SuperflexPy

Release 1.0.0-rc

Sep 11, 2020

Contents

SuperflexPy

The new language
of hydrological modelling

SuperflexPy is an open-source framework written in Python for constructing flexible, conceptual, distributed hydrological models.

SuperflexPy builds on our 10-years-experience with the development and application of Superflex. The new framework is a completely new implementation of Superflex and expands the possibility offered by the old version, allowing to build completely customized, spatially-distributed hydrological models.

Thanks to its object-oriented architecture, SuperflexPy can be easily extended to satisfy your own needs, creating new elements with a completely customized logic, just in a few lines of pure Python code.

Constructing a semi-distributed conceptual hydrological model will be straightforward with SuperflexPy, with a user experience similar to any other Python framework:

- inputs and outputs are handled directly by the modeler using common Python libraries (e.g. Numpy or Pandas for reading from text files) without the need of customized input files and long pre- and post-processing to adapt the data to the model;
- the elements of the framework are declared and initialized through a Python script, without the need of long and complicated setup text files;
- all the elements of the framework are objects with built-in functionalities for handling parameters and states, routing the fluxes, and solving common structures present in conceptual models (e.g. reservoirs, lag functions, etc.);
- the mathematical model is separated from the numerical model, allowing for testing different numerical methods for solving differential equations;
- the framework can be run at any level of complexity, from a single bucket to an entire river network;
- the framework is easy interface with other Python modules for calibration and uncertainty analysis; we will provide an interface to common frameworks.

CHAPTER 1

Team

SuperflexPy is actively developed at Eawag, by researchers in the Hydrological modelling group, with the support of external people.

The core team consists of:

- Marco Dal Molin (implementation and design)
- Dr. Fabrizio Fenicia (design and supervision)
- Prof. Dmitri Kavetski (design and supervision)

CHAPTER 2

Stay in touch

If you want to get e-mails about future developments of the framework, please subscribe to our mailing list clicking here.

CHAPTER 3

Code and demos

The source code can be accessed at the repository.

A demo, implementing GR4J, is available in a Colab Notebook.

Note: Last update 26/06/2020

3.1 Installation

SuperflexPy has been developed and tested using Python 3 (version 3.7.4). It is not compatible with Python 2.

SuperflexPy is available as Python package at PyPI repository, at the link https://pypi.org/project/superflexpy

The simplest way to install SuperflexPy is using the package installer for Python (pip) running the command

```
pip install superflexpy
```

After the first installation, to upgrade to a new version run the command

```
pip install --upgrade superflexpy
```

3.1.1 Dependencies

SuperflexPy needs the following python packages to run

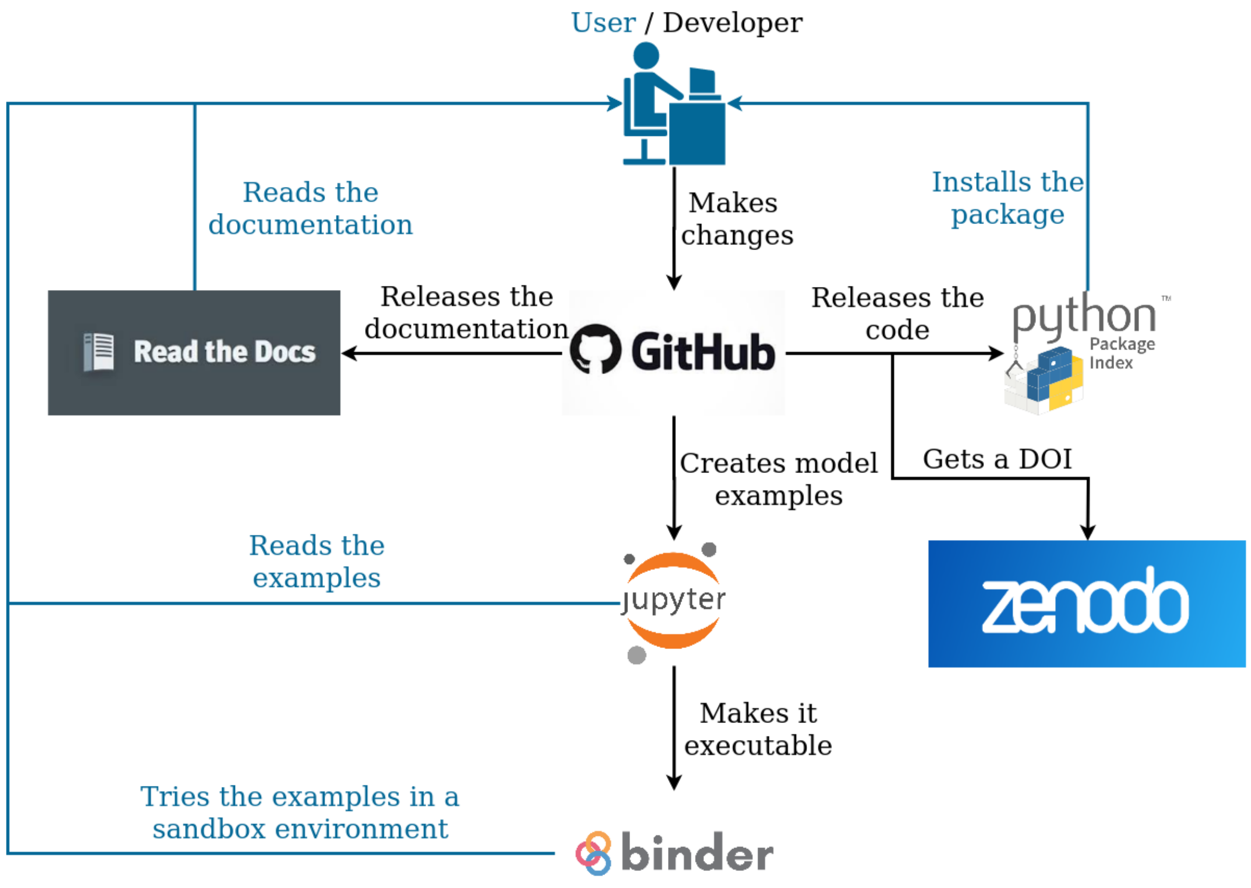
- numpy
- numba

All the packages are available through pip.

The installation of numba is necessary only if the modeler decides to use the numba optimized implementation of the numerical solvers. GPU acceleration (CUDA) is not needed and, therefore, it is not supported.

Note: Last update 07/09/2020

3.2 Software organization and contribution



The superflexPy framework is composed by different pieces that are necessary to fully understand and use the framework:

- **Source code**: it contains all the code necessary to use the framework at its latest (and potentially unstable) version. It should be accessed only by an advanced user who wants to understand the internal functioning, install manually the latest version, or expand the framework.
- **Packaged release**: it allows the user to easily get and install a stable version of the framework.
- **Documentation**: it explains the functioning of the framework in its details.
- **Examples**: they are the "place-to-start" for a new user, providing working models and showcasing potential applications.
- **Scientific publication**: it is a peer-reviewed publication that presents the framework to the public and puts it in prospective with other existing solutions.

Source code, documentation, and examples are part of the official repository of SuperflexPy that is hosted on GitHub. The repository is the only place where code, documentation, and examples should be modified.

New releases of the software are distributed through the official Python Package Index (PyPI) where SuperflexPy has a dedicated page.

Documentation is built automatically from the source folder on GitHub and published online in a dedicated website.

Examples are made available on GitHub as Jupyter notebooks and can be either visualized statically or run in a sandbox environment.

The scientific publication is currently in preparation and it will be linked here once accepted.

3.2.1 Contribution

Contribution to the framework can be made in different ways. Types of contributions include:

- Submit issues on bugs, desired features, etc.
- Solve open issues.
- Extend the documentation with new use cases.
- Extend or modify the framework.
- Use and advertize the framework in your publication.

This page illustrates the typical workflow that should be followed when contributing to a GitHub project. Please, try to follow it.

Branching scheme

Updates to SuperflexPy are made directly in the branch `master`, which represent the most up-to-date branch. The branch `release` is not actively updated since the only action that should be done is a merge from the `master`, once a tangible update is available.

When something gets pushed to the branch `release`, a new version of the package is automatically released on PyPI. Remember to change the version number in the `setup.py` file.

Developers are free to create new branches but pull requests must be directed to `master` and not to `release`.

Documentation and examples are generated from the content of the `master` branch.

Note: Last update 26/06/2020

Tip: If interested in the context in which SuperflexPy operates, please check our publication (**link here when available**)

3.3 Principles of SuperflexPy

Numerical models are widely used in hydrology for prediction, process understanding, and engineering applications.

Models can differ depending on how the processes are represented (conceptual vs. physical based models) or on how the physical domain gets discretized (from lumped configurations to detailed grid-based models).

Conceptual models are, at the catchment scale, among the most used due to their limited number of parameters and high interpretability.

3.3.1 Conceptual models

Conceptual models are hydrological models that describe the dynamics directly at the catchment scale, providing relationship between the storage of the catchment and the outflow. Such models are usually relatively simple and cheap to run; their simplicity allows to use conceptual models to explore the processes directly at the catchment scale.

Thanks to their appealing features, a large variety of conceptual models has been proposed in the last 40 years. These models are usually quite similar, being composed by general elements such as reservoirs, lag functions, and connections but, at the same time, they are all slightly different one from the other, making model selection and comparison complicated.

Differences may appear in several levels:

- **conceptualization**: different models may decide to represent different processes;
- **mathematical model**: the same process (e.g. a flux) may be represented by different equations;
- **numerical model**: the same equation may be solved with different numerical techniques.

In order to overcome these 3 problems and to facilitate the configuration and comparison of different solutions, several flexible modeling frameworks have been proposed in the last decade.

3.3.2 Flexible modelling frameworks

A flexible modeling framework is a software platform that allows the user to build customized hydrological models that, usually, differ in the conceptualization but share the same mathematical and numerical formulation.

In order to achieve this result, flexible modeling frameworks usually offer a library of generic elements (e.g., reservoirs, lag functions, connections, etc.) and the possibility of connecting them freely.

In the last decade several flexible modeling frameworks have been proposed; while representing a step forward compared to classical conceptual models in terms of flexibility, these frameworks still present problematics:

- the promised infinite flexibility is actually lost in the implementation, with some frameworks that have a master structure with the possibility selecting the elements and fluxes to use;
- the choice of the numerical model is sometimes fixed, not allowing user to assess its impact on the results;
- the spatial discretization is usually pre-defined (e.g., some frameworks can operate only in lumped configuration while others are designed to operate on grids) not allowing the user to assess the impact of different discretizations;
- the frameworks are usually difficult to modify or extend by users that are not part of the core development team since these operations require a deep understanding of the source code;
- the source code itself may not be available as open-source and distributed only as executable;

These limitations, mainly due to implementation issues, limit the possibility of fully exploiting the potential of flexible modelling frameworks and can be addressed with a careful software implementation.

3.3.3 Spatial organization

Another important aspect to consider when designing a hydrological model is the spatial resolution to utilize to represent the catchment. Most of the existing models and frameworks can be classified in one or more of the following categories:

- **lumped configuration**, when all the physical domain is considered uniform;
- **grid-based configuration**, when the physical domain is subdivided with a (usually) uniform grid;

- **semi-distributed configuration**, when the physical domain is subdivided in irregular areas that have the same hydrological response.

The first approach produces the simplest model, with a limited number of parameters and usually fairly good predictions; the limitation of this choice is that, if there are areas of the catchment behaving differently, the model will not be able to represent this difference, with consequences on the values of the calibrated parameters and on their interpretation.

The second approach produces models with high computational demand and a large number of parameters; the catchment gets divided with a grid and the underlying assumption, that each pixel has its own hydrological behavior, may be relaxed aggregating different areas.

The third approach, which is in between the other two in terms of spatial complexity and number of parameters, tries to find a subdivision of the catchment that is driven by process understanding; this results is a subdivision in irregular areas that are supposed to have the same hydrological behavior; this approach enables the modeller to reflect his/her understanding of the dominant processes at the catchment scale.

3.3.4 SuperflexPy

In order to overcome most of the problems illustrated above, we have developed SuperflexPy, a new flexible framework for building conceptual hydrological models with different levels of spatial complexity, from lumped to semi-distributed.

SuperflexPy contains the functionalities to build all the common elements that can be found in the conceptual models or in the flexible modeling frameworks and to connect them, constructing spatially distributed configurations.

In order to do that, SuperflexPy is internally organized in four different levels to satisfy different degrees of spatial complexity:

- elements;
- units;
- nodes;
- network.

The lower level is represented by the elements; they can be, for example, reservoirs, lag functions, or connections and are designed to represent specific processes affecting the hydrological cycle (e.g. soil dynamics).

The second level is represented by the units; a unit is a component that connects together several elements creating the structure of a lumped configuration.

The third level is represented by the nodes; a node contains several units that operate in parallel. Each unit should represent the contribution of different hydrological behaving areas of the node.

The fourth level is represented by the network; a network connects different nodes, routing the fluxes from the upstream to the downstream ones. This enables the representation of complex watersheds that are composed by several subcatchments, creating a semi-distributed hydrological model.

Technical details on these components are provided in the *Organization of SuperflexPy* page.

Note: Last update 26/06/2020

3.4 Organization of SuperflexPy

Superflex is designed to operate at several levels of complexity, from a single reservoir to a complex river network. To do so, all the components are designed to operate alone or inside other components.

For this reason, all the components have a series of methods that are implemented to enable the execution of some basic functionality (e.g. parameters handling) at all the levels.

We will first describe the common aspects of all the components of the framework and, then, go specific in describing each one of them.

3.4.1 Generalities

Common methods

All the components share the following common methods.

- **Parameters and states**: each component may have parameters or states that are identified by a unique identifier. Each component of SuperflexPy have implemented some methods that enable to set or get states and parameters of the component and of the components that it contains:
  - `set_parameters`: change the value of the parameters
  - `get_parameters`: get the current value of the parameters
  - `get_parameters_name`: get the identifier of the parameters
  - `set_states`: change the value of the states
  - `get_states`: get the current value of the states
  - `get_states_name`: get the identifier of the parameters
  - `reset_states`: reset the states to their initialization value
- **Time step**: as commonly done in hydrological modeling, inputs and outputs are assumed to have the same constant time step. In SuperflexPy, all the components that require the definition of a time step (e.g. reservoirs that are controlled by a differential equation) contain the methods that enable to set and get the time step.
  - `set_timestep`: set the time step used in the model; all the components at a higher level (e.g. units) have this method; when called, it applies the change to all the elements contained in the component;
  - `get_timestep`: returns the time step used in the model.
- **Inputs and outputs**: all the components have functionalities to handle the inputs and to generate the outputs.
  - `set_input`: set the input fluxes of the component;
  - `get_output`: run the component (and all the components contained in it) and return the output fluxes.

Usage of the identifier

Parameters and states in SuperflexPy are identified using a string. The string can have an arbitrary length with the only requirement that it cannot contain the character underscore _.

Every component of SuperflexPy (except for the network) must have a unique identifier (that cannot contain the character _). When an element is inserted into a unit or when the unit is inserted into the node, the identifier of the component is prepended to the name of the parameter using the character _ as separator.

If, for example, the element with identifier `e1` has the parameter `par1`, the name of the parameter becomes, at initialization, `e1_par1`. When, then, the element is inserted into the unit `u1` its name becomes `u1_e1_par1`, and so on.

In this way, every parameter and state of the model has its own unique identifier that can be used to change its value from every component of SuperflexPy.

Time variant parameters

In hydrological modelling, time variant parameters can be useful for representing seasonal phenomena or stochasticity.

SuperflexPy is designed to operate both with constant and time variant parameters. Parameters, in fact, can be either scalar float numbers or numpy 1D arrays of the same length of the input fluxes; in the first case, the parameter will be interpreted as time constant while, in the second case, the parameter will be considered as time variant and have a specific value for each time step.

Time step and length of the simulation

As common practice in hydrological modeling, SuperflexPy uses a single uniform time step. This means that all the input time series of fluxes must have the same time resolution that will be, then, used to generate the outputs.

It has been decided to not have a parameter of the model fixing the length of the simulation (i.e. the number of time steps that needs to be run); this will be inferred at runtime from the length of the input fluxes that, for this reason, must all have the same length.

Inputs and outputs formats

Inputs and outputs fluxes of SuperflexPy's components are represented using 1D numpy arrays.

For the inputs, regardless the number of fluxes, the method `set_input` takes a list of numpy arrays (one array per flux); the order of the arrays inside the list must follow the indications of the documentation of the method. All the input fluxes must have the same dimension since, as explained in the section *Time step and length of the simulation*, the length of the simulation is defined by this dimension.

The output fluxes are also returned as a list of numpy 1D arrays from the `get_output` method.

Only for the elements, whenever the number of upstream or downstream elements is different from one (e.g. *Connections*), the `set_input` or the `get_output` methods will use bidimensional lists of numpy arrays: this solution is used to route fluxes from and to multiple elements.

3.4.2 Elements

Elements represent the basic level of the SuperflexPy's architecture; they can operate either alone or, connected together, as part of a unit.

Depending on the type, the elements can have parameters or states, can handle multiple fluxes as input or as output, can be designed to operate with one or more elements upstream or downstream, can be controlled by differential equations, or can be designed to operate a convolution operation on the incoming fluxes.

The framework provides a series of basic elements that can be extended by the user to satisfy all these possible modeling needs.

- `BaseElement`: element without states and parameters;
- `StateElement`: element with states but without parameters;

- `ParameterizedElement`: element with parameters but without states;
- `StateParameterizedElement`: element with states and parameters.

All the possible elements can be generated starting from the four general elements proposed; to facilitate the extension of the framework, we offer also some specific elements of common use in hydrological modeling; those are reservoirs, lag functions, and connections.

Reservoirs

A reservoir is an element that receives an input and transforms it, based on its internal state and on some parameters. It is usually governed by the differential equation

$$\frac{\mathrm{d}S}{\mathrm{d}t} = \mathbf{I}(\theta, t) - \mathbf{O}(S, \theta, t)$$

Where S is the internal state of the reservoir, $\mathbf{I}$ represents the incoming fluxes (usually independent from the state), $\mathbf{O}$ represents the outgoing fluxes, and θ is a vector representing the parameters that control the behavior of the reservoir.

The framework provides the class `ODEsElement` that contains all the logic that is needed to solve an element that is controlled by a differential equation. The user needs only to define the equations needed to calculate the fluxes.

The solution of the differential equation is done using a numerical approximation; the choice of the numerical approximation (e.g. implicit Euler) is left to the user, when initializing the reservoir.

SuperflexPy provides already some “numerical approximators” that can be used to create a numerical approximation of the differential equation (e.g. implicit or explicit Euler). These approximators are designed to operate coupled with a “root finder” that finds the solution (root) of the numerical approximation of the differential equation. The user can either use the numerical routines provided by the framework or implement the interface necessary to use an external solver (e.g. from scipy), which may be needed when the numerical problem becomes more complex (e.g. coupled differential equations). For more information about the numerical solver refer to the page *Numerical routines for solving ODEs*.

Lag functions

A lag function is an element that applies a delay to the incoming faxes; mathematically, the lag function applies a convolution to the incoming fluxes. In practice, the result is usually achieved distributing the fluxes at each time step in the following ones, according to weight array.

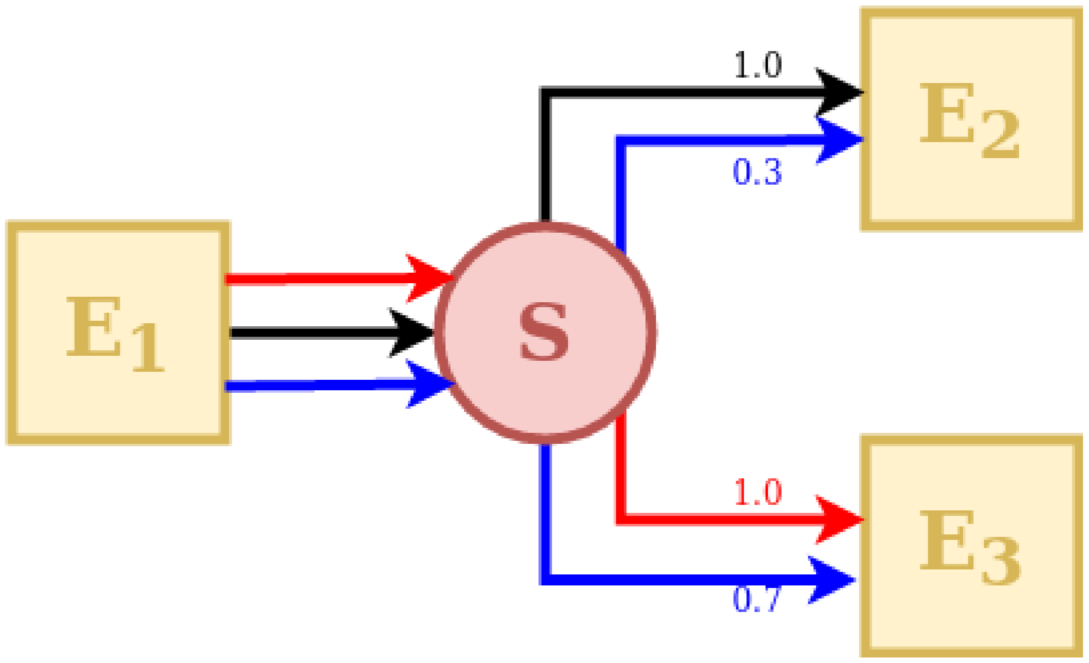
SuperflexPy already provides class, called `LagElement`, that implements all the methods needed to represent a lag function, leaving to user only the duty of defining weight array that has to be used.

Connections

Connection elements are needed to link together different elements, when building a unit. If an element has several elements downstream, for example, its fluxes need to be split using a `Splitter`; on the other hand, when the outflow of several elements is collected by a single one, this operation has to be done through a `Junction` element.

SuperflexPy provides several elements to connect and to fill the gaps in the structure; these elements are designed to operate with an arbitrarily large number of fluxes and upstream or downstream elements.

Splitter



A `Splitter` is an element that takes the outputs of a single upstream element and distributes them to feed several downstream elements.

The behavior of a splitter in SuperflexPy is controlled by two matrices: direction and weight. The first controls into which downstream elements the incoming fluxes are directed; the second defines the proportion of each flux that goes to the downstream elements.

Looking at the picture, the element E1 has 3 incoming fluxes: in order, red, black, and blue. The red flux is taken entirely by the element E3, the black flux is taken entirely by the element E2, and the blue flux is split at 30% to E2 and 70% to E3.

That direction matrix is a 2D matrix that has as many columns as the number of fluxes and as many rows as the number of downstream elements. Each element of the matrix contains the index identifying the incoming flux that is transferred in that position to the downstream element. The blue flux, for example, is the third (index 2) incoming flux and gets distributed as second input (index 1) to both downstream elements; the direction matrix will contain, therefore, the number 2 in position (0,1) and (1,1), with the first number (row) that indicates the downstream element and the second (column) that indicates the flux position. When a flux is not sent to a downstream element (e.g red flux to E2) it will be identified as None in the direction matrix.

The direction matrix for the splitter in the picture is here reported:

$$D = \begin{pmatrix} 1 & 2 & \text{None} \\ 0 & 2 & \text{None} \end{pmatrix}$$

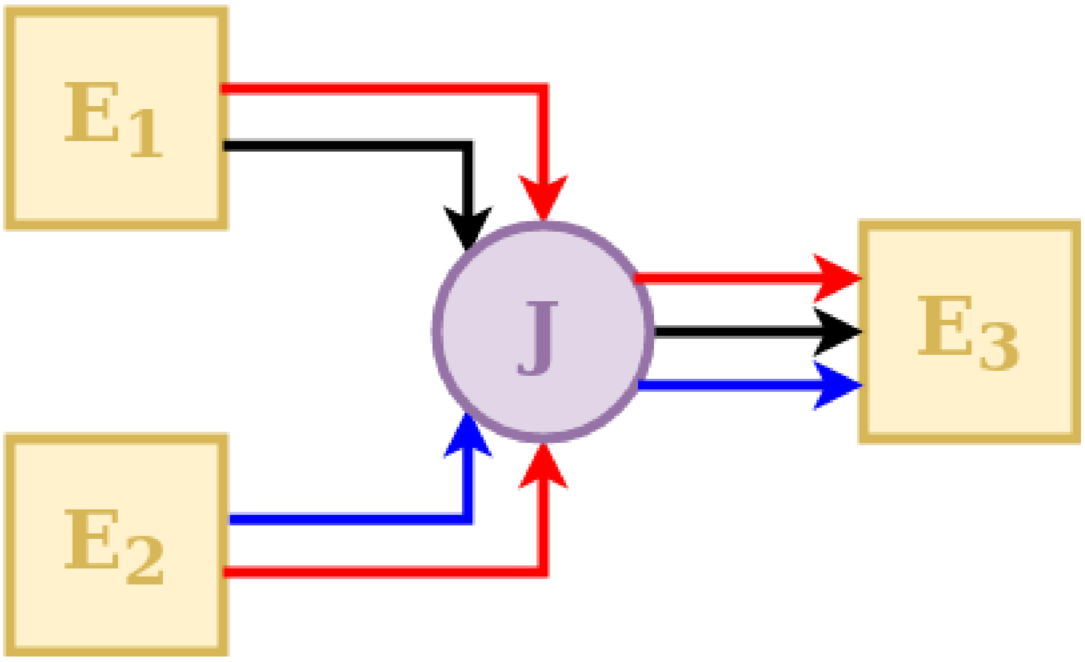
The weight matrix has the same dimensionality of the direction matrix. Each element of this matrix represents the proportion of the respective incoming flux that gets distributed to the specific downstream element. Looking at the blue flux, it will occupy the third column in the weight matrix (because it is the third incoming flux) and have value 0.3 in the first row (first downstream element) and 0.7 in the second row (second downstream element).

The weight matrix for the splitter in the picture is here reported:

$$W = \begin{pmatrix} 0 & 1.0 & 0.3 \\ 1.0 & 0 & 0.7 \end{pmatrix}$$

Note that, as a quick check, the sum of each column of the weight matrix should be 1 otherwise a portion of the flux is lost.

Junction



A `Junction` is an element that takes the outputs of several upstream elements and converges them into a single downstream element.

The behavior of a junction in SuperflexPy is controlled by direction matrix that defines how the incoming fluxes have to be aggregated (summed) to feed the downstream element.

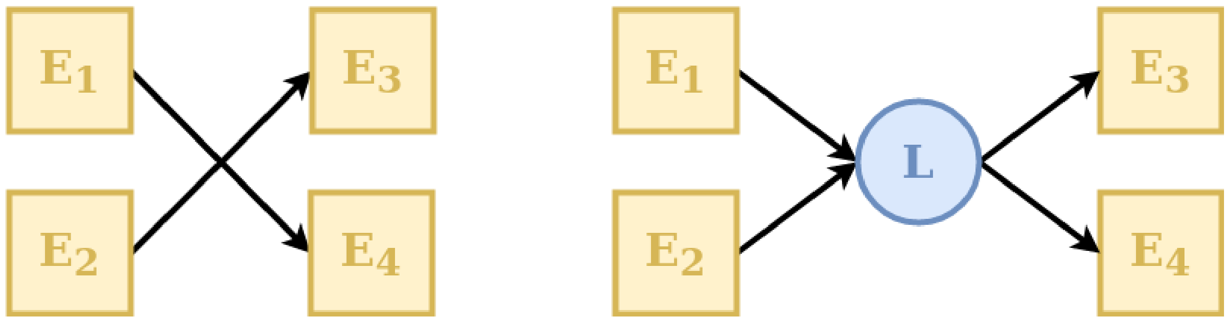
Looking at the picture, the element E3 takes three fluxes as input: in order, red, black, and blue. The red flux comes from both upstream elements; the black flux comes only from E1; the blue flux comes only from E2.

The direction matrix has as many rows as the number of fluxes and as many columns as number of upstream elements. Each entry of the matrix indicates the position of the flux of the upstream elements that compose a specific flux of the downstream element. The blue flux, that is the third incoming flux to E3, for example, is represented by the third row of the matrix with the couple (None, 0) since the flux is not present in E1 and it is the first flux of E2.

The direction matrix for the junction in the picture is here reported:

$$D = \begin{pmatrix} 0 & 1 \\ 1 & \text{None} \\ \text{None} & 0 \end{pmatrix}$$

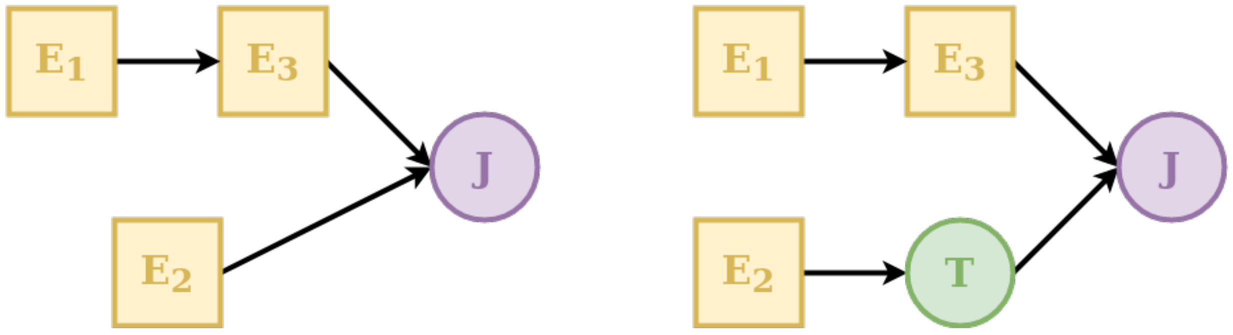
Linker



A `Linker` is an element that can be used to connect multiple elements upstream to multiple elements downstream.

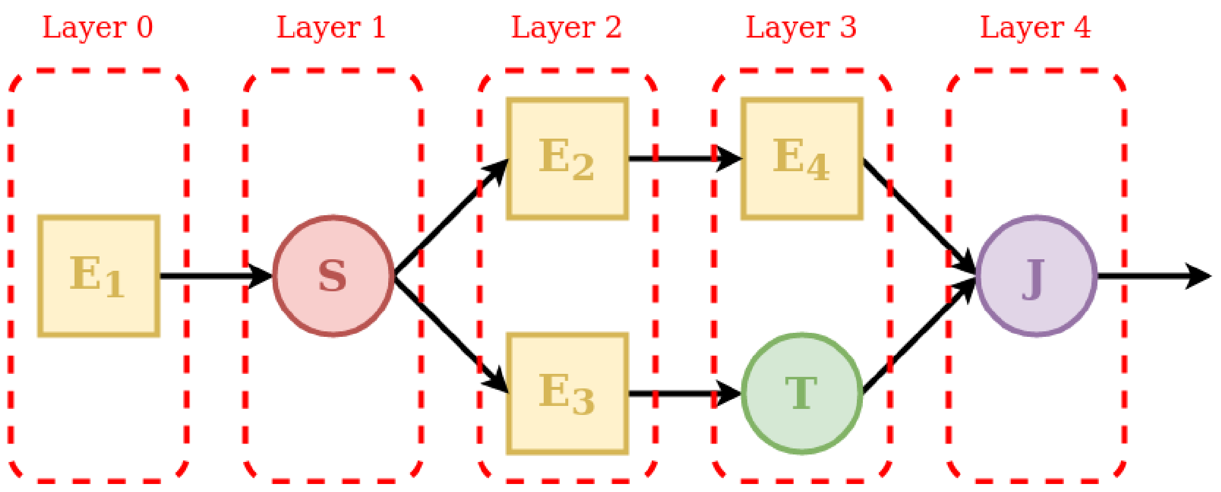
Its usefulness is due to the fact that in SuperflexPy the structure of the model is defined as an ordered list of elements. This means that (refer to the *Unit* section for further details) if we want to connect the first element of a layer with the second element of the following layer (e.g., E1 with E4, in the example above) we have to put an additional layer in between that contains a linker that direct the fluxes to the proper downstream element.

Transparent



A transparent element is an element that does nothing: it returns, as output, the same fluxes that it takes as input. It is needed to fill gaps in the structure defined in the unit.

3.4.3 Unit



The unit represents the second level of components in SuperflexPy and it is used to connect different elements, moving the fluxes from upstream to downstream. The unit can be used either alone in a lumped configuration or, as part of a node, to create a semi-distributed model.

The elements are copied into the unit: this means that an element that belongs to a unit is completely independent from the original element and from any other copy of the same element in another units. Changes to the state or to the parameters of an element inside a unit will, therefore, not reflect outside the unit.

As shown in the picture, the elements are organized as a succession of layers, from left (upstream) to right (downstream).

The first and the last layer must contain only a single element, since the inputs of the unit are transferred to the first element and the outputs of the unit are taken from the last element.

The order of the elements inside each layer defines how they are connected: the first element of a layer (e.g. E2 in the picture) will transfer its outputs to the first element of the downstream layer (e.g. E4); the second element of a layer (e.g. E3) will transfer its outputs to the second element of the downstream layer (e.g. T), and so on.

When the output of an element is split between more downstream elements (e.g. E1) the operation has to be done putting an additional layer in between that contains a splitter: in the example, the splitter S has two downstream elements (E2 and E3); the framework will route the first group of outputs of the splitter to E2 and the second to E3.

Whenever there is a gap in the structure, a transparent element should be used to fill the gap. In the example, the output of E3 have to be aggregated with the output of E4; since the elements belong to different layers, this can be achieved putting a transparent element in the same layer of E4.

Since the unit must have a single element in the last layer, the outputs of E4 and T must be collected using a Junction.

Each element is aware of the number of upstream and downstream elements that it must have (for example, a reservoir must have one element upstream and downstream, a splitter must have one element upstream and can have several elements downstream, and so on). The structure of a unit is valid only if the number downstream elements that a layer must have is equal to the number of upstream elements that the following must have. In the example, layer 1 must have two downstream elements (information contained in the splitter) that is consistent with the configuration of layer 2.

To get more familiar with the definition of the model structure in SuperflexPy and to understand how to reproduce the structures of popular models, refer to the page *Application: implementation of existing conceptual models*.

3.4.4 Node

The node represents the third level of components in SuperflexPy and it is used to aggregate different units, summing their contribution in the creation of the total outflow. The node can be run either alone or as part of a bigger network.

When a unit is inserted into a node, the default behavior is that the states of the elements belonging to the unit get copied while the parameters no. This means that, if same unit belongs to two different nodes (A and B), changes to the values of the parameters of the elements in node A will reflect also in node B while changes to the values of the states of the elements in node A will not reflect in node B. This default behavior can be changed, making also the parameters independent (set `shared_parameters=False` at initialization).

The choice of sharing the parameters between elements of the same unit that belong to different nodes is motivated by the fact that the unit is supposed to represent areas that have the same hydrological response. The idea is that the hydrological response is controlled by the parameters and that, therefore, elements of the same unit belonging to different nodes should have the same parameter values. The states, on the other hand, should be independent because different nodes may get different inputs and, therefore, the evolution of their states should be independent.

Refer to the page *Multiple units configuration* for details on how to incorporate the units inside the node.

Routing

The most common use of a node is to represent catchment, which can be part of a larger system, composing a network.

For this reason, the node has the possibility of defining routing functions that delay the fluxes; two types of routing are possible:

- internal routing;
- external routing.

The first is designed to simulate the delay that the fluxes get when they are collected from the units to the river network; the former is meant to represent the delay that derives from the routing of the fluxes inside the river network, between the outlets of the present node and of the downstream one.

In the default implementation of the node in SuperflexPy, the two routing functions simply return their input (i.e. no delay is applied); the user can change this behavior creating a customized node that implements these functions.

An example on how to do that can be found in the page *Adding the routing to a node*

3.4.5 Network

The network represents the fourth level of components in SuperflexPy and it is used to connect together several nodes, routing the fluxes from upstream to downstream.

The topology of the network is defined assigning to each node the information about its downstream node. The network will then solve the nodes, starting from the most upstream ones and then moving downstream, solving the remaining nodes and routing the fluxes towards the output of the network.

The network is the only component of SuperflexPy that does not have the `set_input` method since the input, which is node-specific, has to be assigned to each node belonging to the network.

When a node is inserted in the network it is not copied, meaning that any change the node (e.g. setting different inputs) outside the network reflects also inside.

To respond to the practical needs of the modeler, the output of the network is not only the output of the most downstream node but a dictionary that contains the output of all the nodes of the network.

Note: Last update 04/09/2020

3.5 Numerical routines for solving ODEs

Reservoirs are among the most common elements in conceptual hyrdological models. While their dynamic changes, reservoirs are still controlled by one (or more) ordinary differential equation (ODE) of the form

$$\frac{\mathrm{d}S}{\mathrm{d}t} = \mathbf{I}(\theta, t) - \mathbf{O}(S, \theta, t)$$

The analytical solution of such differential equation is almost always impossible to obtain and, therefore, a numerical approximation has to be constructed and solved.

In most of the cases, when a robust numerical approximation is used (e.g. Implicit Euler), the solution of the numerical approximation (i.e. finding a value of the state S that solves the algebraic equation) requires an iterative procedure, since an algebraic solution cannot be defined.

The solution of the ODE, therefore, can be seen as a two-steps approach:

1. Find a numerical approximation of the ODE
2. Solve such numerical approximation

This can be done in SuperflexPy using two components: `NumericalApproximator` and `RootFinder`. The first uses the fluxes from the reservoir element to construct a numerical approximation of the ODE, the second finds, numerically, the root of such approximation.

SuperflexPy provides already two numerical approximators (implicit and explicit Euler) and a root finder (which uses the Pegasus method). Other algorithms can be used extending the classes `NumericalApproximator` and `RootFinder`.

3.5.1 Numerical approximator

The implementation of a customized numerical approximator can be done extending the class `NumericalApproximator` and implementing two methods: `_get_fluxes` and `_differential_equation`.

```python
class CustomNumericalApproximator(NumericalApproximator):

    @staticmethod
    def _get_fluxes(fluxes, S, S0, args):

        # Some code here

        return fluxes

    @staticmethod
    def _differential_equation(fluxes, S, S0, dt, args, ind):

        # Some code here

        return [dif_eq, min_val, max_val]
```

where `fluxes` is a list of functions used to calculate the fluxes, `S` is an array of states that solve the ODE for the different time steps, `S0` is the initial state, `dt` is the time step, `args` is a list of additional arguments used by the functions in `fluxes`, and `ind` is the index of the input arrays to use.

The first method (`_get_fluxes`) is responsible for calculating the fluxes after the ODE has been solved and operates with a vector of states. The second method (`_differential_equation`) calculates the approximation of the ODE and it is designed to be interfaced to the root finder, returning the value of the differential equation and the minimum and maximum boundary for the search of the root.

To understand better how these methods work, please have a look at the implementation of `ImplicitEuler` and `ExplicitEuler`.

3.5.2 Root finder

The implementation of a root finder can be done extending the class `RootFinder` implementing the method `solve`.

```
class CustomRootFinder(RootFinder):

    def solve(self, dif_eq, fluxes, S0, dt, ind, args):

        # Some code here

        return root
```

where `dif_eq` is a function that calculates the value of the approximated ODE, `fluxes` is a list of functions used to calculate the fluxes, `S0` is the initial state, `dt` is the time step, `args` is a list of additional arguments used by the functions in `fluxes`, and `ind` is the index of the input arrays to use.

The method `solve` is responsible for finding the numerical solution of the approximated ODE.

To understand better how this method works, please have a look at the implementation of `Pegasus`.

3.5.3 Computational efficiency with Numpy and Numba

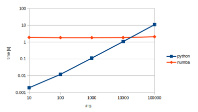
Conceptual hydrological models are often associated to computationally demanding tasks, such as parameter calibration and uncertainty quantification, which require multiple model runs (even millions). Computational efficiency is therefore an important requirement of a SuperflexPy.

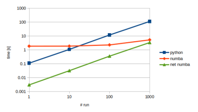
Computational efficiency is not the greatest strength of pure Python but libraries like Numpy and Numba can help in pushing the performance close to Fortran or C.

Numpy provides highly efficient arrays that can be transformed with C-time performance, as long as vector operations (i.e. elementwise operations between arrays) are run; Numba provides a "just-in-time compiler" that can be applied to a normal Python method to compile, at runtime, its content to machine code that interacts efficiently with NumPy arrays. This operation is extremely effective when solving ODEs where the method loops through a vector to perform some element-wise operations.

For this reason we provide a Numba-optimized version of the `NumericalApproximator` and of the `RootFinder` that enables to solve ODEs efficiently.

The figure shows the results of a benchmark that compares the execution times of the pure Python vs. the Numba implementation, in relation to the length of the time series (panel a) and to the number of model runs (panel b). The plots clearly show the tradeoff between compile time (which is zero for Python and around 2 seconds for Numba) and run time, where Numba is 30 times faster than Python. This means that the choice of the implementation to use, which can be done simply using a different `NumericalApproximator` implementation, may depend on the application: a single run of the *HYMOD* model, with the implicit Euler numerical solver and a time series of 1000 time steps takes 0.11 seconds with Python and 1.85 with Numba while, if the same model is run 100 times (for a calibration, for example) the Numba version takes 2.35 seconds while the Python version 11.75 seconds.





Note: Last update 26/06/2020

3.6 Quick demo

In this demo we will build a simple semi-distributed conceptual model with SuperflexPy, showing how the elements are initialized, configured, and run and how to use the model at any level of complexity, from single element to multiple nodes.

3.6.1 Importing SuperflexPy

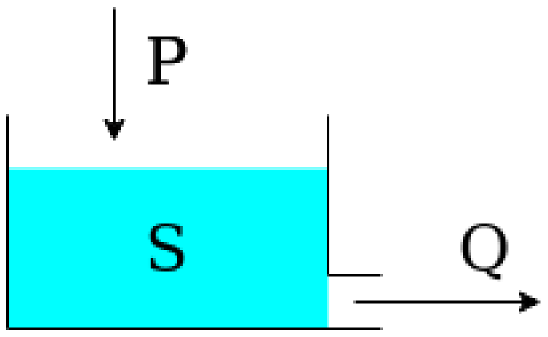
Assuming that SuperflexPy has already been installed (refer to the *Installation* guide), the elements needed to build the model must be imported from the SuperflexPy package. For this demo, this is done with the following lines

```
1 from superflexpy.implementation.elements.hbv import FastReservoir
2 from superflexpy.implementation.elements.gr4j import UnitHydrograph1
3 from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
4 from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
5 from superflexpy.framework.unit import Unit
6 from superflexpy.framework.node import Node
7 from superflexpy.framework.network import Network
```

Lines 1-2 import the two element that we will use (a reservoir and a lag function), lines 3-4 import the numerical solver used to solve the differential equation of the reservoir, lines 5-7 import the components of SuperflexPy needed to make the model spatially distributed.

A complete list of the elements already implemented in SuperflexPy, including the equations used and the import path is available at the *Elements list* page. If the desired element is not available it can be built following the instruction given in the *Expand SuperflexPy: build customized elements* page.

3.6.2 Single-element configuration



The single-element model is composed by a single reservoir that is governed by the following differential equation

$$\frac{\mathrm{d}S}{\mathrm{d}t} = P - Q$$

where S is the state of the reservoir, P is the precipitation input, and Q is the outflow, defined by the equation:

$$Q = kS^{\alpha}$$

where k and α are parameters of the element. It can be noticed that, for simplicity, evapotranspiration is not considered in this demo.

The first step that needs to be done is initializing the numerical approximator, needed to construct the differential equation to solve, and the root finder used to find the value of the state that solves the approximation of the differential equation. In this case, we choose to use the Python implementation of implicit Euler (numerical approximator) and of the Pegasus algorithm (root finder). This can be done with the following code, where the default settings of the solver are used (refer to the solver docstring).

```
solver_python = PegasusPython()

approximator = ImplicitEulerPython(root_finder=solver_python)
```

After that, the element can be initialized

```
1 fast_reservoir = FastReservoir(
2     parameters={'k': 0.01, 'alpha': 2.0},
3     states={'S0': 10.0},
4     approximation=approximator,
5     id='FR'
6 )
```

During initialization, parameters (line 2) and initial state (line 3) are defined, together with the numerical approximator and the identifier (the identifier must be unique and cannot contain the character _).

After initialization, the time step used to solve the differential equation and the inputs of the element must be defined.

```
fast_reservoir.set_timestep(1.0)
fast_reservoir.set_input([precipitation])
```

Precipitation is a numpy array containing the precipitation time series. Note that the length of the simulation (i.e., number of time steps to run the model) cannot be defined and it is automatically set equal to the length of the input arrays.

At this point, the element can be run, calling the method get_output

```
output = fast_reservoir.get_output()[0]
```

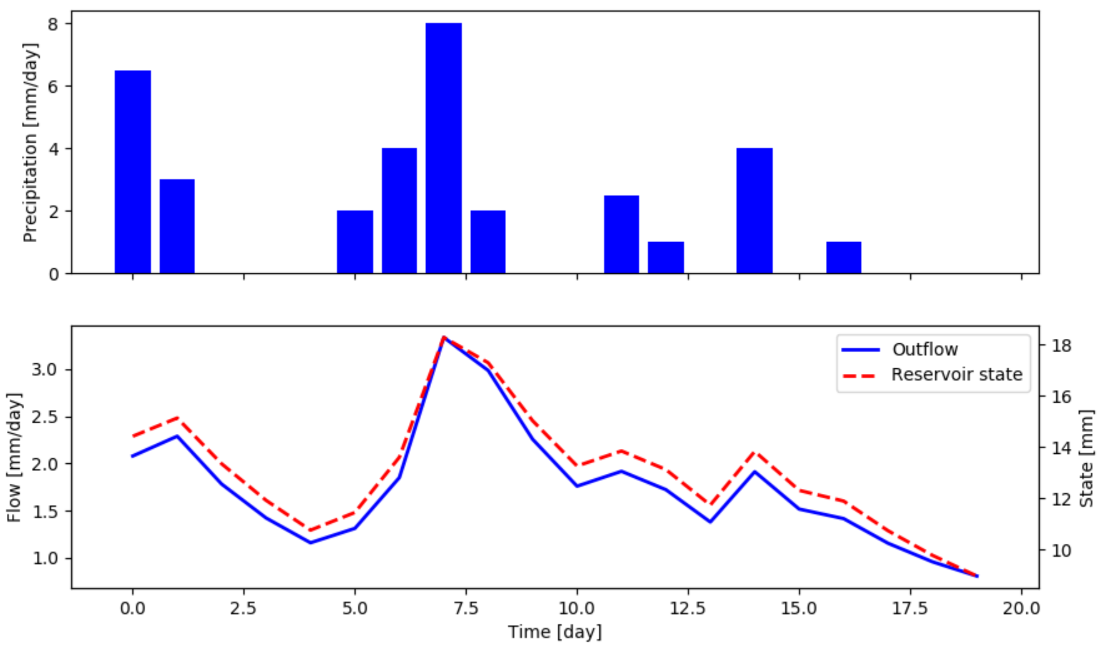
The method will run the element for all the time steps solving the differential equation and return a list containing all the output arrays of the element (in this specific case there is only one output, Q).

After running, the state of the reservoir (for all the time steps) is saved in the state_array attribute of the element and can be inspected

```
reservoir_state = fast_reservoir.state_array[:, 0]
```

the state_array is a 2D array with as many row (first dimension) as the number of time steps and as many columns (second dimension) as the number of states. The order of the states is defined in the documentation of the element.

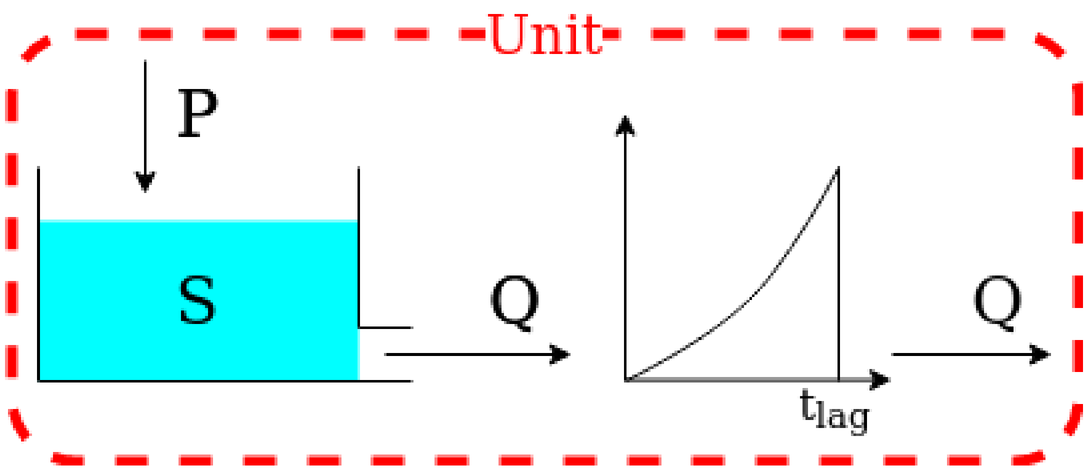
The plot shows the output of the simulation.



Note that the get_output method also sets the element states to their value at the final time step (in this case 8.98). Therefore, if the method is called again, it will use this value as initial state instead to the one defined at initialization. The states of the model can be reset calling the `reset_states` method.

```
fast_reservoir.reset_states()
```

3.6.3 Lumped model structure



We now move from a single-element configuration to multiple elements connected. For simplicity, we limit the complexity at two elements; more complex configurations can be found in the *Application: implementation of existing conceptual models* page.

The structure is composed by a reservoir, which output is taken, as input, by a lag function. The lag function convolutes the incoming flux using the function

$$Q_{\text{out}} = Q_{\text{in}} \left(\frac{t}{t_{\text{lag}}}\right)^{\frac{5}{2}} \qquad \text{for } t < t_{\text{lag}}$$

and its behavior is controlled by the parameter t_{lag}.

The first step consists in initializing the two elements that compose the structure

```
1  fast_reservoir = FastReservoir(
2      parameters={'k': 0.01, 'alpha': 2.0},
3      states={'S0': 10.0},
4      approximation=approximator,
5      id='FR'
6  )
7
8  lag_function = UnitHydrograph1(
9      parameters={'lag-time': 2.3},
10     states={'lag': None},
11     id='lag-fun'
12 )
```

Note that the initial state of the lag function has been set to `None` (line 10); in this case the element will initialize the state to an arrays of zeros of the proper length, depending on the value of t_{lag} (in this specific case, 3).

We can, then, initialize the unit that connect the elements, defining the model structure

```
unit_1 = Unit(
    layers=[[fast_reservoir], [lag_function]],
    id='unit-1'
)
```

Line 2 defines the structure; it is a 2D list where the inner level sets the elements belonging to each layer an the outer level lists the layers. Note that the order of the elements in the list is of primary importance. Refer to *Unit* for further details.

After initialization, time step and inputs must be defined

```
unit_1.set_timestep(1.0)
unit_1.set_input([precipitation])
```

The unit sets the time step of all the elements that contains to the provided value and transfers the inputs to the first element (the reservoir, in this example).

After that, the unit can be run

```
output = unit_1.get_output()[0]
```

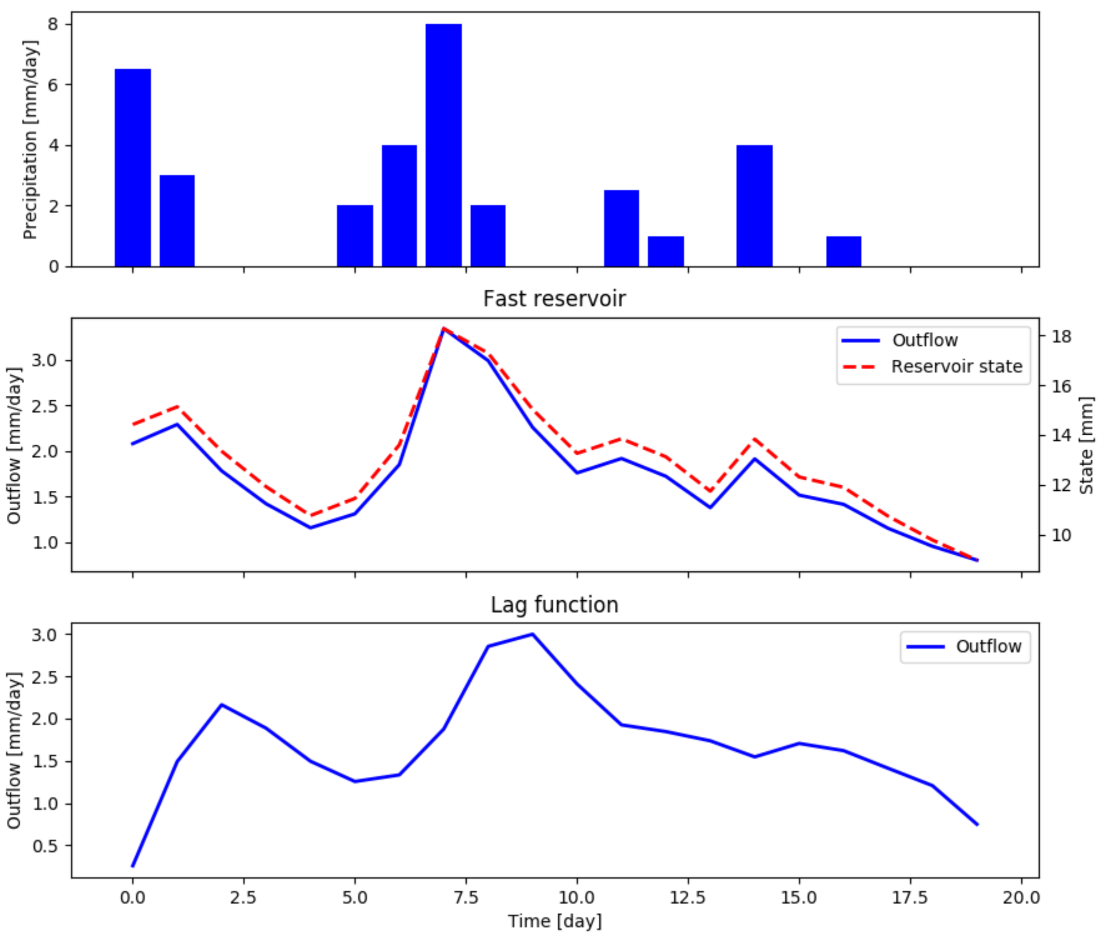
The unit will call the `get_output` method of all its elements, from upstream to downstream, set the inputs of the downstream elements to the output of their respective upstream elements, and return the output of the last element.

The outputs and the states of the internal elements can be inspected after running the model

```
1 fr_state = unit_1.get_internal(id='FR', attribute='state_array')[:, 0]
2 fr_output = unit_1.call_internal(id='FR', method='get_output', solve=False)[0]
```

Note that (line 2) we need to pass to the function `get_output` of the reservoir the argument `solve=False` in order to avoid to re-run the element.

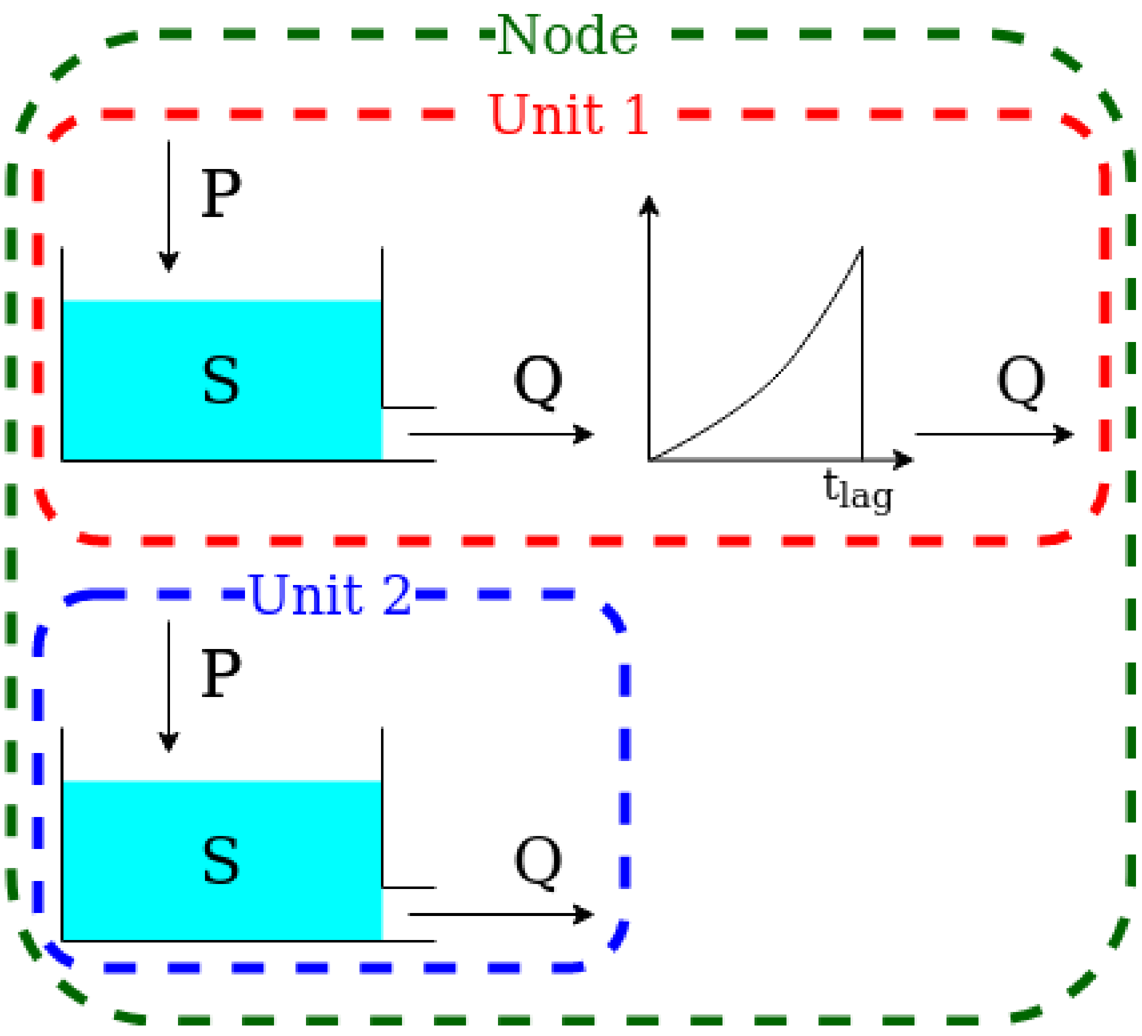
The plot shows the output of the simulation.



The elements of the unit can be re-set to their initial state

```
unit_1.reset_states()
```

3.6.4 Multiple units configuration



A catchment (node) can be composed by different areas that react differently to the same precipitation input. We may have 70% of the catchment that can be represented with the structure described in the previous section and 30% that can be described simply by a single reservoir.

This behavior can be simulated with SuperflexPy creating a node that contains multiple units.

The first step consists in initializing the two units and the elements composing them, as done in the previous sections.

```
fast_reservoir = FastReservoir(
    parameters={'k': 0.01, 'alpha': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
    id='FR'
)

lag_function = UnitHydrograph1(
    parameters={'lag-time': 2.3},
    states={'lag': None},
```

(continues on next page)

(continued from previous page)

```
    id='lag-fun'
)

unit_1 = Unit(
    layers=[[fast_reservoir], [lag_function]],
    id='unit-1'
)

unit_2 = Unit(
    layers=[[fast_reservoir]],
    id='unit-2'
)
```

Note that, once the elements are added to a unit, they become independent, meaning that any change to the reservoir contained in `unit-1` does not affect the reservoir in `unit-2`.

At this point, the node can be initialized, putting together the two units

```
1  node_1 = Node(
2      units=[unit_1, unit_2],
3      weights=[0.7, 0.3],
4      area=10.0,
5      id='node-1'
6  )
```

Line 2 contains the list of the units that belong to the node, line 3 their weight (i.e. part of the node outflow influenced by this unit). The representative area of the node (line 4) will be used, in case, by the network.

After that, time step and inputs must be defined

```
node_1.set_timestep(1.0)
node_1.set_input([precipitation])
```

the same time step will be set to the elements composing all the units of the node, the inputs will be passed to all the units of the node.

We can now run the node and collect its output

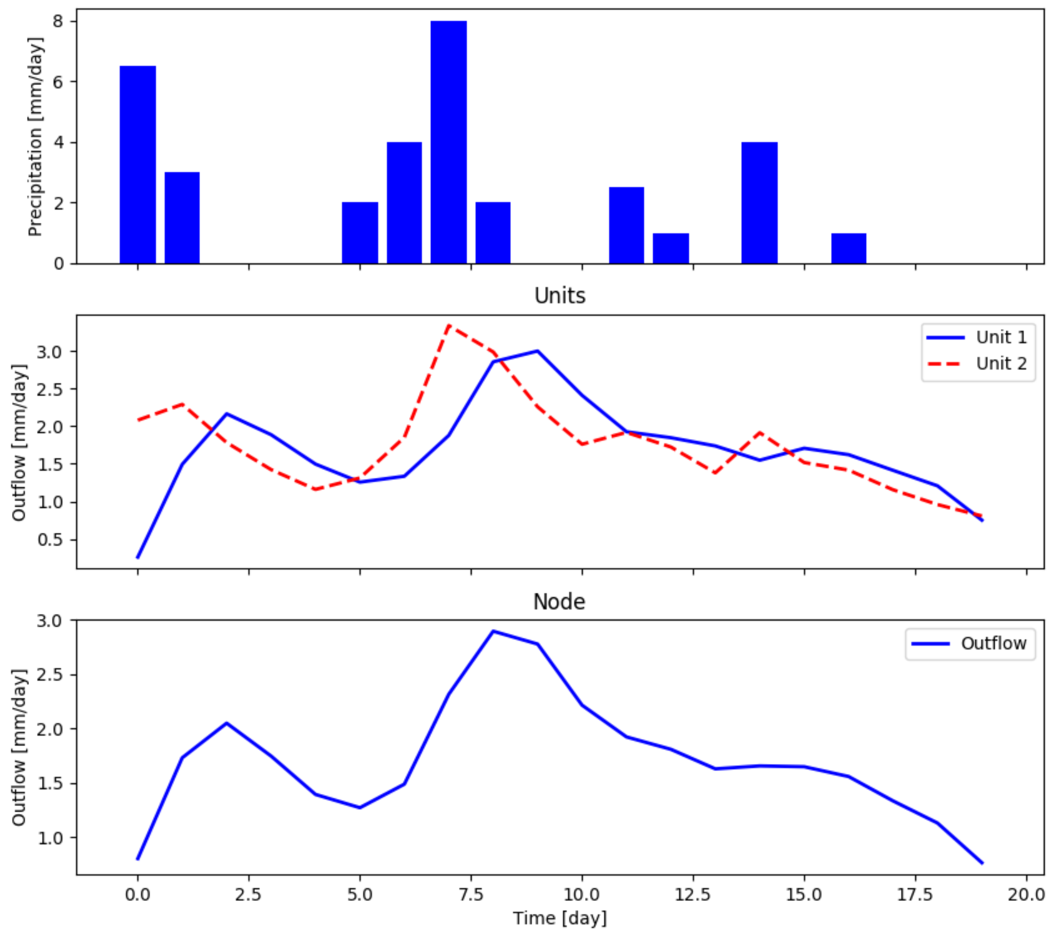
```
output = node_1.get_output()[0]
```

The node will call the get_output method of all the units and aggregate their outputs using the weights. In the case of multiple fluxes (e.g., water and contaminants) their order must be the same in all the units.

The outputs of the single units, as well as the states and fluxes of the elements composing them, can be inspected

```
output_unit_1 = node_1.call_internal(id='unit-1', method='get_output', solve=False)[0]
output_unit_2 = node_1.call_internal(id='unit-2', method='get_output', solve=False)[0]
```

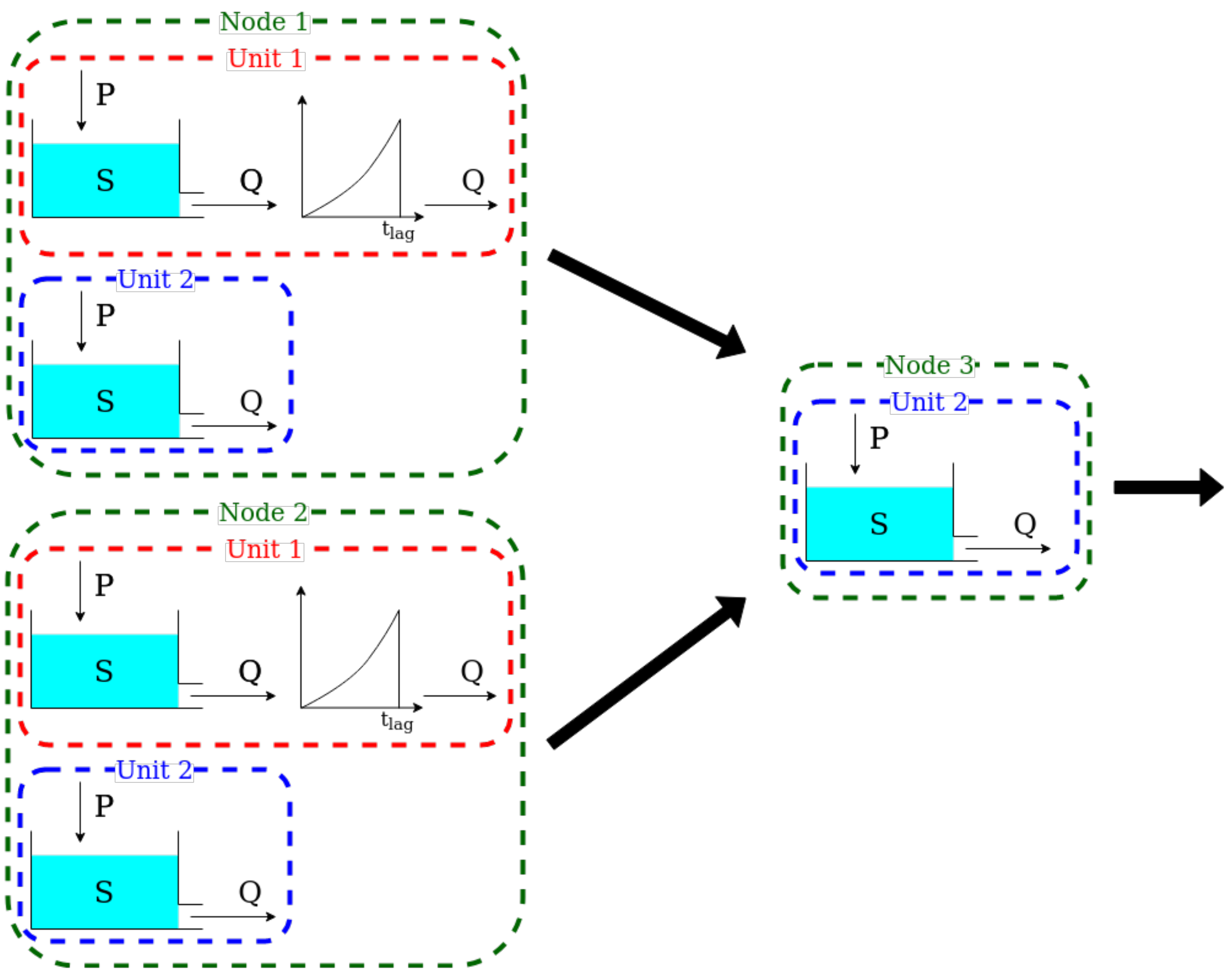
The plot shows the output of the simulation.



All the elements composing the node can be re-set to their initial state

```
node_1.reset_states()
```

3.6.5 Multiple nodes in a network



A watershed can be composed by several catchments connected in a network that have different inputs but share areas with the same hydrological behavior. This can be simulated with SuperflexPy creating a network that contains multiple nodes.

The first step for creating a network is initializing the nodes composing it.

```
fast_reservoir = FastReservoir(
    parameters={'k': 0.01, 'alpha': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
    id='FR'
)

lag_function = UnitHydrograph1(
    parameters={'lag-time': 2.3},
    states={'lag': None},
    id='lag-fun'
)

unit_1 = Unit(
    layers=[[fast_reservoir], [lag_function]],
```

(continues on next page)

(continued from previous page)

```
    id='unit-1'
)

unit_2 = Unit(
    layers=[[fast_reservoir]],
    id='unit-2'
)

node_1 = Node(
    units=[unit_1, unit_2],
    weights=[0.7, 0.3],
    area=10.0,
    id='node-1'
)

node_2 = Node(
    units=[unit_1, unit_2],
    weights=[0.3, 0.7],
    area=5.0,
    id='node-2'
)

node_3 = Node(
    units=[unit_2],
    weights=[1.0],
    area=3.0,
    id='node-3'
)
```

`node-1` and `node-2` contains both the units but with different proportions; `node-3` contains only `unit-2`. When units are added to a catchment the states of the elements belonging to them remain independent while the parameters stay linked, meaning that the change of a parameter in `unit-1` in `node-1` is applied also in `unit-1` in `node-2`. Different behavior can be achieve setting the parameter `shared_parameters` equal to `False` when initializing the nodes.

At this point, the network can be initialized

```
1  net = Network(
2      nodes=[node_1, node_2, node_3],
3      topography={
4          'node-1': 'node-3',
5          'node-2': 'node-3',
6          'node-3': None
7      }
8  )
```

Line 2 lists the nodes belonging to the network, lines 4-6 define the topology of the network; this is done with a dictionary that has the identifier of the nodes as key and the identifier of the single downstream node as value; the most downstream node has value None.

The inputs are catchment-specific and must be provided to each catchment, as done in the single-node case.

```
node_1.set_input([precipitation])
node_2.set_input([precipitation * 0.5])
node_3.set_input([precipitation + 1.0])
```

The time step is set by the network to the same value for all the nodes.

```
net.set_timestep(1.0)
```

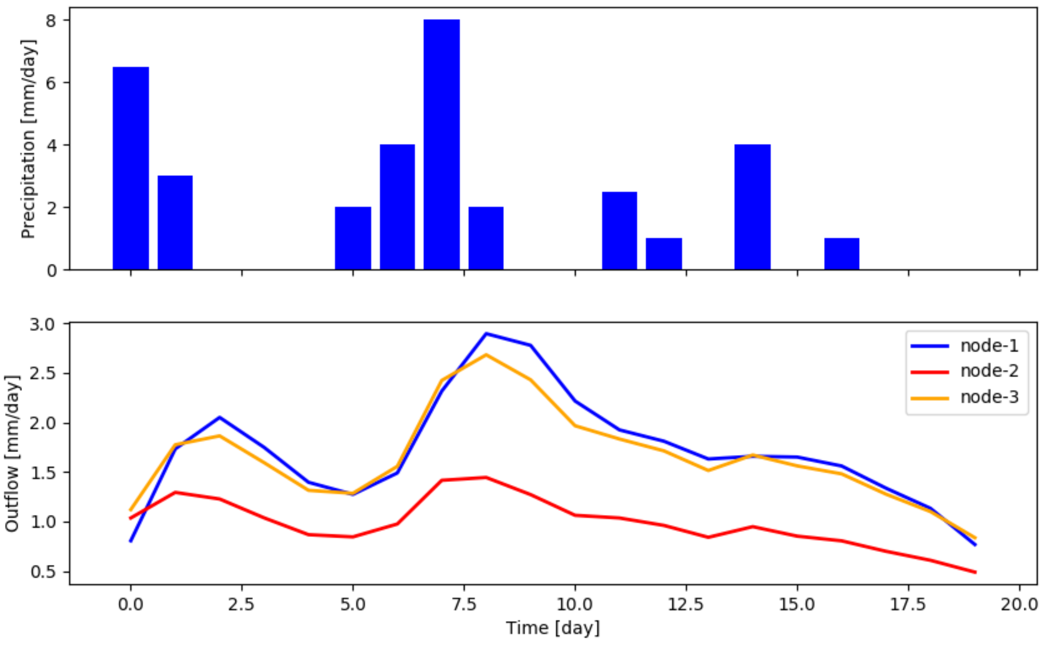
We can now run the network and get the output values

```
output = net.get_output()
```

The network runs the nodes from upstream to downstream, collects their output, and route them to the outlet. The output of the network is a dictionary that has the identifier of the nodes, as key, and the list of output fluxes, as value. It is also possible to inspect the internals of the nodes, as done with the single-node case.

```
output_unit_1_node_1 = net.call_internal(id='node-1_unit-1',
                                         method='get_output',
                                         solve=False)[0]
```

The plot shows the results of the simulation.



Note: Last update 26/06/2020

3.7 Elements list

This page contains all the elements implemented as part of SuperflexPy. The elements are divided in three categories

- Reservoir
- Lag functions
- Connectors

We will now list all the elements in alphabetical order.

3.7.1 Reservoirs

Fast reservoir (HBV)

```
from superflexpy.implementation.elements.hbv import FastReservoir
```

Governing equations

$$
\begin{aligned}
&\frac{\mathrm{d}S}{\mathrm{d}t} = P - Q \\
&Q = kS^{\alpha}
\end{aligned}
$$

Inputs required

- Precipitation

Main outputs

- Total outflow

Interception filter (GR4J)

```
from superflexpy.implementation.elements.gr4j import InterceptionFilter
```

Governing equations

$$
\begin{aligned}
&\text{if } P^{\text{in}} > E^{\text{in}}_{\text{POT}}: \\
&\quad P^{\text{out}} = P^{\text{in}} - E^{\text{in}}_{\text{POT}} \\
&\quad E^{\text{out}}_{\text{POT}} = 0 \\
\\
&\text{if } P^{\text{in}} < E^{\text{in}}_{\text{POT}}: \\
&\quad P^{\text{out}} = 0 \\
&\quad E^{\text{out}}_{\text{POT}} = E^{\text{in}}_{\text{POT}} - P^{\text{in}}
\end{aligned}
$$

Inputs required

- Potential evapotranspiration
- Precipitation

Main outputs

- Net potential evapotranspiration
- Net precipitation

Linear reservoir (Hymod)

```
from superflexpy.implementation.elements.hymod import LinearReservoir
```

Governing equations

$$\begin{aligned}&\frac{\mathrm{d}S}{\mathrm{d}t}=P-Q\\&Q=kS\end{aligned}$$

Inputs required

- Precipitation

Main outputs

- Total outflow

Production store (GR4J)

```
from superflexpy.implementation.elements.gr4j import ProductionStore
```

Governing equations

$$\begin{aligned}&\frac{\mathrm{d}S}{\mathrm{d}t}=P_{\mathrm{s}}-E_{\mathrm{act}}-Perc\\&P_{\mathrm{s}}=P\left(1-\left(\frac{S}{x_1}\right)^{\alpha}\right)\\&E_{\mathrm{act}}=E_{\mathrm{pot}}\left(2\frac{S}{x_1}-\left(\frac{S}{x_1}\right)^{\alpha}\right)\\&Perc=\frac{x^{1-\beta}}{(\beta-1)\mathrm{d}t}\nu^{\beta-1}S^{\beta}\\&P_{\mathrm{r}}=P-P_{\mathrm{s}}+Perc\end{aligned}$$

Inputs required

- Potential evapotranspiration
- Precipitation

Main outputs

- Total outflow (P_{r})

Secondary outputs

- Actual evapotranspiration (E_{act})

Routing store (GR4J)

```
from superflexpy.implementation.elements.gr4j import RoutingStore
```

Governing equations

$$\begin{aligned}
&\frac{\mathrm{d}S}{\mathrm{d}t} = P - Q - F \\
&Q = \frac{x_3^{1-\gamma}}{(\gamma - 1)\mathrm{d}t} S^{\gamma} \\
&F = \frac{x_2}{x_3^{\omega}} S^{\omega}
\end{aligned}$$

Inputs required

- Precipitation

Main outputs

- Outflow (Q)
- Loss term (F)

Snow reservoir (Thur model HESS)

```
from superflexpy.implementation.elements.thur_model_hess import SnowReservoir
```

Governing equations

$$
\begin{aligned}
&\frac{\mathrm{d}S}{\mathrm{d}t} = Sn - M \\
&Sn = P \quad \text{if } T \leq T_0; \quad \text{else } 0 \\
&M = M_{\text{pot}} \left(1 - \exp\left(-\frac{S}{m}\right)\right) \\
&M_{\text{pot}} = kT \quad \text{if } T \geq T_0; \quad \text{else } 0
\end{aligned}
$$

Inputs required

- Precipitation (total, the separation between snow and rain is done internally)
- Temperature

Main outputs

- Melt + rainfall input

Unsaturated reservoir (HBV)

```
from superflexpy.implementation.elements.hbv import UnsaturatedReservoir
```

Governing equations

$$
\begin{aligned}
&\frac{\mathrm{d}S}{\mathrm{d}t} = P - E_{\text{act}} - Q \\
&E_{\text{act}} = C_{\text{e}} E_{\text{pot}} \left(\frac{\left(\frac{S}{S_{\text{max}}}\right)(1+m)}{\frac{S}{S_{\text{max}}} + m} \right) \\
&Q = P \left(\frac{S}{S_{\text{max}}}\right)^{\beta}
\end{aligned}
$$

Inputs required

- Precipitation
- Potential evapotranspiration

Main outputs

- Total outflow

Secondary outputs

- Actual evapotranspiration

Upper zone (Hymod)

```
from superflexpy.implementation.elements.hymod import UpperZone
```

Governing equations

$$
\begin{aligned}
\frac{\mathrm{d}S}{\mathrm{d}t} &= P - E_{\mathrm{act}} - Q \\
E_{\mathrm{act}} &= E_{\mathrm{pot}}\left(\frac{\left(\frac{S}{S_{\mathrm{max}}}\right)(1+m)}{\frac{S}{S_{\mathrm{max}}}+m}\right) \\
Q &= P\left(1-\left(1-\frac{S}{S_{\mathrm{max}}}\right)^{\beta}\right)
\end{aligned}
$$

Inputs required

- Precipitation
- Potential evapotranspiration

Main outputs

- Total outflow

Secondary outputs

- Actual evapotranspiration

3.7.2 Lag functions

All the lag functions implemented in SuperflexPy are designed to take an arbitrary number of input fluxes and to apply a transformation to it based on a weight array that defines the shape of the lag function. It is only this that differentiate different lag functions.

Half triangular lag (Thur model HESS)

```
from superflexpy.implementation.elements.thur_model_hess import HalfTriangularLag
```

Equation used for the lag

The area of the lag is calculated with the following expression

$$A_{\text{lag}}(t) = 0$$

for $t \leq 0$

$$A_{\text{lag}}(t) = \left(\frac{t}{t_{\text{lag}}}\right)^2$$

for $0 < t \leq t_{\text{lag}}$

$$A_{\text{lag}}(t) = 1$$

for $t > t_{\text{lag}}$

The weight array is then calculated as the difference between the value of A_{lag} at two adjacent points.

$$w(t_{\text{j}}) = A_{\text{lag}}(t_{\text{j}}) - A_{\text{lag}}(t_{\text{j-1}})$$

Unit hydrograph 1 (GR4J)

```
from superflexpy.implementation.elements.gr4j import UnitHydrograph1
```

Equation used for the lag

The area of the lag is calculated with the following expression

$$A_{\text{lag}}(t) = 0$$

for $t \leq 0$

$$A_{\text{lag}}(t) = \left(\frac{t}{t_{\text{lag}}}\right)^{\frac{5}{2}}$$

for $0 < t \leq t_{\text{lag}}$

$$A_{\text{lag}}(t) = 1$$

for $t > t_{\text{lag}}$

The weight array is then calculated as the difference between the value of A_{lag} at two adjacent points.

$$w(t_{\text{j}}) = A_{\text{lag}}(t_{\text{j}}) - A_{\text{lag}}(t_{\text{j-1}})$$

Unit hydrograph 2 (GR4J)

```
from superflexpy.implementation.elements.gr4j import UnitHydrograph2
```

Equation used for the lag

The area of the lag is calculated with the following expression

$$A_{\text{lag}}(t) = 0$$
$$\text{for } t \leq 0$$
$$A_{\text{lag}}(t) = \frac{1}{2}\left(\frac{2t}{t_{\text{lag}}}\right)^{\frac{5}{2}}$$
$$\text{for } 0 < t \leq \frac{t_{\text{lag}}}{2}$$
$$A_{\text{lag}}(t) = 1 - \frac{1}{2}\left(2 - \frac{2t}{t_{\text{lag}}}\right)^{\frac{5}{2}}$$
$$\text{for } \frac{t_{\text{lag}}}{2} < t \leq t_{\text{lag}}$$
$$A_{\text{lag}}(t) = 1$$
$$\text{for } t > t_{\text{lag}}$$

The weight array is then calculated as the difference between the value of A_{lag} at two adjacent points.

$$w(t_{\text{j}}) = A_{\text{lag}}(t_{\text{j}}) - A_{\text{lag}}(t_{\text{j-1}})$$

3.7.3 Connectors

SuperflexPy implements, by default four different connectors:

- splitter
- junction
- linker
- transparent element

All of them are designed to operate with an infinite number of fluxes and, when possible, with infinite upstream or downstream elements.

Apart from those, there are also some connectors that have been implemented as part of a specific configuration, to achieve a particular design.

Note: Last update 26/06/2020

Note: If you build your own component using SuperflexPy, you should also share your implementation with the community and contribute to the *Elements list* page to make other users aware of your implementation.

3.8 Expand SuperflexPy: build customized elements

In this page we will illustrate how to create customized elements using the SuperflexPy framework.

The examples include three elements:

- Linear reservoir

- Half-triangular lag function
- Parameterized splitter

The elements presented here are as simple as possible, since the focus is in explaining how the framework works, rather than providing code to use in a practical application. To understand deeper how to exploit all the functionalities of SuperflexPy, please have a look at the elements that have been already implemented (importing path `superflexpy.implementation.elements`).

In this page we report, for brevity, only the code, without docstring. The complete code used to generate this page is available at the path `doc/source/build_element_code.py`.

3.8.1 Linear reservoir

The linear reservoir is one of the simplest reservoir that can be built. The idea is that the output flux is a linear function of the state of the reservoir.

The element is controlled by the following differential equation

$$\frac{\mathrm{d}S}{\mathrm{d}t} = P - Q$$

with

$$Q = kS$$

The solution of the differential equation can be approximated using a numerical method with the equation that, in the general case, becomes:

$$\frac{S_{t+1} - S_t}{\Delta t} = P - Q(S)$$

Several numerical methods exist to approximate the solution of the differential equation and, usually, they differ for the state used to evaluate the fluxes: implicit Euler, for example, uses the state at the end of the time step (S_{t+1})

$$\frac{S_{t+1} - S_t}{\Delta t} = P - kS_{t+1}$$

explicit Euler uses the state at the beginning of the time step (S_t)

$$\frac{S_{t+1} - S_t}{\Delta t} = P - kS_t$$

and so on for other methods.

Note that, even if for this simple case the differential equation can be solved analytically and the solution of the numerical approximation can be found without iteration, we will use anyway the numerical solver offered by SuperflexPy to illustrate how to proceed in a more general case where such option is not available.

The framework provides the class `ODEsElement` that has most of the methods required to solve the element. The class implementing the element will inherit from this and implement only a few methods.

```python
import numba as nb
from superflexpy.framework.element import ODEsElement

class LinearReservoir(ODEsElement):
```

The first method to implement is the class initializer

```
1      def __init__(self, parameters, states, approximation, id):
2
3          ODEsElement.__init__(self,
4                               parameters=parameters,
5                               states=states,
6                               approximation=approximation,
7                               id=id)
8
9          self._fluxes_python = [self._fluxes_function_python]  # Used by get fluxes,␣
   ↪regardless of the architecture
10
11         if approximation.architecture == 'numba':
12             self._fluxes = [self._fluxes_function_numba]
13         elif approximation.architecture == 'python':
14             self._fluxes = [self._fluxes_function_python]
15         else:
16             message = '{}The architecture ({}) of the approximation is not correct'.
   ↪format(self._error_message,
17                                                                                     ␣
   ↪     approximation.architecture)
18             raise ValueError(message)
```

The main purpose of the method (lines 9-16) is to deal with the numerical solver used for solving the differential equation. In this case we can accept two architectures: pure python or numba. The option selected will change the function used to calculate the fluxes. Keep in mind that, since some operations the python implementation of the fluxes is still used, this must be always present.

The second method to define is the one that maps the (ordered) list of input fluxes to a dictionary that gives a name to these fluxes.

```
    def set_input(self, input):

        self.input = {'P': input[0]}
```

Note that the name (key) of the input flux must be the same used for the correspondent variable in the flux functions.

The third method to implement is the one that runs the model, solving the differential equation and returning the output flux.

```
    def get_output(self, solve=True):

        if solve:
            self._solver_states = [self._states[self._prefix_states + 'S0']]
            self._solve_differential_equation()

            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]
```

The method takes, as input, the parameter solve: if `False`, the state array of the reservoir will not be calculated again, potentially producing a different result, and the output will be computed based on the state that is already stored. This is the desired behavior in case of post-run inspection of the element.

Line 4 transforms the states dictionary in an ordered list, line 5 call the built-in solver of the differential equation, line 7 updates the state of the model to the one of the updated value, lines 9-14 call the external numerical solver to get the values of the fluxes (note that, for this operation, the python implementation of the fluxes is used always).

The last method(s) to implement is the one that defines the fluxes: in this case the methods are two, one for the python implementation and one for numba.

```
1       @staticmethod
2       def _fluxes_function_python(S, S0, ind, P, k):
3   
4           if ind is None:
5               return (
6                   [
7                       P,
8                       - k * S,
9                   ],
10                  0.0,
11                  S0 + P
12              )
13          else:
14              return (
15                  [
16                      P[ind],
17                      - k[ind] * S,
18                  ],
19                  0.0,
20                  S0 + P[ind]
21              )
22  
23      @staticmethod
24      @nb.jit('Tuple((UniTuple(f8, 2), f8, f8))(optional(f8), f8, i4, f8[:], f8[:])',
25              nopython=True)
26      def _fluxes_function_numba(S, S0, ind, P, k):
27  
28          return (
29              (
30                  P[ind],
31                  - k[ind] * S,
32              ),
33              0.0,
34              S0 + P[ind]
35          )
```

They are both private static methods. Their input consists of the state used to compute the fluxes (`S`), initial state (`S0`, used to define the maximum possible state for the reservoir), index to use in the arrays (`ind`, all inputs are arrays and, when solving for a single time step, the index indicates the time step to look for), input fluxes (`P`), and parameters (`k`). The output is a tuple containing three elements:

- tuple with the values of the fluxes calculated according to the state; positive sign for incoming fluxes, negative for outgoing;
- lower bound for the search of the state;
- upper bound for the search of the state;

The implementation for the numba solver differs in two aspects:

- the usage of the numba decorator that defines the types of the input variables (lines 24-25)

- the fact that the method works only for a single time step and not for the vectorized solution (use python method for that)

3.8.2 Half-triangular lag function

The half-triangular lag function is a function that has the shape of a right triangle, growing linearly until t_{lag} and then zero. The growth rate (α) is designed such as the total area of the triangle is one.

$$f_{\text{lag}} = \alpha t$$

for $t \leq t_{\text{lag}}$

$$f_{\text{lag}} = 0$$

for $t > t_{\text{lag}}$

SuperflexPy provides the class `LagElement` that contains most of the functionalities needed to solve a lag function. The class implementing a customized lag function will inherit from it and implement only the necessary methods that are needed to calculate the transformation that needs to be applied to the incoming flux.

```python
import numpy as np

class TriangularLag(LagElement):
```

The only method to implement is the private method used to calculate the `weight` array, that is used to distribute the incoming flux.

```python
    def _build_weight(self, lag_time):

        weight = []

        for t in lag_time:
            array_length = np.ceil(t)
            w_i = []

            for i in range(int(array_length)):
                w_i.append(self._calculate_lag_area(i + 1, t)
                           - self._calculate_lag_area(i, t))

            weight.append(np.array(w_i))

        return weight
```

that makes use of a secondary private static method

```python
    @staticmethod
    def _calculate_lag_area(bin, len):

        if bin <= 0:
            value = 0
        elif bin < len:
            value = (bin / len)**2
        else:
            value = 1

        return value
```

This method returns the value of the area of a triangle that is proportional to the lag function, with a smaller base `bin`. The `_build_weight` method, on the other hand, uses the output of this function to calculate the weight array.

Note that this choice of using a second static method to calculate the weight array, while being convenient, is specific to this particular case; other implementation of the `_build_weight` method are possible and welcome.

3.8.3 Parameterized splitter

A splitter is an element that takes the flux coming from one element upstream and divides it to feed multiple elements downstream. Usually, the behavior of such an element is controlled by some parameters that define the part of the flux that goes into a specific element.

While SuperflexPy can support infinite fluxes (e.g., for simulating transport processes) and an infinite number of downstream elements, the simplest case that we are presenting here has a single flux that gets split into two downstream elements. In this example, the system needs only one parameter (α_{split}) to be defined, with the downstream fluxes that are

$$Q_1^{\text{out}} = \alpha_{\text{split}} Q^{\text{in}}$$
$$Q_2^{\text{out}} = \left(1 - \alpha_{\text{split}}\right) Q^{\text{in}}$$

SuperflexPy provides the class `ParameterizedElement` that is designed for all the generic elements that are controlled by some parameters but do not have a state. The class implementing the parameterized splitter will inherit from this and implement only some methods.

```
from superflexpy.framework.element import ParameterizedElement

class ParameterizedSingleFluxSplitter(ParameterizedElement):
```

The first thing to define are two private attributes defining how many upstream and downstream elements the splitter has; this information is used by the unit when constructing the model structure.

```
    _num_downstream = 2
    _num_upstream = 1
```

After that we need to define the function that takes the inputs and the one that calculates the outputs of the splitter.

```
    def set_input(self, input):

        self.input = {'Q_in': input[0]}

    def get_output(self, solve=True):

        split_par = self._parameters[self._prefix_parameters + 'split-par']

        return [
            self.input['Q_in'] * split_par,
            self.input['Q_in'] * (1 - split_par)
        ]
```

The two methods have the same structure of the one implemented as part of the linear reservoir example. Note that, in this case, the argument `solve` of the `get_output` method is not used but it is still required to maintain the interface consistent.

Note: Last update 26/06/2020

3.9 Expand SuperflexPy: modify the existing components

3.9.1 Adding the routing to a node

The nodes in SuperflexPy are designed to provide the possibility of simulating the routing process that may happen in a catchment. The routing is a delay in the fluxes that comes from their propagation within the catchment (internal routing) and in the river network (external routing).

The default implementation of the node (`Node` class in `superflexpy.framework.node`) does not provide the routing functionality. Although the methods `_internal_routing` and `external_routing` exist and are integrate in the code, their implementation simply returns the incoming fluxes without any transformation.

The modeller that wants to implement the routing, therefore, has to implement a customized node that implements those two methods. The object-oriented design of SuperflexPy simplifies this operation since the new node class will inherit all the methods from the original class and has to overwrite only the two that are responsible of the routing.

We propose here an implementation of the routing that uses a lag function that has the shape of an isosceles triangle with base `t_internal` and `t_external`, for internal and external routing respectively. The implementation is similar to the case of the *Half-triangular lag function*.

The first step to do in the implementation is to import the `Node` component from SuperflexPy and implement the class `RoutedNode` with the following code

```python
from superflexpy.framework.node import Node

class RoutedNone(Node):
```

We then need to implement the methods `_internal_routing` and `external_routing`. Both the methods receive, as input, a list of fluxes and return, as output, the same list of fluxes with the delay applied.

```python
1      def _internal_routing(self, flux):
2
3          t_internal = self.get_parameters(names=[self._prefix_local_parameters + 't_
   ↪internal'])
4          flux_out = []
5
6          for f in flux:
7              flux_out.append(self._route(f, t_internal))
8
9          return flux_out
10
11     def external_routing(self, flux):
12
13         t_external = self.get_parameters(names=[self._prefix_local_parameters + 't_
   ↪external'])
14         flux_out = []
15
16         for f in flux:
17             flux_out.append(self._route(f, t_external))
18
19         return flux_out
```

Since, in this simple example, the two routing mechanisms are handled using the same lag function, the methods take advantage of the method `_route` (line 7 and 17) to handle the routing.

The method is implemented with the following code

```
    def _route(self, flux, time):

        state = np.zeros(int(np.ceil(time)))
        weight = self._calculate_weight(time)

        out = []
        for value in flux:
            state = state + weight * value
            out.append(state[0])
            state[0] = 0
            state = np.roll(state, shift=-1)

        return np.array(out)

    def _calculate_weight(self, time):

        weight = []

        array_length = np.ceil(time)

        for i in range(int(array_length)):
            weight.append(self._calculate_lag_area(i + 1, time)
                          - self._calculate_lag_area(i, time))

        return weight

    @staticmethod
    def _calculate_lag_area(portion, time):

        half_time = time / 2

        if portion <= 0:
            value = 0
        elif portion < half_time:
            value = 2 * (portion / time) ** 2
        elif portion < time:
            value = 1 - 2 * ((time - portion) / time)**2
        else:
            value = 1

        return value
```

Note that all the code in this block is highly similar to the one implemented in `RoutedNode` and that, for the implementation of the routing, the only two methods that are strictly necessary are `_internal_routing` and `external_routing` while all the others are only "support methods" to these two, needed only to make the code more organized and easier to maintain.

Note: Last update 01/09/2020

3.10 Application: implementation of existing conceptual models

In this page we propose the implementation of existing conceptual hydrological models. The translation of a model in SuperflexPy requires the following steps:

1. Design of a structure that reflects the original model but satisfy the requirements of SuperflexPy (e.g. does not contain mutual interaction between elements);
2. Extension of the framework, coding the required elements (as explained in the page *Expand SuperflexPy: build customized elements*)
3. Construction of the model structure using the elements implemented at point 2

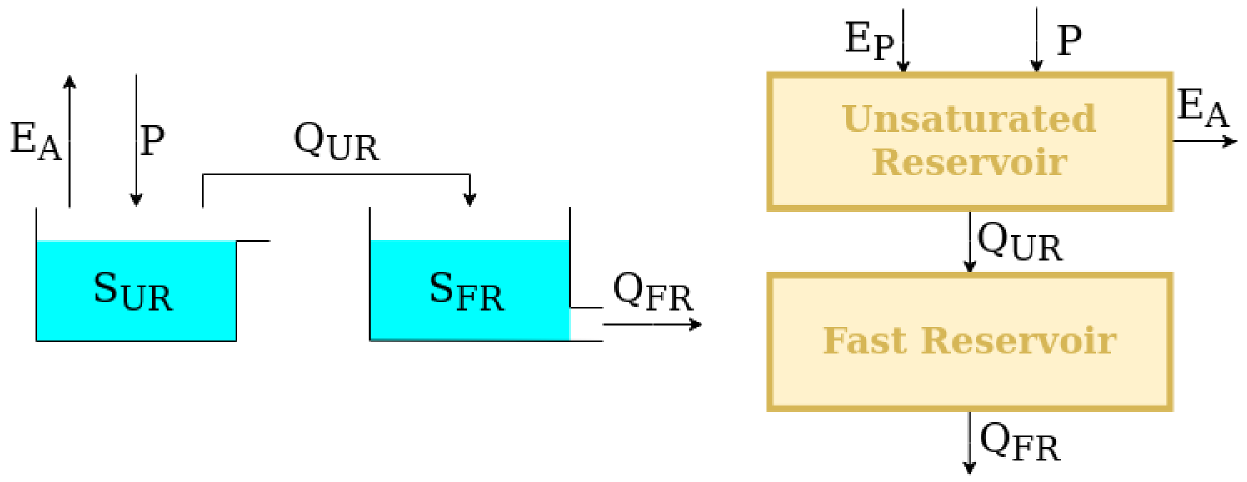
3.10.1 M4 from Kavetski and Fenicia, WRR, 2011

M4 is a simple conceptual model presented, as part of a models comparison study, in the article

> Kavetski, D., and F. Fenicia (2011), **Elements of a flexible approach for conceptual hydrological modeling: 2. Applicationand experimental insights**, WaterResour.Res.,47, W11511, doi:10.1029/2011WR010748.

Design of the structure

The structure of M4 is quite simple and can be implemented directly in SuperflexPy without the need of using connection elements. The figure shows, on the left, the structure as shown in the paper and, on the right, the SuperflexPy implementation. as part of a models comparison study.



The upstream element, the unsaturated reservoir (UR), is designed to represent runoff generation processes (e.g. separation between evaporation and runoff) and it is controlled by the differential equation

$$\frac{\mathrm{d}S_{\mathrm{UR}}}{\mathrm{d}t}=P-E_{\mathrm{P}}\left(\frac{\frac{S_{\mathrm{UR}}}{S_{\mathrm{max}}}\left(1+m\right)}{\frac{S_{\mathrm{UR}}}{S_{\mathrm{max}}}+m}\right)-P\left(\frac{S_{\mathrm{UR}}}{S_{\mathrm{max}}}\right)^{\beta}$$

The downstream element, the fast reservoir (FR), is designed to represent runoff propagation processes (e.g. routing) and it is controlled by the differential equation

$$\frac{\mathrm{d}S_{\mathrm{FR}}}{\mathrm{d}t}=P-kS_{\mathrm{FR}}^{\alpha}$$

S_{UR} and S_{FR} are the states of the reservoirs, P is the input flux, E_{P} is the potential evapotranspiration, and S_{max}, m, β, k, α are parameters of the model.

Elements creation

We now report the code used to implement the elements designed in the previous section. Instruction on how to use the framework to build new elements can be found in the page *Expand SuperflexPy: build customized elements*.

Note that for common applications the elements are already available (refer to the page *Elements list*) and, therefore, the modeller does not need to implement the code presented in this section.

Unsaturated reservoir

The element is a reservoir that can be implemented extending the `ODEsElement` class.

```python
class UnsaturatedReservoir(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._fluxes_function_python]

        if approximation.architecture == 'numba':
            self._fluxes = [self._fluxes_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._fluxes_function_python]

    def set_input(self, input):

        self.input = {'P': input[0],
                      'PET': input[1]}

    def get_output(self, solve=True):

        if solve:
            self._solver_states = [self._states[self._prefix_states + 'S0']]

            self._solve_differential_equation()

            # Update the state
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [-fluxes[0][2]]

    def get_AET(self):

        try:
```

(continues on next page)

(continued from previous page)

```
            S = self.state_array
        except AttributeError:
            message = '{}get_aet method has to be run after running '.format(self._
↪error_message)
            message += 'the model using the method get_output'
            raise AttributeError(message)

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=S,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]

    # PROTECTED METHODS

    @staticmethod
    def _fluxes_function_python(S, S0, ind, P, Smax, m, beta, PET):

        if ind is None:
            return (
                [
                    P,
                    - PET * ((S / Smax) * (1 + m)) / ((S / Smax) + m),
                    - P * (S / Smax)**beta,
                ],
                0.0,
                S0 + P
            )
        else:
            return (
                [
                    P[ind],
                    - PET[ind] * ((S / Smax[ind]) * (1 + m[ind])) / ((S / Smax[ind])␣
↪+ m[ind]),
                    - P[ind] * (S / Smax[ind])**beta[ind],
                ],
                0.0,
                S0 + P[ind]
            )

    @staticmethod
    @nb.jit('Tuple((UniTuple(f8, 3), f8, f8))(optional(f8), f8, i4, f8[:], f8[:],␣
↪f8[:], f8[:], f8[:])',
            nopython=True)
    def _fluxes_function_numba(S, S0, ind, P, Smax, m, beta, PET):

        return (
            (
                P[ind],
                PET[ind] * ((S / Smax[ind]) * (1 + m[ind])) / ((S / Smax[ind]) +␣
↪m[ind]),
                - P[ind] * (S / Smax[ind])**beta[ind],
            ),
```

(continues on next page)

(continued from previous page)

```
            0.0,
            S0 + P[ind]
        )
```

Fast reservoir

The element is a reservoir that can be implemented extending the `ODEsElement` class.

```
class FastReservoir(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._fluxes_function_python]  # Used by get fluxes,␣
↪regardless of the architecture

        if approximation.architecture == 'numba':
            self._fluxes = [self._fluxes_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._fluxes_function_python]

    # METHODS FOR THE USER

    def set_input(self, input):

        self.input = {'P': input[0]}

    def get_output(self, solve=True):

        if solve:
            self._solver_states = [self._states[self._prefix_states + 'S0']]
            self._solve_differential_equation()

            # Update the state
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,  # I can use␣
↪the python method since it is fast
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]

    # PROTECTED METHODS

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    def _fluxes_function_python(S, S0, ind, P, k, alpha):

        if ind is None:
            return (
                [
                    P,
                    - k * S**alpha,
                ],
                0.0,
                S0 + P
            )
        else:
            return (
                [
                    P[ind],
                    - k[ind] * S**alpha[ind],
                ],
                0.0,
                S0 + P[ind]
            )

    @staticmethod
    @nb.jit('Tuple((UniTuple(f8, 2), f8, f8))(optional(f8), f8, i4, f8[:], f8[:],␣
↪f8[:])',
            nopython=True)
    def _fluxes_function_numba(S, S0, ind, P, k, alpha):

        return (
            (
                P[ind],
                - k[ind] * S**alpha[ind],
            ),
            0.0,
            S0 + P[ind]
        )
```

Model initialization

Now that all the elements needed have been implemented, we can put them together to build the model structure. For details refer to *Quick demo*.

The first step consists in initializing all the elements.

```
root_finder = PegasusPython()
numeric_approximator = ImplicitEulerPython(root_finder=root_finder)

ur = UnsaturatedReservoir(
    parameters={'Smax': 50.0, 'Ce': 1.0, 'm': 0.01, 'beta': 2.0},
    states={'S0': 25.0},
    approximation=numeric_approximator,
    id='UR'
)

fr = FastReservoir(
    parameters={'k': 0.1, 'alpha': 1.0},
```

(continues on next page)

(continued from previous page)

```
    states={'S0': 10.0},
    approximation=numeric_approximator,
    id='FR'
)
```

After that, the elements can be put together to create a `Unit` that reflects the structure presented in the figure.

```
model = Unit(
    layers=[
        [ur],
        [fr]
    ],
    id='M4'
)
```

3.10.2 GR4J

GR4J is a conceptual hydrological model originally introduced in the article

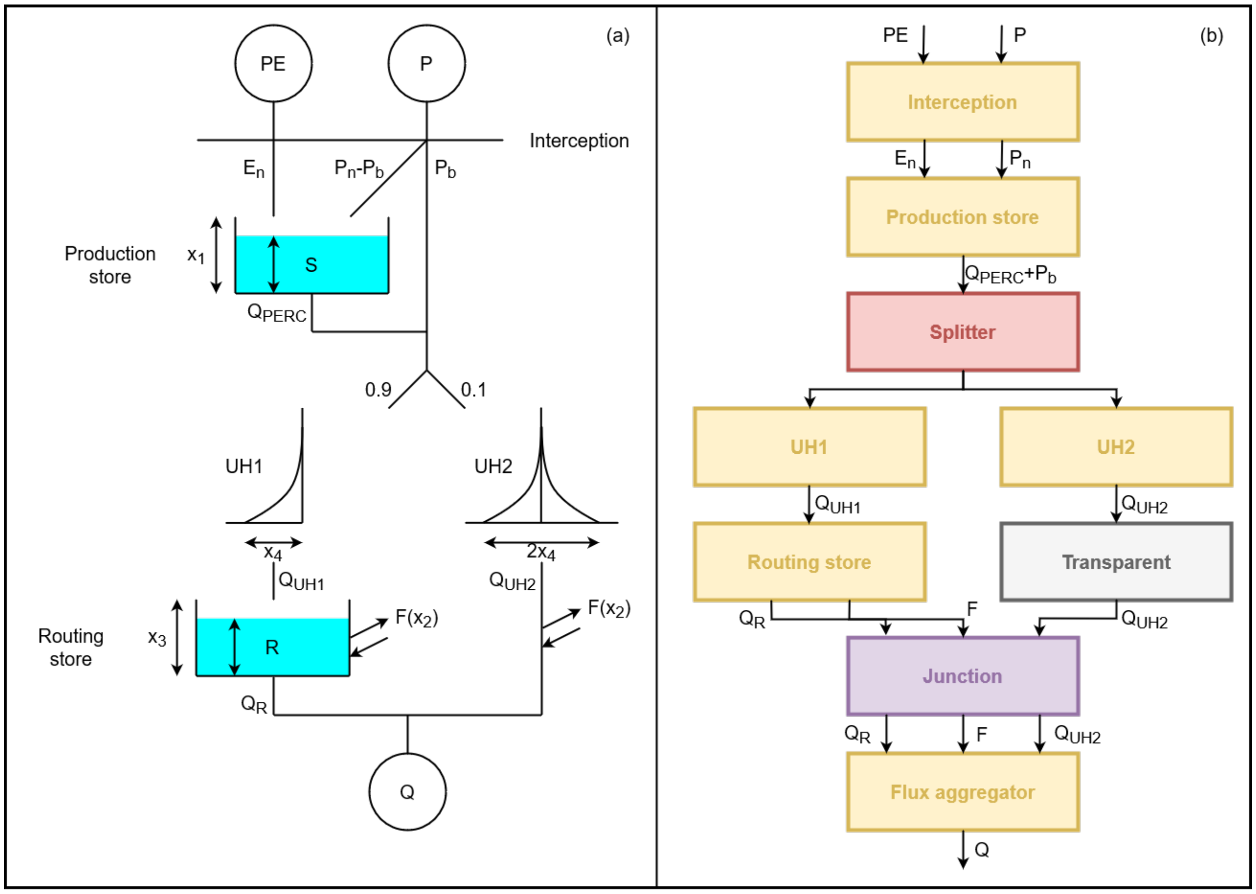
> Perrin, C., Michel, C., and Andréassian, V.: **Improvement of a parsimonious model for streamflow simulation**, Journal of Hydrology, 279, 275-289, https://doi.org/10.1016/S0022-1694(03)00225-7, 2003.

The solution adopted here follows the "continuous state-space representation" presented in

> Santos, L., Thirel, G., and Perrin, C.: **Continuous state-space representation of a bucket-type rainfall-runoff model: a case study with the GR4 model using state-space GR4 (version 1.0)**, Geosci. Model Dev., 11, 1591-1605, 10.5194/gmd-11-1591-2018, 2018.

Design of the structure

The figure shows, on the left, the model structure as proposed in Perrin et al., 2003 and, on the right, the adaptation to SuperflexPy



In the original version, the potential evaporation and the precipitation are "filtered" by an interception layer that sets the smallest of the two fluxes to zero and the biggest to the difference between the two.

$$
\begin{aligned}
&\text{if } P > PE:\\
&\quad P_{\text{NET}} = P - PE\\
&\quad E_{\text{NET}} = 0\\
&\text{else}:\\
&\quad P_{\text{NET}} = 0\\
&\quad E_{\text{NET}} = PE - P
\end{aligned}
$$

This element is reproduced identically in the SuperflexPy implementation by the "interception filter"

In the original model, the precipitation is split between a portion P_s that flows into the production store and the remaining part P_b that bypasses the reservoir. Since P_s and P_b are function of the state of the reservoir

$$
\begin{aligned}
P_s &= P_{\text{NET}}\left(1-\left(\frac{S_{\text{UR}}}{x_1}\right)^{\alpha}\right)\\
P_b &= P_{\text{NET}}\left(\frac{S_{\text{UR}}}{x_1}\right)^{\alpha}
\end{aligned}
$$

when we implement this part of the model in SuperflexPy, these two fluxes cannot be calculated before solving the reservoir. For this reason, in the SuperflexPy implementation of GR4J, all the precipitation flows into an element that incorporates the production store; this element takes care of dividing the precipitation internally, while solving the differential equation

$$
\frac{\mathrm{d}S_{\text{UR}}}{\mathrm{d}t} = P_{\text{NET}}\left(1-\left(\frac{S_{\text{UR}}}{x_1}\right)^{\alpha}\right) - E_{\text{NET}}\left(2\frac{S_{\text{UR}}}{x_1}-\left(\frac{S_{\text{UR}}}{x_1}\right)^{\alpha}\right) - \frac{x_1^{1-\beta}}{(\beta-1)}\nu^{\beta-1}S_{\text{UR}}^{\beta}
$$

where the first term is the precipitation P_s, the second term is the actual evaporation, and the third term represent the output of the reservoir, called "percolation".

Once the reservoir is solved (i.e. the values of S_{UR} that solve the differential equation are found), the element outputs the sum of percolation and bypassing precipitation P_b

After this, the flux is divided between two lag functions (called "unit hydrograph", abbreviated UH): 90% of the flux goes to UH1 while the 10% goes to UH2. In this part of the structure there is a strict correspondence between the elements of GR4J and their SuperflexPy implementation.

The output of UH1 represents the input of the routing store, a reservoir that is controlled by the differential equation

$$\frac{\mathrm{d}S_{\mathrm{RR}}}{\mathrm{d}t} = Q_{\mathrm{UH1}} - \frac{x_3^{1-\gamma}}{(\gamma-1)}S_{\mathrm{RR}}^{\gamma} - \frac{x_2}{x_3^{\omega}}S_{\mathrm{RR}}^{\omega}$$

where the second term is the output of the reservoir and the last is a gain/loss term (called Q_{RF}).

The gain/loss term Q_{RF}, which is a function of the state S_{RR} of the reservoir, is subtracted also to the output of UH2: in SuperflexPy, this operation cannot be done together with the solution of the routing store but it is done afterwards. For this reason, the SuperflexPy implementation of GR4J has an additional element (called "flux aggregator") that collects (through a junction element) the output of the routing store, the gain/loss term, and the output of UH2 and computes the outflow of the model using the equation

$$Q = Q_{\mathrm{RR}} + \max(0; Q_{\mathrm{UH2}} - Q_{\mathrm{RF}})$$

Elements creation

We now report the code used to implement the elements designed in the previous section. Instruction on how to use the framework to build new elements can be found in the page *Expand SuperflexPy: build customized elements*.

Note that for common applications the elements are already available (refer to the page *Elements list*) and, therefore, the modeller does not need to implement the code presented in this section.

Interception

The interception filter can be implemented extending the class `BaseElement`

```python
class InterceptionFilter(BaseElement):

    _num_upstream = 1
    _num_downstream = 1

    def set_input(self, input):

        self.input = {}
        self.input['PET'] = input[0]
        self.input['P'] = input[1]

    def get_output(self, solve=True):

        remove = np.minimum(self.input['PET'], self.input['P'])

        return [self.input['PET'] - remove, self.input['P'] - remove]
```

Production store

The production store is controlled by a differential equation and, therefore, it can be constructed extending the class `ODEsElement`

```python
class ProductionStore(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._flux_function_python]

        if approximation.architecture == 'numba':
            self._fluxes = [self._flux_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._flux_function_python]

    def set_input(self, input):

        self.input = {}
        self.input['PET'] = input[0]
        self.input['P'] = input[1]

    def get_output(self, solve=True):

        if solve:
            # Solve the differential equation
            self._solver_states = [self._states[self._prefix_states + 'S0']]
            self._solve_differential_equation()

            # Update the states
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          dt=self._dt,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        Pn_minus_Ps = self.input['P'] - fluxes[0][0]
        Perc = - fluxes[0][2]
        return [Pn_minus_Ps + Perc]

    def get_aet(self):

        try:
            S = self.state_array
        except AttributeError:
            message = '{}get_aet method has to be run after running '.format(self._
↪error_message)
```

(continues on next page)

(continued from previous page)

```python
            message += 'the model using the method get_output'
            raise AttributeError(message)

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=S,
                                          S0=self._solver_states,
                                          dt=self._dt,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]

    @staticmethod
    def _flux_function_python(S, S0, ind, P, x1, alpha, beta, ni, PET, dt):

        if ind is None:
            return(
                [
                    P * (1 - (S / x1)**alpha),  # Ps
                    - PET * (2 * (S / x1) - (S / x1)**alpha),  # Evaporation
                    - ((x1**(1 - beta)) / ((beta - 1) * dt)) * (ni**(beta - 1)) * (S**beta)  # Perc
                ],
                0.0,
                S0 + P * (1 - (S / x1)**alpha)
            )
        else:
            return(
                [
                    P[ind] * (1 - (S / x1[ind])**alpha[ind]),  # Ps
                    - PET[ind] * (2 * (S / x1[ind]) - (S / x1[ind])**alpha[ind]),  # Evaporation
                    - ((x1[ind]**(1 - beta[ind])) / ((beta[ind] - 1) * dt[ind])) * (ni[ind]**(beta[ind] - 1)) * (S**beta[ind])  # Perc
                ],
                0.0,
                S0 + P[ind] * (1 - (S / x1[ind])**alpha[ind])
            )

    @staticmethod
    @nb.jit('Tuple((UniTuple(f8, 3), f8, f8))(optional(f8), f8, i4, f8[:], f8[:], f8[:], f8[:], f8[:], f8[:], f8[:])',
            nopython=True)
    def _flux_function_numba(S, S0, ind, P, x1, alpha, beta, ni, PET, dt):

        return(
            (
                P[ind] * (1 - (S / x1[ind])**alpha[ind]),  # Ps
                - PET[ind] * (2 * (S / x1[ind]) - (S / x1[ind])**alpha[ind]),  # Evaporation
                - ((x1[ind]**(1 - beta[ind])) / ((beta[ind] - 1) * dt[ind])) * (ni[ind]**(beta[ind] - 1)) * (S**beta[ind])  # Perc
            ),
            0.0,
            S0 + P[ind] * (1 - (S / x1[ind])**alpha[ind])
```

(continues on next page)

(continued from previous page)

```python
        )
```

Unit hydrographs

The unit hydrographs are an extension of the `LagElement` and they can be implemented with the following code

```python
class UnitHydrograph1(LagElement):

    def __init__(self, parameters, states, id):

        LagElement.__init__(self, parameters, states, id)

    def _build_weight(self, lag_time):

        weight = []

        for t in lag_time:
            array_length = np.ceil(t)
            w_i = []
            for i in range(int(array_length)):
                w_i.append(self._calculate_lag_area(i + 1, t)
                           - self._calculate_lag_area(i, t))
            weight.append(np.array(w_i))

        return weight

    @staticmethod
    def _calculate_lag_area(bin, len):
        if bin <= 0:
            value = 0
        elif bin < len:
            value = (bin / len)**2.5
        else:
            value = 1
        return value
```

```python
class UnitHydrograph2(LagElement):

    def __init__(self, parameters, states, id):

        LagElement.__init__(self, parameters, states, id)

    def _build_weight(self, lag_time):

        weight = []

        for t in lag_time:
            array_length = np.ceil(t)
            w_i = []
            for i in range(int(array_length)):
                w_i.append(self._calculate_lag_area(i + 1, t)
                           - self._calculate_lag_area(i, t))
            weight.append(np.array(w_i))

```

(continues on next page)

(continued from previous page)

```
        return weight

    @staticmethod
    def _calculate_lag_area(bin, len):
        half_len = len / 2
        if bin <= 0:
            value = 0
        elif bin < half_len:
            value = 0.5 * (bin / half_len)**2.5
        elif bin < len:
            value = 1 - 0.5 * (2 - bin / half_len)**2.5
        else:
            value = 1
        return value
```

Routing store

The routing store is an `ODEsElement`

```
class RoutingStore(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._flux_function_python]

        if approximation.architecture == 'numba':
            self._fluxes = [self._flux_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._flux_function_python]

    def set_input(self, input):

        self.input = {}
        self.input['P'] = input[0]

    def get_output(self, solve=True):

        if solve:
            # Solve the differential equation
            self._solver_states = [self._states[self._prefix_states + 'S0']]
            self._solve_differential_equation()

            # Update the states
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          dt=self._dt,
```

(continues on next page)

(continued from previous page)

```
                                                **self.input,
                                                **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                                )

        Qr = - fluxes[0][1]
        F = -fluxes[0][2]

        return [Qr, F]

    @staticmethod
    def _flux_function_python(S, S0, ind, P, x2, x3, gamma, omega, dt):

        if ind is None:
            return(
                [
                    P,  # P
                    - ((x3**(1 - gamma)) / ((gamma - 1) * dt)) * (S**gamma),  # Qr
                    - (x2 * (S / x3)**omega),  # F
                ],
                0.0,
                S0 + P
            )
        else:
            return(
                [
                    P[ind],  # P
                    - ((x3[ind]**(1 - gamma[ind])) / ((gamma[ind] - 1) * dt[ind])) *␣
↪(S**gamma[ind]),  # Qr
                    - (x2[ind] * (S / x3[ind])**omega[ind]),  # F
                ],
                0.0,
                S0 + P[ind]
            )

    @staticmethod
    @nb.jit('Tuple((UniTuple(f8, 3), f8, f8))(optional(f8), f8, i4, f8[:], f8[:],␣
↪f8[:], f8[:], f8[:], f8[:])',
            nopython=True)
    def _flux_function_numba(S, S0, ind, P, x2, x3, gamma, omega, dt):

        return(
            (
                P[ind],  # P
                - ((x3[ind]**(1 - gamma[ind])) / ((gamma[ind] - 1) * dt[ind])) *␣
↪(S**gamma[ind]),  # Qr
                - (x2[ind] * (S / x3[ind])**omega[ind]),  # F
            ),
            0.0,
            S0 + P[ind]
        )
```

Flux aggregator

The flux aggregator can be implemented extending a `BaseElement`

```python
class FluxAggregator(BaseElement):

    _num_downstream = 1
    _num_upstream = 1

    def set_input(self, input):

        self.input = {}
        self.input['Qr'] = input[0]
        self.input['F'] = input[1]
        self.input['Q2_out'] = input[2]

    def get_output(self, solve=True):

        return [self.input['Qr']
                + np.maximum(0, self.input['Q2_out'] - self.input['F'])]
```

Model initialization

Now that all the elements needed have been implemented, we can put them together to build the model structure. For details refer to *Quick demo*.

The first step consists in initializing all the elements.

```python
x1, x2, x3, x4 = (50.0, 0.1, 20.0, 3.5)

root_finder = PegasusPython()  # Use the default parameters
numerical_approximation = ImplicitEulerPython(root_finder)

interception_filter = InterceptionFilter(id='ir')

production_store = ProductionStore(parameters={'x1': x1, 'alpha': 2.0,
                                               'beta': 5.0, 'ni': 4/9},
                                   states={'S0': 10.0},
                                   approximation=numerical_approximation,
                                   id='ps')

splitter = Splitter(weight=[[0.9], [0.1]],
                    direction=[[0], [0]],
                    id='spl')

unit_hydrograph_1 = UnitHydrograph1(parameters={'lag-time': x4},
                                    states={'lag': None},
                                    id='uh1')

unit_hydrograph_2 = UnitHydrograph2(parameters={'lag-time': 2*x4},
                                    states={'lag': None},
                                    id='uh2')

routing_store = RoutingStore(parameters={'x2': x2, 'x3': x3,
                                         'gamma': 5.0, 'omega': 3.5},
                             states={'S0': 10.0},
                             approximation=numerical_approximation,
                             id='rs')

transparent = Transparent(id='tr')
```

(continues on next page)

(continued from previous page)

```python

junction = Junction(direction=[[0, None],   # First output
                               [1, None],   # Second output
                               [None, 0]],  # Third output
                    id='jun')

flux_aggregator = FluxAggregator(id='fa')
```

After that, the elements can be put together to create a `Unit` that reflects the structure presented in the figure.

```python
model = Unit(layers=[[interception_filter],
                     [production_store],
                     [splitter],
                     [unit_hydrograph_1, unit_hydrograph_2],
                     [routing_store, transparent],
                     [junction],
                     [flux_aggregator]],
             id='model')
```

3.10.3 HYMOD

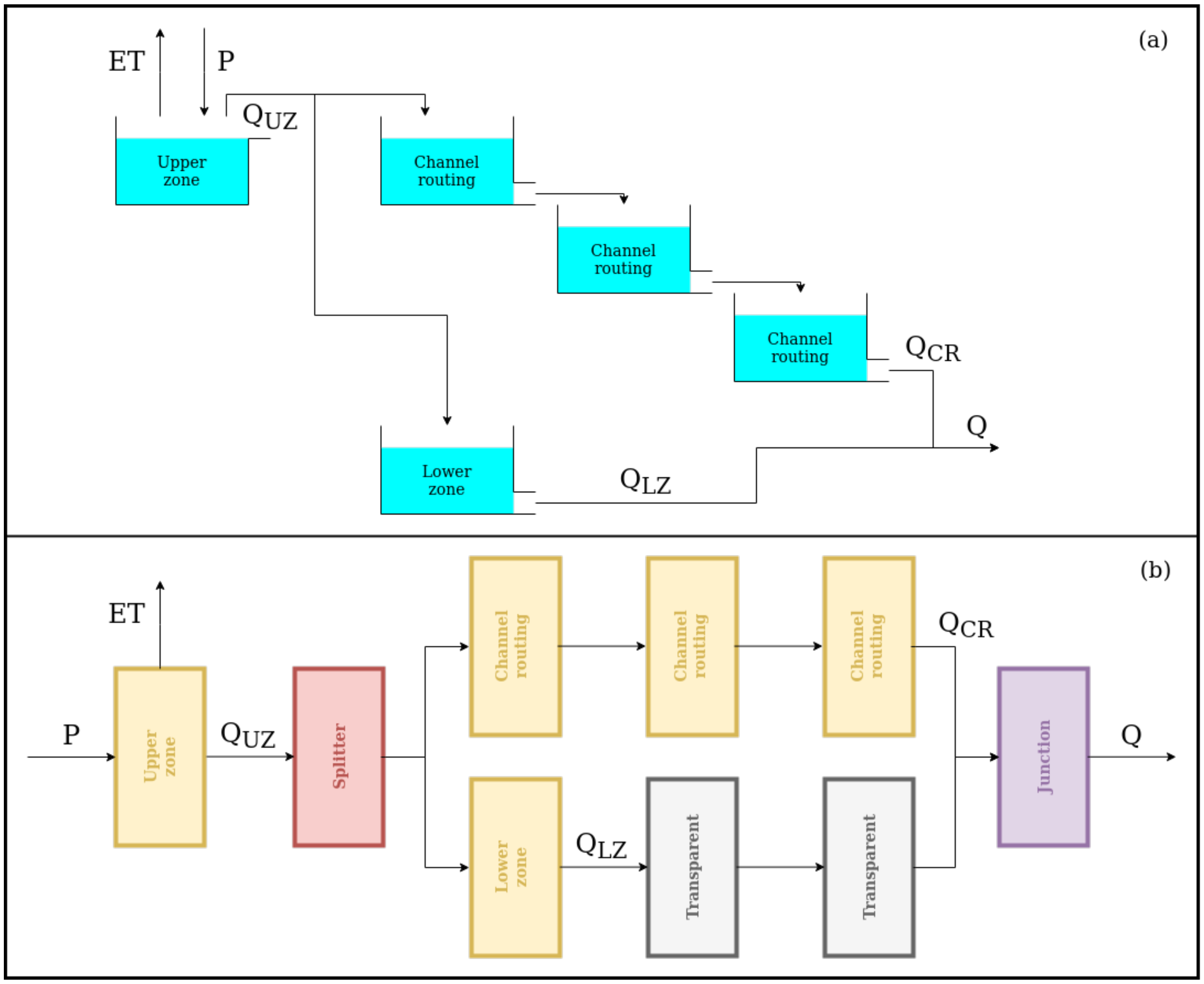
HYMOD is a conceptual hydrological model presented in the Ph.D. thesis

> Boyle, D. P. (2001). **Multicriteria calibration of hydrologic models**, The University of Arizona. Link

The solution proposed here follows the model structure presented in

> Wagener, T., Boyle, D. P., Lees, M. J., Wheater, H. S., Gupta, H. V., and Sorooshian, S.: **A framework for development and application of hydrological models**, Hydrol. Earth Syst. Sci., 5, 13–26, https://doi.org/10.5194/hess-5-13-2001, 2001.

Design of the structure



The structure of HYMOD is composed by three blocks of reservoirs that are designed to represent three mechanism that control the streamflow at the catchment scale: upper zone (soil dynamics), channel routing (surface runoff), and lower zone (subsurface flow).

As it can be seen in the figure, the original structure of HYMOD already satisfy the design constrains of SuperflexPy and therefore it can be implemented simply translating the single elements individually, without the need of workarounds, as done in the case of GR4J.

The first element (upper zone) is a reservoirs intended to represent soil dynamics that simulates streamflow generation processes and evaporation. It is controlled by the differential equation

$$\frac{\mathrm{d}S_{\mathrm{UR}}}{\mathrm{d}t} = P - E - P\left(1 - \left(1 - \frac{S_{\mathrm{UR}}}{S_{\mathrm{max}}}\right)^{\beta}\right)$$

where the first term represent the precipitation input, the second the actual evaporation (which is equal to the potential evaporation as long as there is sufficient storage in the reservoir) and the third the outflow of the reservoir.

The outflow of the reservoir is then split between the channel routing and the lower zone; these are all represented by some linear reservoirs that are controlled by the differential equation

$$\frac{\mathrm{d}S}{\mathrm{d}t} = P - kS$$

where the first term is the input of the reservoir (the outflow of the upstream element, in this case) and the second term represent the outflow of the reservoir.

Channel routing and lower zone differentiate one from the other by two factors: the number of reservoirs used (3 in the first case and 1 in the second) and by the value of the parameter k, which controls the outflow rate, that is bigger for the channel routing.

The output of these two branches of the model are collected together by a junction, which sums them to generate the output of the model.

Comparing the two panels in the figure, the only difference is due to the presence of the two transparent element that are needed to fill the gaps that, otherwise, will be present in the structure.

Elements creation

We now report the code used to implement the elements designed in the previous section. Instruction on how to use the framework to build new elements can be found in the page *Expand SuperflexPy: build customized elements*.

Note that for common applications the elements are already available (refer to the page *Elements list*) and, therefore, the modeller does not need to implement the code presented in this section.

Upper zone

The code used to simulate the upper zone present a change in the equation used to calculate the actual evaporation. In the original version (Wagener et al., 2001) the equation is "described" in the text

> *The actual evapotranspiration is equal to the potential value if sufficient soil moisture is available; otherwise it is equal to the available soil moisture content.*

which translates to the equation

$$\begin{aligned}
&\text{if } S > 0: \\
&\quad E = PE \\
&\text{else}: \\
&\quad E = 0
\end{aligned}$$

Unfortunately this solution is not smooth and this can cause some computational problems. For this reason, it has been decided to use the following equation to calculate the potential evaporation

$$E = PE\left(\frac{\frac{S_{\text{UR}}}{S_{\text{max}}}(1+\theta)}{\frac{S_{\text{UR}}}{S_{\text{max}}}+\theta}\right)$$

The upper zone reservoir can be implemented extending the `ODEsElement` class.

```python
class UpperZone(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._fluxes_function_python]
```

(continues on next page)

(continued from previous page)

```

        if approximation.architecture == 'numba':
            self._fluxes = [self._fluxes_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._fluxes_function_python]

    def set_input(self, input):

        self.input = {'P': input[0],
                      'PET': input[1]}

    def get_output(self, solve=True):

        if solve:
            self._solver_states = [self._states[self._prefix_states + 'S0']]

            self._solve_differential_equation()

            # Update the state
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [-fluxes[0][2]]

    def get_AET(self):

        try:
            S = self.state_array
        except AttributeError:
            message = '{}get_aet method has to be run after running '.format(self._
↪error_message)
            message += 'the model using the method get_output'
            raise AttributeError(message)

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
                                          S=S,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]

    # PROTECTED METHODS

    @staticmethod
    def _fluxes_function_python(S, S0, ind, P, Smax, m, beta, PET):

        if ind is None:
```

(continues on next page)

(continued from previous page)

```
            return (
                [
                    P,
                    - PET * ((S / Smax) * (1 + m)) / ((S / Smax) + m),
                    - P * (1 - (1 - (S / Smax))**beta),
                ],
                0.0,
                S0 + P
            )
        else:
            return (
                [
                    P[ind],
                    - PET[ind] * ((S / Smax[ind]) * (1 + m[ind])) / ((S / Smax[ind])␣
↪+ m[ind]),
                    - P[ind] * (1 - (1 - (S / Smax[ind]))**beta[ind]),
                ],
                0.0,
                S0 + P[ind]
            )

    @staticmethod
    @nb.jit('Tuple((UniTuple(f8, 3), f8, f8))(optional(f8), f8, i4, f8[:], f8[:],␣
↪f8[:], f8[:], f8[:])',
            nopython=True)
    def _fluxes_function_numba(S, S0, ind, P, Smax, m, beta, PET):

        return (
            (
                P[ind],
                - PET[ind] * ((S / Smax[ind]) * (1 + m[ind])) / ((S / Smax[ind]) +␣
↪m[ind]),
                - P[ind] * (1 - (1 - (S / Smax[ind]))**beta[ind]),
            ),
            0.0,
            S0 + P[ind]
        )
```

Channel routing and lower zone

All the elements belonging to the channel routing and to the lower zone are reservoirs that can be implemented extending the `ODEsElement` class.

```
class LinearReservoir(ODEsElement):

    def __init__(self, parameters, states, approximation, id):

        ODEsElement.__init__(self,
                             parameters=parameters,
                             states=states,
                             approximation=approximation,
                             id=id)

        self._fluxes_python = [self._fluxes_function_python]  # Used by get fluxes,␣
↪regardless of the architecture
```

(continues on next page)

(continued from previous page)

```

        if approximation.architecture == 'numba':
            self._fluxes = [self._fluxes_function_numba]
        elif approximation.architecture == 'python':
            self._fluxes = [self._fluxes_function_python]

    # METHODS FOR THE USER

    def set_input(self, input):

        self.input = {'P': input[0]}

    def get_output(self, solve=True):

        if solve:
            self._solver_states = [self._states[self._prefix_states + 'S0']]
            self._solve_differential_equation()

            # Update the state
            self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})

        fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,  # I can use␣
↪the python method since it is fast
                                          S=self.state_array,
                                          S0=self._solver_states,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )

        return [- fluxes[0][1]]

    # PROTECTED METHODS

    @staticmethod
    def _fluxes_function_python(S, S0, ind, P, k):

        if ind is None:
            return (
                [
                    P,
                    - k * S,
                ],
                0.0,
                S0 + P
            )
        else:
            return (
                [
                    P[ind],
                    - k[ind] * S,
                ],
                0.0,
                S0 + P[ind]
            )

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    @nb.jit('Tuple((UniTuple(f8, 2), f8, f8))(optional(f8), f8, i4, f8[:], f8[:])',
            nopython=True)
    def _fluxes_function_numba(S, S0, ind, P, k):

        return (
            (
                P[ind],
                - k[ind] * S,
            ),
            0.0,
            S0 + P[ind]
        )
```

Model initialization

Now that all the elements needed have been implemented, we can put them together to build the model structure. For details refer to *Quick demo*.

The first step consists in initializing all the elements.

```
root_finder = PegasusPython()  # Use the default parameters
numerical_approximation = ImplicitEulerPython(root_finder)

upper_zone = UpperZone(parameters={'Smax': 50.0, 'm': 0.01, 'beta': 2.0},
                       states={'S0': 10.0},
                       approximation=numerical_approximation,
                       id='uz')

splitter = Splitter(weight=[[0.6], [0.4]],
                    direction=[[0], [0]],
                    id='spl')

channel_routing_1 = LinearReservoir(parameters={'k': 0.1},
                                    states={'S0': 10.0},
                                    approximation=numerical_approximation,
                                    id='cr1')

channel_routing_2 = LinearReservoir(parameters={'k': 0.1},
                                    states={'S0': 10.0},
                                    approximation=numerical_approximation,
                                    id='cr2')

channel_routing_3 = LinearReservoir(parameters={'k': 0.1},
                                    states={'S0': 10.0},
                                    approximation=numerical_approximation,
                                    id='cr3')

lower_zone = LinearReservoir(parameters={'k': 0.1},
                             states={'S0': 10.0},
                             approximation=numerical_approximation,
                             id='lz')

transparent_1 = Transparent(id='tr1')

transparent_2 = Transparent(id='tr2')
```

(continues on next page)

(continued from previous page)

```python

junction = Junction(direction=[[0, 0]],  # First output
                    id='jun')
```

After that, the elements can be put together to create a `Unit` that reflects the structure presented in the figure.

```python
model = Unit(layers=[[upper_zone],
                     [splitter],
                     [channel_routing_1, lower_zone],
                     [channel_routing_2, transparent_1],
                     [channel_routing_3, transparent_2],
                     [junction]],
             id='model')
```

Note: Last update 02/09/2020

3.11 Case studies

In this page we propose the model configurations used in publications.

3.11.1 Dal Molin et al., 2020, HESS

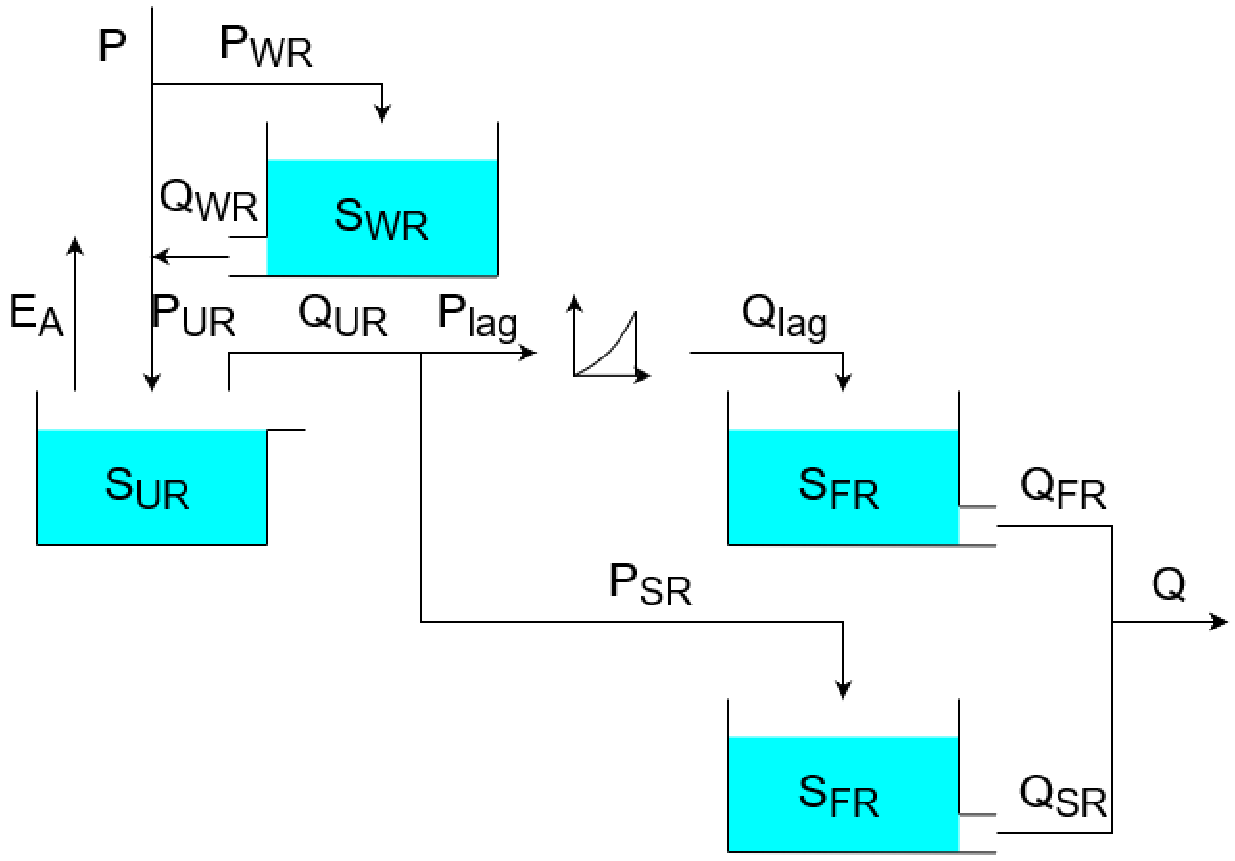
This section contains instructions for the implementation of the semi-distributed hydrological model M02 presented in the article:

Dal Molin, M., Schirmer, M., Zappa, M., and Fenicia, F.: **Understanding dominant controls on streamflow spatial variability to set up a semi-distributed hydrological model: the case study of the Thur catchment**, Hydrol. Earth Syst. Sci., 24, 1319–1345, https://doi.org/10.5194/hess-24-1319-2020, 2020.

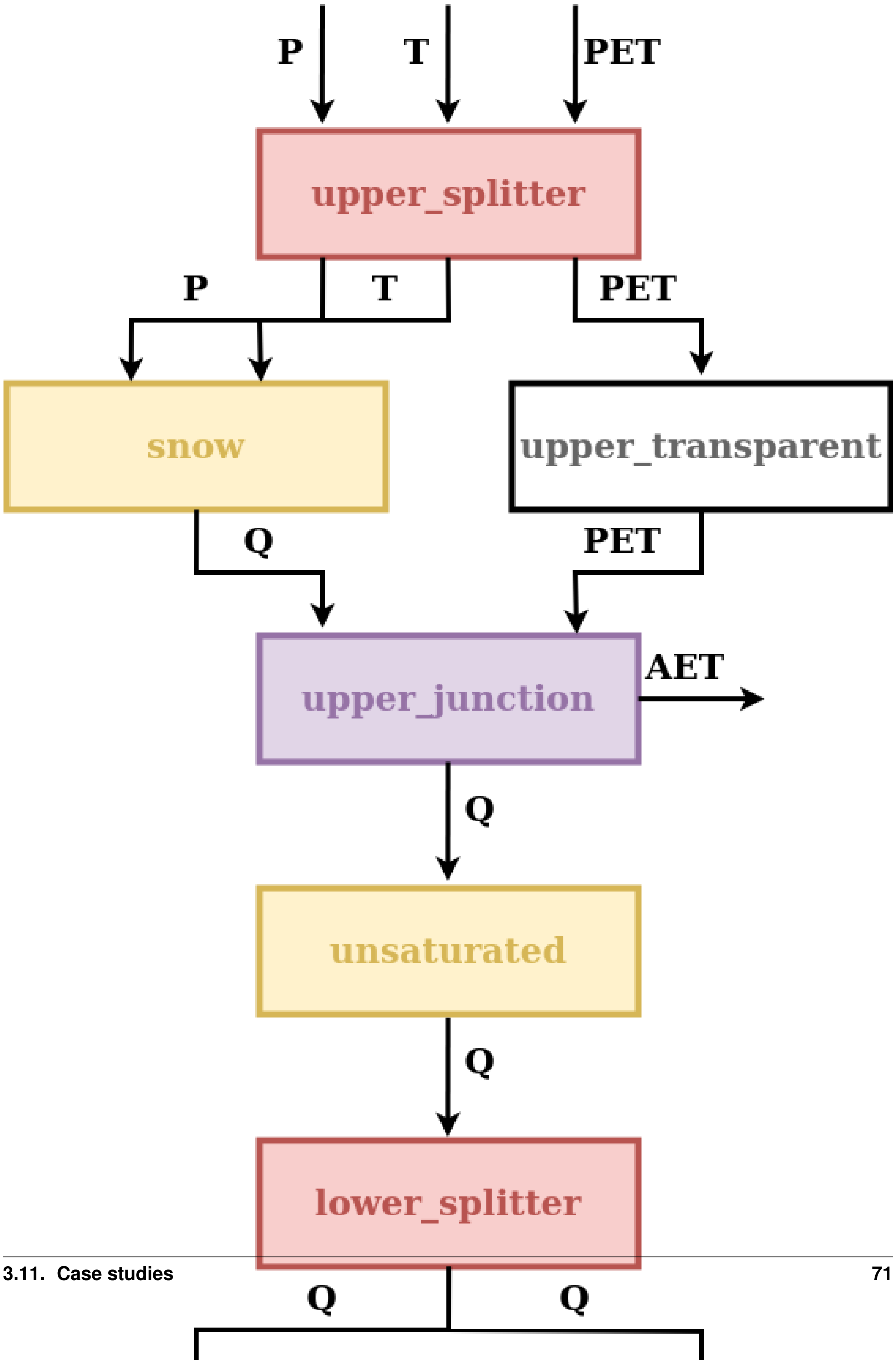
In this application, the Thur catchment has been divided in 10 subcatchments and 2 hydrological response units (HRUs). Please refer to the article for the details; here we propose only the code needed to setup a model that corresponds to the one used in the publication.

Model structure

The two HRUs are represented using the same model structure represented in the figure.



The structure is similar to the one of *HYMOD*; its conversion to SuperflexPy is presented here


P
T
PET
upper_splitter
P
T
PET
snow
upper_transparent
Q
PET
upper_junction
AET
Q
unsaturated
Q
lower_splitter
Q
Q

Note that also the temperature has been threated as a flux: this choice is not forced by the framework but, in this particular case, where it is the first element that needs it, this is particularly convenient. An alternative solution would have been designing the snow reservoir in such a way that the temperature becomes a state of the reservoir; this solution would have been preferable in the case where the element that needed the flux was not at the beginning of the structure.

Defining the elements

We here assume that all the elements are already existing; therefore they just need to be imported.

```
from superflexpy.implementation.elements.thur_model_hess import SnowReservoir, UnsaturatedReservoir, HalfTriangularLag, FastReservoir
from superflexpy.implementation.elements.structure_elements import Transparent, Junction, Splitter
from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
```

After this, all the elements must be initialized, defining the initial state and the parameters.

```
solver = PegasusPython()
approximator = ImplicitEulerPython(root_finder=solver)

upper_splitter = Splitter(
    direction=[
        [0, 1, None],     # P and T go to the snow reservoir
        [2, None, None]   # PET goes to the transparent element
    ],
    weight=[
        [1.0, 1.0, 0.0],
        [0.0, 0.0, 1.0]
    ],
    id='upper-splitter'
)

snow = SnowReservoir(
    parameters={'t0': 0.0, 'k': 0.01, 'm': 2.0},
    states={'S0': 0.0},
    approximation=approximator,
    id='snow'
)

upper_transparent = Transparent(
    id='upper-transparent'
)

upper_junction = Junction(
    direction=[
        [0, None],
        [None, 0]
    ],
    id='upper-junction'
)

unsaturated = UnsaturatedReservoir(
    parameters={'Smax': 50.0, 'Ce': 1.0, 'm': 0.01, 'beta': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
```

(continues on next page)

(continued from previous page)

```
    id='unsaturated'
)

lower_splitter = Splitter(
    direction=[
        [0],
        [0]
    ],
    weight=[
        [0.3],    # Portion to slow reservoir
        [0.7]     # Portion to fast reservoir
    ],
    id='lower-splitter'
)

lag_fun = HalfTriangularLag(
    parameters={'lag-time': 2.0},
    states={'lag': None},
    id='lag-fun'
)

fast = FastReservoir(
    parameters={'k': 0.01, 'alpha': 3.0},
    states={'S0': 0.0},
    approximation=approximator,
    id='fast'
)

slow = FastReservoir(
    parameters={'k': 1e-4, 'alpha': 1.0},
    states={'S0': 0.0},
    approximation=approximator,
    id='slow'
)

lower_transparent = Transparent(
    id='lower-transparent'
)

lower_junction = Junction(
    direction=[
        [0, 0]
    ],
    id='lower-junction'
)
```

Defining the HRUs structure

Once all the elements have been created we can connect them composing the two HRUs.

```
consolidated = Unit(
    layers=[
        [upper_splitter],
        [snow, upper_transparent],
        [upper_junction],
```

(continues on next page)

(continued from previous page)

```python
        [unsaturated],
        [lower_splitter],
        [slow, lag_fun],
        [lower_transparent, fast],
        [lower_junction],
    ],
    id='consolidated'
)

unconsolidated = Unit(
    layers=[
        [upper_splitter],
        [snow, upper_transparent],
        [upper_junction],
        [unsaturated],
        [lower_splitter],
        [slow, lag_fun],
        [lower_transparent, fast],
        [lower_junction],
    ],
    id='unconsolidated'
)
```

Defining the catchments

Now that the HRUs have been created, we need to assign them to the catchments

```python
andelfingen = Node(
    units=[consolidated, unconsolidated],
    weights=[0.24, 0.76],
    area=403.3,
    id='andelfingen'
)

appenzell = Node(
    units=[consolidated, unconsolidated],
    weights=[0.92, 0.08],
    area=74.4,
    id='appenzell'
)

frauenfeld = Node(
    units=[consolidated, unconsolidated],
    weights=[0.49, 0.51],
    area=134.4,
    id='frauenfeld'
)

halden = Node(
    units=[consolidated, unconsolidated],
    weights=[0.34, 0.66],
    area=314.3,
    id='halden'
)

```

(continues on next page)

(continued from previous page)

```
29  herisau = Node(
30      units=[consolidated, unconsolidated],
31      weights=[0.88, 0.12],
32      area=16.7,
33      id='herisau'
34  )
35
36  jonschwil = Node(
37      units=[consolidated, unconsolidated],
38      weights=[0.9, 0.1],
39      area=401.6,
40      id='jonschwil'
41  )
42
43  mogelsberg = Node(
44      units=[consolidated, unconsolidated],
45      weights=[0.92, 0.08],
46      area=88.1,
47      id='mogelsberg'
48  )
49
50  mosnang = Node(
51      units=[consolidated],
52      weights=[1.0],
53      area=3.1,
54      id='mosnang'
55  )
56
57  stgallen = Node(
58      units=[consolidated, unconsolidated],
59      weights=[0.87, 0.13],
60      area=186.6,
61      id='stgallen'
62  )
63
64  waengi = Node(
65      units=[consolidated, unconsolidated],
66      weights=[0.63, 0.37],
67      area=78.9,
68      id='waengi'
69  )
```

Note that all the catchments incorporate the information about their area that will then be used by the network.

Not all the catchment must have all the HRUs; if an HRU is not present in a catchment (e.g. unconsolidated in Mosnang, line 50) it can be simply omitted.

Defining the network

The last step consists in creating the network that connects all the catchments previously declared.

```
model = Network(
    nodes=[
        andelfingen,
        appenzell,
        frauenfeld,
```

(continues on next page)

(continued from previous page)

```
        halden,
        herisau,
        jonschwil,
        mogelsberg,
        mosnang,
        stgallen,
        waengi,
    ],
    topography={
        'andelfingen': None,
        'appenzell': 'stgallen',
        'frauenfeld': 'andelfingen',
        'halden': 'andelfingen',
        'herisau': 'halden',
        'jonschwil': 'halden',
        'mogelsberg': 'jonschwil',
        'mosnang': 'jonschwil',
        'stgallen': 'halden',
        'waengi': 'frauenfeld',
    }
)
```

Running the model

Now that all the components have been initialized, we can run the model.

The first step is to assign the input fluxes to the single elements. For this we assume that the data is available as a pandas DataFrame and that the columns are named `P_name_of_the_catchment`, `T_name_of_the_catchment`, and `PET_name_of_the_catchment`.

The inputs can be set using a for loop

```
for cat, cat_name in zip(catchments, catchments_names):
    cat.set_input([
        df['P_{}'.format(cat_name)].values,
        df['T_{}'.format(cat_name)].values,
        df['PET_{}'.format(cat_name)].values,
    ])
```

After this, the last thing to be done before actually running the model is setting the time step used in the simulations. This can be done directly at the network level and it will be set to all the components.

```
thur_catchment.set_timestep(1.0)
```

The only thing left to do is running the model!

```
thur_catchment.get_output()
```

Note: Last update 01/09/2020

3.12 Examples

The following examples are available as Jupyter notebooks that can be either visualized on GitHub or run in a sandbox environment.

- Run a simple model run - visualize
- Calibrate a model run - visualize
- Initialize a single element model run - visualize
- Initialize a single unit model: run - visualize
- Initialize a simple node model: run - visualize
- Initialize a complete (network) model: run - visualize
- Create a new reservoir: run - visualize
- Replicate GR4J: run - visualize
- Replicate Hymod: run - visualize
- Replicate M02 in Dal Molin et al., HESS, 2020: run - visualize
- Replicate M4 in Kavetski and Fenicia, WRR, 2011: run - visualize
- Modify M4 in Kavetski and Fenicia, WRR, 2011: run - visualize

Note: Last update 06/05/2020

3.13 Reference

This reference provides details about the API of SuperflexPy. This page is limited to the framework; particular implementations of elements or components are not included for brevity

3.13.1 superflexpy.framework.element

3.13.2 superflexpy.utils.generic_component

3.13.3 superflexpy.framework.unit

3.13.4 superflexpy.framework.node

3.13.5 superflexpy.framework.network

3.13.6 superflexpy.utils.root_finder

3.13.7 superflexpy.utils.numerical_approximator

Note: Last update 04/09/2020

3.14 Change log

3.14.1 Version 1.0.0

Version 1.0.0 represents the first mature release of SuperflexPy. Many things have changed since previous 0.* releases both in terms of code organization and conceptualization of the framework. For this reason, models built with versions 0.* are not compatible.

Major changes to existing components

- New numerical solver structure for elements controlled by ordinary differential equations (ODEs). A new component, the `NumericaApproximator` is introduced; its task it to get the fluxes from the elements and construct an approximation of the ODEs. In the previous release of the framework the approximation was hard coded in the element implementation.
- `ODEsElement` have now to implement the methods `_fluxes` and `_fluxes_python` instead of `_differential_equation`
- Added the possibility for nodes and units to have local states and parameters. To this end, some internal functionalities for finding the element given the `id` have been changed to account for the presence of states and parameters at a level higher then the elements.

Minor changes to existing components

- Added implicit or explicit check at initialization of units, nodes, and network that the components that they contain are of the right type (e.g. a node must contain units)
- Some minor changes to the `RootFinder` to accommodate the new numerical implementation.
- Added numba implementation to GR4J elements

New code

- Added `hymod` elements