https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

Geoscientific Model Development
Discussions
Open Access

SuperflexPy 1.2.0: an open source Python framework for building, testing and improving conceptual hydrological models

Marco Dal Molin[1,2], Dmitri Kavetski[1,3], Fabrizio Fenicia[1]

[1]Eawag, Swiss Federal Institute of Aquatic Science and Technology, Dübendorf, Switzerland
[2]The Centre of Hydrogeology and Geothermics (CHYN), University of Neuchâtel, Neuchâtel, Switzerland
[3]School of Civil, Environmental and Mining Engineering, University of Adelaide, SA, Australia

Correspondence to: Marco Dal Molin (marco.dalmolin@eawag.ch)

Abstract

Catchment-scale hydrological models are widely used to represent and improve our understanding of hydrological processes, and to support operational water resources management. Conceptual models, where catchment dynamics are approximated using relatively simple storage and routing elements, offer an attractive compromise in terms of predictive accuracy, computational demands and amenability to interpretation. This paper introduces SuperflexPy, an open-source Python framework implementing the SUPERFLEX principles (Fenicia et al., 2011) for building conceptual hydrological models from generic components, with a high degree of control over all aspects of model specification. SuperflexPy can be used to build models of a wide range of spatial complexity, ranging from simple lumped models (e.g. a reservoir) to spatially distributed configurations (e.g. nested sub-catchments), with the ability to customize all individual model elements. SuperflexPy is a Python package, enabling modelers to exploit the full potential of the framework without the need for separate software installations, and making it easier to use and interface with existing Python code for model deployment. This paper presents the general architecture of SuperflexPy, and illustrates its usage to build conceptual models of varying degrees of complexity. The illustration includes the usage of existing SuperflexPy model elements, as well as their extension to implement new functionality. SuperflexPy is available as open-source code, and can be used by the hydrological community to investigate improved process representations, for model comparison, and for operational work. A comprehensive documentation is available online and provided as supplementary material to this paper.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

Table of Contents

1 Introduction

1.1 Conceptual hydrological models

Catchment-scale hydrological models are widely used to predict catchment behavior under natural and human-impacted conditions, as well as to represent and improve our understanding of internal catchment functioning (e.g. Beven, 1989). For example, catchment models underlie projections of climate change impact on groundwater recharge and streamflow (e.g., Eckhardt and Ulbrich, 2003), are used as tools for hypothesis testing to identify dominant hydrological processes (e.g., Clark et al., 2011b;Hrachowitz et al., 2014;Wrede et al., 2015), and are used to inform agricultural practices such as irrigation scheduling (e.g., McInerney et al., 2018) and pesticide application (e.g., Moser et al., 2018;Ammann et al., 2020). The typical use of hydrological models is to simulate or forecast the streamflow response (runoff) of a catchment to rainfall; for this reason they are often referred to as rainfall-runoff (RR) models (e.g., Moradkhani and Sorooshian, 2009). However, their application extends to the simulation of other environmental variables, including internal catchment variables such as groundwater levels (e.g., Seibert and McDonnell, 2002) and soil moisture (e.g., Matgen et al., 2012), as well as water chemistry (e.g., Bertuzzo et al., 2013;Ammann et al., 2020)

An important class of catchment models are "process based" models, which describe the cascade of processes transforming catchment inputs (e.g. precipitation) into outputs (e.g. streamflow) though conservation equations. These models are an appealing choice due to their broad physical underpinnings, as well as their ability to represent internal catchment processes and potential for predicting catchment responses under changing environmental conditions. Process based models can be classified according to the nature of their constitutive equations (e.g. conceptual or physically based) and their spatial resolution (e.g. lumped or distributed) (e.g., Refsgaard, 1996).

Conceptual models, where catchment dynamics are approximated using relatively simple storage and routing elements (e.g. Fenicia et al., 2011), are popular in practice because they offer an attractive compromise in terms of predictive accuracy, computational demands, and amenability to interpretation. Popular conceptual models include TopModel (Beven and Kirkby, 1979), HBV (Lindstrom et al., 1997), GR4J (Perrin et al., 2003), and HyMod (Boyle, 2001).

In terms of spatial resolution, conceptual models can be applied in a lumped configuration (treating the entire catchment as a single unit) if the interest is in modeling integrated catchment outputs (e.g. streamflow). Alternatively, distributed configurations can be used when the interest is in modeling hydrological behavior at individual landscape sections (e.g., sub-catchments). In such distributed setups, the catchment is subdivided into spatial elements such as sub-catchments (e.g., Feyen et al., 2008;Lerat et al., 2012), Hydrological Response Units (HRUs) (e.g., Arnold et al., 1998;Fenicia et al., 2016;Dal

Molin et al., 2020b), or grids (e.g., Samaniego et al., 2010). A common strategy for developing distributed conceptual models is to represent individual landscape elements using independent (non-interacting) lumped models, and then obtain total catchment outflow by aggregating the outflows from these individual models, potentially incorporating flow routing elements to represent routing delays. This strategy is often referred to as "semi-distributed" modelling, and typically employs discretization based on principles of "hydrological similarity" (e.g., Sivapalan et al., 1987), such as HRUs (e.g., Leavesley, 1984). In many cases, semi-distributed modelling achieves good predictive ability while greatly simplifying model representation and reducing computational demands compared to fully-integrated 2D/3D distributed models such as Parflow (Maxwell, 2013) or Mike She (Refsgaard and Storm, 1995), which typically use much smaller landscape elements and explicitly model lateral exchanges. For the purposes of this presentation, we consider semi-distributed modelling to be a special case of distributed modelling.

1.2 Hydrological model structure and flexible modeling frameworks

The selection of model structure has preoccupied researchers and practitioners since the early days of hydrological modelling (e.g., Ibbitt and O'Donnell, 1971;Moore and Clarke, 1981;Jakeman and Hornberger, 1993). Although in principle the physical laws governing hydrological processes are the same everywhere, the diversity of catchment conditions in terms of topography, soil, geology, vegetation, and anthropogenic influence, results in remarkably different manifestations of these physical laws at the catchment scale. These local differences, also termed "uniqueness of place" (Beven, 2000), considerably limit our ability to develop generalizable hydrological hypotheses (e.g., Wagener et al., 2007).

Model structure selection has led to multiple research directions, including the search for a single model structure that achieves good prediction across all catchments (the "fixed" model paradigm), and the search for model structures best suited for particular types of environmental conditions (the "flexible" model paradigm). Whether in search of a single model or multiple models, model selection necessarily relies on a process of model development, comparison, and refinement. Approaches to formalize this process include the top-down approach (e.g. Sivapalan et al., 2003), the system identification approach (e.g Young, 1998), and the method of multiple working hypotheses (e.g., Clark et al., 2011a). These approaches are not mutually exclusive, as the idea of comparing different model representations is ubiquitous in model development and empirical science in general.

The process of model development, comparison, and refinement can be facilitated using flexible modeling frameworks, which enable hydrologists to hypothesize, implement, and (eventually) test and refine different model structures. Flexible frameworks have themselves developed along multiple directions. For example, GEOframe-NewAge (Formetta et al., 2014), SUMMA (Clark et al., 2015), RAVEN (Craig et al., 2020), and CHM (Marsh et al., 2020) focus on the realm of physically based models; the CAPTAIN toolbox (Young et al., 2009) is a general toolkit for time series analysis; machine learning frameworks such as scikit-learn (Pedregosa et al., 2011) or PyTorch (Paszke et al., 2019) can be used to construct data driven models.

In this paper, we focus on flexible frameworks intended for conceptual hydrological modeling. Many frameworks have been developed for such purpose, and offer different degrees of flexibility. For example, FUSE (Clark et al., 2008) allows exchanging the components of 4 common models (Sacramento, PRMS, TOPMODEL and ARNO/VIC). SUPERFLEX (Fenicia et al., 2011) allows building model structures from generic elements (reservoirs, lag function and connections). CMF (Kraft et al., 2011), represents the model as an abstract network of elements and can be adopted for conceptual models. PERSiST (Futter et al., 2014) allows to create semi-distributed bucket-type models and is designed to be coupled with a water quality model. ECHSE (Kneis, 2015) is a framework for development of object-based conceptual hydrological model engines. MARRMoT (Knoben et al., 2019) provides a unified implementation of 46 existing conceptual models (including GR4J, HBV, and others).

When discussing any mathematical model, it is relevant to distinguish its conceptual principles from its software implementation. For example, common conceptual models, such as GR4J, HBV and TOPMODEL, exist in many public and in-house versions and in many computer languages (Excel, R, Matlab, Fortran, C, and others). FUSE, to our knowledge, has implementations in Fortran (Clark et al., 2008) and R (Vitolo et al., 2016). SUPERFLEX, besides its original Fortran implementation (Fenicia et al., 2011), has also been coded in Matlab (David et al., 2019).

Ideally, the software implementation of a flexible framework should fulfill its goals, i.e., (1) cover the envisioned range of applications (e.g., flexibility, spatial extension, processes representation, etc.), and (2) achieve this in a sufficiently "practical" way (i.e., it should not require complicated and abstract setup procedures to define its configuration).

For conceptual models, a flexible model framework should arguably cover the following "realms":

1. Lumped models;
2. Distributed setups, including simulation of sub-catchments and flows/processes at internal points;
3. Substance transport modelling, including water isotopes, pesticides, etc;
4. Ability to reproduce existing models, when necessary.

A modeling code should also meet certain practical criteria:

1. Ease of use, including installation, learning, and operation. Interoperability with external software, for example for model calibration and uncertainty analysis, is of obvious relevance because hydrological models are often used as parts of larger-scale projects and operations.
2. Ease of modifications and extensions. Even a comprehensive software implementation will eventually require extension. For example, a modeling framework intended for streamflow simulation may require extension to simulate water chemistry. Another type of modification might be a switch to a different numerical implementation better suited for parallel computing, etc.
3. Computational efficiency. Hydrological model applications, especially including calibration and uncertainty quantification, may require thousands or even millions of model runs.

Arguably, these requirements are not collectively fulfilled by available software implementations of flexible frameworks. For example, only CMF and ECHSE provide full flexibility in terms of model structure selection. In some frameworks, the intended flexibility is obtained by enabling and tuning the components of a master structure (e.g. FUSE, PERSiST). The original implementation of SUPERFLEX in Fortran (Fenicia et al., 2011), used for research case studies though not released publically, also used this “master structure” approach. In other flexible frameworks, the user can chose from an extensive library of existing model configuration (e.g. MARMMoT). Some flexible frameworks are limited to lumped configurations (e.g. FUSE, MARMMoT). Substance transport is currently partially possible with PERSiST, by interfacing with additional software.

Clearly, achieving all these objectives simultaneously is challenging, and entails juggling multiple obvious and less obvious tradeoffs. For example, the intended flexibility of a framework may come at the expense of ease of use, similar to how computer languages have varying degrees of abstraction from the hardware behavior. Implementing a practical flexible framework therefore requires careful code design, experimentation, and inevitably, some compromises.

This work pursues the flexible framework objectives by building upon the concept of SUPERFLEX (Fenicia et al., 2011;Kavetski and Fenicia, 2011;Fenicia et al., 2014;Fenicia et al., 2016). A key attractive feature of SUPERFLEX as a modelling concept is the fine “granularity” of model structures it can support, which enables systematic and detailed hypothesis testing (Fenicia et al., 2011). For example, the hydrologist should have the ability to change the functional form of a single flux expression in one of the model elements, as well as to change such flux expressions in multiple specific parts of the model.

The development of the proposed framework capitalizes on the authors’ collective experience in using the earlier implementations of SUPERFLEX in a series of empirical case studies over the last decade, ranging from lumped model implementations (e.g., Kavetski and Fenicia, 2011;Fenicia et al., 2014), to distributed setups (e.g. Fenicia et al., 2016;Dal Molin et al., 2020b), interpretation in the context of fieldwork insights (e.g., Wrede et al., 2015), large scale model intercomparisons (e.g., van Esse et al., 2013), and the inclusion of pesticide/substance transport (e.g. Ammann et al., 2020). These applications have highlighted the versatility of SUPERFLEX principles to solve increasingly complex modelling problems, and have led to insights into software design and configuration aspects not available in the earlier implementations. This study reports on these developments and offers an open source implementation of SUPERFLEX for use by the hydrological community.

1.3 Aims

This work introduces SuperflexPy, an open-source Python software that implements the principles of the SUPERFLEX framework for conceptual hydrological model development. Our objectives are as follows:

1. Present SuperflexPy and its basic building blocks (*components*): *elements*, *units*, *nodes,* and *network*.
2. Illustrate how SuperflexPy can help hydrologists implement a conceptual model structure at the desired level of internal complexity and spatial resolution – including recreating existing models or developing new ones.
3. Provide a broad discussion of how the SuperflexPy contributes to the toolkits available to the hydrological community, including existing flexible frameworks, in terms of intended scope of application, advantages, and limitations.

The paper is organized as follows: Sect. 2 presents SuperflexPy to the hydrological community; Sect. 3 illustrates selected applications of the framework including the setup of SUPERFLEX configurations used in earlier case studies, as well as how to use SuperflexPy to create new *elements*; Sect. 4 provides SuperflexPy implementation details useful for understanding the usage and general potential of the framework; Sect. 5 discusses the scope of SuperflexPy, its current limitations, and future developments. Finally, Sect. 6 draws the conclusions.

2 Description of SuperflexPy

2.1 General organization

The SuperflexPy framework has a hierarchical organization with four nested levels: “*element*”, “*unit*”, “*node*”, and “*network*”, collectively referred as “*components*”. These *components* are shown in Figure 1 and described below:

1. ***Element*** (Figure 1a). This level represents the basic model building blocks and is used to create reservoirs, lag functions, and connections. An *element* can be used to represent an entire

catchment, or, more commonly, a specific process within the catchment.

The **reservoir** element is described mathematically by ordinary differential equations (ODEs):

$$\frac{\mathrm{d}\mathbf{S}(t)}{\mathrm{d}t} = \mathbf{g}_{\mathrm{S}}\left(\mathbf{S}(t), \mathbf{X}(t); \boldsymbol{\theta}\right) \tag{1}$$

$$\mathbf{Y}(t) = \mathbf{g}_{\mathrm{Y}}\left(\mathbf{S}(t), \mathbf{X}(t); \boldsymbol{\theta}\right) \tag{2}$$

where $\mathbf{S}$ are the state variables (e.g., water storages, substance concentrations, etc.), $\mathbf{X}$ are the inputs (e.g., precipitation), $\mathbf{Y}$ are the outputs (e.g., streamflow), and $\mathbf{g}_{\mathrm{S}}$ and $\mathbf{g}_{\mathrm{Y}}$ are specified constitutive functions (e.g., storage-discharge relationships). 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).

The **lag function** element is described mathematically by a convolution integral:

$$\mathbf{Y}(t) = \mathbf{X}(t) * \mathbf{g}_{\mathrm{H}}(t; \boldsymbol{\theta}) = \int_0^T \mathbf{X}(t-\tau) * \mathbf{g}_{\mathrm{H}}(\tau; \boldsymbol{\theta}) \mathrm{d}\tau \tag{3}$$

where $*$ denotes the convolution operator, $\mathbf{X}$ is the input (e.g., water flux), $\mathbf{g}_{\mathrm{H}}$ is the impulse response function, and T is the time of influence of $\mathbf{g}_{\mathrm{H}}$ (i.e. the maximum lag). Lag functions are used to represent delays due, for example, to routing.

The **connection** element joins or splits fluxes from other elements. It has parameters but no states:

$$\mathbf{Y}(t) = \mathbf{g}_{\mathrm{C}}\left(\mathbf{X}(t); \boldsymbol{\theta}\right) \tag{4}$$

where $\mathbf{g}_{\mathrm{C}}$ describes the connectivity between input fluxes and output fluxes, and $\boldsymbol{\theta}$ represents connectivity parameters (if any).

2. ***Unit*** (Figure 1b). A *unit* is a collection of multiple connected *elements*, and is generally intended to implement a lumped catchment model. Multiple reservoir and lag function elements within a *unit* can be connected to each other, either directly (one-to-one connections), or using connection elements such as splitters and junctions (when a single *element* is connected to multiple *elements*). *Elements* can be combined to build a *unit*, with the only restriction being that feedback loops *between* the *elements* are not allowed. In technical terms, the overall model structure must be an acyclic directional graph. This design restriction is motivated by computational efficiency reasons, as it enables the numerical solution of elements from upstream to downstream. Note that the restriction is not absolute, because it does not preclude feedback between the states *within* a given *element*. Hence, if feedbacks are deemed necessary, they can

handled within an individual *element*, for example by creating a reservoir element with multiple storages.

3. ***Node*** (Figure 1c). A *node* is a collection of multiple *units* that operate in parallel. In the context of distributed models, the *node* can be used to represent a single catchment and the *units* can be used to represent multiple landscape elements (areas) within the catchment. Each *unit* within a *node* is characterized by a weight (e.g. representing its area fraction) specified by the modeler. The weights are used to combine the output fluxes from the *units* into the total output flux of the *node*.
4. ***Network*** (Figure 1d). A *network* connects multiple *nodes* into a tree structure, and is typically intended to develop a distributed model that generates predictions at internal sub-catchment locations (e.g. to reflect a nested catchment setup). The *network* routs the fluxes from upstream *nodes* (leaves of the tree) to the final downstream *node* (root of the tree). The routing in the river network can be simulated adding delays (lag) to the *nodes* outputs.

This hierarchical organization makes the effort required to configure SuperflexPy to a new problem proportional to the problem complexity. In particular:

- Level 1 is sufficient to create single-*element* models, e.g., a single-reservoir model or a unit hydrograph model (e.g. Kirchner, 2009);
- Level 2 is sufficient to create a lumped model structure, such as GR4J (Perrin et al., 2003) or Hymod (Boyle, 2001);
- Level 3 is sufficient create a distributed model that represents spatial heterogeneity but generates predictions only at the catchment outlet (e.g. Gao et al., 2014;Nijzink et al., 2016);
- Level 4 is needed only in models that generate predictions at interior points (e.g. Fenicia et al., 2016;Dal Molin et al., 2020b).

Examples of SuperflexPy models implemented at Levels 2 and 4 are given later in Sect. 3.

From a software design prospective, SuperflexPy embraces the object-oriented paradigm (e.g., Meyer, 1988). All framework *components* are represented by objects that can operate either alone or together, interacting with each other and with external libraries (e.g. for calibration) through defined interfaces. More details are provided in Sect. 4.2.

2.2 A simple illustration of SuperflexPy: creating a new model from existing components

This section illustrates the key steps needed to configure and run a hydrological model using the SuperflexPy framework. The illustration presents a distributed model intended to represent a catchment with 2 HRUs and 3 sub-catchments. The model structure is shown in Figure 1d. Within SuperflexPy, the entire catchment is represented using a *network*, the sub-catchments are represented using *nodes*, and the HRUs are represented using *units*. The corresponding SuperflexPy code is shown in Figure 2.

For this example, an implementation of the necessary *elements* with SuperflexPy already exists, therefore the *elements* only need to be imported. The case where the model structure requires *elements* for which an implementation is not yet available is considered in Sect. 2.3. Even more complex setups are described in Sect. 3 and in the online documentation (see code availability section).

We start by importing the model *components* required by the model structure, namely the *elements* (`LinearReservoir` and `HalfTriangularLag`), *unit*, *node*, and *network*. The numerical solvers `PegasusPython` and `ImplicitEulerPython` needed to solve the reservoir elements are also imported (more on this in Sect. 4.3). The import operation is shown in lines 1-7.

The imported *components* are then initialized, which entails specifying the model architecture (connectivity between model *components*) and the initial values of parameters and states. The initialisation sequence starts from the numerical routines (lines 10-11) and then proceeds from the lowest-level *components* (*elements*) to the highest-level *component* (*network*).

In detail:

L1. *Elements* are initialized by specifying parameters, states, identifier (`id`) and, when needed, the numerical solver (lines 14-16).

L2. *Units* are initialized by specifying the *elements* that compose them and the identifier (lines 19-20). The connectivity between *elements* is defined by conceptualizing the *unit* as a succession of layers that contain the *elements*. Further examples on this are given in Sect. 3.
The parameters and states of these *elements* can be changed after initialization using the methods `set_parameters` and `set_states` of the containing units. This procedure is shown on line 23 for the `LinearReservoir` element.

L3. *Nodes* are initialized by specifying the *units* that compose them, their contribution (weight) to the *node* output, the influence area of the *node* (here, the area of the sub-catchment), and the identifier (lines 26-28).
Within a given node, *units* operate independently from each other.

L4. The *network* is initialized by specifying the *nodes* that compose it and their connectivity, called `topography` (line 31). The connectivity is defined indicating, for each *node*, the *node* downstream of it.

The next step is to set the model inputs and time step. Lines 34-36 show how the inputs are assigned directly to the *nodes*, enabling the model to receive spatially-varying rainfall and PET. The time step is set on line 39 (note that variable time steps are also supported, see the documentation).

The model can now be run by calling the `get_output` method of the highest-level *component*, as shown on line 42.

2.3 Creating new model *components* with SuperflexPy

We now consider the case where the intended model structure has *components* beyond those already available in SuperflexPy. New model *components* can be created by extending existing SuperflexPy *components*. We anticipate that the SuperflexPy *components* most likely to require extension are the *elements*, where new reservoir constitutive functions may be required for new applications. In contrast, it is less likely that *unit*, *node* and *network* functionalities would require extension.

The extension of existing SuperflexPy elements to create new elements relies on the object-oriented paradigm underlying the SuperflexPy software design. The inheritance principle, one of the core concepts of the object-oriented paradigm, allows the user to construct new *components* by "inheriting" most of the functionalities (methods) from existing classes. Separate implementation is then required only for methods where model differences are to be introduced. This approach reduces substantially the amount of coding required to introduce a new model *component*. To this end, SuperflexPy provides a library of built-in high-level *components* that can be easily extended to achieve the desired functionality.

A detailed example of making use of this design is given in Sect. 3.2, which shows how to implement a reservoir with a new storage-discharge relationship.

3 Examples of building hydrological models using SuperflexPy

This section provides more examples of using SuperflexPy to implement hydrological models, including the use of built-in *elements* and the creation of new *elements*. We follow a progression from simple to complex. Section 3.1 shows the implementation of model M4, a lumped model built solely from reservoir elements and used in the original SUPERFLEX case study (Kavetski and Fenicia, 2011). Section 3.2 shows how to define a new *element* with a different storage-discharge relationship for one of the reservoirs of M4. Section 3.3 shows the implementation of a distributed model from a recent application of SUPERFLEX in the Thur catchment (Dal Molin et al., 2020b). Further details and more examples are provided in the model documentation (see code availability section).

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

3.1 Implementing SUPERFLEX configuration M4

M4 is a simple lumped model presented in Kavetski and Fenicia (2011). As shown in Figure 3, M4 comprises two reservoirs connected in series: an “unsaturated” reservoir (UR) intended to represent the partitioning of precipitation between evaporation and runoff, and a “fast” reservoir (FR) intended to represent subsequent streamflow generation mechanisms.

UR partitions precipitation $P^{(\mathrm{UR})}$ into a portion that enters the UR storage and eventually evaporates through flux $E_{\mathrm{A}}^{(\mathrm{UR})}$, and a portion $Q^{(\mathrm{UR})}$ that is directed to the downstream FR reservoir:

$$\frac{\mathrm{d}S^{(\mathrm{UR})}}{\mathrm{d}t} = P^{(\mathrm{UR})} - E_{\mathrm{A}}^{(\mathrm{UR})} - Q^{(\mathrm{UR})} \tag{5}$$

where:

$$\overline{S^{(\mathrm{UR})}} = \frac{S^{(\mathrm{UR})}}{S_{\max}^{(\mathrm{UR})}} \tag{6}$$

$$Q^{(\mathrm{UR})} = P^{(\mathrm{UR})} \times \left(\overline{S^{(\mathrm{UR})}}\right)^{\beta^{(\mathrm{UR})}} \tag{7}$$

$$E_{\mathrm{A}}^{(\mathrm{UR})} = E_{\mathrm{P}}^{(\mathrm{UR})} \times \frac{\overline{S^{(\mathrm{UR})}}\left(1+m^{(\mathrm{UR})}\right)}{\overline{S^{(\mathrm{UR})}} + m^{(\mathrm{UR})}} \tag{8}$$

In equations (6)-(8), $S_{\max}^{(\mathrm{UR})}$ and $\beta^{(\mathrm{UR})}$ are model parameters. The quantity $m^{(\mathrm{UR})}$ is used to approximate a “smooth” threshold behavior; we typically fix $m^{(\mathrm{UR})} = 0.01$.

FR is a power-law reservoir,

$$\frac{\mathrm{d}S^{(\mathrm{FR})}}{\mathrm{d}t} = P^{(\mathrm{FR})} - Q^{(\mathrm{FR})} \tag{9}$$

with the storage-discharge relationship given by

$$Q^{(\mathrm{FR})} = k^{(\mathrm{FR})}\left(S^{(\mathrm{FR})}\right)^{\alpha^{(\mathrm{FR})}} \tag{10}$$

where $k^{(\mathrm{FR})}$ and $\alpha^{(\mathrm{FR})}$ are model parameters.

The inflow $P^{(\mathrm{FR})}$ is given by the outflow from UR, i.e., $P^{(\mathrm{FR})} = Q^{(\mathrm{UR})}$.

M4 is a lumped model with multiple *elements*, and hence can be implemented using SuperflexPy levels L1 and L2 (*element* and *unit*, see Section 2.1). Figure 4 shows the code needed to implement M4. Similar to the model described in Sect. 2.2, the two model *elements* (UR and FR) are already implemented. Hence, the user only needs to import (lines 1-3) and initialize (lines 7-13) the *elements* together with the numerical

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

routines. Next, the *unit* that comprises the two reservoirs is imported (line 4) and initialized (line 15). The input data, namely precipitation and PET time series, are set on line 20. Input data is provided using Numpy arrays. The reading of input data (from text file(s), databases, etc.) is done separately from SuperflexPy, using any suitable Python library or function. In this case, we use Numpy to read from a text file, as shown in lines 17-18.

The model configuration is then complete – line 23 runs the model with given input data to produce the model outputs. The outputs contain streamflow time series in the form of Numpy arrays.

3.2 Changing the equations of the fast reservoir in M4

Suppose the modeler wishes to modify model M4 by changing the storage-discharge equation of the fast reservoir given in equation (10) to a new relationship

$$Q^{(\mathrm{FR})} = \frac{k^{(\mathrm{FR})}\left(S^{(\mathrm{FR})}\right)^{\alpha^{(\mathrm{FR})}}}{S^{(\mathrm{FR})} + b^{(\mathrm{FR})}} \tag{11}$$

where $k^{(\mathrm{FR})}$, $\alpha^{(\mathrm{FR})}$, and $b^{(\mathrm{FR})}$ are model parameters.

An *element* with this storage-discharge relationship has not been implemented in SuperflexPy yet (as of version 1.2.0). The following sections give two approaches for creating such an *element*.

3.2.1 Standard approach for creating a new reservoir

The standard approach for creating a new reservoir in SuperflexPy is to define a new class that inherits most of its functionality (methods) from the class `ODEsElement`. This operation is illustrated in the code snippet in Figure 5. The new class must override the following methods:

- `__init__`: constructor of the class. Its main purpose is to call the constructor of the parent class (lines 5-6) and to point at the method used to calculate the fluxes, here, `_fluxes_function_python` (see also Sect. 4.3, which shows the benefits of using Numba-optimized methods for calculating the fluxes);
- `set_input`: takes the input fluxes in a predefined order (here, just precipitation) and assigns them a key (line 13) that is then used when setting up and solving the model equations;
- `get_output`: calls the functionalities implemented by the `ODEsElement` to solve the *element* equation over the entire simulation (all time steps). Lines 18-20 get the current state of the reservoir, call the ODE solver, and set the state to the final value. Lines 22-26 get the output flux arrays from the numerical approximator (see Sect. 4.3). Line 28 returns a list with the outflow of the *element* (here, the streamflow);

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

- `_fluxes_function_python`: calculates the fluxes for a given state, inputs, and parameters. Line 34 implements the vector version while line 36 implements the scalar version. Both versions are needed by the numerical approximator.

The new *element* `NewFastReservoir` is now defined and can be used in the "new" version of M4, as shown in Sect. 3.1 for the pre-existing *element* `PowerReservoir`. The Object-Oriented features of Python are very useful here to enable the new class to inherit most of the methods from the base class `ODEsElement`. Otherwise, in addition to the methods listed above, we would have needed to implement many other methods, e.g., for interfacing with numerical solvers, for setting parameters and states, etc.

3.2.2 Simplified method for creating a new reservoir

The same new reservoir element can be implemented in a simpler way by noting that `NewFastReservoir` differs from `PowerReservoir` solely in the definition of the outflow equation. This difference affects only one of the four methods implemented in Figure 5, namely `_fluxes_function_python`. A simpler implementation of `NewFastReservoir` can be achieved by making this class inherit directly from class `PowerReservoir` instead of from class `ODEsElement`. The code in Figure 6 illustrates this approach and implements only the method `_fluxes_function_python`. All other methods are inherited from class `PowerReservoir`.

Note that this simplified implementation is a consequence of the required modification being relatively minor, i.e., a change solely in the constitutive function equation. More complex modifications, such as the inclusion/exclusion of input/output fluxes (e.g. inclusion of evapotranspiration into the `PowerReservoir`), would require the standard implementation approach described in Sect. 3.2.1.

3.3 Implementing a distributed model

This section illustrates the use of SuperflexPy to implement a distributed hydrological model. The example follows the procedure illustrated in Sect. 2.2, creating the more realistic model M02, developed in Dal Molin et al. (2020b) to provide streamflow predictions at 10 sub-catchments of the Thur catchment in Switzerland, see Figure 7a. Each sub-catchment receives its own forcing (precipitation, potential evapotranspiration, and temperature). Two HRU types are defined based on geology: consolidated and unconsolidated formations (Figure 7b). Both HRU types are characterized by the same model structure, which is shown in Figure 8 and represents an enhancement of the structure of model M4 with the following additions: (1) a "snow" reservoir, WR, which controls the partition of incoming precipitation between rainfall and snowfall based on temperature, (2) a lag function between UR and FR, and (3) a

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

"slow" reservoir, SR, which acts in parallel to FR and is controlled by the same equations as FR but with different parameter values.

This example represents a higher degree of complexity compared to the previous examples, both in terms of lumped model structure used for the HRU types (*unit*) and in terms of introducing a spatial discretization.

Given the spatial organization of the model, *nodes* are used to represent sub-catchments and *units* are used to implement HRU types. Note that the sub-catchments may share (one or more) HRU types, which in SuperflexPy translates into the *nodes* sharing (one or more) *units*. The *network* level is used to connect multiple *nodes*, and enables predictions at internal catchment locations. Figure 10 shows the SuperflexPy representation of the spatial organization shown in Figure 7.

We start by implementing the *units*. As can be seen in Figure 8, the model structure used to represent the HRUs has *elements* operating in parallel and, therefore, requires the use of connections. Figure 9 shows how the model structure is "translated" into the SuperflexPy framework. Recall, from Section 2.1, that the connection of *elements* within a *unit* must correspond to an acyclic directional graph, i.e., it should not contain feedback loops. Furthermore, *elements* can be connected only if they belong to two consecutive layers, which implies that "gaps" in the structure must be filled using a *transparent* element (which outputs the same fluxes it receives as inputs).

Comparing Figure 8 with Figure 9, we see how the HRUs structure has been implemented within SuperflexPy. The following implementation aspects are noteworthy:

1. The incoming precipitation is partitioned into rainfall and snowfall. This partitioning is done internally in WR. The SuperflexPy implementation of WR, in fact, takes care of two processes: (i) partitioning of precipitation into rainfall and snowfall; and (ii) simulation of snow processes (accumulation and melting). The output of WR is, logically, the sum of rainfall and snowmelt. Alternatively, a (new) splitter element could be defined to partition the fluxes between UR (rainfall) and WR (snowfall) based on temperature.
2. WR, as currently implemented, does not receive as input the potential evapotranspiration (PET), which is needed by the downstream *element* UR. The transfer of the PET to the UR, therefore, is done using the system "upper splitter-upper transparent-upper junction" (Figure 9) that allows to bypass the WR.
3. The parallel part of the structure is composed by two *elements* on one branch (lag and FR) and only one *element* on the other branch (SR). To satisfy the requirement of not having "gaps" in the *unit* structure, a transparent element (lower transparent) is added after.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

The code to setup this model is listed in Figure 11. As shown in the simplified example in Sect. 2.2, the user initializes and connects all model *components*, proceeding sequentially from the lowest level (*elements*) to the highest level (*network*). The procedure can be summarized as follows:

1. Lines 10-29: Initialize the *elements* needed for the lumped model structures used in the HRUs;
2. Lines 32-39: Initialize the units used to represent the HRUs, linking all the *elements*;
3. Lines 42-51: Initialize the nodes used to represent the sub-catchments. Both *units* are assigned to 9 *nodes*; the Mosnang sub-catchment contains a single HRU and hence only a single *unit* is assigned to the corresponding *node* (line 49).
4. Lines 54-60: Connect the *nodes* using a *network*; the topological structure of the *network* is defined by labeling each *node* with a unique identifier and specifying the downstream *node*.

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 Python dictionary, with keys given by the node identifiers and values given by the list of Numpy arrays representing the output fluxes.

4 Implementation details of SuperflexPy

This section presents additional technical details of SuperflexPy. A more detailed description is provided in the model documentation.

4.1 Parameters and states

All SuperflexPy *components* can have parameters and states. Parameters specify *component* characteristics, whereas states keep track of the *component* history. States and parameters are set as part of initializing the model *components*, and can be manipulated using `get` and `set` methods provided by the framework at all levels of its hierarchy (see the example in Sect. 2.2).

The parameters can be either constant or variable in time. Constant parameters represent the most common application of hydrological models. Time-variant parameters have been proposed in research applications to represent ”deterministic” system variability (e.g. seasonality, Westra et al., 2014) and/or “stochastic” system variability (e.g. Reichert and Mieleitner, 2009).

4.2 Modular design following the Object-Oriented paradigm

As noted in Sect. 2.3, SuperflexPy embraces the object-oriented paradigm (e.g. Meyer, 1988), which is widely used in general software and is increasingly adopted in scientific software.

The object-oriented design provides several advantages in the context of SuperflexPy:

- The inheritance principle enables the creation of new classes by extending existing ones. Inheritance reduces drastically the amount of new code that needs to be generated to implement a new model *component* (an example was provided in Sect. 3.2);
- Changes to a class (e.g. a *component*) and the creation of new classes can be carried out in isolation from the rest of the code, as long as the interfaces between classes are respected;
- When creating a model, only the necessary objects need to be initialized and used. This principle makes the model configuration effort roughly proportional to required model complexity, i.e., simple model structures can be constructed from the minimal set of required components. This capability avoids the overhead imposed in frameworks where simple model structures are explicitly constructed as special cases of more complex model structures. The simpler implementation may also reduce computational costs;
- Objects retain their history (states), which can be accessed post-run to undertake model analysis and/or subsequent computation;
- The modular nature of objects facilitates the development and testing of new code.

These benefits make it easier to achieve clean and maintainable code, which is essential for any practical modelling framework.

4.3 Numerical solution of ODEs

Reservoir elements are described using ordinary differential equations (ODEs), which are typically solved using numerical time-stepping approximations. There are many such approximations, e.g. Euler methods, Runge-Kutta methods, etc.

SuperflexPy separates the formulation of model equations from the solution of these equations. More specifically, flux equations are defined internally in *elements* (as shown in the example in Sect. 3.2), while the numerical method is specified externally (to the *element*) by defining a so-called “numerical approximator”. This separation enables the modeller to select the numerical method without making any changes to the model equations.

SuperflexPy provides two built-in numerical approximators, namely the fixed-step implicit and explicit Euler methods. The user can implement additional solvers, either by coding them directly or by interfacing with external ODEs solvers (e.g. from SciPy). As detailed in Sect. 4.4, the choice of numerical implementation, and its compatibility with optimizing compilers, may have a strong impact on the overall computational speed of the model.

4.4 Computational efficiency

Computational efficiency is a key requirement of a practical modelling framework. Conceptual hydrological models are often used in Monte Carlo uncertainty quantification, which requires 100's - 1000's of model runs (even millions in some cases). Model calibration is another common computationally demanding task required by most hydrological models.

The choice of programming languages inevitably requires a trade-off between computational efficiency and ease of use. The choice of Python for SuperflexPy was motivated by the attraction of a flexible and widely used scripting language in conjunction with two efficient numerical libraries: NumPy (Walt et al., 2011) and Numba (Lam et al., 2015). 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 interacts efficiently with NumPy arrays.

The combined use of Numpy and Numba is extremely effective when solving ODEs, where the method loops through a vector to perform element-wise operations. The built-in approaches for solving ODEs are compatible with such numerical infrastructure, and therefore enable fast computation times. Note that switching to ODEs solvers that do not take advantage of such libraries might dramatically increase the model runtime.

Numba offers drastic computational speed ups compared to native Python; our experimentation suggests runtime reductions by factors of up to 30. However, a drawback of Numba is the requirement to compile the code at runtime. For a lumped model composed of a few reservoirs, the Numba compilation time is of the order of a few seconds. Therefore, Numba will outperform Python when the simulation is long (e.g. 100,000 time steps, corresponding to roughly 12 years of hourly data) and/or when the model needs to be run a large number of times. For example, calibration to observed data (1000's of model runs) takes a few seconds with the Numba implementation compared to a couple of minutes with native Python execution (we here report only the runtime of the model itself, and exclude the runtime of the calibration tool procedures; more details on benchmarking in the documentation).

4.5 Ability to represent multiple fluxes and states

SuperflexPy can operate with multiple fluxes and state variables. In particular, connection elements, *units*, *nodes*, and the *network* are designed to deal with an arbitrary large number of fluxes.

This generality is intended to support the extension of SuperflexPy to model the transport of chemical substances contained in the water (e.g., Fenicia et al., 2010;Ammann et al., 2020). Representing fluxes of chemical substances requires state variables and fluxes in addition to those corresponding to water storages and water fluxes. The calculation of such fluxes also requires additional equations and parameters. The available examples in SuperflexPy do not include transport processes. However, the framework architecture foresee this need, and has been designed to be readily extended to accommodate such processes. For example, *elements* (e.g. a reservoir) can be created to implement the functionality required to simulate such fluxes (e.g. specifying the governing equations for substance transport, etc).

5 Summary and discussion

5.1 Defining the right complexity for a flexible model implementation

Achieving the envisaged flexibility of the SUPERFLEX framework in practical software is challenging, because flexible modelling functionality may come at the expense of ease of use and computational cost. The challenge in designing SuperflexPy has been to determine an appropriate level of abstraction for typical conceptual model applications. On one hand, high level of mathematical generality and abstraction offers the most flexibility, but risks losing a clear hydrological interpretation and may increase user effort in customizing the model. On the other hand, a framework that is over-restricted in terms of component behaviour may be easy to manage (as the number of modelling options is low), but it may not fulfil the promise of flexible models and may result in a limited range of application (e.g. limiting to lumped configurations only).

Conceptual models vary significantly in terms of complexity (e.g. from single bucket models to distributed models, from modelling streamflow alone to modelling water isotopes and/or other chemicals, etc.). In order to accommodate this range of potential model complexity in a flexible and practical way, we have organized the SuperflexPy software into four hierarchical levels, namely *element*, *unit*, *node*, *network*. As shown in Sect. 2.1 and in the examples of Sect. 3, these levels map to many types of conceptual modelling applications, from a single element (e.g. a reservoir), to a lumped model (typically composed of several elements, such as a combination of reservoirs), to a composition of lumped models, designed to provide prediction at a single outlet (e.g. a catchment with several HRUs, characterized by lumped models), and eventually to a distributed model capable of making predictions at multiple internal sub-catchments. An important outcome of this design choice is that the model configuration effort by the user becomes proportional to the required model complexity, so that simple models are much easier to configure.

The flexibility in customizing SuperflexPy elements is enhanced through its Object-Oriented design. As shown in Section 3.2, new components can be built by inheriting most methods from existing components and specifying only the required new features of interest. The potential downsides of using a scripted language Python in terms of computational speed are mitigated by the ability to use the Numba package.

5.2 Current limitations in model structure specification

As part of balancing the flexibility, ease of use, and computational performance of SuperflexPy, some restrictions have been imposed on the connectivity between model *components*.

The first restriction is that the *elements* within the *unit* must represent an acyclic directional graph, with no feedback loops from downstream to upstream *elements* (Sect. 2.1). This restriction enables the numerical solvers to proceed in a single pass from upstream to downstream and improves the computational performance of the framework. It also simplifies the specification of its structure. The restriction on internal model feedbacks is not expected to be overly limiting when developing conceptual hydrological models, as the fluxes in these models typically flow only in one direction (e.g. in the model M4 the water flows only from UR to FR and not vice versa). An example where internal model feedbacks may be required is given by the bidirectional interaction between surface water and groundwater in the hyporheic zone, where the direction of the fluxes changes depending on the state of the two components. Such interactions can still be modelled in SuperflexPy by introducing *elements* that embed such feedbacks internally. For example, the hyporheic zone, can be represented using a two-state reservoir with interacting states (e.g., Seibert et al., 2003). In other words, the restriction on model feedbacks applies to interactions *between* elements but not to the internal structure *within* an element.

The second restriction regards the topology of a *network*, which must represent a tree where any given *node* can connect and transfer fluxes to only a single downstream *node* (Sect. 2.1). This requirement is met by typical conceptual distributed models, as discussed in the introduction and illustrated in Sect. 3.3. However, fully integrated distributed models, such as Parflow or Mike She, do consider mutual dependencies between spatial elements, leading for example to 2D or 3D groundwater flows. Such configurations are considered beyond the scope of conceptual distributed models, and therefore are currently not supported in SuperflexPy.

5.3 Current usage and future developments

SuperflexPy is easy to install and run; it is written in pure Python and its dependencies are limited to the packages NumPy and Numba (Sect. 4.4). Installation can be done directly using the package installer for Python (pip) without the need of installing external libraries. We stress that SuperflexPy is not a wrapper of earlier SUPERFLEX code but offers a completely new implementation that is not constrained by choices taken in the earlier code versions.

SuperflexPy has already been used for research applications. Jansen et al. (2020) performed a "model mimicry" study where similarities and differences within the HBV "family" models were investigated. SuperflexPy was used to construct a compare a set of HBV-like models, assessing, among other things, the behavior of single *components* and the impact of numerical implementation. A list of publications using SuperflexPy is maintained on the documentation website.

In terms of future developments, we hope that SuperflexPy offers the hydrological community a new tool for research work and practical applications. Further SuperflexPy developments are likely to follow from such work and collaborations, including: (1) expansion of the library of model *components* beyond the ones here presented (as shown in the example in Sect. 3.2), and (2) more fundamental changes in response to future model applications. It is important to highlight that SuperflexPy can be used to create and combine new model *components*, thereby enabling experimentation with new model structures and general conceptualizations. The framework, therefore, is not limited to *components* and structures taken from existing models. Such collection, however, may be produced by storing the configurations that allow reproducing such models. The model documentation already contains a small sample of such configurations, which may grow as new users share their implementations with the community. In order to facilitate the use of the code, the code is accessible on GitHub with license LGPL-3.0 and distributed using the Python package installer PyPI (refer to the section at the end of the paper on code availability). The documentation provides a guide on how interested colleagues can contribute to the framework.

6 Conclusions

SuperflexPy is a new Python flexible modelling framework for building conceptual models ranging from lumped to distributed configurations. The framework offers detailed control over each aspect of model configuration, and caters to a wide range of typical conceptual model applications. In order to facilitate the model building process, the framework is organized using four hierarchical levels, namely *element*, *unit*, *node* and *network*, which correspond to conceptual model setups of increasing levels of complexity, namely a single element (e.g. a reservoir), a lumped model (e.g. a collection of interconnected reservoirs), a collections of lumped models designed to provide prediction at a single outlet, and a distributed model designed to provide predictions at internal sub-catchments. As the construction of a model that requires a certain hierarchical level does not require specifying the levels above it, the model configuration effort is proportional to the complexity of the application. The framework supports multiple states and fluxes in each *component*, and foresees an extension to water quality applications.

SuperflexPy builds on the experience of the authors and their colleagues in a series of hydrological case studies using the SUPERFLEX principles. SuperflexPy offers an open source implementation of the SUPERFLEX principles. In order to facilitate usability and further developments, we focused on several aspects of the model code, including ease of use and interfacing, availability, amenability of extensions, and computational efficiency.

We presented two examples illustrating common use cases of the SuperflexPy framework, including the implementation of a simple lumped reservoir model, and the implementation of a distributed model to represent a system of multiple sub-catchments and HRUs. It is hoped that the framework will contribute to ongoing efforts in the hydrological modelling community to develop more robust and representative models. The framework is open source, available with license LGPL-3.0 on GitHub.

Code availability

The organization of SuperflexPy as a software project is shown in Figure 12. The public GitHub repository contains all the source code of the framework, as well as examples and documentation. Package releases are distributed using the Python package index (https://pypi.org/project/superflexpy/). Releases are identified using a version number based on Semantic Versioning 2.0.0 (this paper refers to version 1.2.0) and assigned a DOI (Dal Molin et al. (2020a) for the release associated with this paper) through Zenodo. Documentation (https://superflexpy.readthedocs.io) and examples are updated periodically, and made available through Read the Docs and Binder respectively.

SuperflexPy is implemented using Python 3.7 and depends on NumPy (version used 1.19) and Numba (version used 0.50); compatibility with future versions will be assured through future releases of SuperflexPy.

SuperflexPy is available under the license LGPL-3.0. Users of the framework are invited to share their modelling solutions with the community by contributing to the GitHub repository.

Author contributions

All authors contributed to writing the paper. MDM designed, implemented, and documented the Python package, with input from FF and DK.

Competing interests

The authors declare that they have no conflict of interest.

Acknowledgements

References

Ammann, L., Doppler, T., Stamm, C., Reichert, P., and Fenicia, F.: Characterizing fast herbicide transport in a small agricultural catchment with conceptual models, Journal of Hydrology, 586, 124812, https://doi.org/10.1016/j.jhydrol.2020.124812, 2020.

Arnold, J. G., Srinivasan, R., Muttiah, R. S., and Williams, J. R.: LARGE AREA HYDROLOGIC MODELING AND ASSESSMENT PART I: MODEL DEVELOPMENT1, JAWRA Journal of the American Water Resources Association, 34, 73-89, 10.1111/j.1752-1688.1998.tb05961.x, 1998.

Bertuzzo, E., Thomet, M., Botter, G., and Rinaldo, A.: Catchment-scale herbicides transport: Theory and application, Advances in Water Resources, 52, 232-242, https://doi.org/10.1016/j.advwatres.2012.11.007, 2013.

Beven, K.: Changing ideas in hydrology — The case of physically-based models, Journal of Hydrology, 105, 157-172, https://doi.org/10.1016/0022-1694(89)90101-7, 1989.

Beven, K. J., and Kirkby, M. J.: A physically based, variable contributing area model of basin hydrology / Un modèle à base physique de zone d'appel variable de l'hydrologie du bassin versant, Hydrological Sciences Bulletin, 24, 43-69, 10.1080/02626667909491834, 1979.

Beven, K. J.: Uniqueness of place and process representations in hydrological modelling, Hydrol. Earth Syst. Sci., 4, 203-213, 10.5194/hess-4-203-2000, 2000.

Boyle, D. P.: Multicriteria calibration of hydrologic models, The University of Arizona., 2001.

Clark, M. P., Slater, A. G., Rupp, D. E., Woods, R. A., Vrugt, J. A., Gupta, H. V., Wagener, T., and Hay, L. E.: Framework for Understanding Structural Errors (FUSE): A modular framework to diagnose differences between hydrological models, Water Resources Research, 44, Artn W00b02

10.1029/2007wr006735, 2008.

Clark, M. P., Kavetski, D., and Fenicia, F.: Pursuing the method of multiple working hypotheses for hydrological modeling, Water Resources Research, 47, Artn W09301

10.1029/2010wr009827, 2011a.

Clark, M. P., McMillan, H. K., Collins, D. B. G., Kavetski, D., and Woods, R. A.: Hydrological field data from a modeller's perspective: Part 2: process-based evaluation of model hypotheses, Hydrological Processes, 25, 523-543, 10.1002/hyp.7902, 2011b.

Clark, M. P., Nijssen, B., Lundquist, J. D., Kavetski, D., Rupp, D. E., Woods, R. A., Freer, J. E., Gutmann, E. D., Wood, A. W., Brekke, L. D., Arnold, J. R., Gochis, D. J., and Rasmussen, R. M.: A unified approach for process-based hydrologic modeling: 1. Modeling concept, Water Resources Research, 51, 2498-2514, 10.1002/2015wr017198, 2015.

Craig, J. R., Brown, G., Chlumsky, R., Jenkinson, R. W., Jost, G., Lee, K., Mai, J., Serrer, M., Sgro, N., Shafii, M., Snowdon, A. P., and Tolson, B. A.: Flexible watershed simulation with the Raven hydrological modelling framework, Environ Modell Softw, 129, 104728, https://doi.org/10.1016/j.envsoft.2020.104728, 2020.

Dal Molin, M., Kavetski, D., and Fenicia, F.: SuperflexPy: The flexible language of hydrological modelling. v.1.2.0, Zenodo, 10.5281/zenodo.4303938, 2020a.

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, 10.5194/hess-24-1319-2020, 2020b.

David, P. C., Oliveira, D. Y., Grison, F., Kobiyama, M., and Chaffe, P. L. B.: Systematic increase in model complexity helps to identify dominant streamflow mechanisms in two small forested basins, Hydrological Sciences Journal, 64, 455-472, 10.1080/02626667.2019.1585858, 2019.

Eckhardt, K., and Ulbrich, U.: Potential impacts of climate change on groundwater recharge and streamflow in a central European low mountain range, Journal of Hydrology, 284, 244-252, https://doi.org/10.1016/j.jhydrol.2003.08.005, 2003.

Fenicia, F., Wrede, S., Kavetski, D., Pfister, L., Hoffmann, L., Savenije, H. H. G., and McDonnell, J. J.: Assessing the impact of mixing assumptions on the estimation of streamwater mean residence time, Hydrological Processes, 24, 1730-1741, 10.1002/hyp.7595, 2010.

Fenicia, F., Kavetski, D., and Savenije, H. H. G.: Elements of a flexible approach for conceptual hydrological modeling: 1. Motivation and theoretical development, Water Resources Research, 47, Artn W11510

10.1029/2010wr010174, 2011.

Fenicia, F., Kavetski, D., Savenije, H. H. G., Clark, M. P., Schoups, G., Pfister, L., and Freer, J.: Catchment properties, function, and conceptual model representation: is there a correspondence?, Hydrological Processes, 28, 2451-2467, 10.1002/hyp.9726, 2014.

Fenicia, F., Kavetski, D., Savenije, H. H. G., and Pfister, L.: From spatially variable streamflow to distributed hydrological models: Analysis of key modeling decisions, Water Resources Research, 52, 954-989, 10.1002/2015wr017398, 2016.

Feyen, L., Kalas, M., and Vrugt, J. A.: Semi-distributed parameter optimization and uncertainty assessment for large-scale streamflow simulation using global optimization/Optimisation de paramètres semi-distribués et évaluation de l'incertitude pour la simulation de débits à grande échelle par l'utilisation d'une optimisation globale, Hydrological Sciences Journal, 53, 293-308, 2008.

Formetta, G., Antonello, A., Franceschi, S., David, O., and Rigon, R.: Hydrological modelling with components: A GIS-based open-source framework, Environ Modell Softw, 55, 190-200, https://doi.org/10.1016/j.envsoft.2014.01.019, 2014.

Futter, M. N., Erlandsson, M. A., Butterfield, D., Whitehead, P. G., Oni, S. K., and Wade, A. J.: PERSiST: a flexible rainfall-runoff modelling toolkit for use with the INCA family of models, Hydrol. Earth Syst. Sci., 18, 855-873, 10.5194/hess-18-855-2014, 2014.

Gao, H., Hrachowitz, M., Fenicia, F., Gharari, S., and Savenije, H. H. G.: Testing the realism of a topography-driven model (FLEX-Topo) in the nested catchments of the Upper Heihe, China, Hydrol. Earth Syst. Sci., 18, 1895-1915, 10.5194/hess-18-1895-2014, 2014.

Hrachowitz, M., Fovet, O., Ruiz, L., Euser, T., Gharari, S., Nijzink, R., Freer, J., Savenije, H. H. G., and Gascuel-Odoux, C.: Process consistency in models: The importance of system signatures, expert knowledge, and process complexity, Water Resources Research, 50, 7445-7469, 10.1002/2014wr015484, 2014.

Ibbitt, R. P., and O'Donnell, T.: Designing conceptual catchment models for automatic fitting methods, IAHS Publication, 101, 462-475, 1971.

Jakeman, A. J., and Hornberger, G. M.: How Much Complexity Is Warranted in a Rainfall-Runoff Model, Water Resources Research, 29, 2637-2649, Doi 10.1029/93wr00877, 1993.

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

Kavetski, D., and Fenicia, F.: Elements of a flexible approach for conceptual hydrological modeling: 2. Application and experimental insights, Water Resources Research, 47, Artn W11511

10.1029/2011wr010748, 2011.

Kirchner, J. W.: Catchments as simple dynamical systems: Catchment characterization, rainfall-runoff modeling, and doing hydrology backward, Water Resources Research, 45, Artn W02429

10.1029/2008wr006912, 2009.

Kneis, D.: A lightweight framework for rapid development of object-based hydrological model engines, Environ Modell Softw, 68, 110-121, https://doi.org/10.1016/j.envsoft.2015.02.009, 2015.

Knoben, W. J. M., Freer, J. E., Fowler, K. J. A., Peel, M. C., and Woods, R. A.: Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT) v1.2: an open-source, extendable framework providing implementations of 46 conceptual hydrologic models as continuous state-space formulations, Geosci Model Dev, 12, 2463-2480, 10.5194/gmd-12-2463-2019, 2019.

Kraft, P., Vaché, K. B., Frede, H.-G., and Breuer, L.: CMF: A Hydrological Programming Language Extension For Integrated Catchment Models, Environ Modell Softw, 26, 828-830, https://doi.org/10.1016/j.envsoft.2010.12.009, 2011.

Lam, S. K., Pitrou, A., and Seibert, S.: Numba: a LLVM-based Python JIT compiler, Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, Association for Computing Machinery, Austin, Texas, Article 7 pp., 2015.

Leavesley, G. H.: Precipitation-runoff modeling system: User's manual, 4238, US Department of the Interior, 1984.

Lerat, J., Andreassian, V., Perrin, C., Vaze, J., Perraud, J.-M., Ribstein, P., and Loumagne, C.: Do internal flow measurements improve the calibration of rainfall-runoff models?, Water Resources Research, 48, 2012.

Lindstrom, G., Johansson, B., Persson, M., Gardelin, M., and Bergstrom, S.: Development and test of the distributed HBV-96 hydrological model, Journal of Hydrology, 201, 272-288, Doi 10.1016/S0022-1694(97)00041-3, 1997.

Marsh, C. B., Pomeroy, J. W., and Wheater, H. S.: The Canadian Hydrological Model (CHM) v1.0: a multi-scale, multi-extent, variable-complexity hydrological model – design and overview, Geosci. Model Dev., 13, 225-247, 10.5194/gmd-13-225-2020, 2020.

Matgen, P., Fenicia, F., Heitz, S., Plaza, D., de Keyser, R., Pauwels, V. R. N., Wagner, W., and Savenije, H.: Can ASCAT-derived soil wetness indices reduce predictive uncertainty in well-gauged areas? A comparison with in situ observed soil moisture in an assimilation application, Advances in Water Resources, 44, 49-65, https://doi.org/10.1016/j.advwatres.2012.03.022, 2012.

Maxwell, R. M.: A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling, Advances in Water Resources, 53, 109-117, https://doi.org/10.1016/j.advwatres.2012.10.001, 2013.

McInerney, D., Thyer, M., Kavetski, D., Githui, F., Thayalakumaran, T., Liu, M., and Kuczera, G.: The Importance of Spatiotemporal Variability in Irrigation Inputs for Hydrological Modeling of Irrigated Catchments, Water Resources Research, 54, 6792-6821, 10.1029/2017wr022049, 2018.

Meyer, B.: Object-oriented software construction, Prentice hall New York, 1988.

Moore, R. J., and Clarke, R. T.: A distribution function approach to rainfall runoff modeling, Water Resources Research, 17, 1367-1382, 10.1029/WR017i005p01367, 1981.

Moradkhani, H., and Sorooshian, S.: General review of rainfall-runoff modeling: model calibration, data assimilation, and uncertainty analysis, in: Hydrological modelling and the water cycle, Springer, 1-24, 2009.

Moser, A., Wemyss, D., Scheidegger, R., Fenicia, F., Honti, M., and Stamm, C.: Modelling biocide and herbicide concentrations in catchments of the Rhine basin, Hydrol. Earth Syst. Sci., 22, 4229-4249, 10.5194/hess-22-4229-2018, 2018.

Nijzink, R. C., Samaniego, L., Mai, J., Kumar, R., Thober, S., Zink, M., Schäfer, D., Savenije, H. H. G., and Hrachowitz, M.: The importance of topography-controlled sub-grid process heterogeneity and semi-quantitative prior constraints in distributed hydrological models, Hydrol. Earth Syst. Sci., 20, 1151-1176, 10.5194/hess-20-1151-2016, 2016.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., and Antiga, L.: PyTorch: An imperative style, high-performance deep learning library, Advances in Neural Information Processing Systems, 2019, 8024-8035,

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., and Dubourg, V.: Scikit-learn: Machine learning in Python, the Journal of machine Learning research, 12, 2825-2830, 2011.

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.

Refsgaard, J.: TERMINOLOGY, MODELLING PROTOCOL AND CLASSIFICATION OF HYDROLOGICAL MODEL CODES, in: Distributed Hydrological Modelling, 17, 1996.

Refsgaard, J. C., and Storm, B.: MIKE SHE, in: Computer Models of Watershed Hydrology, edited by: Singh, V. P., Water Resources Publications, Colorado, 809-846, 1995.

Reichert, P., and Mieleitner, J.: Analyzing input and structural uncertainty of nonlinear dynamic models with stochastic, time-dependent parameters, Water Resources Research, 45, 10.1029/2009wr007814, 2009.

Samaniego, L., Kumar, R., and Attinger, S.: Multiscale parameter regionalization of a grid-based hydrologic model at the mesoscale, Water Resources Research, 46, 10.1029/2008wr007327, 2010.

Seibert, J., and McDonnell, J. J.: On the dialog between experimentalist and modeler in catchment hydrology: Use of soft data for multicriteria model calibration, Water Resources Research, 38, 23-21-23-14, 10.1029/2001wr000978, 2002.

Seibert, J., Rodhe, A., and Bishop, K.: Simulating interactions between saturated and unsaturated storage in a conceptual runoff model, Hydrological Processes, 17, 379-390, 2003.

Sivapalan, M., Beven, K., and Wood, E. F.: On hydrologic similarity: 2. A scaled model of storm runoff production, Water Resources Research, 23, 2266-2278, https://doi.org/10.1029/WR023i012p02266, 1987.

Sivapalan, M., Blöschl, G., Zhang, L., and Vertessy, R.: Downward approach to hydrological prediction, Hydrological Processes, 17, 2101-2111, 10.1002/hyp.1425, 2003.

van Esse, W. R., Perrin, C., Booij, M. J., Augustijn, D. C. M., Fenicia, F., Kavetski, D., and Lobligeois, F.: The influence of conceptual model structure on model performance: a comparative study for 237 French catchments, Hydrology and Earth System Sciences, 17, 4227-4239, 10.5194/hess-17-4227-2013, 2013.

Vitolo, C., Wells, P., Dobias, M., and Buytaert, W.: fuse: An R package for ensemble Hydrological Modelling, Journal of Open Source Software, 1, 52, 10.21105/joss.00052, 2016.

Wagener, T., Sivapalan, M., Troch, P., and Woods, R.: Catchment Classification and Hydrologic Similarity, Geography Compass, 1, 901-931, doi:10.1111/j.1749-8198.2007.00039.x, 2007.

Walt, S. v. d., Colbert, S. C., and Varoquaux, G.: The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30, 10.1109/mcse.2011.37, 2011.

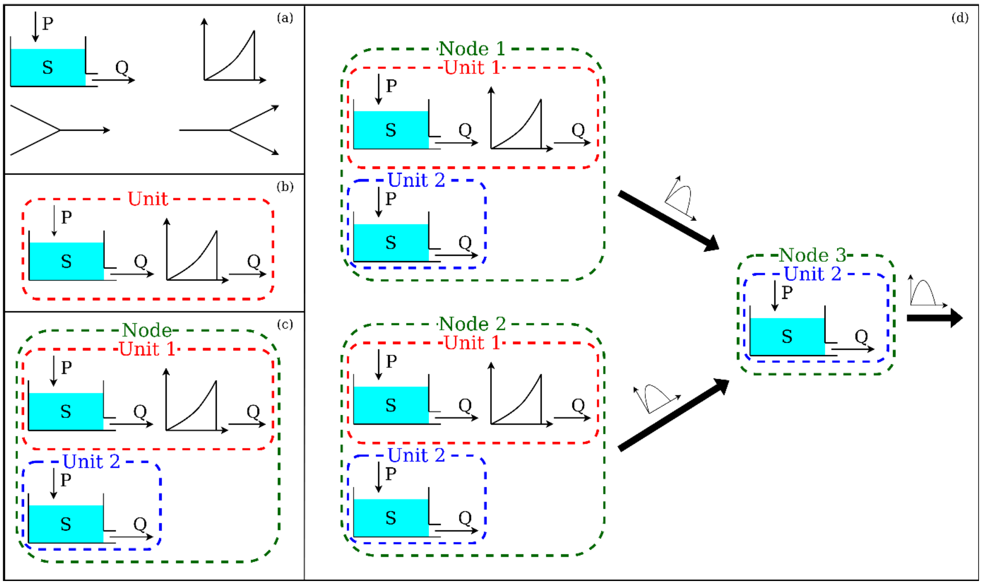
Westra, S., Thyer, M., Leonard, M., Kavetski, D., and Lambert, M.: A strategy for diagnosing and interpreting hydrological model nonstationarity, Water Resources Research, 50, 5090-5113, 10.1002/2013wr014719, 2014.

Wrede, S., Fenicia, F., Martínez-Carreras, N., Juilleret, J., Hissler, C., Krein, A., Savenije, H. H. G., Uhlenbrook, S., Kavetski, D., and Pfister, L.: Towards more systematic perceptual model development: a case study using 3 Luxembourgish catchments, Hydrological Processes, 29, 2731-2750, 10.1002/hyp.10393, 2015.

Young, P.: Data-based mechanistic modelling of environmental, ecological, economic and engineering systems, Environ Modell Softw, 13, 105-122, https://doi.org/10.1016/S1364-8152(98)00011-5, 1998.

Young, P. C., Tych, W., and Taylor, C. J.: The Captain Toolbox for Matlab, IFAC Proceedings Volumes, 42, 758-763, https://doi.org/10.3182/20090706-3-FR-2004.00126, 2009.

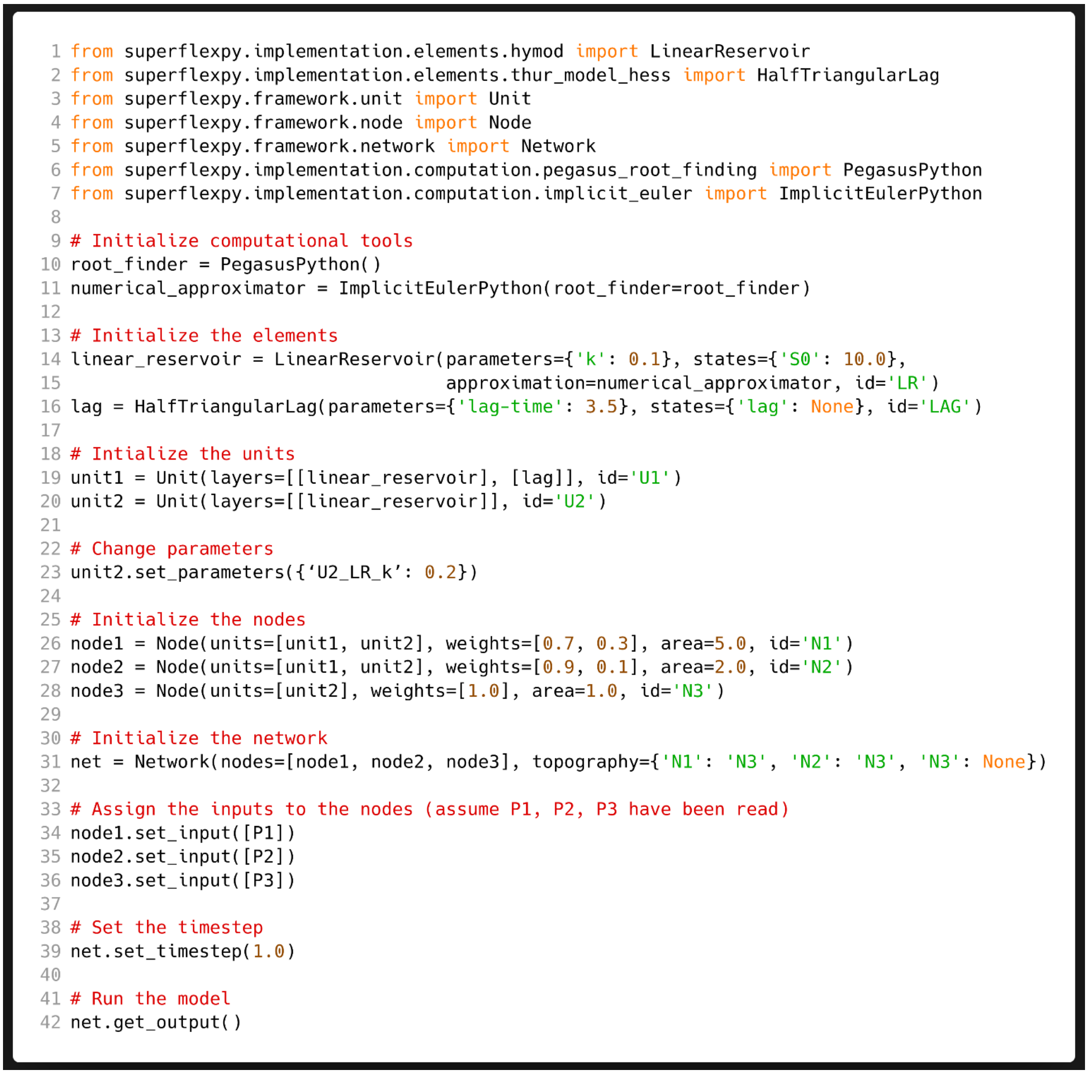
Figures



835

Figure 1. Four basic *components* of SuperflexPy. (a) *Elements* (e.g. reservoirs, lags, connections) are used to represent specific processes; (b) *Units* connect multiple elements and are intended to implement lumped catchment models; (c) *Nodes* collect multiple units that operate in parallel representing different landscape elements within a catchment; (d) *Network* connects multiple nodes and is used to represent
840 distributed setups.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020



```python
from superflexpy.implementation.elements.hymod import LinearReservoir
from superflexpy.implementation.elements.thur_model_hess import HalfTriangularLag
from superflexpy.framework.unit import Unit
from superflexpy.framework.node import Node
from superflexpy.framework.network import Network
from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython

# Initialize computational tools
root_finder = PegasusPython()
numerical_approximator = ImplicitEulerPython(root_finder=root_finder)

# Initialize the elements
linear_reservoir = LinearReservoir(parameters={'k': 0.1}, states={'S0': 10.0},
                                   approximation=numerical_approximator, id='LR')
lag = HalfTriangularLag(parameters={'lag-time': 3.5}, states={'lag': None}, id='LAG')

# Intialize the units
unit1 = Unit(layers=[[linear_reservoir], [lag]], id='U1')
unit2 = Unit(layers=[[linear_reservoir]], id='U2')

# Change parameters
unit2.set_parameters({‘U2_LR_k’: 0.2})

# Initialize the nodes
node1 = Node(units=[unit1, unit2], weights=[0.7, 0.3], area=5.0, id='N1')
node2 = Node(units=[unit1, unit2], weights=[0.9, 0.1], area=2.0, id='N2')
node3 = Node(units=[unit2], weights=[1.0], area=1.0, id='N3')

# Initialize the network
net = Network(nodes=[node1, node2, node3], topography={'N1': 'N3', 'N2': 'N3', 'N3': None})

# Assign the inputs to the nodes (assume P1, P2, P3 have been read)
node1.set_input([P1])
node2.set_input([P2])
node3.set_input([P3])

# Set the timestep
net.set_timestep(1.0)

# Run the model
net.get_output()
```

Figure 2. Code implementing the model in Figure 1d.

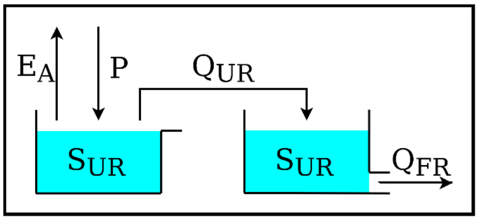
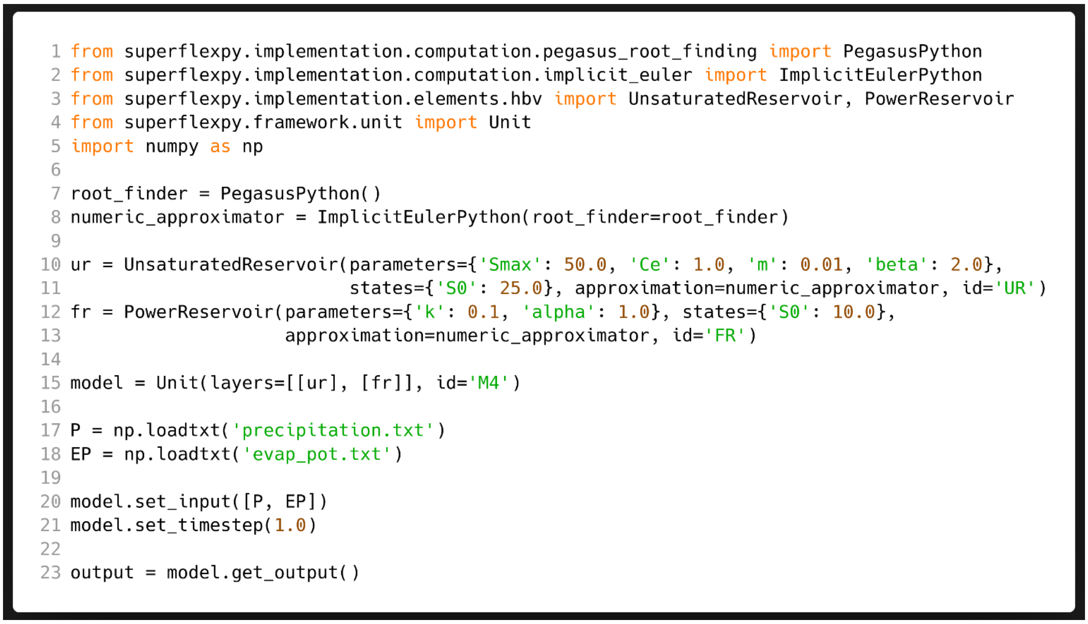


Figure 3. Schematic of model M4 used in the original SUPERFLEX case studies of Kavetski and Fenicia (2011).

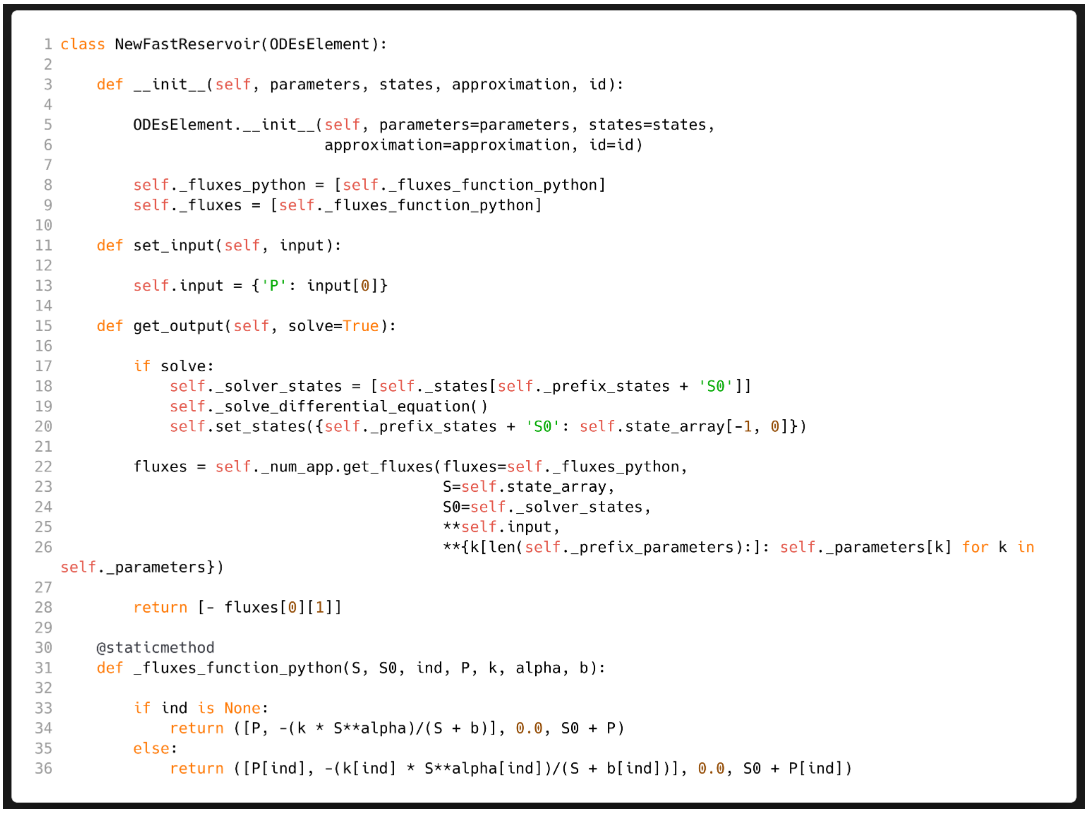
850



```python
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
import numpy as np

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')

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

P = np.loadtxt('precipitation.txt')
EP = np.loadtxt('evap_pot.txt')

model.set_input([P, EP])
model.set_timestep(1.0)

output = model.get_output()
```

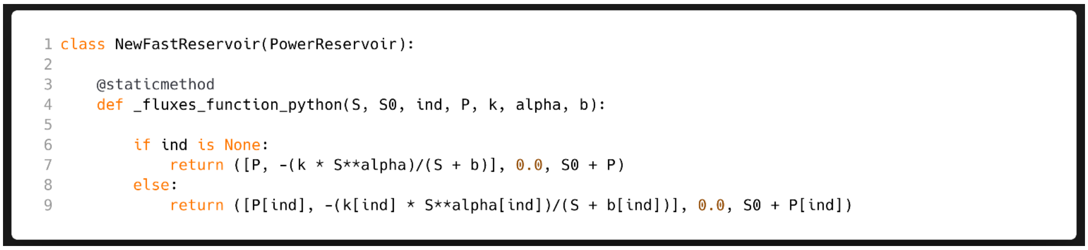
Figure 4. Code implementing model M4 in Figure 3.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020



```python
class NewFastReservoir(ODEsElement):

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

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

        self._fluxes_python = [self._fluxes_function_python]
        self._fluxes = [self._fluxes_function_python]

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

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

        return [- fluxes[0][1]]

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

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

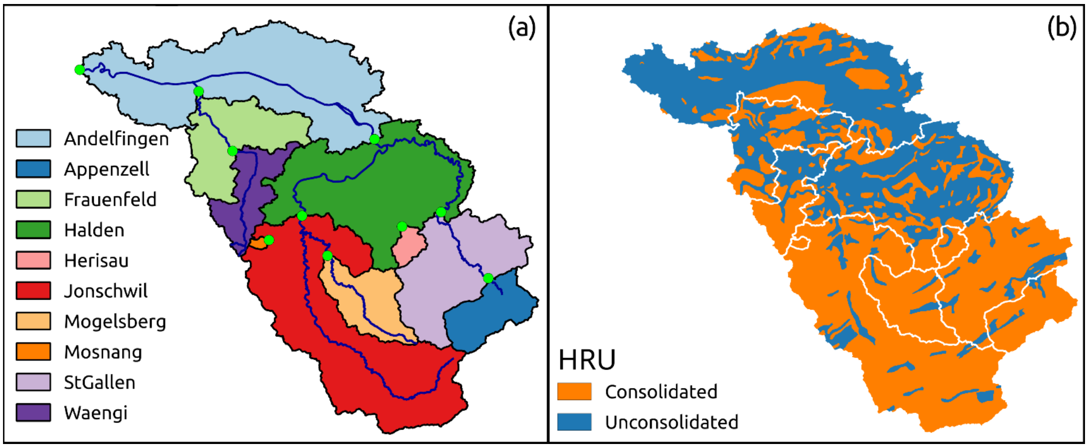
Figure 5. Standard method for implementing a new reservoir element `NewFastReservoir` by extending the class `ODEsElement`.



```
class NewFastReservoir(PowerReservoir):

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

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

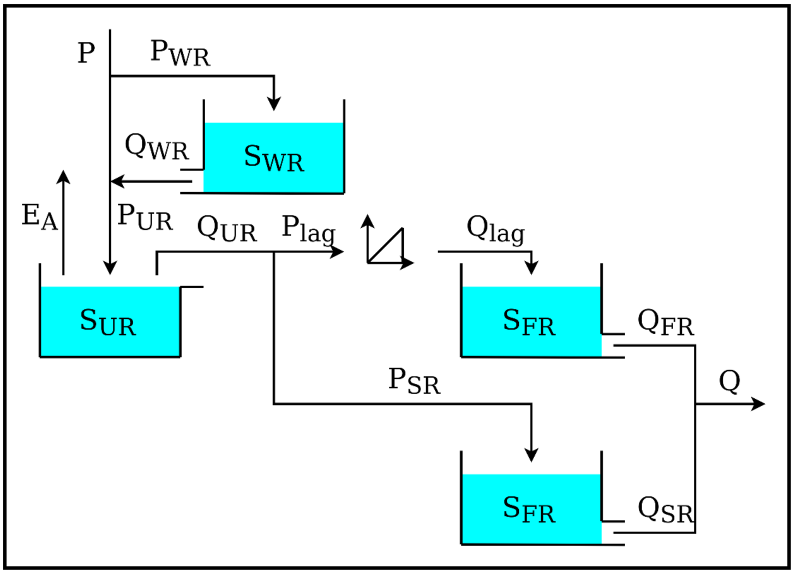
Figure 6. Simplified code implementing the `NewFastReservoir`.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020



865 **Figure 7.** Illustration of a distributed application of SuperflexPy: (a) subdivision of the Thur catchment into sub-catchments and (b) hydrological response units (HRUs) as presented in model M02 in Dal Molin et al. (2020b). The panels of the figure were originally published in figures 1a and 6 of Dal Molin et al. (2020b).

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020



870

Figure 8. Model structure used to represent the HRUs in model M02 in Dal Molin et al. (2020b).

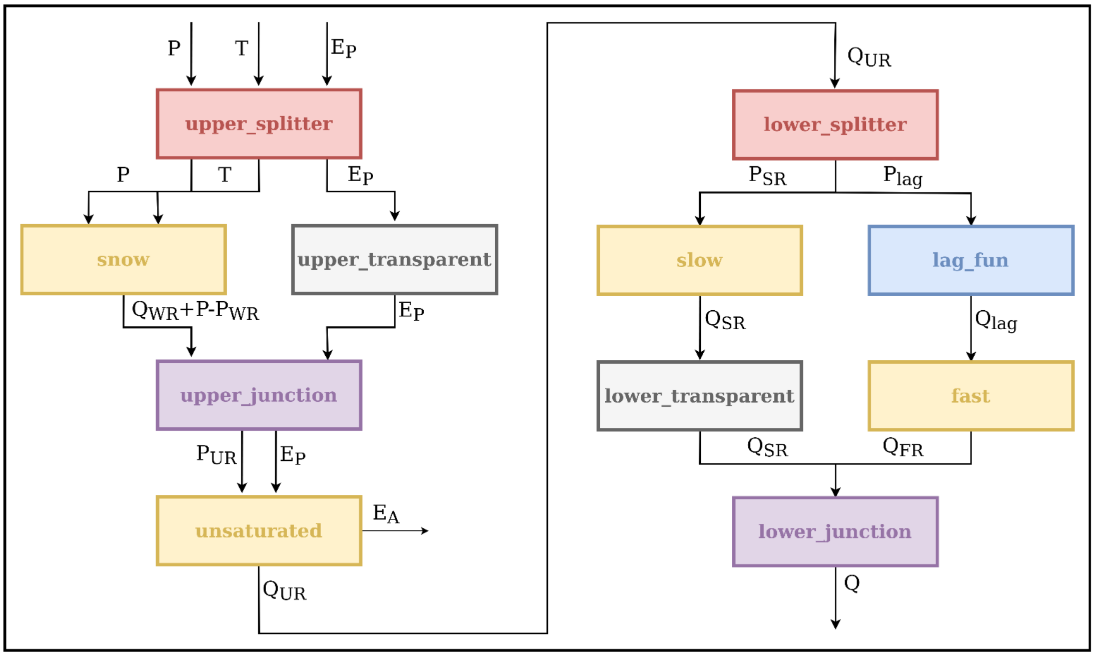


Figure 9. Translation to SuperflexPy of the model structure M02 presented in Figure 8.

875

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

Geoscientific Model Development
Discussions
Open Access

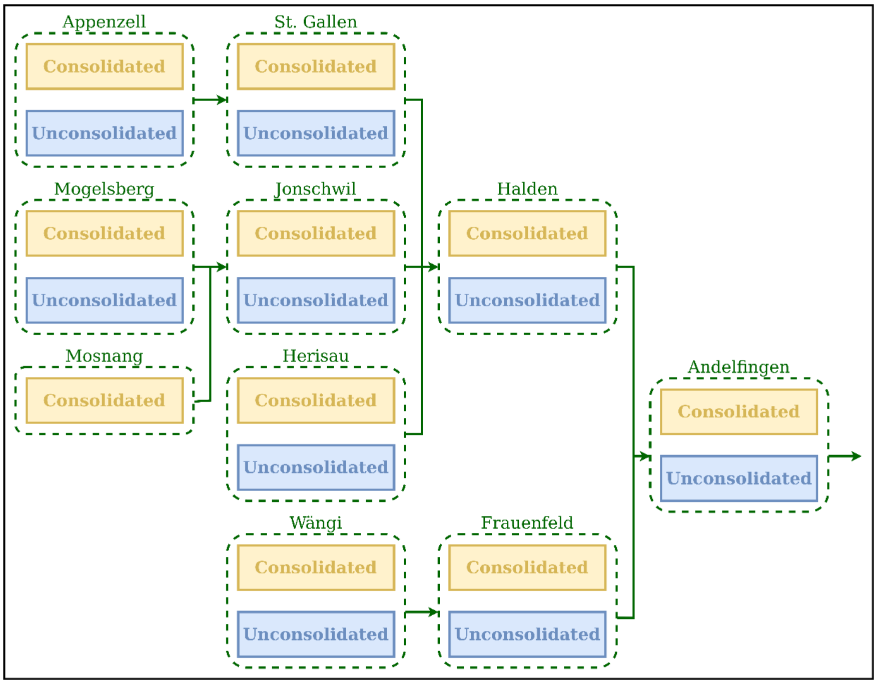
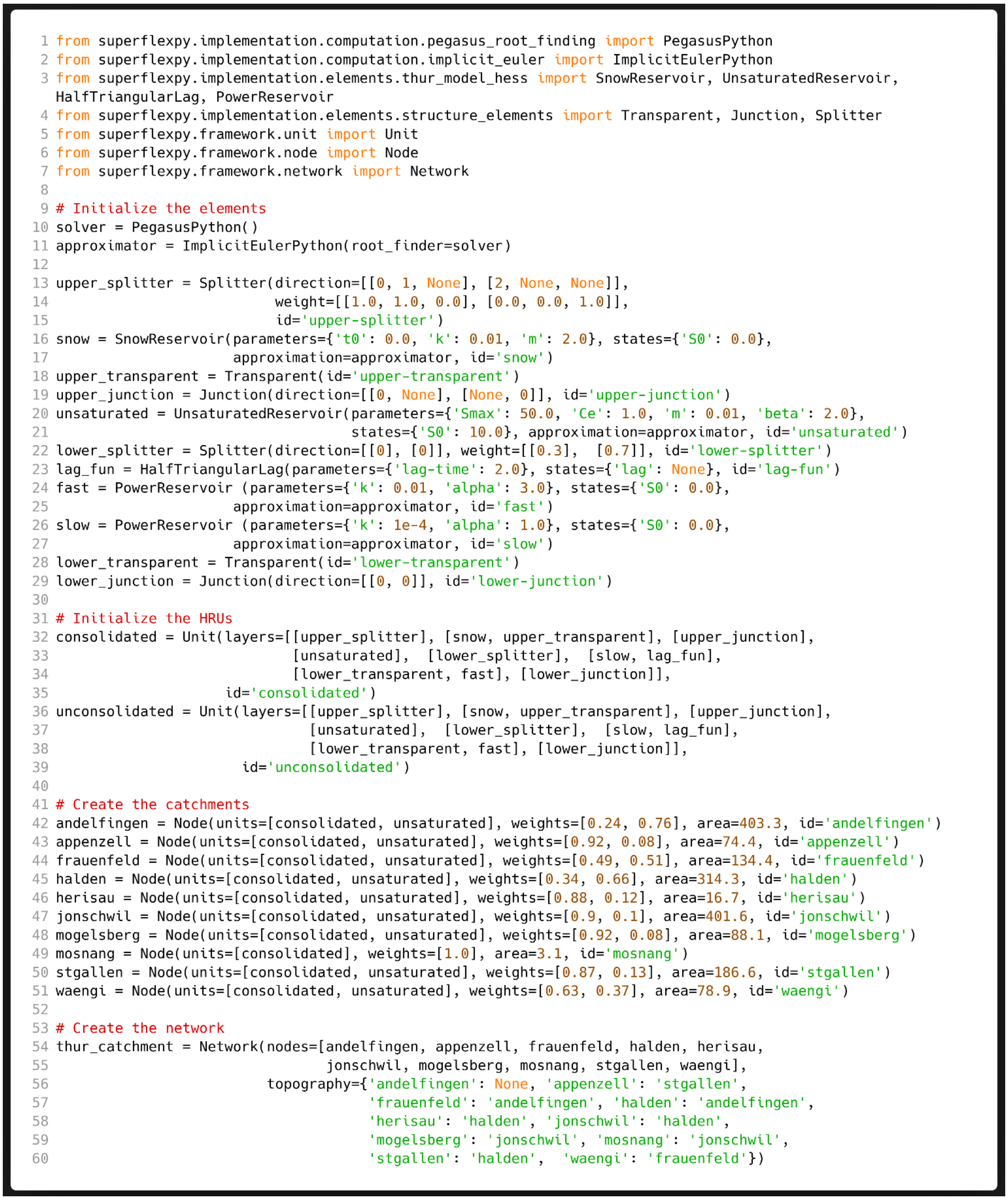


Figure 10. Spatial organization of the SuperflexPy model configuration used to simulate water fluxes in 880 the Thur catchment (M02 inDal Molin et al., 2020b). The units, used to represent the HRUs, are shown using the blue and yellow boxes. The nodes, used to represent the sub-catchments, are shown using the green dashed boxes. The group of nodes connected together (green arrows) creates a network.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

Geoscientific Model Development Discussions
Open Access



```
from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
from superflexpy.implementation.elements.thur_model_hess import SnowReservoir, UnsaturatedReservoir,
HalfTriangularLag, PowerReservoir
from superflexpy.implementation.elements.structure_elements import Transparent, Junction, Splitter
from superflexpy.framework.unit import Unit
from superflexpy.framework.node import Node
from superflexpy.framework.network import Network

# Initialize the elements
solver = PegasusPython()
approximator = ImplicitEulerPython(root_finder=solver)

upper_splitter = Splitter(direction=[[0, 1, None], [2, None, None]],
                          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],  [0.7]], 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},
                     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')

# Initialize 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')

# Create the catchments
andelfingen = Node(units=[consolidated, unsaturated], weights=[0.24, 0.76], area=403.3, id='andelfingen')
appenzell = Node(units=[consolidated, unsaturated], weights=[0.92, 0.08], area=74.4, id='appenzell')
frauenfeld = Node(units=[consolidated, unsaturated], weights=[0.49, 0.51], area=134.4, id='frauenfeld')
halden = Node(units=[consolidated, unsaturated], weights=[0.34, 0.66], area=314.3, id='halden')
herisau = Node(units=[consolidated, unsaturated], weights=[0.88, 0.12], area=16.7, id='herisau')
jonschwil = Node(units=[consolidated, unsaturated], weights=[0.9, 0.1], area=401.6, id='jonschwil')
mogelsberg = Node(units=[consolidated, unsaturated], weights=[0.92, 0.08], area=88.1, id='mogelsberg')
mosnang = Node(units=[consolidated], weights=[1.0], area=3.1, id='mosnang')
stgallen = Node(units=[consolidated, unsaturated], weights=[0.87, 0.13], area=186.6, id='stgallen')
waengi = Node(units=[consolidated, unsaturated], weights=[0.63, 0.37], area=78.9, id='waengi')

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

885 **Figure 11.** Code implementing the distributed model in Figure 9 and Figure 10.

https://doi.org/10.5194/gmd-2020-409
Preprint. Discussion started: 16 December 2020

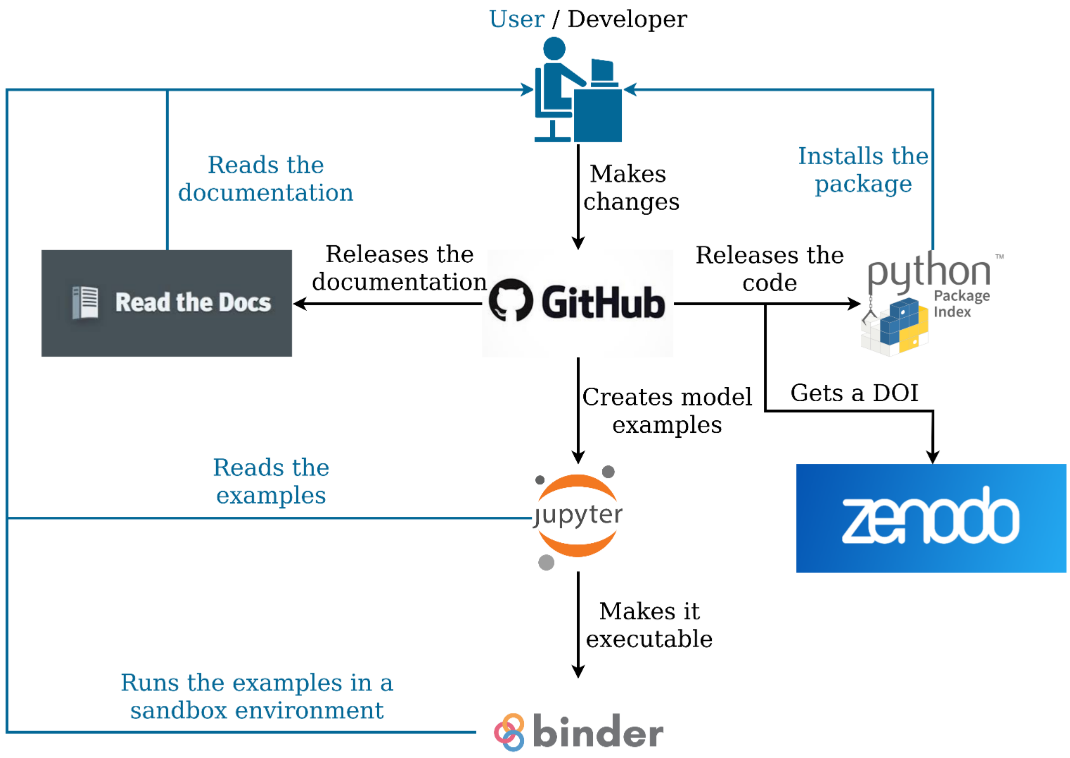


Figure 12: Organization of the SuperflexPy project.