58,0.14,0
08/01/1981,0.14,-11.47,81.73,0
09/01/1981,0.91,-7.37,0.1,0.05
10/01/1981,0.54,-3.23,0.09,0
11/01/1981,0.02,-4.57,1.94,0
```

(continues on next page)

(continued from previous page)

```
12/01/1981,2.28,-4.01,69.95,0
13/01/1981,7.03,-6.39,0.04,0
14/01/1981,9.68,-7.54,73.98,0
15/01/1981,16.23,-3.95,0.23,0.01
16/01/1981,2.77,-7.28,0.18,0.19
17/01/1981,6.49,-1.57,1.29,0.19
18/01/1981,5.53,-3.7,0.07,0
...
```

2.4.1 Spatialization

The spatialization operation needs to be specified to generate per-unit timeseries. This definition needs information on the variable, the method to use and its parameters:

```
forcing.define_spatialization(
    variable='temperature', method='additive_elevation_gradient',
    ref_elevation=1250, gradient=-0.6)
```

As we might also want to calibrate the parameters for such operations, these can also be specified as a reference to a parameter instead of a fixed value. In such case, one must add a data parameter as in the following example:

```
forcing.define_spatialization(
    variable='temperature', method='additive_elevation_gradient',
    ref_elevation=1250, gradient='param:temp_gradients')

parameters.add_data_parameter('temp_gradients', -0.6, min_value=-1, max_value=0)
```

The variables supported so far are: `temperature`, `precipitation`, `pet`. The methods and parameters are described in *the Python API*.

2.5 Running the model

Once the *hydro units*, *parameters* and *forcing* defined, the model can be set up and run:

```
socont.setup(spatial_structure=hydro_units, output_path='/path/to/dir',
             start_date='1981-01-01', end_date='2020-12-31')

socont.run(parameters=parameters, forcing=forcing)
```

Then, the outlet discharge (in mm/d) can be retrieved:

```
sim_ts = socont.get_outlet_discharge()
```

More outputs can be extracted and saved to a netCDF file for further analysis:

```
socont.dump_outputs('/output/dir/')
```

The state variables can be initialized using the `initialize_state_variables()` function between the `setup()` and the `run()` functions. The initialization runs the model for the given period and saves the final state variables. These values are then used as initial state variables for the next run:

```
socont.initialize_state_variables(parameters=parameters, forcing=forcing)
socont.run(parameters=parameters, forcing=forcing)
```

When the model is executed multiple times successively, it clears its previous states. When the states initialization provided by `initialize_state_variables()` has been used, the model resets its state variables to these saved values.

2.5.1 Evaluation

Some metrics can be computed by providing the observation time series (in mm/d):

```
# Preparation of the obs data
obs = hb.Observations()
obs.load_from_csv('/path/to/obs.csv', column_time='Date', time_format='%d/%m/%Y',
                  content={'discharge': 'Discharge (mm/d)'})
obs_ts = obs.data_raw[0]

nse = socont.eval('nse', obs_ts)
kge_2012 = socont.eval('kge_2012', obs_ts)
```

The metrics are provided by the HydroErr package . All the metrics listed under their website can be used and are named according to their function names.

2.6 Outputs

The results can be accessed in different ways and with different levels of detail:

1. The *direct outputs* from the model instance.
2. A *dumped netCDF file* containing more details for each hydro unit.
3. Other outputs such as the spatialized forcing or the SPOTPY outputs.

2.6.1 Direct outputs

Some outputs from the model instance are available after a model run as long as the Python session is still alive. The first one is the discharge time series at the outlet, provided by `get_outlet_discharge()`:

```
sim_ts = model.get_outlet_discharge()
```

Some outputs provide integrated values over the simulation period:

- `get_total_outlet_discharge()`: Integrated discharge at the outlet
- `get_total_et()`: Integrated ET
- `get_total_water_storage_changes()`: Changes in all water reservoirs between the beginning of the period and the end.
- `get_total_snow_storage_changes()`: Changes in snow storage between the beginning of the period and the end.

2.6.2 Dumped netCDF file

A detailed netCDF file can be dumped with `model.dump_outputs('some/path')`. The content of the file depends on the option `record_all` provided at model creation. When True, all fluxes and states are recorded, which slows down the model execution.

The file has the following dimensions:

- `time`: The temporal dimension
- `hydro_units`: The hydro units (e.g., elevation bands)
- `aggregated_values`: Elements recorded at the catchment scale (lumped)
- `distributed_values`: Elements recorded at each hydro unit ([semi-]distributed)
- `land_covers`: The different land covers

It contains three important global attributes:

- `labels_aggregated`: The labels of the lumped elements (fluxes and states)
- `labels_distributed`: The labels of the distributed elements (fluxes and states)
- `labels_land_covers`: The labels of the land covers

For example, for the GSM-Socont model with two different glacier types provides the following attributes:

```
labels_aggregated =
   "glacier-area-rain-snowmelt-storage:content",
   "glacier-area-rain-snowmelt-storage:outflow:output",
   "glacier-area-icemelt-storage:content",
   "glacier-area-icemelt-storage:outflow:output",
   "outlet";

labels_distributed =
   "ground:content",
   "ground:infiltration:output",
   "ground:runoff:output",
   "glacier-ice:content",
   "glacier-ice:outflow-rain-snowmelt:output",
   "glacier-ice:melt:output",
   "glacier-debris:content",
   "glacier-debris:outflow-rain-snowmelt:output",
   "glacier-debris:melt:output",
   "ground-snowpack:content",
   "ground-snowpack:snow",
   "ground-snowpack:melt:output",
   "glacier-ice-snowpack:content",
   "glacier-ice-snowpack:snow",
   "glacier-ice-snowpack:melt:output",
   "glacier-debris-snowpack:content",
   "glacier-debris-snowpack:snow",
   "glacier-debris-snowpack:melt:output",
   "slow-reservoir:content",
   "slow-reservoir:et:output",
   "slow-reservoir:outflow:output",
   "slow-reservoir:percolation:output",
   "slow-reservoir:overflow:output",
```

(continues on next page)

(continued from previous page)

```
   "slow-reservoir-2:content",
   "slow-reservoir-2:outflow:output",
   "surface-runoff:content",
   "surface-runoff:outflow:output";

labels_land_covers =
   "ground",
   "glacier-ice",
   "glacier-debris";
```

Then, it provides the following variables:

- `time` (1D): The dates as Modified Julian Dates (days since 1858-11-17 00:00).
- `hydro_units_ids` (1D): The IDs of the hydro units.
- `hydro_units_areas` (1D): The area of the hydro units.
- `sub_basin_values` (2D): The time series of the aggregated elements (c.f. labels_aggregated)
- `hydro_units_values` (2D): the time series of the distributed elements (c.f. labels_distributed). Please not here the differences between:
  - the fluxes (mm), i.e. `output` elements are already weighted by the land cover fraction and the relative hydro unit area. Thus, these elements can be directly summed over all hydro units to obtain the total contribution of a given component (e.g., ice melt), even when the hydro units have different areas.
  - the state variables (mm) such as `content` or `snow` elements represent the water stored in the respective reservoirs. In this case, this value is not weighted and cannot be summed over the catchment, but must be weighted by the land cover fraction and the relative hydro unit area.
- `land_cover_fractions` (2D, optional): the temporal evolution of the land cover fractions.

2.6.3 Others

Some other outputs are available:

- Dumbed forcing: the forcing object can also be saved as a netCDF file using the `forcing.create_file()`. It thus contains the spatialized forcing time series.
- During the calibration procedure, SPOTPY saves all assessments in csv or sql tables.

CHAPTER

THREE

MODELS

The only model structure implemented so far is GSM-Socont.

3.1 Common options

All models have the following options that can be provided at model creation:

- `solver`: choice of the solver to use; the options are: `heun_explicit` (default), `runge_kutta`, and `euler_explicit`.
- `record_all` (default False): when True, the model will record all fluxes and state values for each time step. This slows down the computations and create large output files. Therefore, it should not be enabled during the calibration phase, but only when one needs to analyse the behaviour of the model in details. When False, the model will output the catchment discharge and some selected timeseries.
- `land_cover_types`: a list of the land cover types to use (e.g., `glacier`). See *the section on the spatial structure*.
- `land_cover_names`: a list of the land cover names to use. Each element must match the land cover types explained above. The names are used in the model to distinguish similar land cover types, for example when using a bare-ice glacier and a debris-covered glacier. See *the section on the spatial structure*.

For example:

```
socont = models.Socont(solver="heun_explicit", record_all=False)
```

3.2 GSM-Socont

GSM-Socont is a conceptual glacio-hydrological model described in *Schaefli2005*.

Some basic properties are given in the following table.

Table 1: Properties of the GSM-Socont model

Spatial structure	semi-lumped (elevation bands)
Time step	daily

3.2.1 Specific options

The implemented GSM-Socont version comes with some options:

- `soil_storage_nb`: 1 or 2. This is the number of soil reservoirs to consider (the second one represents the baseflow).
- `surface_runoff`: `socont_runoff` (the original non-linear quick reservoir) or `linear_storage` (a classic linear storage).

3.2.2 Parameters

It has the parameters listed below.

Table 2: Parameters of the GSM-Socont model

Component	Name	Def. value	Unit	Comments
Precipitation (snow/rain transition)	`prec_t_s`	0 [-2, 2]	°C	Temperature below which precipitation is 100% snow. The snow/rain transition is linear between transition_start and transition_end Optional parameter. Full name: snow_rain_transition: transition_start
…	`prec_t_e`	2 [0, 4]	°C	Temperature above which precipitation is 100% liquid. Optional parameter. Full name: snow_rain_transition: transition_end
Snow	`a_snow`	– [1, 12]	mm/	Degree day snow melting factor. a_{snow} in *Schaefli2005* Full name: snowpack: degree_day_factor
…	`melt_t_s`	0 [0, 5]	°C	Temperature above which the snow starts to melt. Optional parameter. Full name: snowpack: melting_temperature
Glacier	`a_ice` (single type), `a_ice_<n` `a_ice_<i`	– [5, 20]	mm/	With <name> being the provided name of the land cover (e.g. glacier_debris) and <i> the number of similar land cover provided. For example: a_ice_glacier_debris or a_ice_1. Degree day ice melting factor. a_{ice} in *Schaefli2005* Full name: <name>: degree_day_factor
…	`melt_t_i`	0 [0, 5]	°C	Temperature above which the ice starts to melt. Optional parameter. Full name: <name>: melting_temperature, with <name> being the provided name of the land cover (e.g. glacier_debris)
Glacier area lumped reservoir	`k_snow`	– [0.0 0.25	1/d	Response factor for the glacier area lumped reservoir receiving rain and snowmelt water. Similar to k_{snow} in *Schaefli2005*, but different units. Full name: glacier_area_rain_snowmelt_storage: response_factor
…	`k_ice`	– [0.0 1]	1/d	Response factor for the glacier area lumped reservoir receiving ice melt water. Similar to k_{ice} in *Schaefli2005*, but different units. Full name: glacier_area_icemelt_storage: response_factor
Quick runoff (non-	`beta`	–	m^(	Parameter to calibrate.

The pre-defined constraints on the parameters are defined below.

3.3 References

CHAPTER

FOUR

CALIBRATION

4.1 Calibration/analysis using SPOTPY

The calibration and sensitivity analyses are performed by the SPOTPY package. The links to SPOTPY are provided by hydrobricks so that it can be used directly.

As we might not want to calibrate all parameters, those that can change have to be specified in the `parameters` instance (see *parameters*):

```
parameters.allow_changing = ['a_snow', 'k_quick', 'A', 'k_slow_1', 'percol',
                             'k_slow_2', 'precip_corr_factor']
```

Then, an instance of the SPOTPY setup can be created by providing the *model instance*, the *parameters*, the *forcing*, the observation time series, a warmup duration (period that will not be used for the evaluation; in days), and the objective function to use:

```
spot_setup = hb.SpotpySetup(socont, parameters, forcing, obs, warmup=365,
                            obj_func='mse')
```

SPOTPY only maximizes the metric value. Thus, when the metric needs to be minimized, we need to invert the objective function:

```
spot_setup = hb.SpotpySetup(socont, parameters, forcing, obs, warmup=365,
                            obj_func='kge_2012', invert_obj_func=True)
```

Once the setup defined, one can use any SPOTPY algorithm. For example, an optimization using the SCE-UA algorithm can be performed:

```
# Select number of maximum repetitions and run spotpy
sampler = spotpy.algorithms.sceua(spot_setup, dbname='socont_SCEUA', dbformat='csv')
max_rep = 10000
sampler.sample(max_rep)
```

Similarly, a Monte-Carlo analysis can be performed:

```
sampler = spotpy.algorithms.mc(spot_setup, dbname='socont_MC', dbformat='csv',
                               save_sim=False)
sampler.sample(10000)
```

Then, the SPOTPY results can be loaded for analysis:

```
# Load the results
results = sampler.getdata()

# Plot parameter interaction
spotpy.analyser.plot_parameterInteraction(results)
plt.tight_layout()
plt.show()

# Plot posterior parameter distribution
posterior = spotpy.analyser.get_posterior(results, percentage=10)
spotpy.analyser.plot_parameterInteraction(posterior)
plt.tight_layout()
plt.show()
```

4.2 Prior distributions

The default prior distribution is a uniform distribution in the range provided by the min/max parameter values. The prior distribution can be changed before the calibration/analysis using the `set_prior()` function on the `parameters` instance:

```
parameters.set_prior('a_snow', spotpy.parameter.Normal(mean=4, stddev=2))
```

Prebuild parameter distribution functions provided by SPOTPY: Uniform, Normal, logNormal, Chisquare, Exponential, Gamma, Wald, Weilbull.

CHAPTER

FIVE

ADVANCED FEATURES

5.1 Land cover evolution

The land cover types in hydrobricks are defined by the user (see the *hydro units section*). Each hydro unit is thus internally defined by a total area and fractional land covers. These land covers can have a dynamic evolution, externally driven. One can provide the model with a timeseries of dates and new land cover areas, such as:

```
changes = behaviours.BehaviourLandCoverChange()
changes.load_from_csv(
    '/path/to/surface_changes_glacier_debris.csv',
    hydro_units, area_unit='km2', match_with='elevation'
)
model.add_behaviour(changes)
```

The definition of a land cover evolution does not replace the original definition of the hydro units, which need to be also provided to the function. The areas provided in the definition of the hydro units are the starting point of the model, and these changes will be enforced in due time. However, if some changes are defined for dates prior to the start of the modelling period, these changes will also be applied. The function `changes.load_from_csv()` can be called multiple times for different files. The corresponding csv file must contain the name of the land cover to change on the first row (for example here `glacier_debris`), the dates of these changes on the second row, and then the change time series. These changes list all hydro units that need to change; those that do not need to change should not be listed in the file. There are two ways to identify the hydro units: by elevation (`match_with='elevation'`) or by ID (`match_with='id'`). In the following example, these changes start with the unit elevation and contain the time series of the area (here in km2) for every date given above.

Listing 1: Example of a csv file containing a land cover evolution.

```
bands,glacier_debris,glacier_debris,glacier_debris,glacier_debris,glacier_debris,glacier_
↪debris,glacier_debris,glacier_debris,glacier_debris,glacier_debris,glacier_debris,
↪glacier_debris,glacier_debris,glacier_debris,glacier_debris,glacier_debris,glacier_
↪debris
,01/08/2020,01/08/2025,01/08/2030,01/08/2035,01/08/2040,01/08/2045,01/08/2050,01/08/2055,
↪01/08/2060,01/08/2065,01/08/2070,01/08/2075,01/08/2080,01/08/2085,01/08/2090,01/08/
↪2095,01/08/2100
4274,0.013,0.003,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4310,0.019,0.009,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4346,0.052,0.042,0.032,0.022,0.012,0.002,0,0,0,0,0,0,0,0,0,0,0,0
4382,0.072,0.062,0.052,0.042,0.032,0.022,0.012,0.002,0,0,0,0,0,0,0,0,0,0
4418,0.129,0.119,0.109,0.099,0.089,0.079,0.069,0.059,0.049,0.039,0.029,0.019,0.009,0,0,0,
↪0
4454,0.252,0.242,0.232,0.222,0.212,0.202,0.192,0.182,0.172,0.162,0.152,0.142,0.132,0.122,
```

(continues on next page)

(continued from previous page)

```
↪0.112,0.102,0.092
4490,0.288,0.278,0.268,0.258,0.248,0.238,0.228,0.218,0.208,0.198,0.188,0.178,0.168,0.158,
↪0.148,0.138,0.128
4526,0.341,0.331,0.321,0.311,0.301,0.291,0.281,0.271,0.261,0.251,0.241,0.231,0.221,0.211,
↪0.201,0.191,0.181
4562,0.613,0.603,0.593,0.583,0.573,0.563,0.553,0.543,0.533,0.523,0.513,0.503,0.493,0.483,
↪0.473,0.463,0.453
4598,0.648,0.638,0.628,0.618,0.608,0.598,0.588,0.578,0.568,0.558,0.548,0.538,0.528,0.518,
↪0.508,0.498,0.488
4634,0.618,0.608,0.598,0.588,0.578,0.568,0.558,0.548,0.538,0.528,0.518,0.508,0.498,0.488,
↪0.478,0.468,0.458
4670,0.478,0.468,0.458,0.448,0.438,0.428,0.418,0.408,0.398,0.388,0.378,0.368,0.358,0.348,
↪0.338,0.328,0.318
4706,0.306,0.296,0.286,0.276,0.266,0.256,0.246,0.236,0.226,0.216,0.206,0.196,0.186,0.176,
↪0.166,0.156,0.146
4742,0.338,0.328,0.318,0.308,0.298,0.288,0.278,0.268,0.258,0.248,0.238,0.228,0.218,0.208,
↪0.198,0.188,0.178
4778,0.199,0.189,0.179,0.169,0.159,0.149,0.139,0.129,0.119,0.109,0.099,0.089,0.079,0.069,
↪0.059,0.049,0.039
4814,0.105,0.095,0.085,0.075,0.065,0.055,0.045,0.035,0.025,0.015,0.005,0,0,0,0,0,0
4850,0.051,0.041,0.031,0.021,0.011,0.001,0,0,0,0,0,0,0,0,0,0,0
4886,0.019,0.009,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4922,0.008,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4958,0.003,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

There is no need to specify the corresponding changes in the generic `ground` land cover as it will be automatically computed to preserve the total hydro unit area.

CHAPTER

SIX

UPGRADE GUIDE

6.1 v0.4 to v0.5

Breaking change:

- Removing hyphens for underscores. Any component (including land cover elements) have to use underscores and not hyphens (e.g., glacier_ice instead of glacier-ice, slow_reservoir instead of slow-reservoir).

CHAPTER

SEVEN

API REFERENCE

7.1 Models

7.1.1 Base model

7.1.2 Socont

class hydrobricks.models.socont.**Socont**(*name='socont'*, ***kwargs*)

Bases: Model

Socont model implementation

generate_parameters()

7.2 Components

7.2.1 HydroUnits

class hydrobricks.**HydroUnits**(*land_cover_types=None*, *land_cover_names=None*)

Bases: object

Class for the hydro units

create_file(*path*)

Create a file containing the hydro unit properties. Such a file can be used in the command-line version of hydrobricks.

Parameters

path (*str*) – Path of the file to create.

get_ids()

Get the hydro unit ids.

load_from_csv(*path*, *area_unit*, *column_elevation=None*, *column_area=None*, *column_fractions=None*, *columns_areas=None*)

Read hydro units properties from csv file.

Parameters

- **path** (*str* | *Path*) – Path to the csv file containing hydro units data.
- **area_unit** (*str*) – Unit for the area values: “m2” or “km2”

- **column_elevation** (*str*) – Column name containing the elevation values in [m] (optional).
- **column_area** (*str*) – Column name containing the area values (optional).
- **column_fractions** (*dict*) – Column name containing the area fraction values for each land cover (optional).
- **columns_areas** (*dict*) – Column name containing the area values for each land cover (optional).

7.2.2 ParameterSet

class hydrobricks.**ParameterSet**

Bases: object

Class for the parameter sets

add_data_parameter(*name*, *value=None*, *min_value=None*, *max_value=None*, *unit=None*)

Add a parameter related to the data.

Parameters

- **name** (*str*) – The name of the parameter.
- **value** (*float/list*) – The parameter value.
- **min_value** (*float/list*) – Minimum value allowed for the parameter.
- **max_value** (*float/list*) – Maximum value allowed for the parameter.
- **unit** (*str*) – The unit of the parameter.

property allow_changing

change_range(*parameter*, *min_value*, *max_value*)

Change the value range of a parameter.

Parameters

- **parameter** (*str*) – Name (or alias) of the parameter
- **min_value** – New minimum value
- **max_value** – New maximum value

constraints_satisfied() → bool

Check if the constraints between parameters are satisfied.

Return type
True is constraints are satisfied, False otherwise.

create_file(*directory*, *name*, *file_type='both'*)

Create a configuration file containing the parameter values.

Such a file can be used when using the command-line version of hydrobricks. It contains the model parameter values.

Parameters

- **directory** (*str*) – The directory to write the file.
- **name** (*str*) – The name of the generated file.

- **file_type** (*file_type*) – The type of file to generate: 'json', 'yaml', or 'both'.

define_constraint(*parameter_1*, *operator*, *parameter_2*)

Defines a constraint between 2 parameters (e.g., paramA > paramB)

Parameters

- **parameter_1** (*str*) – The name of the first parameter.
- **operator** (*str*) – The operator (e.g. '<=').
- **parameter_2** (*str*) – The name of the second parameter.

Examples

parameter_set.define_constraint('paramA', '>=', 'paramB')

define_parameter(*component*, *name*, *unit=None*, *aliases=None*, *min_value=None*, *max_value=None*, *default_value=None*, *mandatory=True*)

Define a parameter by setting its properties.

Parameters

- **component** (*str*) – The component (brick) name to which the parameter refer (e.g., snowpack, glacier, surface-runoff).
- **name** (*str*) – The name of the parameter in the C++ code of hydrobricks (e.g., degree_day_factor, response_factor).
- **unit** (*str*) – The unit of the parameter.
- **aliases** (*list*) – Aliases to the parameter name, such as names used in other implementations (e.g., kgl, an). Aliases must be unique.
- **min_value** (*float/list*) – Minimum value allowed for the parameter.
- **max_value** (*float/list*) – Maximum value allowed for the parameter.
- **default_value** (*float/list*) – The parameter default value.
- **mandatory** (*bool*) – If the parameter needs to be defined or if it can silently use the default value.

get(*name*)

Get the value of a parameter by name.

Parameters
name (*str*) – The name of the parameter.

Return type
The parameter value.

get_for_spotpy()

Get the parameters to assess ready to be used in spotpy.

Return type
A list of the parameters as spotpy objects.

get_model_parameters()

Get the model-only parameters (excluding data-related parameters).

has(*name*)

Check if a parameter exists.

Parameters
name (*str*) – The name of the parameter.

Return type
True if found, False otherwise.

is_for_forcing(*parameter_name*)

Check if the parameter relates to forcing data.

Parameters
parameter_name – The name of the parameter.

Return type
True if relates to forcing data, False otherwise.

list_constraints()

List the constraints currently defined.

needs_random_forcing()

Check if one of the parameters to assess involves the meteorological data.

Return type
True if one of the parameters to assess involves the meteorological data.

range_satisfied() → bool

Check if the parameter value ranges are satisfied.

Return type
True is ranges are satisfied, False otherwise.

remove_constraint(*parameter_1*, *operator*, *parameter_2*)

Removes a constraint between 2 parameters (e.g., paramA > paramB)

Parameters

- **parameter_1** (*str*) – The name of the first parameter.
- **operator** (*str*) – The operator (e.g. '<=').
- **parameter_2** (*str*) – The name of the second parameter.

Examples

parameter_set.remove_constraint('paramA', '>=', 'paramB')

set_prior(*parameter*, *prior*)

Change the value range of a parameter.

Parameters

- **parameter** (*str*) – Name (or alias) of the parameter
- **prior** (*spotpy.parameter*) – The prior distribution (instance of spotpy.parameter)

set_random_values(*parameters*)

Set the provided parameter to random values.

Parameters
parameters (*list*) – The name or alias of the parameters to set to random values. Example: ['kr', 'A']

Return type
A dataframe with the assigned parameter values.

set_values(*values*, *check_range=True*, *allow_adapt=False*)

Set the parameter values.

Parameters

- **values** (*dict*) – The values must be provided as a dictionary with the parameter name with the related component or one of its aliases as the key. Example: {'k': 32, 'A': 300} or {'slow-reservoir:capacity': 300}
- **check_range** (*bool*) – Check that the parameter value falls into the allowed range.
- **allow_adapt** (*bool*) – Allow the parameter values to be adapted to enforce defined constraints (e.g.: min, max).

7.2.3 Forcing

class hydrobricks.**Forcing**(*hydro_units*)

Bases: TimeSeries

Class for forcing data

apply_defined_spatialization(*parameters*, *parameters_to_apply=None*)

Apply the spatialization operations defined by define_spatialization().

Parameters

- **parameters** (ParameterSet) – The parameter object instance.
- **parameters_to_apply** (*list*) – A list of parameters to apply. The spatialization will only be applied for the variables related to parameters in this list. If None, all variables are spatialized.

create_file(*path*, *max_compression=False*)

Read hydro units properties from csv file.

Parameters

- **path** (*str*) – Path of the file to create.
- **max_compression** (*bool*) – Option to allow maximum compression for data in file.

define_spatialization(***kwargs*)

Define the spatialization operations.

Parameters
kwargs – All the parameters needed by the function spatialize() to perform the spatialization for the given forcing variable.

get_total_precipitation()

spatialize(*variable*, *method='constant'*, *ref_elevation=None*, *gradient=0*, *gradient_1=0*, *gradient_2=0*, *elevation_threshold=None*, *correction_factor=None*)

Spatializes the provided variable to all hydro units using the defined method.

Parameters

- **variable** (*str*) – Name of the variable to spatialize.
- **method** (*str*) – Name of the method to use. Can be: * constant: the same value will be used * additive_elevation_gradient: use of an additive elevation gradient that

  is either constant or changes for every month. Parameters: 'ref_elevation', 'gradient'.

  - multiplicative_elevation_gradient: use of a multiplicative elevation gradient that is either constant or changes for every month. Parameters: 'ref_elevation', 'gradient'.
  - multiplicative_elevation_threshold_gradients: same as multiplicative_elevation_gradient, but with an elevation threshold with a gradient below and a gradient above. Parameters: 'ref_elevation', 'gradient', 'gradient_2', 'elevation_threshold'

- **ref_elevation** (*float*) – Reference elevation. For method(s): 'elevation_gradient'
- **gradient** (*float/list*) – Gradient of the variable to apply per 100m (e.g., °C/100m). Can be a unique value or a list providing a value for every month. For method(s): 'elevation_gradient', 'elevation_multi_gradients'
- **gradient_1** (*float/list*) – Same as parameter 'gradient'
- **gradient_2** (*float/list*) – Gradient of the variable to apply per 100m (e.g., °C/100m) for the units above the elevation threshold defined by 'elevation_threshold'. For method(s): 'elevation_multi_gradients'
- **elevation_threshold** (*int/float*) – Threshold elevation to switch from gradient to gradient_2
- **correction_factor** (*float*) – Correction factor to apply to the precipitation data before spatialization

spatialize_pet(*ref_elevation=None*, *gradient=0*)

Spatializes the PET using a gradient that is either constant or changes for every month.

Parameters

- **ref_elevation** (*float*) – Elevation of the reference station.
- **gradient** (*float/list*) – Gradient [mm/100m] to compute the PET for every hydro unit. Can be a unique value or a list providing a value for every month.

spatialize_precipitation(*ref_elevation*, *gradient=None*, *gradient_1=None*, *gradient_2=None*, *elevation_threshold=None*, *correction_factor=None*)

Spatializes the precipitation using a single gradient for the full elevation range or a two-gradients approach with an elevation threshold.

Parameters

- **ref_elevation** (*float*) – Elevation of the reference station.
- **gradient** (*float*) – Precipitation gradient (ratio) per 100 m of altitude.
- **gradient_1** (*float*) – Same as parameter 'gradient'
- **gradient_2** (*float*) – Precipitation gradient (ratio) per 100 m of altitude for the units above the threshold elevation (optional).
- **elevation_threshold** (*float*) – Threshold to switch from gradient 1 to gradient 2 (optional).

- **correction_factor** (*float*) – Correction factor to apply to the precipitation data before spatialization

spatialize_temperature(*ref_elevation*, *lapse*)

Spatializes the temperature using a temperature lapse that is either constant or changes for every month.

Parameters

- **ref_elevation** (*float*) – Elevation of the reference station.
- **lapse** (*float/list*) – Temperature lapse [°C/100m] to compute the temperature for every hydro unit. Can be a unique value or a list providing a value for every month.

7.2.4 Observations

class hydrobricks.**Observations**

Bases: TimeSeries

Class for forcing data

7.3 Submodules

7.3.1 hydrobricks.plotting module

hydrobricks.plotting.**plot_hydro_units_values**(*results*, *index*, *units*, *units_labels*)

hydrobricks.plotting.**plot_precip_per_unit**(*units_precip*, *hydro_units*)

7.3.2 hydrobricks.utils module

class hydrobricks.utils.**Timer**(*text=None*)

Bases: object

Timer to time code execution. Based on: https://pypi.org/project/codetiming/

start()

Start a new timer.

stop(*show_time=True*)

Stop the timer, and report the elapsed time.

hydrobricks.utils.**area_in_m2**(*area*, *unit*)

hydrobricks.utils.**date_as_mjd**(*date*)

hydrobricks.utils.**days_to_hours_mins**(*days*)

Transform a number of days to hours and minutes

hydrobricks.utils.**dump_config_file**(*content*, *directory*, *name*, *file_type='yaml'*)

hydrobricks.utils.**jd_to_date**(*jd*)

Transform julian date numbers to year, month and day (array-based). From https://gist.github.com/jiffyclub/1294443

hydrobricks.utils.**mjd_to_datetime**(*mjd*)

Transform modified julian dates to datetime instances (array-based).

hydrobricks.utils.**validate_kwargs**(*kwargs*, *allowed_kwargs*)

Checks the keyword arguments against a set of allowed keys.

7.4 Preprocessing

7.4.1 Compute elevation bands

class hydrobricks.preprocessing.catchment.**Catchment**(*outline=None*)

Bases: object

Creation of catchment-related data

extract_dem(*dem_path*) → bool

Extract the DEM data for the catchment. Does not handle change in coordinates.

Parameters
dem_path (*str|Path*) – Path of the DEM file.

Return type
True if successful, False otherwise.

get_elevation_bands(*method='isohypse'*, *number=100*, *distance=50*)

Construction of the elevation bands based on the chosen method.

Parameters

- **method** (*str*) – The method to build the elevation bands: 'isohypse' = fixed contour intervals (provide the 'distance' parameter) 'quantiles' = quantiles of the catchment area (same surface; provide the 'number' parameter)
- **number** (*int*) – Number of bands to create when using the 'quantiles' method.
- **distance** (*int*) – Distance (m) between the contour lines when using the 'isohypse' method.

Return type
A dataframe with the elevation bands.

get_mean_elevation()

Get the catchment mean elevation.

Return type
The catchment mean elevation.

7.5 C++ binding

This reference only describes the C++ Python binding. For a full documentation of the C++ code, please refer to the C++ reference.

hydrobricks Python interface

7.5.1 ModelHydro class

class _hydrobricks.**ModelHydro**

Bases: pybind11_object

add_behaviour(*self:* _hydrobricks.ModelHydro, *behaviour: Behaviour*) → bool

Adding a behaviour to the model.

add_time_series(*self:* _hydrobricks.ModelHydro, *time_series:* _hydrobricks.TimeSeries) → bool

Adding a time series to the model.

attach_time_series_to_hydro_units(*self:* _hydrobricks.ModelHydro) → bool

Attach the time series.

clear_time_series(*self:* _hydrobricks.ModelHydro) → None

Clear time series. Use only if the time series were created with ModelHydro::ClearTimeSeries.

create_time_series(*self:* _hydrobricks.ModelHydro, *data_name: str*, *time: numpy.ndarray[numpy.float64[m, 1]]*, *ids: numpy.ndarray[numpy.int32[m, 1]]*, *data: numpy.ndarray[numpy.float64[m, n]]*) → bool

Create a time series and add it to the model.

dump_outputs(*self:* _hydrobricks.ModelHydro, *path: str*) → bool

Dump the model outputs to file.

forcing_loaded(*self:* _hydrobricks.ModelHydro) → bool

Check if the forcing data were loaded.

get_behaviour_items_nb(*self:* _hydrobricks.ModelHydro) → int

Get the number of behaviour items.

get_behaviours_nb(*self:* _hydrobricks.ModelHydro) → int

Get the number of behaviours.

get_outlet_discharge(*self:* _hydrobricks.ModelHydro) → numpy.ndarray[numpy.float64[m, 1]]

Get the outlet discharge.

get_total_et(*self:* _hydrobricks.ModelHydro) → float

Get the total amount of water lost by evapotranspiration.

get_total_outlet_discharge(*self:* _hydrobricks.ModelHydro) → float

Get the outlet discharge total.

get_total_snow_storage_changes(*self:* _hydrobricks.ModelHydro) → float

Get the total change in snow storage.

get_total_water_storage_changes(*self:* _hydrobricks.ModelHydro) → float

Get the total change in water storage.

init_with_basin(*self:* _hydrobricks.ModelHydro, *model_settings:* _hydrobricks.SettingsModel, *basin_settings:* _hydrobricks.SettingsBasin) → bool

Initialize the model and create the sub basin.

is_ok(*self:* _hydrobricks.ModelHydro) → bool

Check if the model is correctly set up.

reset(*self:* _hydrobricks.ModelHydro) → None

Reset the model before another run.

run(*self:* _hydrobricks.ModelHydro) → bool

Run the model.

save_as_initial_state(*self:* _hydrobricks.ModelHydro) → None

Save the model state as initial conditions.

update_parameters(*self:* _hydrobricks.ModelHydro, *model_settings:* _hydrobricks.SettingsModel) → None

Update the parameters with the provided values.

7.5.2 SettingsModel class

class _hydrobricks.**SettingsModel**

Bases: pybind11_object

generate_socont_structure(*self:* _hydrobricks.SettingsModel, *land_cover_types: List[str]*, *land_cover_names: List[str]*, *soil_storage_nb: int = 1*, *surface_runoff: str = 'socont_runoff'*) → bool

Generate the GSM-SOCONT structure.

log_all(*self:* _hydrobricks.SettingsModel, *log_all: bool = True*) → None

Logging all components.

set_parameter(*self:* _hydrobricks.SettingsModel, *component: str*, *name: str*, *value: float*) → bool

Setting one of the model parameter.

set_solver(*self:* _hydrobricks.SettingsModel, *name: str*) → None

Set the solver.

set_timer(*self:* _hydrobricks.SettingsModel, *start_date: str*, *end_date: str*, *time_step: int*, *time_step_unit: str*) → None

Set the modelling time properties.

7.5.3 SettingsBasin class

class _hydrobricks.**SettingsBasin**

Bases: pybind11_object

add_hydro_unit(*self:* _hydrobricks.SettingsBasin, *id: int*, *area: float*, *elevation: float*) → None

Add a hydro unit to the spatial structure.

add_land_cover(*self:* _hydrobricks.SettingsBasin, *name: str*, *type: str*, *fraction: float*) → None

Add a land cover element.

7.5.4 SubBasin class

class _hydrobricks.**SubBasin**

Bases: pybind11_object

init(*self:* _hydrobricks.SubBasin, *spatial_structure:* _hydrobricks.SettingsBasin) → bool

Initialize the basin.

7.5.5 Parameter class

class _hydrobricks.**Parameter**

Bases: pybind11_object

get_name(*self:* _hydrobricks.Parameter) → str

Get the parameter name.

get_value(*self:* _hydrobricks.Parameter) → float

Get the parameter value.

property name

set_name(*self:* _hydrobricks.Parameter, *arg0: str*) → None

Set the parameter name.

set_value(*self:* _hydrobricks.Parameter, *arg0: float*) → None

Set the parameter value.

property value

class _hydrobricks.**ParameterVariableYearly**

Bases: *Parameter*

set_values(*self:* _hydrobricks.ParameterVariableYearly, *year_start: int*, *year_end: int*, *values: List[float]*) → bool

Set the parameter values.

7.5.6 TimeSeries class

class _hydrobricks.**TimeSeries**

Bases: pybind11_object

static create(*data_name: str*, *time: numpy.ndarray[numpy.float64[m, 1]]*, *ids: numpy.ndarray[numpy.int32[m, 1]]*, *data: numpy.ndarray[numpy.float64[m, n]]*) → *_hydrobricks.TimeSeries*

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[Schaefli2005] Schaefli, B., Hingray, B., Niggli, M., & Musy, A. (2005). A conceptual glacio-hydrological model for high mountainous catchments. Hydrology and Earth System Sciences Discussions, 9(1), 95–109. https://doi.org/10.5194/hessd-2-73-2005

PYTHON MODULE INDEX

INDEX

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V