SuperflexPy

Version 1.2.1

Marco Dal Molin, Dmitri Kavetski, Fabrizio Fenicia

May 04, 2021

NOTE

This PDF is a snapshot of the documentation created on the date reported on the cover page. The sole purpose of this document is to become the supplement of the paper submitted to Geoscientific Model Development.

The current (most up-to-date) version of the documentation can be found at the address superflexpy.readthedocs.io. From the website, it is also possible to generate a PDF snapshot of the documentation. At the moment, this can be done accessing the menu on the bottom left of the webpage.

CONTENTS

SUPERFLEXPY

SuperflexPy is an open-source Python framework for constructing conceptual hydrological models for lumped and semi-distributed applications.

SuperflexPy builds on our 10 year experience with the development and application of Superflex, including collaborations with colleagues at the Eawag (Switzerland), TU-Delft (Netherlands), LIST (Luxembourg), University of Adelaide (Australia), and others. The SuperflexPy framework offers a brand new implementation of Superflex, allowing the modeler to build fully customized, spatially-distributed hydrological models.

Thanks to its object-oriented architecture, SuperflexPy can be easily extended to meet your modelling requirements, including the creation of new components with customized internal structure, in just a few lines of Python code.

Constructing a hydrological model is straightforward with SuperflexPy:

- inputs and outputs are handled directly by the modeler using common Python libraries (e.g. Numpy or Pandas). The modeller can use hence data files of their own design, without the need to pre- and/or post- process data into text formats prescribed by the framework itself;
- the framework components are declared and initialized through a Python script;
- the framework components are implemented as classes with built-in functionalities for handling parameters and states, routing fluxes, and solving the model equations (e.g. describing reservoirs, lag functions, etc.);
- the numerical implementation is separated from the conceptual model, allowing the use of different numerical methods for solving the model equations;
- the framework can be run at multiple levels of complexity, from a single-bucket model to a model that represents an entire river network;
- the framework is available as an open source Python package from Github;
- the framework can be easily interfaced with other Python modules for calibration and uncertainty analysis.

Team

SuperflexPy is actively developed by researchers in the Hydrological Modelling Group at Eawag, with the support of external collaborators.

The core team consists of:

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

Stay in touch

If you wish to receive e-mails about future SuperflexPy developments, please subscribe to our mailing list clicking here.

Note: Using SuperflexPy requires a general knowledge of Python and Numpy. Other Python libraries may be needed for pre- and post- processing of the data.

In line with the Python terminology, we will use the word **define** when referring to the definition of a class, and **initialize** when referring to the creation of an instance of a class, i.e. an object.

CHAPTER

ONE

INSTALLATION

SuperflexPy is implemented using Python 3 (version 3.7.3). It is not compatible with Python 2.

SuperflexPy is available as a Python package at PyPI repository

The simplest way to install SuperflexPy is to use the package installer for Python (pip). Open the operating system command prompt and run the command

```
pip install superflexpy
```

To upgrade to a newer SuperflexPy version (when available), run the following command

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

1.1 Dependencies

SuperflexPy requires the following Python packages

- Numpy
- Numba

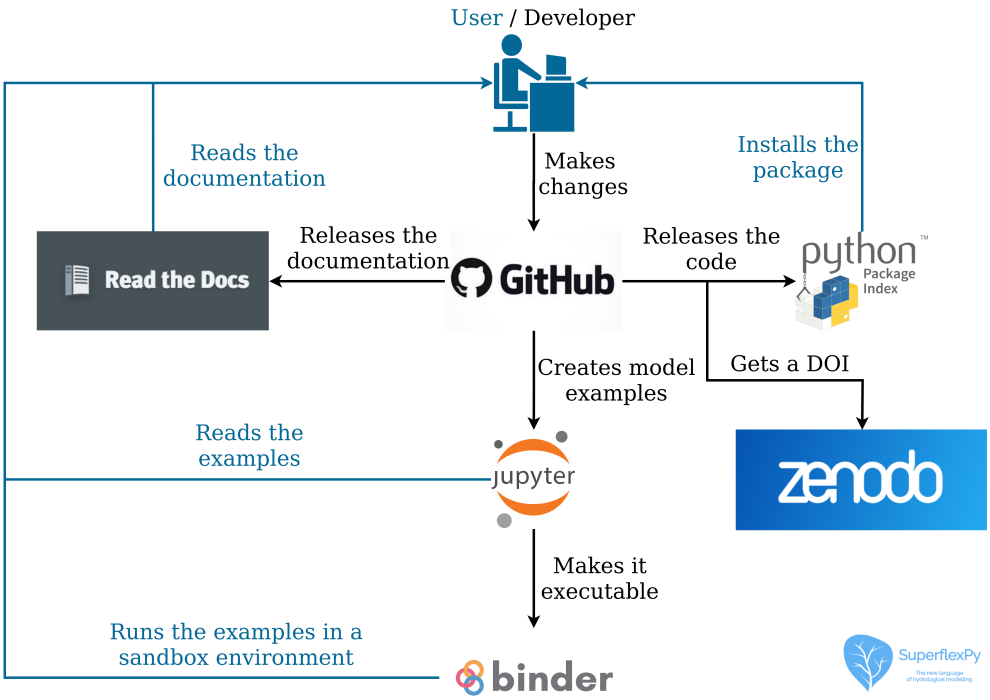
All dependencies are available through pip and will be installed automatically when installing SuperflexPy.

Note that Numba is required only if the modeler wishes to use the Numba optimized implementation of the numerical solvers. GPU acceleration (CUDA) is currently not supported but will be explored in future versions.

CHAPTER

TWO

SOFTWARE ORGANIZATION AND CONTRIBUTION



The SuperflexPy framework comprises the following components:

- **Source code**: Latest version of all the code necessary to use the framework. The source code would normally be accessed only by advanced users, e.g. to understand the internal organization of the framework, to install manually the latest version, to extend the framework with new functionality, etc.
- **Packaged release**: Latest stable version of the framework available for users.
- **Documentation**: Detailed explanation of the framework.
- **Examples**: Introduction to SuperflexPy for a new user, providing working models and demonstrating potential applications.
- **Scientific references**: Publications that present and/or use the framework in scientific contexts.

The source code, documentation, and examples are part of the official repository of SuperflexPy hosted on GitHub. A user who wishes to read the source code and/or modify any aspect of SuperflexPy (source code, documentation, and examples) can do it using GitHub.

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

The documentation builds automatically from the source folder on GitHub and is published online in Read the Docs.

Examples are available on GitHub as Jupyter notebooks. These examples can be visualized statically or run in a sandbox environment (see *Examples* for further details).

The scientific publication introducing SuperflexPy has been submitted to *Geoscientific Model Development* and is currently under review. The draft is available here.

2.1 Contributions

Contributions to the framework can be made in the following ways:

- Submit issues on bugs, desired features, etc;
- Solve open issues;
- Extend the documentation with new demos and examples;
- Extend and/or modify the framework;
- Use and cite the framework in your publications.

Code contribution by external users will be mainly additive (i.e., adding new components, as illustrated in *Expand SuperflexPy: Build customized elements* and *Expand SuperflexPy: Build customized components*) and should include also appropriate testing (*Automated testing*).

Contributors will maintain authorship of the contributed code and are invited to include, in all files, their contact information to facilitate future collaboration. The authors and maintainers of SuperflexPy will undertake a basic inspection of the contributed code to identify any quality issues.

The typical workflow that should be followed when contributing to a GitHub project is described here.

In summary, the following steps should be followed:

1. Fork the SuperflexPy repository to the user GitHub account;
2. Clone the fork on the user computer;
3. Modify the code, commit the changes, and push them to the GitHub fork of SuperflexPy;
4. Make a pull request on GitHub to the SuperflexPy repository.

2.1.1 Branching scheme of the GitHub repository

Updates to SuperflexPy are made directly in the branch `master`, which is the most up-to-date branch. The branch `release` is used only for the staging of new software releases and, therefore, code should not be pushed directly to it.

When a code update is merged from `master` to `release`, a new version of the package is automatically released on PyPI. Remember to update the version number in the `setup.py` file to avoid conflicts.

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 `master` branch.

CHAPTER

THREE

PRINCIPLES OF SUPERFLEXPY

Hydrological models are widely used in environmental science and engineering for process understanding and prediction.

Models can differ depending on how the processes are represented (conceptual vs. physical based models), and how the physical domain is discretized (from simple lumped configurations to detailed fully-distributed models).

At the catchment scale, conceptual models are the most widely used class of models, due to their ability to capture hydrological dynamics in a parsimonious and computationally fast way.

3.1 Conceptual models

Conceptual models describe hydrological dynamics directly at the scale of interest. For example, in catchment-scale applications, they are based on relationships between catchment storage and outflow. Such models are usually relatively simple and cheap to run; their simplicity allows extensive explorations of many different process representations, uncertainty quantification using Monte Carlo methods, and so forth.

Many conceptual models have been proposed over the last 40 years. These models have in common that they are composed by general elements such as reservoirs, lag functions, and connections. That said, existing models do differ from each other in a multitude of major and minor aspects, which complicates model comparison and selection.

Model differences may appear on several levels:

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

Several flexible modeling frameworks have been proposed in the last decade to facilitate the implementation and comparison of the diverse set of hydrological models.

3.2 Flexible modelling frameworks

A flexible modeling framework can be seen as a language for building conceptual hydrological models, which allows to build a (potentially complex) model from simpler low-level components.

The main objective of a flexible modeling framework is to facilitate the process of model building and comparison, giving modelers the possibility to adjust the model structure to help achieve their application objectives.

Although several flexible modeling frameworks have been proposed in the last decade, there are still some notable challenges. For example:

- implementation constraints can limit the originally envisaged flexibility of the framework;

- the choice of numerical model can be fixed;
- the spatial organization can be limited to lumped configurations;
- the ease of use can be limited by a complex software design.

These challenges can impact on usability, practicality and performance, and ultimately limit the types of modeling problems that can be tackled. The SuperflexPy framework is designed to address many of these challenges, providing a framework suitable for a wide range of research and operational applications.

3.3 Spatial organization

Hydrologists are increasingly interested in modeling large catchments where spatial heterogeneity becomes important. The following categories of spatial model organization can be distinguished:

- **lumped configuration**, where the entire physical domain is considered uniform;
- **semi-distributed configuration**, where the physical domain is subdivided into (usually coarse) areal fractions that are assumed to have the same hydrological response and operate in parallel (usually without connectivity between them);
- **fully-distributed configuration**, where the physical domain is subdivided into a (usually fine) grid. This configuration typically includes flux exchanges between neighboring grid cells.

The lumped approach yields the simplest models, with a low number of parameters and often sufficiently good predictions. However, the obvious limitation is that if the catchment properties vary substantially in space, the lumped model will not capture these variations. Nor can a lumped model produce spatially distributed streamflow predictions.

The fully-distributed approach typically yields models with a large number of parameters and high computational demands, usually related to the resolution of the grid that is used.

The semi-distributed approach is intermediate between the other two approaches in terms of spatial complexity and number of parameters. A typical example is the discretisation of the catchment into Hydrological Response Units (HRUs), defined as catchment areas assumed to behave in a hydrologically "similar" way. The definition of HRUs represents a modelling choice and depends on the process understanding available in the catchment of interest.

3.4 SuperflexPy

SuperflexPy is a new flexible framework for building hydrological models. It is designed to accommodate models with a wide range of structural complexity, and to support spatial configurations ranging from lumped to distributed. The design of SuperflexPy is informed by the extensive experience of its authors and their colleagues in developing and applying conceptual hydrological models.

In order to balance flexibility and ease of use, SuperflexPy is organized in four different levels, which correspond to different degrees of spatial complexity:

1. elements;
2. units;
3. nodes;
4. network.

The first level is represented by "elements", which comprise reservoirs, lag functions, and connections. Elements can represent entire models or individual model components, and are intended to represent specific processes within the hydrological cycle (e.g. soil dynamics).

The second level is represented by "units", which connect together multiple elements. This level can be used to build lumped models or to represent HRUs within a spatially distributed model.

The third level is represented by "nodes", where each node contains several units that operate in parallel. Nodes can be used to distinguish the behavior of distinct units within a catchment, e.g., when building a (semi)-distributed model where the units are used to represent HRUs (defined according to soil, vegetation, topography, etc).

The fourth level is represented by the "network", which connects multiple nodes and routes the fluxes from upstream to downstream nodes. This level enables the representation of large watersheds and river networks that comprise several subcatchments with substantial flow routing delays. A SuperflexPy model configuration can contain only a single network.

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

CHAPTER

FOUR

ORGANIZATION OF SUPERFLEXPY

SuperflexPy is designed to operate at multiple levels of complexity, from a single reservoir to a complex river network.

All SuperflexPy components, namely elements, units, nodes, network, are designed to operate alone or as part of other components. For this reason, all components have methods that enable the execution of basic functionality (e.g. parameter handling) at all levels. For example, consider a unit that contains multiple elements. The unit will then provide the functionality for setting the parameter values for its elements.

Note that, programmatically, SuperflexPy component types are classes, and the actual model components are then class instances (objects).

We will first describe each component type in specific detail, and then highlight some *Generalities* that apply to all components.

4.1 Elements

Elements represent the basic level of the SuperflexPy. Conceptually, SuperflexPy uses the following elements: reservoirs, lag functions, and connections. Elements can be used to represent a complete model structure, or combined together to form one or more *Unit*.

Depending on their type, conceptual elements can have parameters and/or states, can handle multiple fluxes as inputs and/or as outputs, can be designed to operate with one or more elements upstream or downstream, can be controlled by differential equations or by a convolution operations, etc.

Programmatically, the conceptual elements can be implemented by extending the following classes:

- `BaseElement`: for elements without states and parameters (e.g., junctions);
- `StateElement`: for elements with states but without parameters;
- `ParameterizedElement`: for elements with parameters but without states (e.g., junctions);
- `StateParameterizedElement`: for elements with states and parameters (e.g., reservoirs and lag functions).

For example, consider the conceptual element "junction", which sums the fluxes coming from multiple elements upstream without needing states or parameters. This element can be built by extending the class `BaseElement` to implement the method that sums the fluxes.

To facilitate usage, SuperflexPy provides a set of "pre-packaged" classes that already implement already most of the functionality needed to specify reservoirs, lag functions, and connections. The next sections focus on these classes.

4.1.1 Reservoirs

A reservoir is a storage element described by the differential equation (or, more generally, a system of differential equations)

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

where $\mathbf{S}$ represents the internal states of the reservoir, $\mathbf{I}$ represents the sum of all input fluxes, $\mathbf{O}$ represents the sum of all output fluxes, and θ represents the parameters that control the behavior of the reservoir. In most conceptual models, reservoir elements have a single state variable (representing water storage), however multiple state variables can be accommodated when necessary (e.g., to represent transport).

SuperflexPy provides the class `ODEsElement` that contains all the logic needed to represent an element controlled by a differential equation. The user needs only to specify the equations defining input and output fluxes.

The differential equation is solved numerically, though analytical solutions could be possible. The choice of solution method (e.g. the implicit Euler scheme) is made by the user when initializing the reservoir element.

SuperflexPy provides several "numerical approximators" to solve decoupled ODEs, including the implicit and the explicit Euler schemes. The user can either employ 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 implementation*.

4.1.2 Lag functions

A lag function is an element that applies a delay to the incoming fluxes. In mathematical terms, the lag function represents a convolution of the incoming fluxes with a weight function. Here, the convolution is implemented by distributing the fluxes at a given time step into the subsequent time steps, according to a weight array. The same procedure is then repeated over multiple time steps, adding together the contributions originating from the preceding time steps.

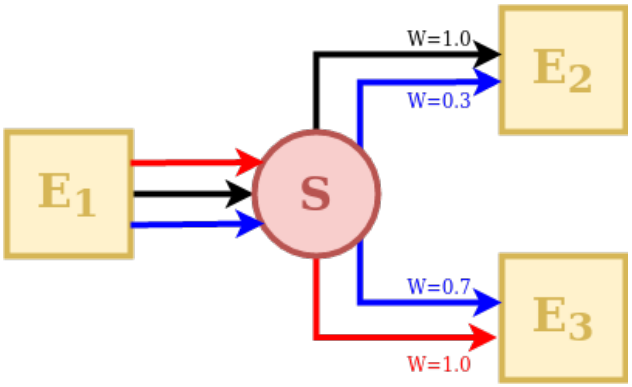
SuperflexPy provides the class `LagElement` that implements all the methods needed to represent a lag function. The user only needs to define the weight array.

4.1.3 Connections

Connection elements are used to link together multiple elements when building a unit.

SuperflexPy provides several types of connection elements. For example, a `Splitter` is used to split the output flux from a single upstream element and distribute the respective portions to multiple downstream elements. Conversely, a `Junction` is used to collect the output fluxes from multiple upstream elements and feed them into a single downstream element. Connection elements are designed to operate with an arbitrarily number of fluxes and upstream/downstream elements.

Splitter



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

The behavior of a splitter in SuperflexPy is controlled by two matrices: "direction" and "weight". The direction matrix specifies *which* input fluxes contribute (even fractionally) to the downstream elements and in which order. The weight matrix defines the *proportion* of each of the input fluxes that goes into each the downstream element.

In the illustration schematic, element S receives 3 input fluxes, which are coloured and indexed according to their order: red (index 0), black (index 1), and blue (index 2). Element E2 receives the black flux as its first input (index 0), the blue flux as its second input (index 1), and does not receive any portion of the third flux. Element E3 receives the blue flux as its first input (index 0), the red flux as its second input (index 1), and does not receive any portion of the black flux.

This information is represented by the direction matrix $\mathbf{D}$ as follows:

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

The direction matrix is a 2D matrix with as many columns as the number of fluxes and as many rows as the number of downstream elements. The row index refers to a downstream element (in this case the first row refers to element E2, and the second row to element E3). The column index refers to the input fluxes received by to the downstream element. Note that care must be taken when indexing the elements and fluxes to correctly reflect the intended model structure.

The values of $\mathbf{D}$ can be an integer referring to the index of the input flux to the splitter S, or `None` if an input flux to the splitter S does not reach a downstream element.

As such, the direction matrix can be used to select the fluxes and change the order in which they are transmitted to downstream elements.

Next, we consider the weight matrix $\mathbf{W}$, which describes the fraction of each flux directed to each downstream element. The red flux is taken entirely by element E3, the black flux is taken entirely by element E2, and the blue flux is split at 30% to E2 and 70% to E3. This information is represented as follows:

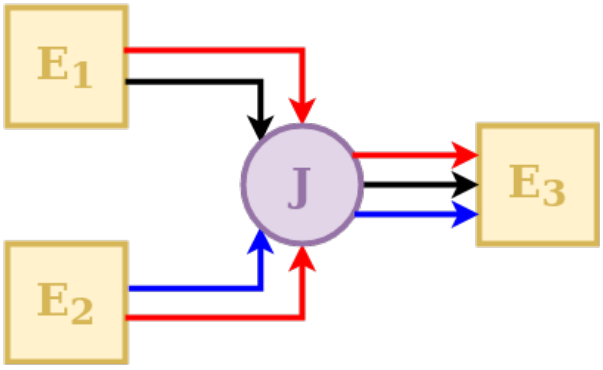
$$\mathbf{W} = \begin{pmatrix} 0 & 1.0 & 0.3 \\ 1.0 & 0 & 0.7 \end{pmatrix}$$

The weight matrix has the same shape as the direction matrix. The row index refers to the downstream element, in the same order as in the direction matrix $\mathbf{D}$, whereas the column index refers to the input flux to the splitter S.

The elements of $\mathbf{W}$ represent the fraction of each input flux received by the splitter S and directed to the downstream element. In the example, the first downstream element (first row of the matrix $\mathbf{W}$) receives 0% of the first (red) flux, 100% of the second (black) flux, and 30% of the third (blue) flux.

Note that the columns of the weight matrix should sum up to 1 to ensure conservation of mass.

Junction



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

The behavior of a junction in SuperflexPy is controlled by a direction matrix, which defines how the incoming fluxes are to be combined (summed) to feed the downstream element.

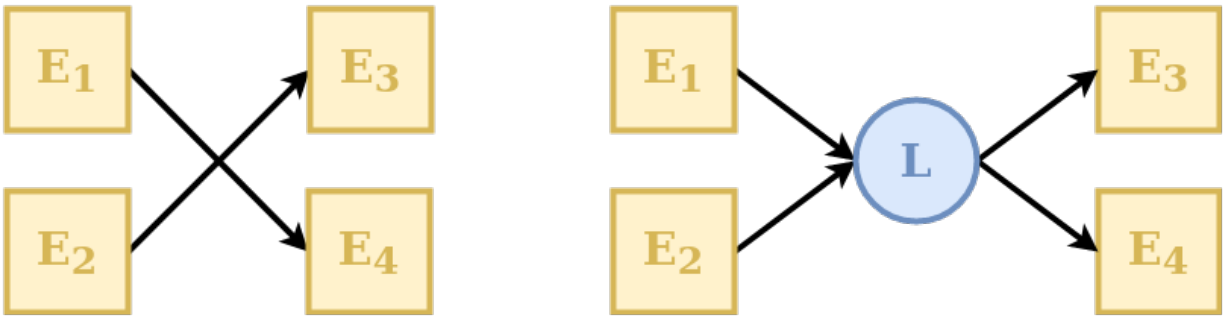
In the schematic, element E3 receives 3 input fluxes, which are indexed based on their order: red (index 0), black (index 1), and blue (index 2). The red flux comes from both upstream elements (index 0 and 1, respectively); the black flux comes only from element E1 (index 1); the blue flux comes only from element E2 (index 2). This information is represented by the direction matrix $\mathbf{D}$ as follows:

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

The direction matrix is a 2D matrix that has as many rows as the number of fluxes and as many columns as number of upstream elements. The row index refers to the flux (in this case the first row refers to the red flux, the second row to the black flux, and the third row to the blue flux). The column index refers to the upstream element input flux to the junction (in this case the first column refers to element E1, the second column to element E2).

The value of the matrix element can be an integer referring to the index of the input flux to junction J coming from the specific upstream element, or `None` if an input flux to junction J does not come from the upstream element.

Linker

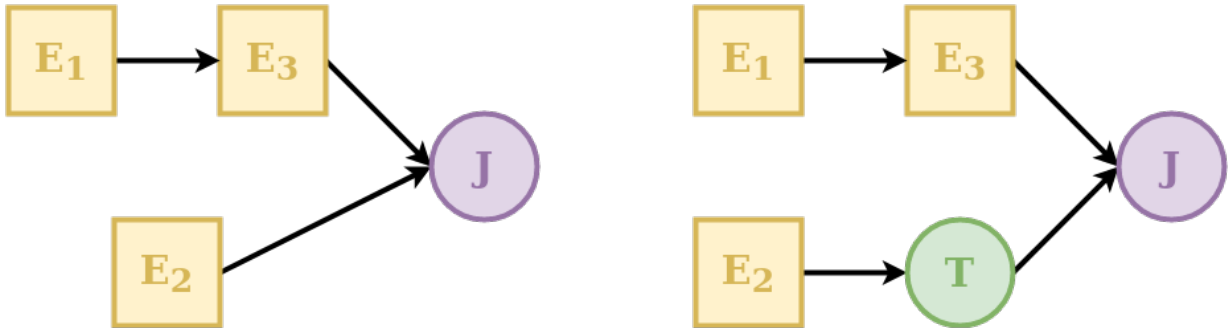


A `Linker` is an element that can be used to connect multiple elements upstream to multiple elements downstream without mixing fluxes.

Linkers are useful in SuperflexPy because the structure of the unit is defined as an ordered list of elements (see *Unit*). This means that if we want to connect the first element of a layer to the second element of the following layer (e.g.,

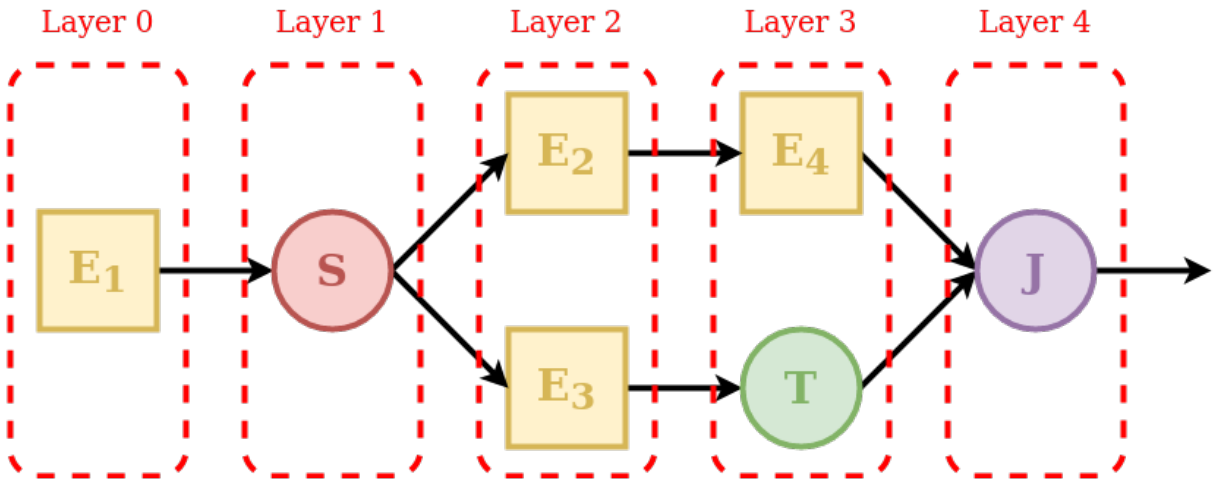
direct the output from upstream element E1 to downstream element E4, in the example above) we have to insert an additional intermediate layer with a linker that directs the fluxes to the correct downstream element. Further details on the organization of the units in layers are presented in section *Unit*.

Transparent



A transparent element is an element that returns, as output, the same fluxes that it receives as input. This element is needed to fill "gaps" in the structure defining a unit. See *Unit* for further details. An example is shown in the schematic above where the transparent element is used to make the two rows have the same number of elements.

4.2 Unit



A unit is a collection of multiple connected elements. The unit can be used either alone, when intended to represent a lumped catchment model, or as part of a *Node*, to create a semi-distributed model.

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

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

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

When the output of an element is split between multiple downstream elements an additional intermediate layer with a splitter is needed. For example, element E1 is intended to provide its outputs to elements E2 and E3. In this case the splitter S has two downstream elements (E2 and E3); the framework will route the first group of outputs of the splitter to element E2 and the second group of outputs to element 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 element E3 is combined with the output of element E4. Since these elements belong to different layers, making this connection directly would create a gap in Layer 3. This problem is solved by specifying a transparent element in Layer 3, i.e., in the same layer as element E4.

Finally, since the unit must have a single element in its last layer, the outputs of elements E4 and T must be collected using the junction J.

Each element is aware of its expected number of upstream and downstream elements. For example, a reservoir must have a single upstream element and a single downstream element, a splitter must have a single upstream element and potentially multiple downstream elements, and so on. A unit is valid only if all layers connect to each other using the expected number of elements. In the example, Layer 1 must have two downstream elements that is consistent with the configuration of Layer 2.

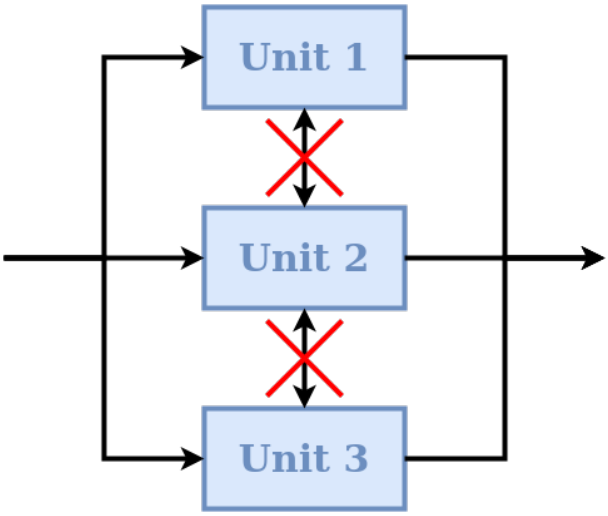
Elements are *copied* into the unit. This means that an element that belongs to a unit is completely independent from the originally defined element and from any other copy of the same element in other units. This SuperflexPy design choice ensures that changes to the state or to the parameters of an element within a given unit will not affect any element outside of that unit. The code below illustrates this behavior:

```
1 e1 = Element(parameters={'p1': 0.1}, states={'S': 10.0})
2
3 u1 = Unit([e1])
4 u2 = Unit([e1])
5
6 e1.set_parameters({'e1_p1': 0.2})
7 u1.set_parameters({'u1_e1_p1': 0.3})
8 u2.set_parameters({'u2_e1_p1': 0.4})
```

In the code, element `e1` is included in units `u1` and `u2`. In lines 6-8 the value of parameter `p1` of element `e1` is changed at the element level and at the unit level. Since elements are *copied* into a unit, these changes apply to three different elements (in the sense of different Python objects in memory), the "originally defined" `e1` and the copies of `e1` in `u1` and `u2`.

For more information on how to define a unit structure in SuperflexPy, refer to the page *Application: implementation of existing conceptual models*, where the framework is used to reproduce some existing lumped models.

4.3 Node



A node is a collection of multiple units assumed to operate in parallel. In the context of semi-distributed models, a node represents a single catchment and the units represent multiple landscape elements (areas) within the catchment. A node can be run either alone or as part of a bigger *Network*.

The default behavior of nodes is that parameters are *shared* between elements of the same unit, even if that unit belongs to multiple nodes. This SuperflexPy design choice is motivated by the unit being intended to represent areas that have the same hydrological response. The idea is that the hydrological response is controlled by the parameters, and therefore elements of the same unit (e.g. HRU) belonging to multiple nodes should have the *same* parameter values.

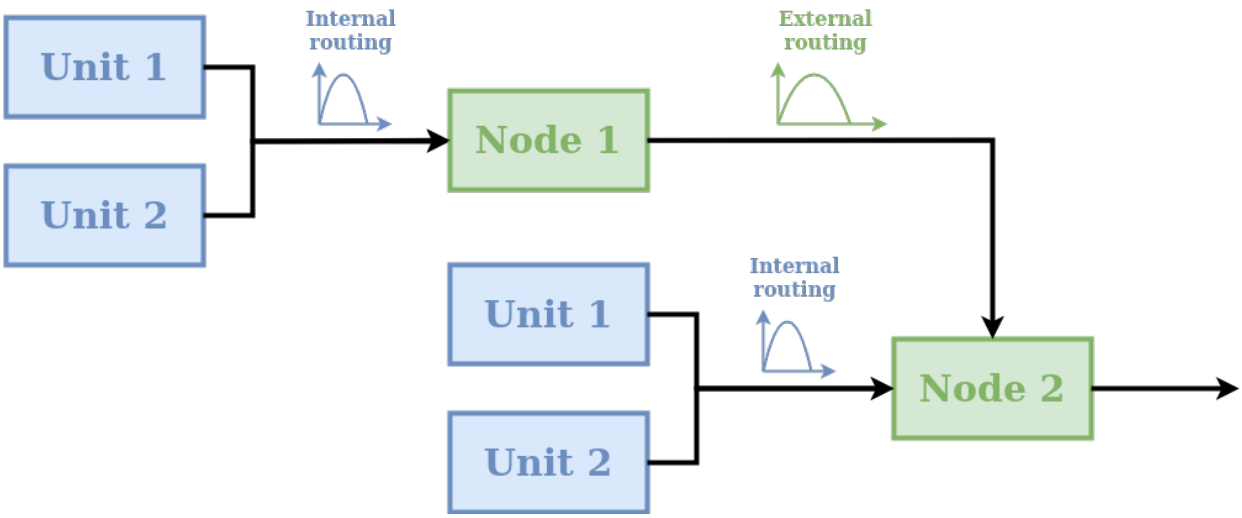
On the other hand, each node has its own states that are tracked separately from the states of other nodes. In particular, when multiple nodes that share the same parameter values receive different inputs (e.g., rainfall), their states will evolve differently. This SuperflexPy design choice supports the most common use of nodes, which is the discretisation of a catchment into potentially overlapping HRUs and subcatchments. Parameters are then assumed constant within HRUs (units), and inputs are assumed to be constant within subcatchments (nodes).

In term of SuperflexPy code, this behavior is achieved by (1) copying the states of the elements belonging to the unit when this unit becomes part of a node; (2) sharing, rather than copying, the parameter values. This means that changes to the parameter values of an element within a node will affect the parameter values of that elements of all other nodes that share the same unit. In contrast, changes to the states will be node-specific.

This default behavior can be changed by setting `shared_parameters=False` at the initialization of the node. In this case, all parameters become node-specific, with no sharing of parameter values even within the same unit.

Refer to the section *Simple semi-distributed model* for details on how to incorporate units into nodes.

4.3.1 Routing



A node can include routing functions that delay the fluxes. As shown in the schematic, two types of routing are possible:

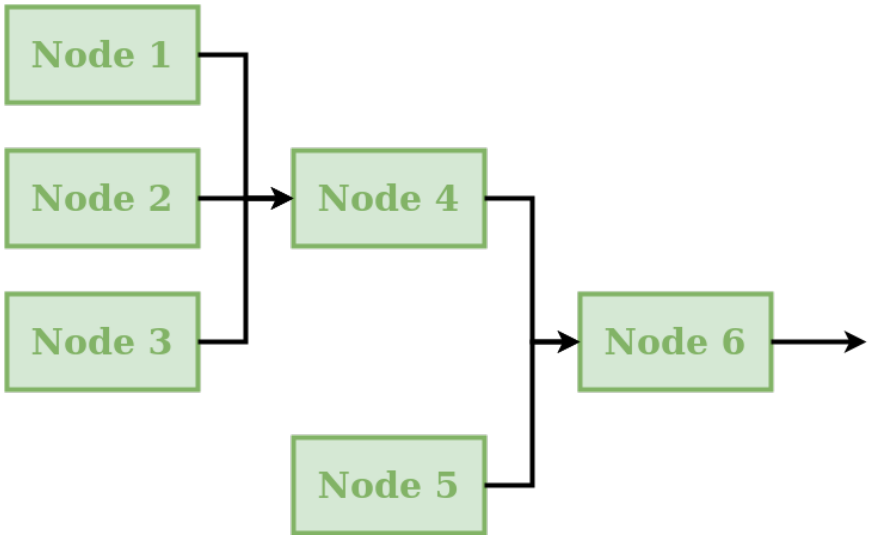
- internal routing;
- external routing.

A typical usage of these routing functions in semi-distributed hydrological modelling is as follows. Internal routing is used to represent delays associated with the routing of fluxes across the catchment towards the river network. External routing is used to represent delays associated with the routing of fluxes within the river network, i.e., from the outlets of the given node to the inlet of the downstream node.

More generally, routing functions can be used for representing any type of delay between the units and the node, as well as delays between nodes.

In the default implementation of a node in SuperflexPy, the two routing functions simply return their input (i.e. no delay is applied). The user can implement a different behavior, e.g., see section *Adding routing to a node*.

4.4 Network



A network connects multiple nodes into a tree structure, and is typically intended to develop a distributed model that generates predictions at internal subcatchment locations (e.g. to represent a "nested" catchment setup).

The connectivity of the network is defined by assigning to each node the information about its downstream node. The network will then compute the node output fluxes, starting from the inlets and then moving downstream, calculating the outflows of the remaining nodes and routing the fluxes towards the outlet.

The network is the only component of SuperflexPy that does not have the `set_input` method (see *Generalities*), because inputs are assumed to be node-specific and hence have to be assigned to each node within the network.

A node is *inserted* (rather than *copied*) into the network. In other words, we initialize a node object and then insert it into the network. This node can then be configured either directly or through the network. Any changes occurring within the node as part of the network affect also the originally defined node (because they are the same Python object).

The output of the network is a dictionary that contains the output of all nodes within the network.

4.5 Generalities

4.5.1 Common methods

All components share the following methods.

- **Parameters and states**: each component has its own parameters and/or states with unique identifiers. Each component of SuperflexPy has methods to set and get the states and parameters of the component itself as well as the states and parameters of its contained components:
  - `set_parameters`: change the current parameter values
  - `get_parameters`: get the current parameter values
  - `get_parameters_name`: get the identifiers of the parameters
  - `set_states`: change the current state values
  - `get_states`: get the current state value
  - `get_states_name`: get the identifiers of the states

  - `reset_states`: reset the states to their initialization value

- **Time step**: as common in hydrological modeling, inputs and outputs are assumed to have the same time resolution, i.e., the input and output data must share the same time stamps. There is no requirement for timestamps to be uniformly spaced, meaning that the time series can have irregular time step sizes. In SuperflexPy, all components that require the definition of a time step (e.g. reservoirs described by a differential equation) contain methods that set and get the time step size. In case of non-uniform time resolution, an array of time steps needs to be provided by the user.
  - `set_timestep`: set the time step used in the model. All components at a higher level (e.g. units) have this method; when called, it applies the change to all elements within the component;
  - `get_timestep`: returns the time step size used in the model.
- **Inputs and outputs**: all components have functionalities to receive inputs and generate outputs.
  - `set_input`: set the component inputs; inputs can be fluxes (e.g., precipitation) or other relevant variables (e.g., temperature influencing the behavior of a snow element).
  - `get_output`: run the component (and all components contained in it) and return the output fluxes.

4.5.2 Component identifiers

In SuperflexPy, ll parameters, states, and components (except for the network) are identified using an identifier string assigned by the user. The identifier string can have an arbitrary length, with the only restriction being that it cannot contain the underscore _, as this is a special character used internally by SuperflexPy.

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 underscore _ as separator.

For example, if the element with identifier `e1` has the parameter `par1`, the name of the parameter becomes, at initialization, `e1_par1`. If element `e1` is inserted into unit `u1`, the parameter 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 within any component of the model.

4.5.3 Time varying parameters

In hydrological modelling, time varying parameters can be useful for representing certain types of model variability, e.g., seasonal phenomena and/or stochasticity.

SuperflexPy can be used with both constant and time varying parameters. Parameters can be specified as either scalar float numbers or as Numpy 1D arrays of the same length as the arrays of input fluxes. In the first case, the parameter will be interpreted as time constant. In the second case, the parameter will be considered as time varying and may have a different value at each time step.

4.5.4 Length of the simulation

In SuperflexPy, there is no model parameter controlling the length of the simulation. The number of model time steps that need to be run is determined automatically at runtime from the length of the arrays containing the input fluxes. For this reason, all input data time series must have the same length.

4.5.5 Format of inputs and outputs

The input and output fluxes of all SuperflexPy components are represented using 1D Numpy arrays.

For the inputs, regardless of the number of fluxes, the method `set_input` takes a list of Numpy arrays (one array per flux). The order of arrays inside the list is important and must follow the indications of the docstring of the method. All input fluxes must have the same length because the number of time steps in the model simulation is determined by the length of the input time series; see also *Length of the simulation*.

The outputs are also returned as a list of Numpy 1D arrays, using the `get_output` method.

Note an important exception for *Connections*: whenever the number of upstream or downstream elements is different from one, the `set_input` or the `get_output` methods will use 2D lists of Numpy arrays. This solution is used to route fluxes between multiple elements.

CHAPTER

FIVE

NUMERICAL IMPLEMENTATION

5.1 Numerical routines for solving ODEs

Reservoirs are the most common elements in conceptual hydrological models. Reservoirs are controlled by one (or more) ordinary differential equations (ODEs) of the form

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

and associated initial conditions.

Such differential equations are usually difficult or impossible to solve analytically, therefore, numerical approximations are employed. These numerical approximations take the form of time stepping schemes.

Moreover, many robust numerical approximations (e.g. implicit Euler) require an iterative root-finding procedure at each time step.

The current implementation of SuperflexPy conceptualizes the solution of the ODE as a two-step procedure:

1. Construct the discrete-time equations defining the numerical approximation of the ODEs at a single time step
2. Solve the numerical approximation for the storage(s)

These steps can be performed using two SuperflexPy components: `NumericalApproximator` and `RootFinder`.

SuperflexPy provides two built-in numerical approximators (implicit and explicit Euler) and a root finder (Pegasus method). These methods are best suited when dealing with smooth flux functions. If a user wants to experiment with discontinuous flux functions, other ODE solution algorithms should be considered.

Other ODE solution algorithms, e.g. Runge-Kutta, can be implemented by extending the classes `NumericalApproximator` and/or `RootFinder`. Even more generally, ODE solvers from external libraries can be used within SuperflexPy by incorporating them in a class that respects the interface expected by the `NumericalApproximator`.

The following sections describe the standard approach to create customized numerical approximators and root finders.

5.1.1 Numerical approximator

A customized numerical approximator can be implemented by extending the class `NumericalApproximator` and implementing two methods: `_get_fluxes` and `_differential_equation`.

```
1 class CustomNumericalApproximator(NumericalApproximator):
2
3     @staticmethod
```

(continues on next page)

(continued from previous page)

```
    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 [diff_eq, min_val, max_val]
```

where `fluxes` is a list of functions used to calculate the fluxes, `S` is the state that solves the ODE, `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 `_get_fluxes` is responsible for calculating the fluxes after the ODE has been solved. This method operates with a vector of states.

The method `_differential_equation` calculates the approximation of the ODE. It returns the residual of the approximated mass balance equations for a given value of `S` and the minimum and maximum bounds for the search of the solution. This method is designed to be interfaced with the root finder.

For further details, please see the implementation of `ImplicitEuler` and `ExplicitEuler`.

5.1.2 Root finder

A customized root finder can constructed by extending the class `RootFinder` implementing the method `solve`.

```
class CustomRootFinder(RootFinder):

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

        # Some code here

        return root
```

where `diff_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. In case of failure, the method should either raise a `RuntimeError` (Python implementation) or return `numpy.nan` (this is not ideal but it is the suggested workaround because Numba does not support exceptions handling).

To understand better how the method `solve` works, please see the implementation of the Pegasus root finder that is currently used in the SuperflexPy applications.

5.2 Sequential solution of the elements

The SuperflexPy framework is built on a model representation that maps to a directional acyclic graph. Model elements are solved sequentially from upstream to downstream, with the output from each element being used as input to its downstream elements.

When fixed-step solvers are used (e.g. implicit Euler), this "one-element-at-a-time" strategy is equivalent to applying the same (fixed-step) solver to the entire ODE system simultaneously.

When solvers with internal substepping are used, the "one-element-at-a-time" strategy does introduces additional approximation error. This additional approximation error is due to treating all fluxes as constant over the time step, whereas in reality fluxes vary within the time step. In most practical applications, this "uniform flux" approximation is already applied to the meteorological inputs (precipitation and PET), hence applying it to internal fluxes does not represent a large additional approximation.

5.3 Computational efficiency with Numpy and Numba

Conceptual hydrological models are often used in computationally demanding contexts, such as parameter calibration and uncertainty quantification, which require many model runs (thousands or even millions). Computational efficiency is therefore an important requirement of SuperflexPy.

Computational efficiency is a potential limitation of pure Python, but libraries like Numpy and Numba can help in pushing the performance closer to traditionally fast languages such as Fortran and C.

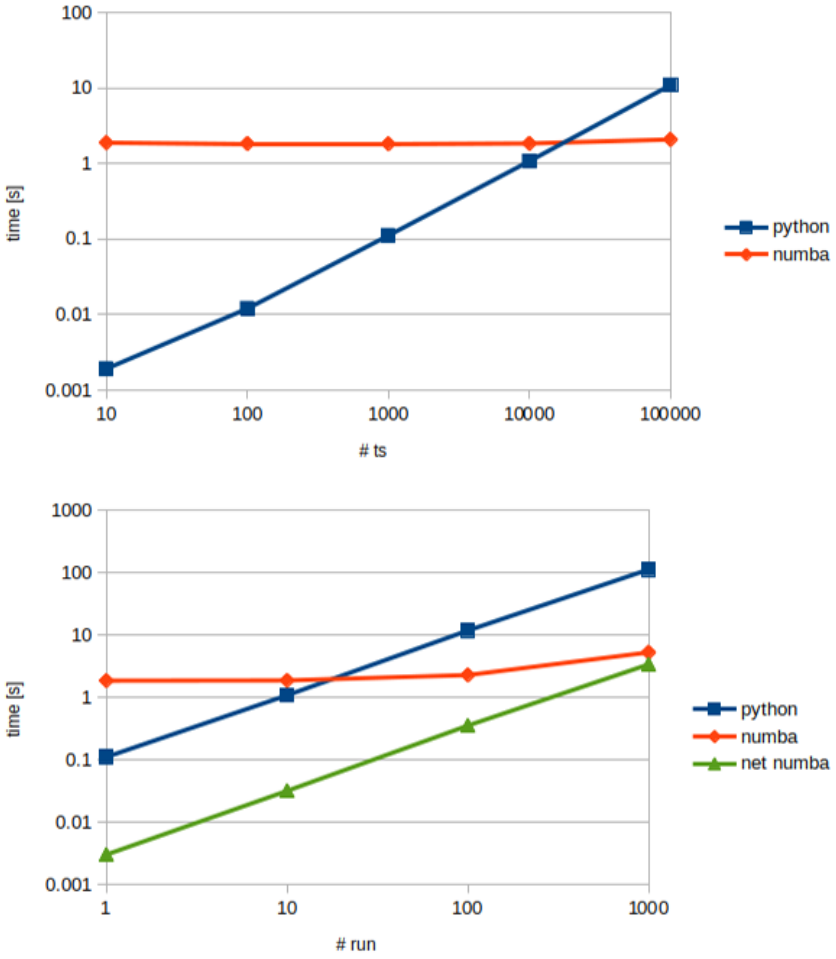
Numpy provides highly efficient arrays for vectorized operations (i.e. elementwise operations between arrays). Numba provides a "just-in-time compiler" that can be used to compile (at runtime) a normal Python method to machine code that operates efficiently with Numpy arrays. The combined use of Numpy and Numba is extremely effective when solving ODEs using time stepping schemes, where the method loops through a vector to perform elementwise operations.

SuperflexPy includes Numba-optimized versions of `NumericalApproximator` and `RootFinder`, which enable efficient solution of ODEs describing the reservoir elements.

The figure below compares the execution times of pure Python vs. the Numba implementation, as a function of the length of the time series (upper panel) and the number of model runs (lower panel). Simulations were run on a laptop (single thread), using the *HYMOD* model, solved using the implicit Euler numerical solver.

The plot clearly shows the tradeoff between compilation time (which is zero for Python and around 2 seconds for Numba) versus run time (where Numba is 30 times faster than Python). For example, a single run of 1000 time steps takes 0.11 seconds with Python and 1.85 seconds with Numba. In contrast, if the same model is run 100 times (e.g., as part of a calibration) the Python version takes 11.75 seconds while the Numba version takes 2.35 seconds.

Note: The objective of these plots is to give an idea of time that is topically required to perform common modelling applications (e.g., calibration) with SuperflexPy, to show the impact of the Numba implementation, and to explain the tradeoff between compilation and run time. The results do not have to be considered as accurate measurements of the performance of SuperflexPy (i.e., rigorous benchmarking).



The green line “net numba” in the lower panel express the run time of the Numba implementation, i.e., excluding the compilation time.

CHAPTER

SIX

HOW TO BUILD A MODEL WITH SUPERFLEXPY

This page shows how to build a complete semi-distributed conceptual model using SuperflexPy, including:

1. how the elements are initialized, configured, and run
2. how to use the model at any level of complexity, from single element to multiple nodes.

All models presented in this page are available as runnable examples (see *Examples*).

Examples of the implementation of more realistic models are given in the pages *Application: implementation of existing conceptual models* and *Case studies*.

6.1 Importing SuperflexPy

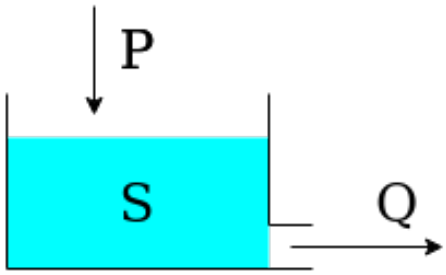
Assuming that SuperflexPy is already installed (see *Installation* guide), the elements needed to build the model are imported from the SuperflexPy package. In this demo, the import is done with the following lines

```
1 from superflexpy.implementation.elements.hbv import PowerReservoir
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 two elements (a reservoir and a lag function), lines 3-4 import the numerical solver used to solve the reservoir equation, and lines 5-7 import the SuperflexPy components needed to implement spatially distributed model.

A complete list of the elements already implemented in SuperflexPy, including their equations and import path, is available in page *List of currently implemented elements*. If the desired element is not available, it can be built following the instructions given in page *Expand SuperflexPy: Build customized elements*.

6.2 Simplest lumped model structure with single element



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

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

where S is the state (storage) of the reservoir, P is the precipitation input, and Q is the outflow.

The outflow is defined by the equation:

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

where k and α are parameters of the element.

For simplicity, evapotranspiration is not considered in this demo.

The first step is to initialize the numerical approximator (see *Numerical implementation*). In this case, we will use the native Python implementation (i.e. not Numba) of the implicit Euler algorithm (numerical approximator) and the Pegasus algorithm (root finder). The initialization 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)
```

Note that the object `approximator` does not have internal states. Therefore, the same instance can be assigned to multiple elements.

The element is initialized next

```
reservoir = PowerReservoir(
    parameters={'k': 0.01, 'alpha': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
    id='R'
)
```

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

After initialization, we specify the time step used to solve the differential equation and the inputs of the element.

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

Here, `precipitation` is a Numpy array containing the precipitation time series. The length of the simulation (i.e., the number of time steps to run the model) is automatically set to the length of the input arrays.

The element can now be run

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

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

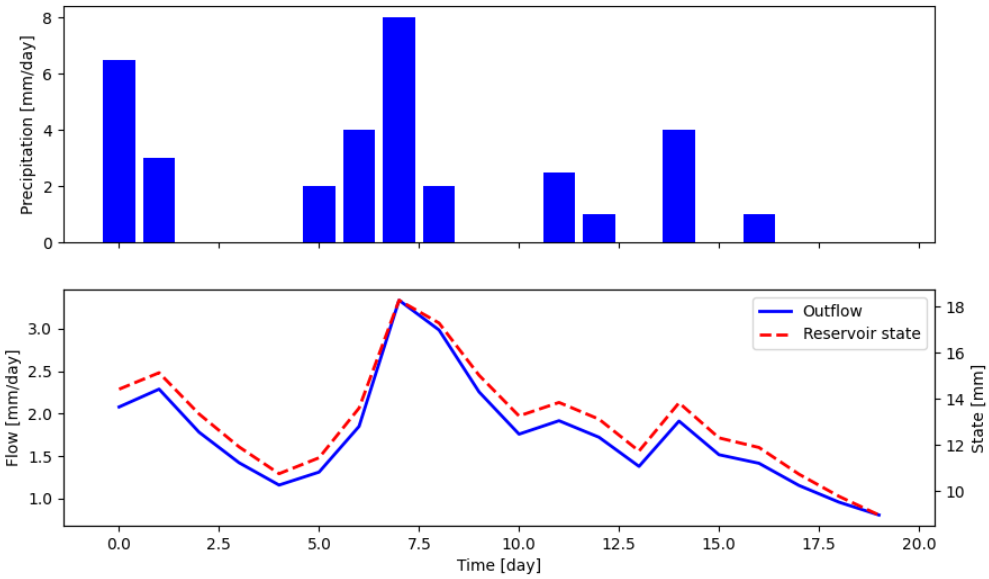
The state of the reservoir at all time steps is saved in the attribute `state_array` of the element and can be accessed as follows

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

Here, `state_array` is a 2D array with the number of rows equal to the number of time steps, and the number of columns equal to the number of states. The order of states is defined in the docstring of the element.

Finally, the simulation outputs can be plotted using standard Matplotlib functions, as follows

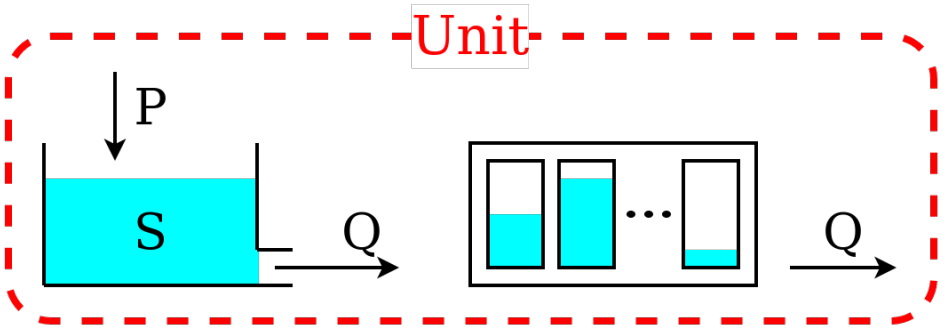
```
fig, ax = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
ax[0].bar(x=range(len(precipitation)), height=precipitation, color='blue')
ax[1].plot(range(len(precipitation)), output, color='blue', lw=2, label='Outflow')
ax_bis = ax[1].twinx()
ax_bis.plot(range(len(precipitation)), reservoir_state, color='red', lw=2, ls='--',␣
↪label='Reservoir state')
```



Note that the method `get_output` also sets the element states to their value at the final time step (in this case 8.98). As a consequence, if the method is called again, it will use this value as initial state instead of the one defined at initialization. This enables the modeler to continue the simulation at a later time, which can be useful in applications where new inputs arrive in real time. The states of the model can be reset using the method `reset_states`.

```
reservoir.reset_states()
```

6.3 Lumped model structure with 2 elements



We now move to a more complex model structure, where multiple elements are connected in a unit. For simplicity, we limit the complexity to two elements; more complex configurations can be found in the *Application: implementation of existing conceptual models* page.

The unit structure comprises a reservoir that feeds a lag function. The lag function applies a convolution operation on the incoming fluxes

$$Q_{\text{out}}(t) = \int_0^t Q_{\text{in}}(t-\tau)h(\tau, t_{\text{lag}})\text{d}\tau$$

The behavior of the lag function is controlled by parameter t_{lag}.

First, we initialize the two elements that compose the unit structure

```
1  reservoir = PowerReservoir(
2      parameters={'k': 0.01, 'alpha': 2.0},
3      states={'S0': 10.0},
4      approximation=approximator,
5      id='R'
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 is set to `None` (line 10). The element will then initialize the state to an arrays of zeros of appropriate length, depending on the value of t_{lag}; in this specific case, `ceil(2.3) = 3`.

Next, we initialize the unit that combines the elements

```
1  unit_1 = Unit(
2      layers=[[reservoir], [lag_function]],
3      id='unit-1'
4  )
```

Line 2 defines the unit structure: it is a 2D list where the inner level sets the elements belonging to each layer and the outer level lists the layers.

After initialization, time step size and inputs are defined

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

The unit sets the time step size for all its elements and transfers the inputs to the first element (in this example, to the reservoir).

The unit can now be run

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

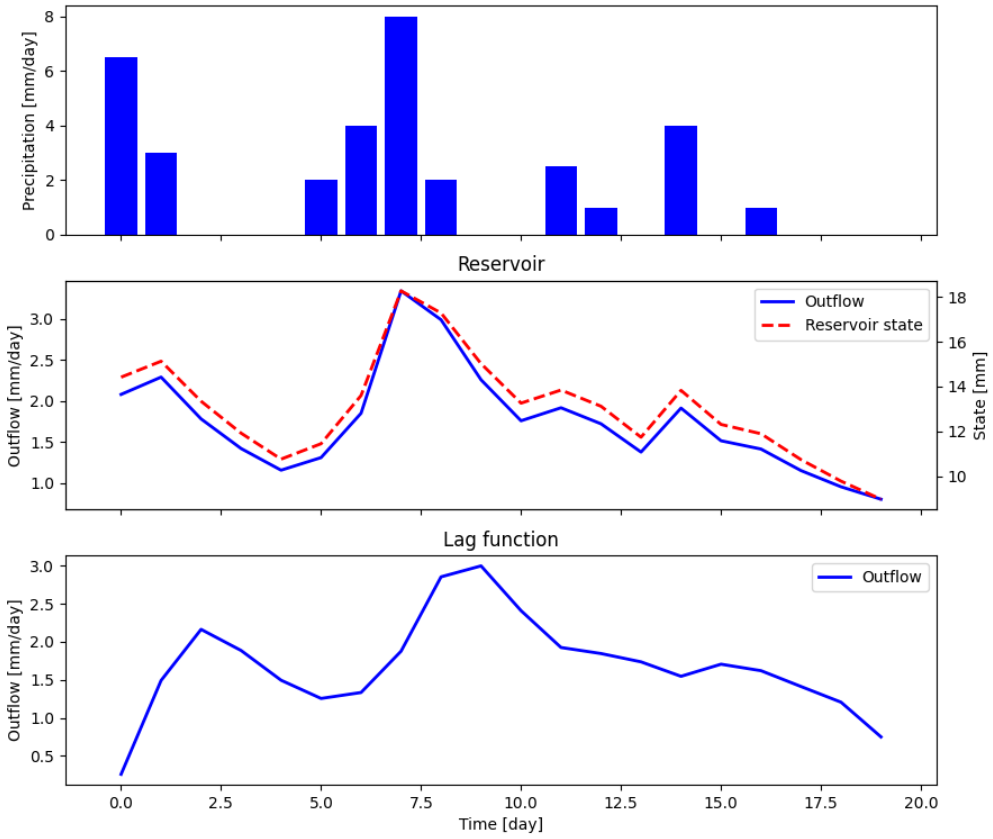
In this code, the unit will call the `get_output` method of all its elements (from upstream to downstream), set the inputs of the downstream elements to the output of their respective upstream elements, and return the output of the last element.

After the unit simulation has completed, the outputs and the states of its elements can be retrieved as follows

```
r_state = unit_1.get_internal(id='R', attribute='state_array')[:, 0]
r_output = unit_1.call_internal(id='R', method='get_output', solve=False)[0]
```

Note that in line 2 we pass the argument `solve=False` to the function `get_output`, in order to access the computed states and outputs without re-running the reservoir element.

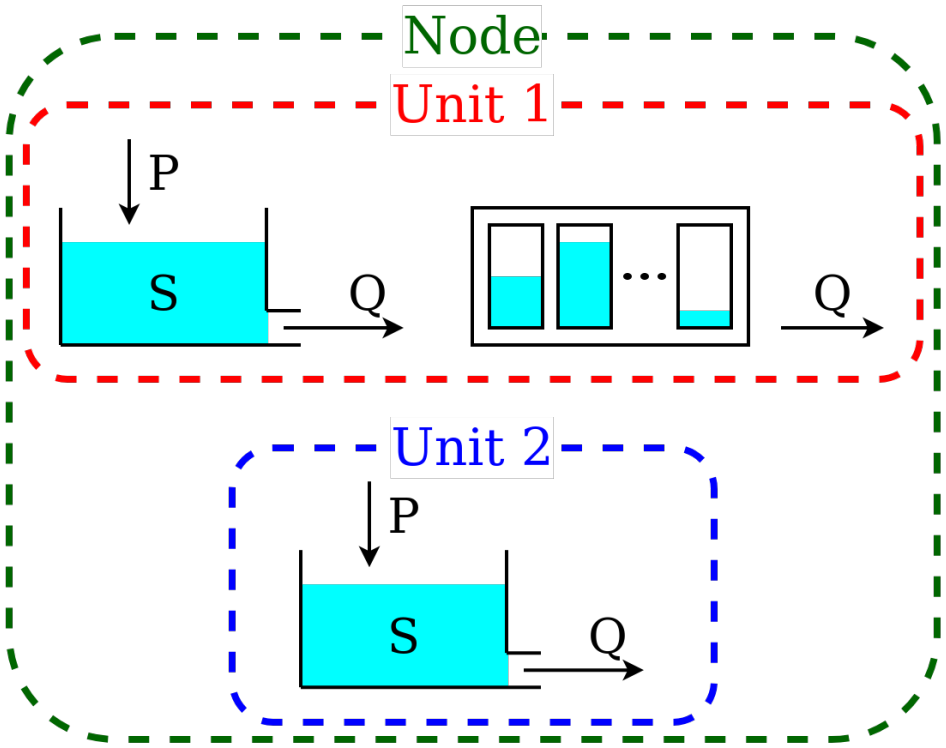
The plot shows the output of the simulation (obtained by plotting `output`, `r_state`, and `r_output`).



If we wish to re-run the model, the elements of the unit can be re-set to their initial state

```
unit_1.reset_states()
```

6.4 Simple semi-distributed model



This model is intended to represent a spatially semi-distributed configuration. A node is used to represent a catchment with multiple areas that react differently to the same inputs. In this example, we represent 70% of the catchment using the structure described in *Lumped model structure with 2 elements*, and the remaining 30% using a single reservoir.

This model configuration is achieved using a node with multiple units.

First, we initialize the two units and the elements composing them, in the same way as in the previous sections.

```
reservoir = PowerReservoir(
    parameters={'k': 0.01, 'alpha': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
    id='R'
)

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

unit_1 = Unit(
```

(continues on next page)

(continued from previous page)

```
    layers=[[reservoir], [lag_function]],
    id='unit-1'
)

unit_2 = Unit(
    layers=[[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 contained in `unit_2` (see *Unit*).

The next step is to initialize the node, which combines the two units

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

Line 2 contains the list of units that belong to the node, and line 3 gives their weight (i.e. the portion of the node outflow coming from each unit). Line 4 specifies the representative area of the node.

Next, we define the time step size and the model inputs

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

The same time step size will be assigned to all elements within the node, and the inputs will be passed to all the units of the node.

We can now run the node and collect its output

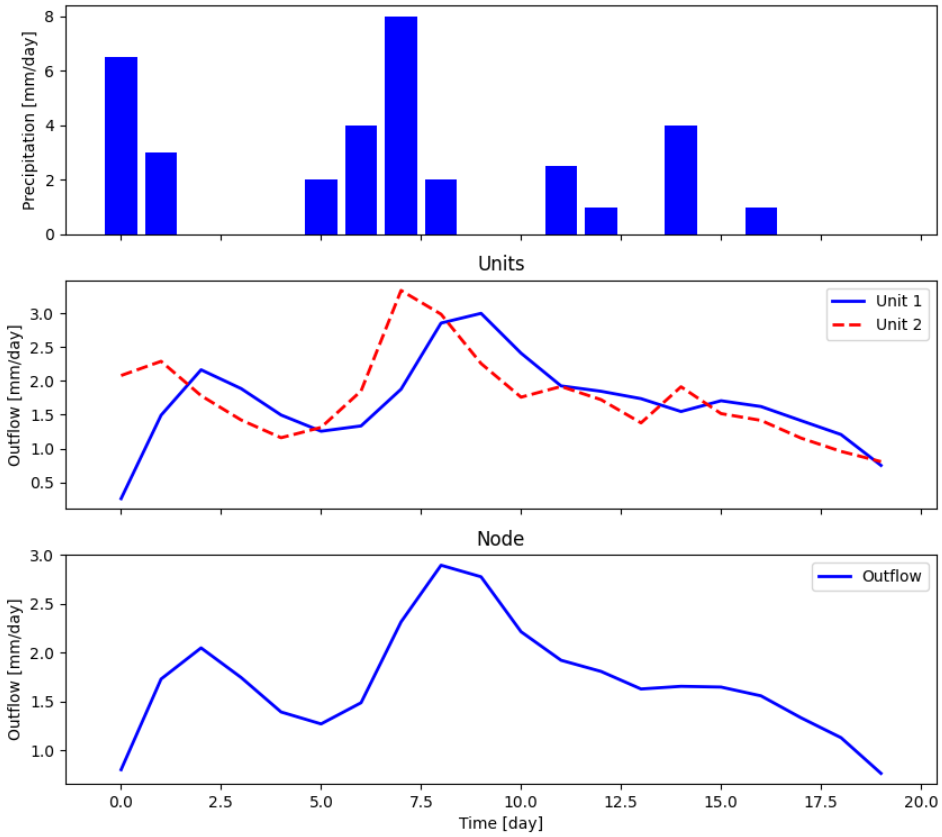
```
output = node_1.get_output()[0]
```

The node will call the method `get_output` of all its units and aggregate their outputs using the weights.

The outputs of the single units, as well as the states and fluxes of the elements composing them, can be retrieved using the method `call_internal`

```
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]
```

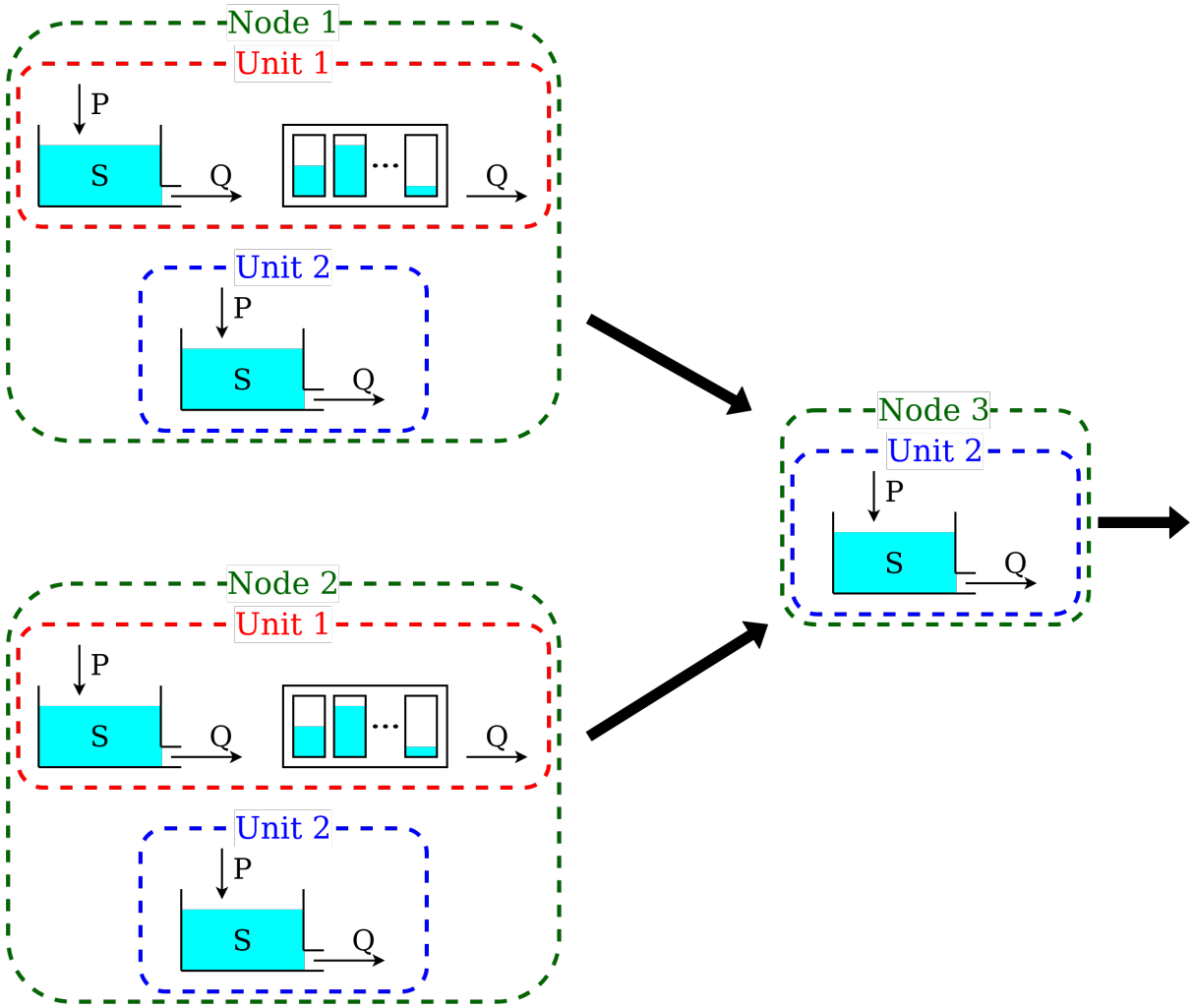
The plot shows the output of the simulation.



All elements within the node can be re-set to their initial states

```
node_1.reset_states()
```

6.5 Semi-distributed model with multiple nodes



A catchment can be composed by several subcatchments (nodes) connected in a network. Each subcatchment receives its own inputs, but may share parameter values with other subcatchments with the same units.

This semi-distributed configuration can be implemented in SuperflexPy by creating a network with multiple nodes.

First, we initialize the nodes

```
reservoir = PowerReservoir(
    parameters={'k': 0.01, 'alpha': 2.0},
    states={'S0': 10.0},
    approximation=approximator,
    id='R'
)

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

(continues on next page)

(continued from previous page)

```
13
14 unit_1 = Unit(
15     layers=[[reservoir], [lag_function]],
16     id='unit-1'
17 )
18
19 unit_2 = Unit(
20     layers=[[reservoir]],
21     id='unit-2'
22 )
23
24 node_1 = Node(
25     units=[unit_1, unit_2],
26     weights=[0.7, 0.3],
27     area=10.0,
28     id='node-1'
29 )
30
31 node_2 = Node(
32     units=[unit_1, unit_2],
33     weights=[0.3, 0.7],
34     area=5.0,
35     id='node-2'
36 )
37
38 node_3 = Node(
39     units=[unit_2],
40     weights=[1.0],
41     area=3.0,
42     id='node-3'
43 )
```

Here, nodes `node_1` and `node_2` contain both units, `unit_1` and `unit_2`, but in different proportions. Node `node_3` contains only a single unit, `unit_2`.

When units are added to a node, the states of the elements within the units remain independent while the parameters stay linked. In this example the change of a parameter in `unit_1` in `node_1` is applied also to `unit_1` in `node_2`. This "shared parameters" behavior can be disabled by setting the parameter `shared_parameters` to `False` when initializing the nodes (see *Node*)

The network is initialized as follows

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

Line 2 provides the list of the nodes belonging to the network. Lines 4-6 define the connectivity of the network; this is done using a dictionary with the keys given by the node identifiers and values given by the single downstream node. The most downstream node has, by convention, its value set to `None`.

The inputs are catchment-specific and must be provided to each node.

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

The time step size is defined at the network level.

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

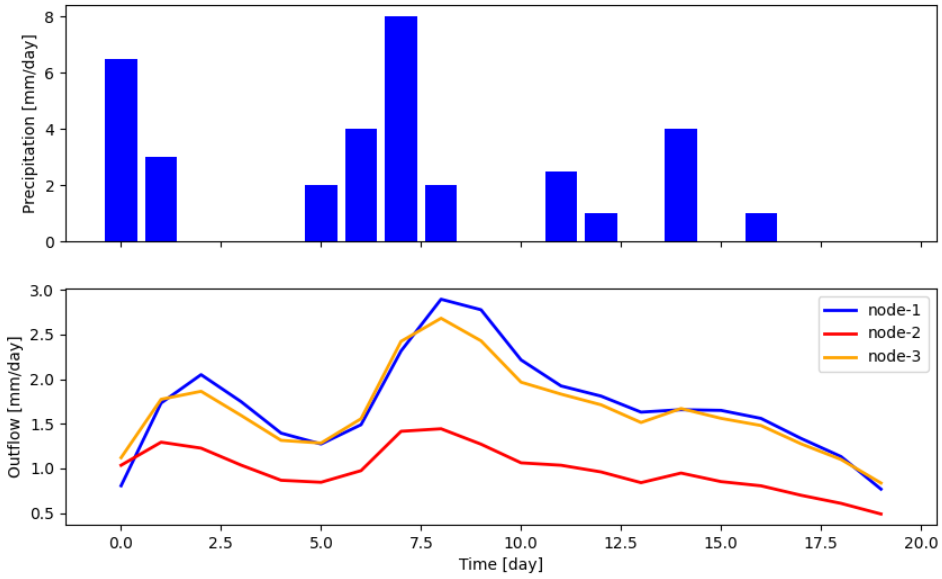
We can now run the network and get the output values

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

The network runs the nodes from upstream to downstream, collects their outputs, and routes them to the outlet. The output of the network is a dictionary, with keys given by the node identifiers and values given by the list of output fluxes of the nodes. It is also possible to retrieve the internals (e.g. fluxes, states, etc.) of the nodes.

```
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.



CHAPTER

SEVEN

LIST OF CURRENTLY IMPLEMENTED ELEMENTS

SuperflexPy provides four levels of components (elements, units, nodes and network) for constructing conceptual hydrological models. The components presented in the page *Organization of SuperflexPy* represent the core of SuperflexPy. These components can be extended to create customized models.

Most of the customization efforts will be required for elements (i.e., reservoirs, lag, and connection elements). This page describes all elements that have been created and shared by the community of SuperflexPy. These elements can be used to construct a wide range of model structures.

This section lists elements according to their type, namely

- Reservoir
- Lag elements
- Connections

Within each section, the elements are listed in alphabetical order.

7.1 Reservoirs

7.1.1 Interception filter

This reservoir is used to simulate interception in models, including GR4J. Further details are provided in the page *GR4J*.

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

Inputs

- Potential evapotranspiration $E_{\text{POT}}^{\text{in}}$ $[LT^{-1}]$
- Precipitation P^{in} $[LT^{-1}]$

Outputs from `get_output`

- Net potential evapotranspiration $E_{\text{POT}}^{\text{out}}$ $[LT^{-1}]$
- Net precipitation P^{out} $[LT^{-1}]$

Governing equations

$$\text{if } P^{\text{in}} > E^{\text{in}}_{\text{POT}}:$$
$$P^{\text{out}} = P^{\text{in}} - E^{\text{in}}_{\text{POT}}$$
$$E^{\text{out}}_{\text{POT}} = 0$$

$$\text{if } P^{\text{in}} < E^{\text{in}}_{\text{POT}}:$$
$$P^{\text{out}} = 0$$
$$E^{\text{out}}_{\text{POT}} = E^{\text{in}}_{\text{POT}} - P^{\text{in}}$$

7.1.2 Linear reservoir

This reservoir assumes a linear storage-discharge relationship. It represents arguably the simplest hydrological model. For example, it is used in the model HYMOD to simulate channel routing and lower-zone storage processes. Further details are provided in the page *HYMOD*.

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

Inputs

- Precipitation P $[LT^{-1}]$

Outputs from `get_output`

- Total outflow Q $[LT^{-1}]$

Governing equations

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

7.1.3 Power reservoir

This reservoir assumes that the storage-discharge relationship is described by a power function. This type of reservoir is common in hydrological models. For example, it is used in the HBV family of models to represent the fast response of a catchment.

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

Inputs

- Precipitation P $[LT^{-1}]$

Outputs from `get_output`

- Total outflow Q $[LT^{-1}]$

Governing equations

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

7.1.4 Production store (GR4J)

This reservoir is used to simulate runoff generation in the model GR4J. Further details are provided in the page *GR4J*.

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

Inputs

- Potential evapotranspiration E_{pot} $[LT^{-1}]$
- Precipitation P $[LT^{-1}]$

Outputs from `get_output`

- Total outflow P_{r} $[LT^{-1}]$

Secondary outputs

- Actual evapotranspiration E_{act} $[LT^{-1}]$ `get_aet()`

Governing equations

$$\frac{\mathrm{d}S}{\mathrm{d}t} = P_{\text{s}} - E_{\text{act}} - Q_{\text{perc}}$$
$$P_{\text{s}} = P\left(1 - \left(\frac{S}{x_1}\right)^{\alpha}\right)$$
$$E_{\text{act}} = E_{\text{pot}}\left(2\frac{S}{x_1} - \left(\frac{S}{x_1}\right)^{\alpha}\right)$$
$$Q_{\text{perc}} = \frac{x^{1-\beta}}{(\beta-1)}\nu^{\beta-1}S^{\beta}$$
$$P_{\text{r}} = P - P_{\text{s}} + Q_{\text{perc}}$$

7.1.5 Routing store (GR4J)

This reservoir is used to simulate routing in the model GR4J. Further details are provided in the page *GR4J*.

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

Inputs

- Precipitation P $[LT^{-1}]$

Outputs from `get_output`

- Outflow Q $[LT^{-1}]$
- Loss term F $[LT^{-1}]$

Governing equations

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

7.1.6 Snow reservoir

This reservoir is used to simulate snow processes based on temperature. Further details are provided in the section *Dal Molin et al., 2020, HESS*.

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

Inputs

- Precipitation P $[LT^{-1}]$
- Temperature T $[C]$

Outputs from `get_output`

- Sum of snow melt and rainfall input $= P - P_{\text{snow}} + M$ $[LT^{-1}]$

Governing equations

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

7.1.7 Unsaturated reservoir (inspired to HBV)

This reservoir specifies the actual evapotranspiration as a smoothed threshold function of storage, in combination with the storage-discharge relationship being set to a power function. It is inspired by the HBV family of models, where a similar approach (but without smoothing) is used to represent unsaturated soil dynamics.

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

Inputs

- Precipitation P $[LT^{-1}]$
- Potential evapotranspiration E_{pot} $[LT^{-1}]$

Outputs from `get_output`

- Total outflow Q $[LT^{-1}]$

Secondary outputs

- Actual evapotranspiration E_{act} `get_AET()`

Governing equations

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

7.1.8 Upper zone (Hymod)

This reservoir is part of the HYMOD model and is used to simulate the upper soil zone. Further details are provided in the page *HYMOD*.

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

Inputs

- Precipitation P $[LT^{-1}]$
- Potential evapotranspiration E_{pot} $[LT^{-1}]$

Outputs from `get_output`

- Total outflow Q $[LT^{-1}]$

Secondary outputs

- Actual evapotranspiration E_{act} $[LT^{-1}]$ `get_AET()`

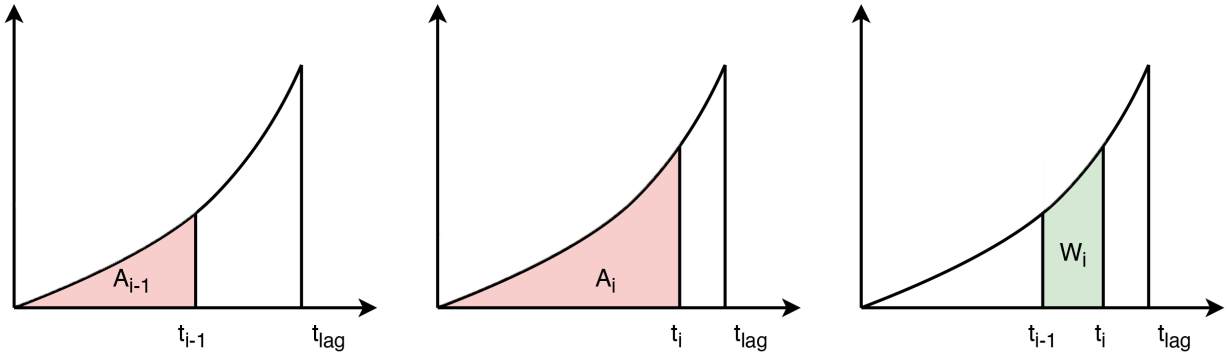
Governing equations

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

7.2 Lag elements

All lag elements implemented in SuperflexPy can accommodate an arbitrary number of input fluxes, and apply a convolution based on a weight array that defines the shape of the lag function.

Lag elements differ solely in the definition of the weight array. The nature (i.e., number and order) of inputs and outputs depend on the element upstream of the lag element.



The weight array can be defined by giving the area below the lag function as a function of the time coordinate. The maximum lag t_{lag} must also be specified. The weights are then given by differences between the values of the area at consecutive lags. This approach is shown in the figure above, where the weight W_i is calculated as the difference between areas A_i and A_{i-1}.

7.2.1 Half triangular lag

This lag element implements the element present in the case study *Dal Molin et al., 2020, HESS* and used in other Superflex studies.

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

Definition of weight array

The area below the lag function is given by

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

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

$$A_{\text{lag}}(t) = 1 \quad \text{for } t > t_{\text{lag}}$$

The weight array is then calculated as

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

7.2.2 Unit hydrograph 1 (GR4J)

This lag element implements the unit hydrograph 1 of *GR4J*.

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

Definition of weight array

The area below the lag function is given by

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

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

$$A_{\text{lag}}(t) = 1 \quad \text{for } t > t_{\text{lag}}$$

The weight array is then calculated as

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

7.2.3 Unit hydrograph 2 (GR4J)

This lag element implements the unit hydrograph 2 of *GR4J*.

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

Definition of weight array

The area below the lag function is given by

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

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

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

$$A_{\text{lag}}(t) = 1 \quad \text{for } t > t_{\text{lag}}$$

The weight array is then calculated as

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

7.3 Connections

SuperflexPy implements four connection elements:

- splitter
- junction
- linker
- transparent element

In addition, customized connectors have been implemented to achieve specific model designs. These customized elements are listed in this section.

7.3.1 Flux aggregator (GR4J)

This element is used to combine routing, exchange and outflow fluxes in the GR4J model. Further details are provided in the page *GR4J*.

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

Inputs

- Outflow routing store Q_{RR} $[LT^{-1}]$
- Exchange flux Q_{RF} $[LT^{-1}]$
- Outflow UH2 Q_{UH2} $[LT^{-1}]$

Main outputs

- Outflow Q $[LT^{-1}]$

Governing equations

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

CHAPTER

EIGHT

EXPAND SUPERFLEXPY: BUILD CUSTOMIZED ELEMENTS

Note: If you build your own SuperflexPy element, we would appreciate if you share your implementation with the community (see *Software organization and contribution*). Please remember to update the *List of currently implemented elements* page to make other users aware of your implementation.

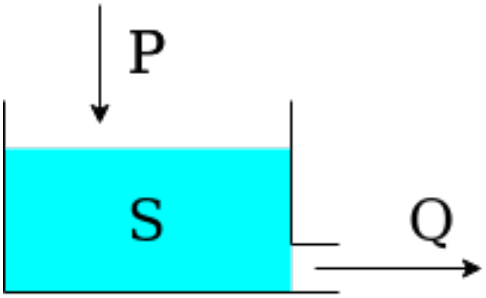
This page illustrates how to create customized elements using the SuperflexPy framework.

The examples include three elements:

- Linear reservoir
- Half-triangular lag function
- Parameterized splitter

The customized elements presented here are relatively simple, in order to provide a clear illustration of the programming approach. To gain a deeper understanding of SuperflexPy functionalities, please familiarize with the code of existing elements (importing "path" `superflexpy.implementation.elements`).

8.1 Linear reservoir



The reservoir is controlled by the following differential equation

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

with

$$Q = kS$$

Note that the differential equation can be solved analytically, and (if applied) the implicit Euler numerical approximation does not require iteration. However, we will still use the numerical approximator offered by SuperflexPy

(see *Numerical implementation*) to illustrate how to proceed in a more general case where analytical solutions are not available.

The SuperflexPy framework provides the class `ODEsElement`, which has most of the methods required to implement the linear reservoir element. The class implementing the reservoir will inherit from `ODEsElement` and implement only a few methods needed to specify its behavior.

```
1 import numba as nb
2 from superflexpy.framework.element import ODEsElement
3
4 class LinearReservoir(ODEsElement):
```

The first method to implement is the class initializer `__init__`

```
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)
```

In the context of SuperflexPy, the main purpose of the method `__init__` (lines 9-16) is to deal with the numerical solver. In this case we can accept two architectures: pure Python or Numba. The option selected will control the method used to calculate the fluxes. Note that, since some methods of the approximator may require the Python implementation of the fluxes, the Python implementation must be always provided.

The second method to implement is `set_input`, which converts the (ordered) list of input fluxes to a dictionary that gives a name to these fluxes.

```
1     def set_input(self, input):
2
3         self.input = {'P': input[0]}
```

Note that the key used to identify the input flux must match the name of the corresponding variable in the flux functions.

The third method to implement is `get_output`, which runs the model and returns the output flux.

```
1     def get_output(self, solve=True):
2
3         if solve:
4             self._solver_states = [self._states[self._prefix_states + 'S0']]
5             self._solve_differential_equation()
6
```

(continues on next page)

(continued from previous page)

```
            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},
                                          )
        return [- fluxes[0][1]]
```

The method receives, as input, the argument `solve`: if `False`, the state array of the reservoir will not be recalculated and the outputs will be computed based on the current state (e.g., computed in a previous run of the reservoir). This option is needed for post-run inspection, when we want to check the output of the reservoir without solving it again.

Line 4 transforms the states dictionary to an ordered list. Line 5 calls the built-in ODE solver. Line 7 updates the state of the model to the last value calculated. Lines 9-14 call the external numerical approximator to get the values of the fluxes. Note that, for this operation, the Python implementation of the fluxes method is always used because the vectorized operation executed by the method `get_fluxes` of the numerical approximator does not benefit from the Numba optimization.

The last methods to implement are `_fluxes_function_python` (pure Python) and `_fluxes_function_numba` (Numba optimized), which calculate the reservoir fluxes.

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

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

        return (
            (
                P[ind],
                - k[ind] * S,
```

(continues on next page)

(continued from previous page)

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

`_fluxes_function_python` and `_fluxes_function_numba` are both private static methods.

Their inputs are: the state used to compute the fluxes (`S`), the initial state (`S0`), the index to use in the arrays (`ind`), the input fluxes (`P`), and the parameters (`k`). All inputs are arrays and `ind` is used, when solving for a single time step, to indicate the time step to look for.

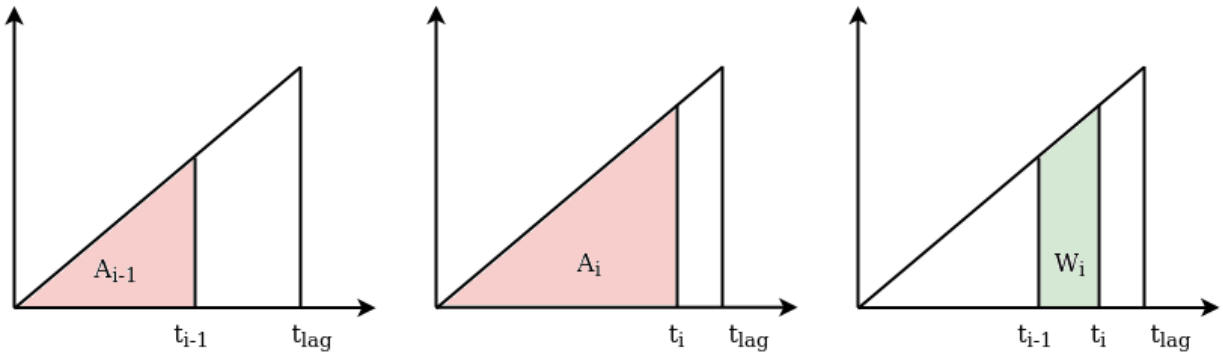
The output is a tuple containing three elements:

- tuple with the computed values of the fluxes; positive sign for incoming fluxes (e.g. precipitation, `P`), negative sign for outgoing fluxes (e.g. streamflow, `- k * S`);
- lower bound for the search of the state;
- upper bound for the search of the state;

The implementation for the Numba solver differs from the pure Python implementation in two aspects:

- the usage of the Numba decorator that defines the types of input variables (lines 24-25)
- the method works only for a single time step and not for the vectorized solution. For the vectorized solution the Python implementation (with Numpy) is considered sufficient, and hence a Numba implementation is not pursued.

8.2 Half-triangular lag function



The half-triangular lag function grows linearly until t_{lag} and then drops to zero, as shown in the schematic. The slope α is determined from the constraint that the total area of the triangle must be equal to 1.

$$f_{\text{lag}} = \alpha t$$
$$\text{for } t \leq t_{\text{lag}}$$
$$f_{\text{lag}} = 0$$
$$\text{for } t > t_{\text{lag}}$$

SuperflexPy provides the class `LagElement` that contains most of the functionalities needed to calculate the output of a lag function. The class implementing a customized lag function will inherit from `LagElement`, and implement only the methods needed to compute the weight array.

```
import numpy as np

class TriangularLag(LagElement):
```

The only method requiring implementation is the private method used to calculate the `weight` array.

```
    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
```

The method `_build_weight` makes use of a secondary private static method

```
    @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 area A_i of the red triangle in the figure, which has base t_i (`bin`). The method `_build_weight` uses this function to calculate the weight array W_i, as the difference between A_i and A_{i-1}.

Note that the method `_build_weight` can be implemented using other approaches, e.g., without using auxiliary methods.

8.3 Parameterized splitter

A splitter is an element that takes the flux from an upstream element and distributes it to feed multiple downstream elements. The element is controlled by parameters that define the portions of the flux that go into specific elements.

The simple case that we consider here has a single input flux that is split to two downstream elements. In this case, the splitter requires only one parameter α_{split}. The fluxes to the downstream elements are

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

SuperflexPy provides the class `ParameterizedElement`, which can be extended to implement all elements that are controlled by parameters but do not have a state. The class implementing the parameterized splitter will inherit from `ParameterizedElement` and implement only the methods required for the new functionality.

```
from superflexpy.framework.element import ParameterizedElement

class ParameterizedSingleFluxSplitter(ParameterizedElement):
```

First, we define two private attributes that specify the number of upstream and downstream elements of the splitter. This information is used by the unit when constructing the model structure.

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

We then define the method that receives the inputs, and the method that calculates the outputs.

```
    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 as the ones implemented as part of the earlier *Linear reservoir* example. Note that, in this case, the argument `solve` of the method `get_output` is not used, but is still required to maintain a consistent interface.

CHAPTER

NINE

EXPAND SUPERFLEXPY: BUILD CUSTOMIZED COMPONENTS

9.1 Adding routing to a node

Nodes in SuperflexPy have the capability to apply a lag to the fluxes simulated by the units. Such lags can represent routing delays in the fluxes as they propagate through the catchment ("internal" routing), or routing delays associated with the river network ("external" routing). Both types of routing can be implemented within a SuperflexPy node.

The default implementation of the node (`Node` class in `superflexpy.framework.node`) does not provide the routing functionality. The methods `_internal_routing` and `external_routing` exist but are set to simply return the incoming fluxes without any transformation.

To support routing within a node, we need to create a customized node that implements the methods `_internal_routing` and `external_routing` for given lag functions. The object-oriented design of SuperflexPy simplifies this operation, because the new node class inherits all the methods from the original class, and has to overwrite only the two methods that are responsible for the routing.

In this example, we illustrate an implementation of routing with a lag function in the shape of an isosceles triangle with base `t_internal` and `t_external`, for internal and external routing respectively. This new implementation is similar to the implementation of the *Half-triangular lag function*.

The first step is to import the `Node` component from SuperflexPy and define the class `RoutedNode`

```
from superflexpy.framework.node import Node

class RoutedNone(Node):
```

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

```
    def _internal_routing(self, flux):

        t_internal = self.get_parameters(names=[self._prefix_local_parameters + 't_
↪internal'])
        flux_out = []

        for f in flux:
            flux_out.append(self._route(f, t_internal))

        return flux_out

    def external_routing(self, flux):

        t_external = self.get_parameters(names=[self._prefix_local_parameters + 't_
↪external'])
```

(continues on next page)

(continued from previous page)

```
14          flux_out = []
15
16          for f in flux:
17              flux_out.append(self._route(f, t_external))
18
19          return flux_out
```

In this simple example, the two routing mechanisms are handled using the same lag functional form. Hence, the methods `_internal_routing` and `external_routing` take advantage of the method `_route` (line 7 and 17).

The method `_route` is implemented as follows

```
1       def _route(self, flux, time):
2
3           state = np.zeros(int(np.ceil(time)))
4           weight = self._calculate_weight(time)
5
6           out = []
7           for value in flux:
8               state = state + weight * value
9               out.append(state[0])
10              state[0] = 0
11              state = np.roll(state, shift=-1)
12
13          return np.array(out)
14
15      def _calculate_weight(self, time):
16
17          weight = []
18
19          array_length = np.ceil(time)
20
21          for i in range(int(array_length)):
22              weight.append(self._calculate_lag_area(i + 1, time)
23                            - self._calculate_lag_area(i, time))
24
25          return weight
26
27      @staticmethod
28      def _calculate_lag_area(portion, time):
29
30          half_time = time / 2
31
32          if portion <= 0:
33              value = 0
34          elif portion < half_time:
35              value = 2 * (portion / time) ** 2
36          elif portion < time:
37              value = 1 - 2 * ((time - portion) / time)**2
38          else:
39              value = 1
40
41          return value
```

Note that the code in this block is similar to the code implemented in *Half-triangular lag function*. The methods in this last code block are "support" methods that make the code more organized and easier to maintain. The same

numerical results can be obtained by moving the functionality of these methods directly into `_internal_routing` and `external_routing`, though the resulting code would be less modular.

CHAPTER

TEN

APPLICATION: IMPLEMENTATION OF EXISTING CONCEPTUAL MODELS

This page describes the SuperflexPy implementation of several existing conceptual hydrological models. The "translation" of a model into SuperflexPy requires the following steps:

1. Design of a structure that reflects the original model but satisfies the requirements of SuperflexPy (e.g. does not contain mutual interaction between elements, see *Unit*);
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 step 2

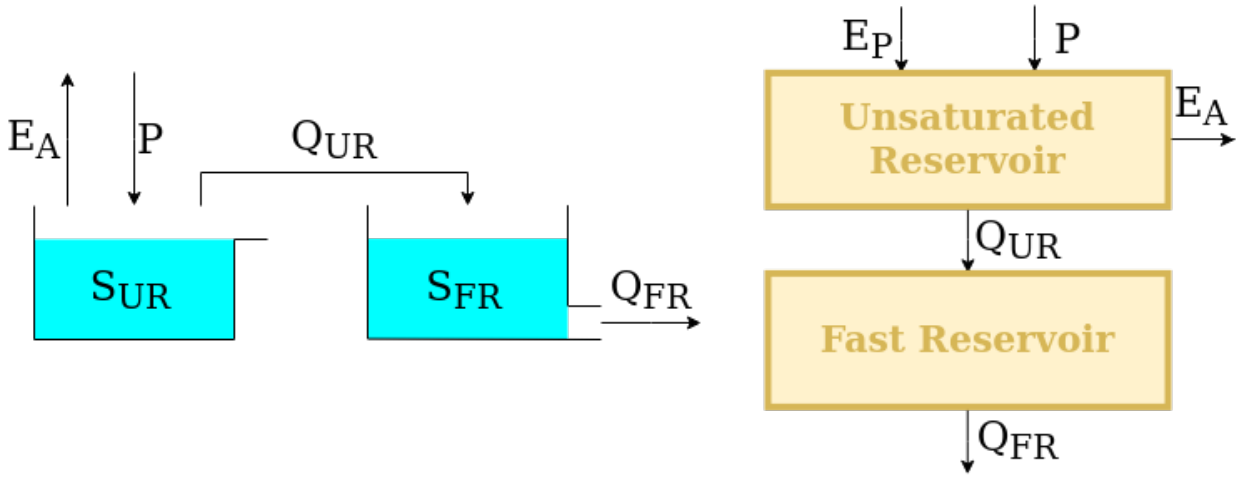
10.1 Model M4 from Kavetski and Fenicia, WRR, 2011

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

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

10.1.1 Design of the model structure

M4 has a simple structure that can be implemented in SuperflexPy without using connection elements. The figure shows, on the left, the structure as presented in the original M4 publication; on the right, a schematic of the SuperflexPy implementation is shown.



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

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

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

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

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

10.1.2 Element creation

We now show how to use SuperflexPy to implement the elements described in the previous section. A detailed explanation of how to use the framework to build new elements can be found in the page *Expand SuperflexPy: Build customized elements*.

Note that, most of the times, when implementing a model structure with SuperflexPym the elements have already been implemented in SuperflexPy and, therefore, the modeller does not need to implement them. A list of the currently implemented elements is provided in the page *List of currently implemented elements*.

Unsaturated reservoir

This element can be implemented by extending the class `ODEsElement`.

```
1  class UnsaturatedReservoir(ODEsElement):
2
3      def __init__(self, parameters, states, approximation, id):
4
5          ODEsElement.__init__(self,
6                               parameters=parameters,
7                               states=states,
```

(continues on next page)

(continued from previous page)

```
                                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,
                                          dt=self._dt,
                                          **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,
                                          dt=self._dt,
                                          **self.input,
                                          **{k[len(self._prefix_parameters):]: self._
↪parameters[k] for k in self._parameters},
                                          )
        return [- fluxes[0][1]]

    # PROTECTED METHODS

```

(continues on next page)

(continued from previous page)

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

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

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

        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] * dt[ind]
        )
```

Fast reservoir

This element can be implemented by extending the class `ODEsElement`.

```
class PowerReservoir(ODEsElement):

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

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

(continues on next page)

(continued from previous page)

```
        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,
                                          dt=self._dt,
                                          **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, alpha, dt):

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

(continues on next page)

(continued from previous page)

```
                S0 + P[ind] * dt[ind]
            )

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

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

10.1.3 Model initialization

Now that all elements are implemented, they can be combined to build the model structure. For details refer to *How to build a model with SuperflexPy*.

First, we initialize all 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 = PowerReservoir(
    parameters={'k': 0.1, 'alpha': 1.0},
    states={'S0': 10.0},
    approximation=numeric_approximator,
    id='FR'
)
```

Next, 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'
)
```

10.2 GR4J

GR4J is a widely used conceptual hydrological model introduced in the article

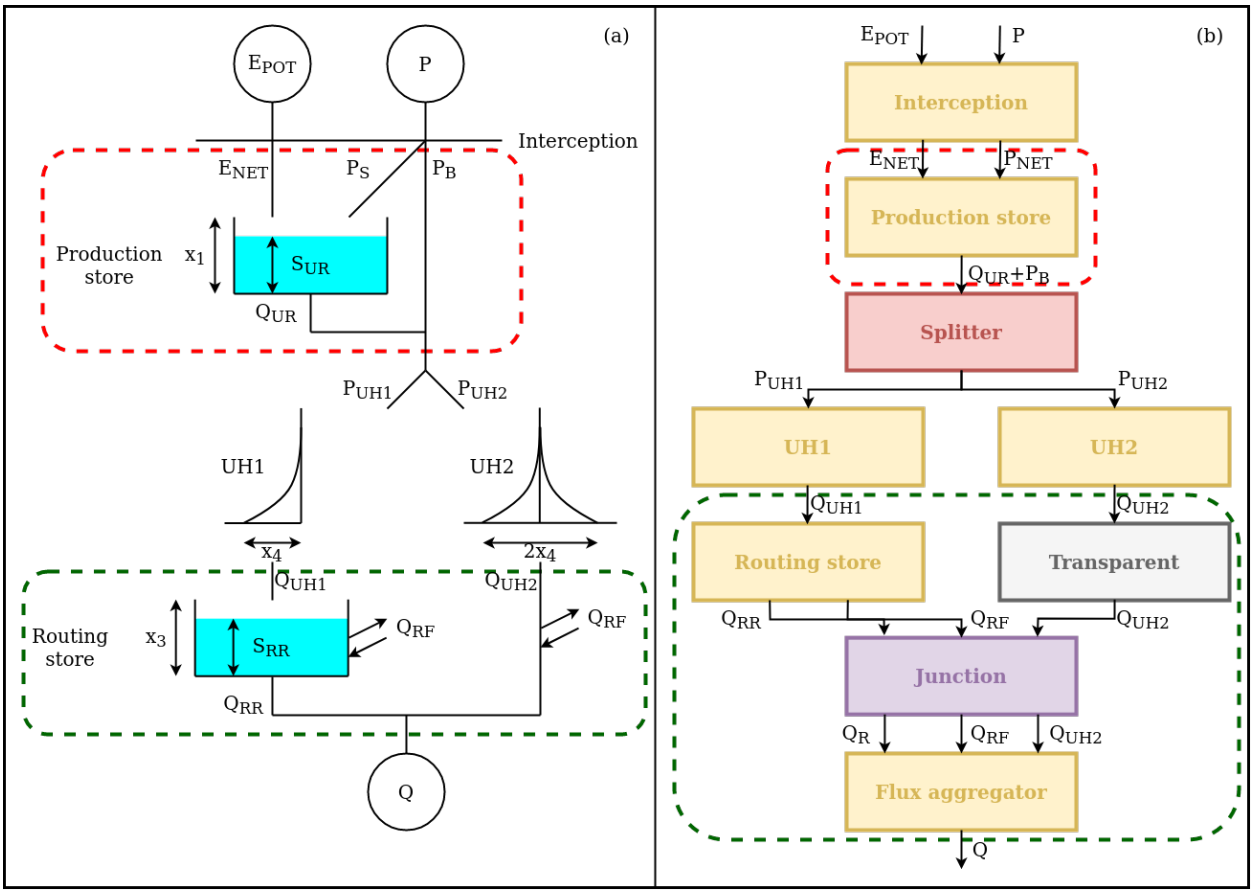
Perrin, C., Michel, C., and Andréassian, V.: **Improvement of a parsimonious model for streamflow simulation**, Journal of Hydrology, 279, 275-289, https://doi.org/10.1016/S0022-1694(03)00225-7, 2003.

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.

10.2.1 Design of the model structure

The figure shows, on the left, the model structure as presented in Perrin et al., 2003; on the right, the adaptation to SuperflexPy is shown.



The potential evaporation and the precipitation are "filtered" by an interception element, that calculates the net fluxes by setting the smallest to zero and the largest to the difference between the two fluxes.

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

This element is implemented in SuperflexPy using the "interception filter".

After the interception filter, the SuperflexPy implementation starts to differ from the original. In the original implementation of GR4J, the precipitation is split between a part P_s that flows into the production store and the remaining part P_b that bypasses the reservoir. P_s and P_b are both functions of the state of the reservoir

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

When we implement this part of the model in SuperflexPy, these two fluxes cannot be calculated before solving the reservoir (due to the representation of the *Unit* as a succession of layers).

To solve this problem, in the SuperflexPy implementation of GR4J, all precipitation (and not only P_s) flows into an element that incorporates the production store. This element takes care of dividing the precipitation internally, while solving the differential equation

$$\frac{dS_{UR}}{dt} = P_{NET}\left(1 - \left(\frac{S_{UR}}{x_1}\right)^{\alpha}\right) - E_{NET}\left(2\frac{S_{UR}}{x_1} - \left(\frac{S_{UR}}{x_1}\right)^{\alpha}\right) - \frac{x_1^{1-\beta}}{(\beta-1)}\nu^{\beta-1}S_{UR}^{\beta}$$

where the first term is the precipitation P_s, the second term is the actual evaporation, and the third term represents the output of the reservoir, which here corresponds to "percolation".

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

The flux is then divided between two lag functions, referred to as "unit hydrographs" and abbreviated UH: 90% of the flux goes to UH1 and 10% goes to UH2. In this part of the model structure the correspondence between the elements of GR4J and their SuperflexPy implementation is quite clear.

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

$$\frac{dS_{RR}}{dt} = Q_{UH1} - \frac{x_3^{1-\gamma}}{(\gamma-1)}S_{RR}^{\gamma} - \frac{x_2}{x_3^{\omega}}S_{RR}^{\omega}$$

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 also subtracted from the output of UH2. In SuperflexPy, this operation cannot be done in the same unit layer as the solution of the routing store, and instead it is done afterwards. For this reason, the SuperflexPy implementation of GR4J has an additional element (called "flux aggregator") that uses a junction element to combine the output of the routing store, the gain/loss term, and the output of UH2. The flux aggregator then computes the outflow of the model using the equation

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

10.2.2 Elements creation

We now show how to use SuperflexPy to implement the elements described in the previous section. A detailed explanation of how to use the framework to build new elements can be found in the page *Expand SuperflexPy: Build customized elements*.

Note that, most of the times, when implementing a model structure with SuperflexPym the elements have already been implemented in SuperflexPy and, therefore, the modeller does not need to implement them. A list of the currently implemented elements is provided in the page *List of currently implemented elements*.

Interception

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

```
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, can be constructed by extending the class `ODEsElement`

```
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()

```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
                    - ((x1[ind]**(1 - beta[ind])) / ((beta[ind] - 1))) *␣
↪(ni[ind]**(beta[ind] - 1)) * (S**beta[ind])  # Perc
                ],
                0.0,
                S0 + P[ind] * (1 - (S / x1[ind])**alpha[ind]) * dt[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))) *␣
↪(ni[ind]**(beta[ind] - 1)) * (S**beta[ind])  # Perc
            ),
            0.0,
            S0 + P[ind] * (1 - (S / x1[ind])**alpha[ind]) * dt[ind]
        )
```

Unit hydrographs

The unit hydrographs are an extension of the `LagElement`, and can be implemented as follows

```
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
```

(continues on next page)

(continued from previous page)

```
        else:
            value = 1
        return value
```

```
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))

        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]
```

(continues on next page)

(continued from previous page)

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

```

(continues on next page)

(continued from previous page)

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

Flux aggregator

The flux aggregator can be implemented by extending a `BaseElement`

```
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'])]
```

10.2.3 Model initialization

Now that all elements are implemented, we can combine them to build the model structure. For details refer to *How to build a model with SuperflexPy*.

First, we initialize all elements.

```
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},
```

(continues on next page)

(continued from previous page)

```
                                    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')

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

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

The elements are then put together to define a `Unit` that reflects the GR4J structure presented in the figure.

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

10.3 HYMOD

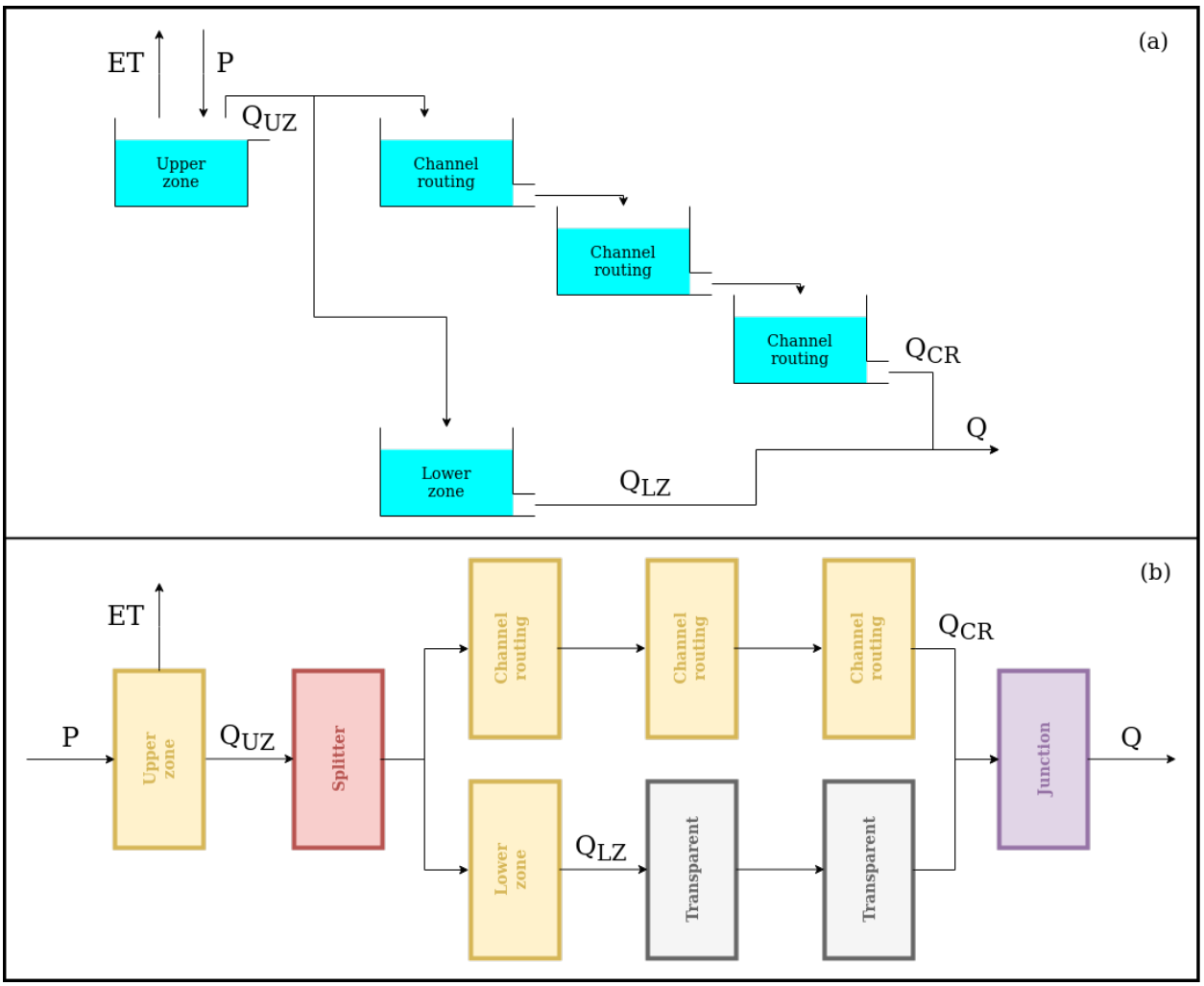
HYMOD is another widely used conceptual hydrological model. It was first published in

> 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.

10.3.1 Design of the structure



HYMOD comprises three groups of reservoirs intended to represent, respectively, the upper zone (soil dynamics), channel routing (surface runoff), and lower zone (subsurface flow).

As can be seen in the figure, the original structure of HYMOD already meets the design constrains of SuperflexPy (it does not contains feedbacks between elements). Therefore the SuperflexPy implementation of HYMOD is more straightforward than for GR4J.

The upper zone is a reservoir intended to represent streamflow generation processes and evaporation. It is controlled by the differential equation

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

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

The outflow from the reservoir is then split between the channel routing (3 reservoirs) and the lower zone (1 reservoir). All these elements are represented by linear reservoirs controlled by the differential equation

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

where the first term is the input (here, the outflow from the upstream element) and the second term represents the outflow from the reservoir.

Channel routing and lower zone differ from each other in the number of reservoirs used (3 in the first case and 1 in the second), and in the value of the parameter k, which controls the outflow rate. Based on intended model operation, k should have a larger value for channel routing because this element is intended to represent faster processes.

The outputs of these two flowpaths are collected by a junction, which generates the final model output.

Comparing the two panels in the figure, the only difference is the presence of the two transparent elements that are needed to fill the "gaps" in the SuperflexPy structure (see *Unit*).

10.3.2 Elements creation

We now show how to use SuperflexPy to implement the elements described in the previous section. A detailed explanation of how to use the framework to build new elements can be found in the page *Expand SuperflexPy: Build customized elements*.

Note that, most of the times, when implementing a model structure with SuperflexPym the elements have already been implemented in SuperflexPy and, therefore, the modeller does not need to implement them. A list of the currently implemented elements is provided in the page *List of currently implemented elements*.

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., 1) 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

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

Note that this solution is not smooth and the resulting sharp threshold can cause problematic discontinuities in the model behavior. A smooth version of this equation is given by

$$\begin{aligned} \overline{S} &= \frac{S_{\text{UR}}}{S_{\text{max}}} \\ E &= E_{\text{POT}} \left(\frac{\overline{S}(1+m)}{\overline{S}+m} \right) \end{aligned}$$

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

```
class UpperZone(ODEsElement):

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

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

(continues on next page)

(continued from previous page)

```
        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,
                                          dt=self._dt,
                                          **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,
                                          dt=self._dt,
                                          **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, dt):
```

(continues on next page)

(continued from previous page)

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

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

        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] * dt[ind]
        )
```

Channel routing and lower zone

The elements representing channel routing and lower zone are all linear reservoirs that can be implemented by extending the class `ODEsElement`.

```
class LinearReservoir(ODEsElement):

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

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

```

(continues on next page)

(continued from previous page)

```
        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,
                                          dt=self._dt,
                                          **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, dt):

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

(continues on next page)

(continued from previous page)

```

    @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, dt):

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

10.3.3 Model initialization

Now that all elements are implemented, we can combine them to build the HYMOD model structure. For details refer to *How to build a model with SuperflexPy*.

First, we initialize all 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')

```

(continues on next page)

(continued from previous page)

```
transparent_1 = Transparent(id='tr1')

transparent_2 = Transparent(id='tr2')

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

The elements are now combined to define a `Unit` that reflects the structure shown in the figure.

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

CHAPTER

ELEVEN

CASE STUDIES

This page describes the model configurations used in research publications based on Superflex and SuperflexPy.

11.1 Dal Molin et al., 2020, HESS

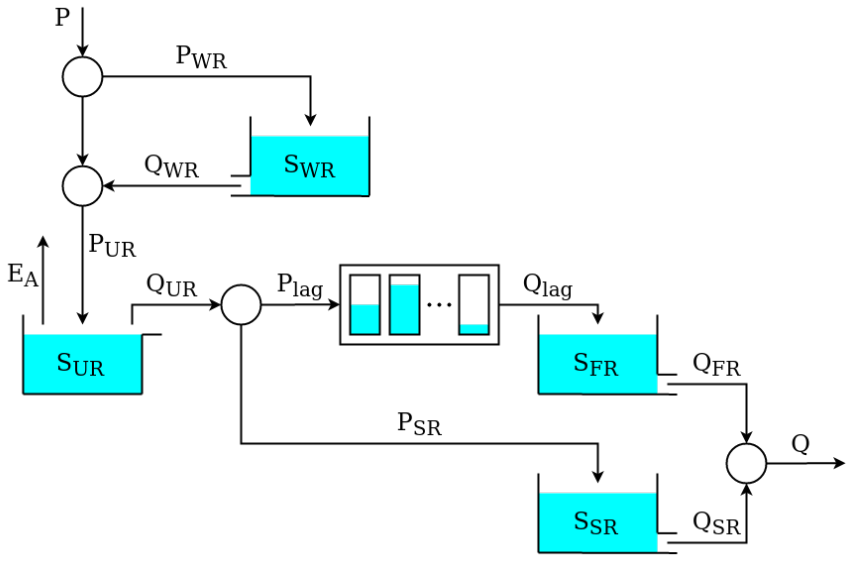
This section describes the implementation of the semi-distributed hydrological model M02 presented in the article:

> 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.

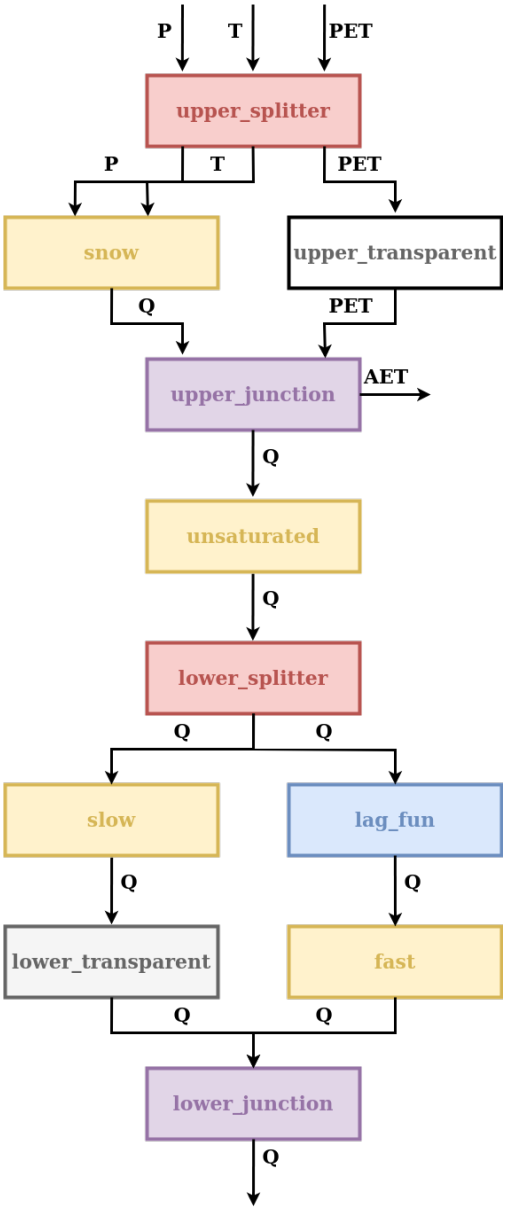
In this application, the Thur catchment is discretized into 10 subcatchments and 2 hydrological response units (HRUs). Please refer to the article for the details; here we only show the SuperflexPy code needed to reproduce the model from the publication.

11.1.1 Model structure

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



This model structure is similar to *HYMOD*; its implementation using SuperflexPy is presented next.

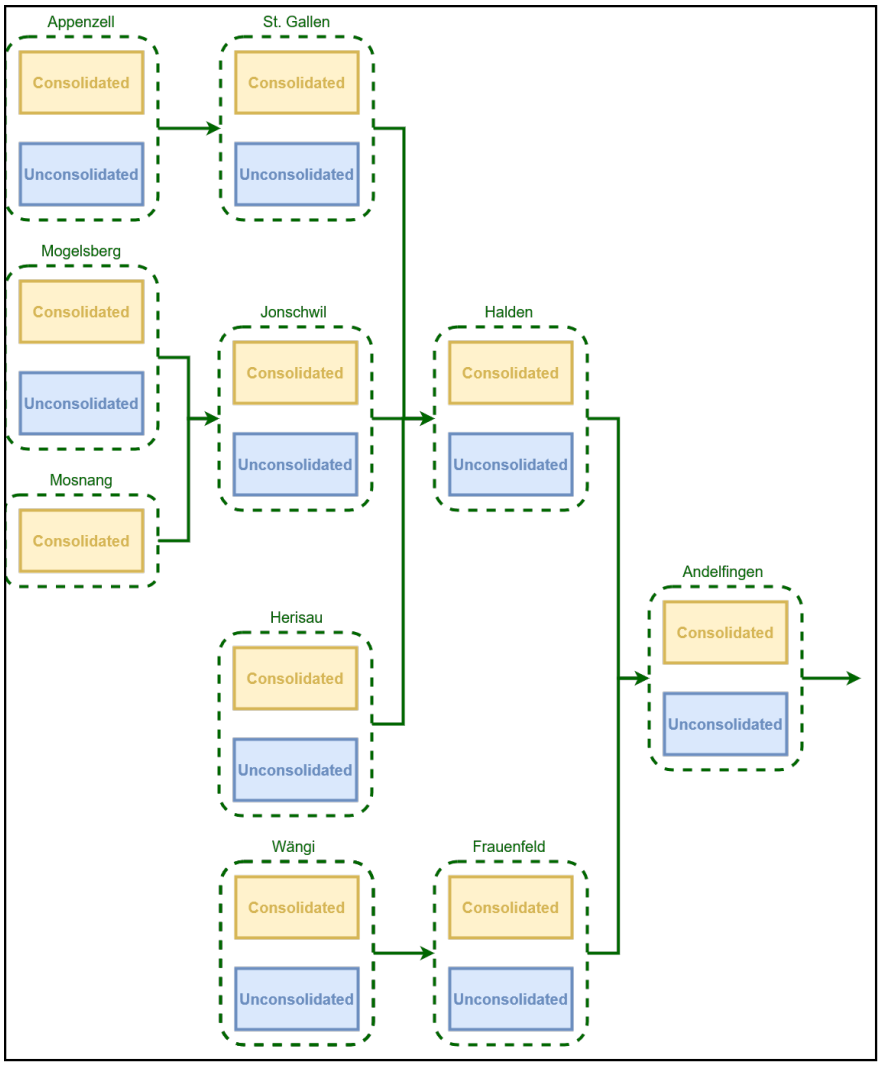


This model structure includes a snow model, and hence requires time series of temperature as an input in addition to precipitation and PET. Inputs are assigned to the element in the first layer of the unit and the model structure then propagates these inputs through all the elements until they reach the element (snow reservoir) where they are actually needed. Consequently, all the elements upstream of the snow reservoir have to be able to handle (i.e. to input and output) that input.

In this model, the choice of temperature as input is convenient because temperature is required by the element appearing first in the model structure.

In other cases, an alternative solution would have been to design the snow reservoir such that the temperature is one of its state variables. This solution would be preferable if the snow reservoir is not the first element of the structure, given that temperature is not an input commonly used by other elements.

The discretization of the Thur catchment into units (HRUs) and nodes (subcatchments) is represented in the figure below.



11.1.2 Initializing the elements

All elements required for this model structure are already available in SuperflexPy. Therefore they just need to be imported.

```
from superflexpy.implementation.elements.thur_model_hess import SnowReservoir, UnsaturatedReservoir, HalfTriangularLag, PowerReservoir
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
```

Elements are then initialized, defining the initial state and parameter values.

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

upper_splitter = Splitter(
    direction=[
```

(continues on next page)

(continued from previous page)

```
        [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,
    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 = PowerReservoir(
    parameters={'k': 0.01, 'alpha': 3.0},
    states={'S0': 0.0},
```

(continues on next page)

(continued from previous page)

```
    approximation=approximator,
    id='fast'
)

slow = PowerReservoir(
    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'
)
```

11.1.3 Initializing the HRUs structure

We now define the two units that represent the HRUs.

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

11.1.4 Initializing the catchments

We now assign the units (HRUs) to the nodes (catchments).

```
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'
)

herisau = Node(
    units=[consolidated, unconsolidated],
    weights=[0.88, 0.12],
    area=16.7,
    id='herisau'
)

jonschwil = Node(
    units=[consolidated, unconsolidated],
    weights=[0.9, 0.1],
    area=401.6,
    id='jonschwil'
)

mogelsberg = Node(
    units=[consolidated, unconsolidated],
    weights=[0.92, 0.08],
    area=88.1,
    id='mogelsberg'
)

mosnang = Node(
    units=[consolidated],
    weights=[1.0],
    area=3.1,
    id='mosnang'
```

(continues on next page)

(continued from previous page)

```
)

stgallen = Node(
    units=[consolidated, unconsolidated],
    weights=[0.87, 0.13],
    area=186.6,
    id='stgallen'
)

waengi = Node(
    units=[consolidated, unconsolidated],
    weights=[0.63, 0.37],
    area=78.9,
    id='waengi'
)
```

Note that all nodes incorporate the information about their `area`, which is used by the network to calculate their contribution to the total outflow.

There is no requirement for a node to contain all units. For example, the unit `unconsolidated` is not present in the Mosnang subcatchment. Hence, as shown in line 50, the node `mosnang` is defined to contain only the unit `consolidated`.

11.1.5 Initializing the network

The last step consists in creating the network that connects all the nodes previously initialized.

```
thur_catchment = Network(
    nodes=[
        andelfingen,
        appenzell,
        frauenfeld,
        halden,
        herisau,
        jonschwil,
        mogelsberg,
        mosnang,
        stgallen,
        waengi,
    ],
    topology={
        'andelfingen': None,
        'appenzell': 'stgallen',
        'frauenfeld': 'andelfingen',
        'halden': 'andelfingen',
        'herisau': 'halden',
        'jonschwil': 'halden',
        'mogelsberg': 'jonschwil',
        'mosnang': 'jonschwil',
        'stgallen': 'halden',
        'waengi': 'frauenfeld',
    }
)
```

11.1.6 Running the model

Before the model can be run, we need to set the input fluxes and the time step size.

The input fluxes are assigned to the individual nodes (catchments). Here, the data is available as a Pandas DataFrame, with columns names `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,
    ])
```

The model time step size is set next. This can be done directly at the network level, which automatically sets the time step size to all lower-level model components.

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

We can now run the model and access its output (see demo_network for details).

```
output = thur_catchment.get_output()
```

CHAPTER

TWELVE

SUPERFLEXPY IN THE SCIENTIFIC LITERATURE

This page lists the scientific publications presenting SuperflexPy or using it in specific applications.

12.1 Publications on SuperflexPy

- Dal Molin, M., Kavetski, D., and Fenicia, F.: **SuperflexPy 1.2.0: an open source framework for building, testing and improving conceptual hydrological models**, Geosci. Model Dev., *under review*, 2020.

12.2 Publications using SuperflexPy

- Jansen, K. F., Teuling, A.J., Craig, J. R., Dal Molin, M., Knoben, W. J. M., Parajka, J., Vis, M., and Melsen, L. A.: **Mimicry of a conceptual hydrological model (HBV): what's in a name?**, Environ. Modell. Softw., *under review*, 2020.

CHAPTER

THIRTEEN

SHARING MODEL CONFIGURATIONS

A key goal of SuperflexPy is to facilitate collaboration between research groups, and to help compare and improve modelling solutions. To this end, users can share model configurations that can be imported and run by other users. Note that models built with SuperflexPy are Python objects that, once initialized, can be imported into other scripts.

A user who wishes to share their model configuration with the community can create a Python script (with a descriptive name) that initializes the model (without running it) and "upload" it to the GitHub repository in the folder `superflexpy/implementation/models/`. This "upload" requires the following steps: (1) fork the SuperflexPy repository, (2) add the script to the local fork of the repository, and (3) make a pull request to the original repository (see *Software organization and contribution* for further details). The contributed code will be checked by the repository maintainers. Assuming all checks are passed the newly incorporated code will be incorporated in the following release of SuperflexPy and thus made available to other SuperflexPy users.

The user will maintain authorship on the contributed code, which will be released with the same *License* as SuperflexPy. It is good practice to include unit tests to enable users to ensure the new code is operating as expected (see *Automated testing*).

13.1 Practical example with M4

We illustrate of how to distribute SuperflexPy models to colleagues using as an example the model *Model M4 from Kavetski and Fenicia, WRR, 2011*.

First, we create the file `m4_sf_2011.py` that contains the code to initialize the model

```
from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
from superflexpy.implementation.elements.hbv import UnsaturatedReservoir,␣
↪PowerReservoir
from superflexpy.framework.unit import Unit

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 = PowerReservoir(
    parameters={'k': 0.1, 'alpha': 1.0},
    states={'S0': 10.0},
```

(continues on next page)

(continued from previous page)

```
    approximation=numeric_approximator,
    id='FR'
)

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

Then we incorporate the file `m4_sf_2011.py` into the SuperflexPy repository in the folder `superflexpy/implementation/models/` following the steps illustrated in the previous section (fork, change, and pull request).

Once the next release of SuperflexPy is available, the M4 model implementation will be available in the updated installed package. General users can then use this new model in their own application, by importing it as shown below.

```
from superflexpy.implementation.models.m4_sf_2011 import model

model.set_input([P, E])
model.set_timestep(1.0)
model.reset_states()

output = model.get_output()
```

13.2 Sharing models "privately" with other users

Model configurations can be shared "privately" between research groups without waiting for a new release of the framework.

This can be done by creating a `my_new_model.py` file that initializes the model and then sharing the file "privately" with other users.

The recipients of the new file can then save it on their machines and use local importing. Assuming that the script that the recipients use to run the model is in the same folder as the file initializing the model, the new model can be used as follows

```
1 from .my_new_model import model
2
3 model.set_input([P, E])
4 model.set_timestep(1.0)
5 model.reset_states()
6
7 output = model.get_output()
```

Note the local import in line 1.

As we believe in the F.A.I.R. principles, we encourage modelers to share their models with the whole community, using the procedure detailed earlier.

13.3 Dumping objects with Pickle

Python offers the module Pickle to serialize objects to binary files. This approach enables the distribution of binary files, but has the disadvantage of lacking transparency in the model structure.

CHAPTER

FOURTEEN

INTERFACING SUPERFLEXPY WITH OTHER FRAMEWORKS

SuperflexPy does not integrate tools for calibration or uncertainty analysis. In this page we show an example on how a model built using SuperflexPy can be interfaced with other tools to perform this task.

14.1 SuperflexPy + SPOTPY

Note: This example is for illustration purposes only, and as such does not represent a specific recommendation of SPOTPY or of any specific calibration algorithm.

SPOTPY is a Python framework for calibration, uncertainty, and sensitivity analysis.

A model can be interfaced with SPOTPY by defining a class that wraps the model and implements the following methods:

- `__init__`: initializes the class, defining some attributes;
- `parameters`: returns the parameters considered in the analysis (note that they may not be all the parameters used by the SuperflexPy model but only the ones that we want to vary in the analysis);
- `simulation`: returns the output of the simulation;
- `evaluation`: returns the observed output;
- `objectivefunction`: defines the objective function to use to evaluate the simulation results.

14.1.1 `__init__`

```
import spotpy

class spotpy_model(object):

    def __init__(self, model, inputs, dt, observations, parameters, parameter_names,␣
↪output_index):

        self._model = model
        self._model.set_input(inputs)
        self._model.set_timestep(dt)

        self._parameters = parameters
        self._parameter_names = parameter_names
        self._observarions = observations
        self._output_index = output_index
```

The class `spotpy_model` is initialized defining the SuperflexPy `model` that is used. The `model`, which can be any SuperflexPy component (from element to network), must be defined before; the `spotpy_model` class sets only the `inputs` and the `dt`.

Other variables necessary to initialize the class `spotpy_model` are:

- `parameters` and `parameters_names`, which define the parameters considered in the calibration. The first variable is a list of `spotpy.parameter` objects, the second variable is a list of the names of the SuperflexPy parameters;
- `observations`, which is an array of observed output values;
- `output_index`, which is the index of the output flux to be considered when evaluating the SuperflexPy simulation. This specification is necessary in the case of multiple output fluxes.

14.1.2 parameters

```
1     def parameters(self):
2         return spotpy.parameter.generate(self._parameters)
```

The method `parameters` generates a new parameter set using the SPOTPY functionalities.

14.1.3 simulation

```
1     def simulation(self, parameters):
2
3         named_parameters = {}
4         for p_name, p in zip(self._parameter_names, parameters):
5             named_parameters[p_name] = p
6
7         self._model.set_parameters(named_parameters)
8         self._model.reset_states()
9         output = self._model.get_output()
10
11        return output[self._output_index]
```

The method `simulation` sets the parameters (lines 3-7), resets the states to their initial value (line 8), runs the SuperflexPy model (line 9), and returns the output flux for the evaluation of the objective function (line 11).

14.1.4 evaluation

```
1     def evaluation(self):
2         return self._observarions
```

The method `evaluation` returns the observed flux, used for the evaluation of the objective function.

14.1.5 objectivefunction

```
1     def objectivefunction(self, simulation, evaluation):
2
3         obj_fun = spotpy.objectivefunctions.nashsutcliffe(evaluation=evaluation,
4                                                           simulation=simulation)
```

(continues on next page)

(continued from previous page)

```

        return obj_fun
```

The method `objectivefunction` defines the objective function used to measure the model fit to the observed data. In this case, the Nash-Sutcliffe efficiency is used.

14.1.6 Example of use

We now show how to employ the implementation above to calibrate a lumped model composed of 2 reservoirs.

First, we initialize the SuperflexPy model, as follows (see *How to build a model with SuperflexPy* for more details on how to set-up a model).

```
from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
from superflexpy.implementation.elements.hbv import PowerReservoir
from superflexpy.framework.unit import Unit

root_finder = PegasusPython()
num_app = ImplicitEulerPython(root_finder=root_finder)

reservoir_1 = PowerReservoir(parameters={'k': 0.1, 'alpha': 2.0},
                             states={'S0': 10.0},
                             approximation=num_app,
                             id='FR1')
reservoir_2 = PowerReservoir(parameters={'k': 0.5, 'alpha': 1.0},
                             states={'S0': 10.0},
                             approximation=num_app,
                             id='FR2')

hyd_mod = Unit(layers=[[reservoir_1],
                       [reservoir_2]],
               id='model')
```

Then, we initialize an instance of the `spotpy_model` class

```
spotpy_hyd_mod = spotpy_model(
    model=hyd_mod,
    inputs=[P],
    dt=1.0,
    observations=Q_obs,
    parameters=[
        spotpy.parameter.Uniform('model_FR1_k', 1e-4, 1e-1),
        spotpy.parameter.Uniform('model_FR2_k', 1e-3, 1.0),
    ],
    parameter_names=['model_FR1_k', 'model_FR2_k'],
    output_index=0
)
```

The arrays `P` and `Q_obs` in lines 3 and 5 contain time series of precipitation (input) and observed streamflow (output). In this example, lines 6-10 indicate the two parameters that we calibrate (`model_FR1_k` and `model_FR2_k`) together with their range of variability.

We can now call the SPOTPY method to calibrate the model. Here, the SCE algorithm option is used.

```
sampler = spotpy.algorithms.sceua(spotpy_hyd_mod, dbname='calibration', dbformat='csv
↪')
sampler.sample(repetitions=5000)
```

CHAPTER

FIFTEEN

EXAMPLES

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

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

CHAPTER

SIXTEEN

AUTOMATED TESTING

Current testing of SuperflexPy consists of validating its numerical results against the original implementation of Superflex. This testing is done for selected model configurations and selected sets of parameters and inputs.

This testing strategy implicitly checks auxiliary methods, including setting parameters and states, retrieving the internal fluxes of the model, setting inputs and getting outputs, etc..

The testing code is contained in folder `test` and uses the Python module `unittest`. The folder contains `reference_results` and `unittest` containing the scripts that run the tests.

Current testing covers:

- Specific elements (reservoirs and lag functions) that are implemented in Superflex (e.g. `01_FR.py`, `02_UR.py`);
- Multiple elements in a unit (e.g. `03_UR_FR.py`, `04_UR_FR_SR.py`);
- Multiple units in a node (e.g. `05_2HRUs.py`);
- Multiple nodes inside a network (e.g. `06_3Cats_2HRUs.py`);
- Auxiliary methods, which are tested implicitly, i.e. assuming that errors in the auxiliary methods propagate to the results.

Current testing does not cover:

- Elements for which numerical results are not available (e.g. some components of GR4J);
- Usage of the Explicit Euler solver;
- Edge cases (e.g. extreme values of parameters and states)

Users contributing SuperflexPy extensions should provide reference results and the code that tests them (including input data and model parameter values).

As the SuperflexPy framework continues to develop, additional facilities for unit-testing and integrated-testing will be employed.

16.1 Automation

Any push of new code to any branch on the github repository will trigger automatic testing based on the scripts contained in the folder `test/unittest`.

CHAPTER

SEVENTEEN

LICENSE

```
                   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


  This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
```

(continues on next page)

(continued from previous page)

```
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
```

(continues on next page)

(continued from previous page)

```
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.

  The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

  Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
```

(continues on next page)

(continued from previous page)

```
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

CHAPTER

EIGHTEEN

CHANGE LOG

18.1 Version 1.2.0

18.1.1 Major changes to existing components

- The abbreviation of "differential equation" changes, in the code, from `dif_eq` to `diff_eq`. This change regards variables names, both in the methods arguments and implementation.
- The class `FastReservoir` has been changed to `PowerReservoir`. No changes in the functionality of the class.

18.1.2 Minor changes

- Testing improved.

18.2 Version 1.1.0

18.2.1 Major changes to existing components

- Form this version, SuperflexPy is released under license LGPL. For details, read *License*

18.2.2 Minor changes to existing components

- Bug fix on the solution of the differential equations of the reservoirs. The calculation of the maximum storage was not correct.

18.3 Version 1.0.0

Version 1.0.0 represents the first mature release of SuperflexPy. Many aspects have changed since earlier 0.x releases both in terms of code organization and conceptualization of the framework. **Models built with versions 0.x are not compatible with this version and with the following releases**.

18.3.1 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.

18.3.2 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)
- Minor changes to `RootFinder` to accommodate the new numerical implementation.
- Added Numba implementation of GR4J elements

18.3.3 New code

- Added `hymod` elements

CHAPTER

NINETEEN

CODE ORGANIZATION

The following schematic follows the standards of UML and shows the organization of the classes composing the framework. This schematic is limited to the core framework (i.e. content of `superflexpy/framework/` and `superflexpy/utils/`). Particular implementations of components (i.e. the content of `superflexpy/implementation/`) are not included. All the classes in the diagram can be extended through inheritance to create customized components.