SuperflexPy 1.2.~~0: an~~1: An open source Python framework for building, testing and improving conceptual hydrological models

Marco Dal Molin[1,2], Dmitri Kavetski[1,3,4], Fabrizio Fenicia[1]

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

Correspondence to: ~~Marco Dal Molin (marco.dalmolin~~Fabrizio Fenicia (fabrizio.fenicia@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~~which approximate 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~~components. 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, discusses the software design and implementation choices, 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. Comprehensive documentation is available online and provided as supplementary material to this paper. 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.~~

35 # Table of Contents

1 Introduction 4

1.1 Conceptual hydrological models 4
40 1.2 Hydrological model structure and flexible modeling frameworks 5
1.3 Aims 9

2 Description of SuperflexPy 10

2.1 General organization 10
2.2 A simple illustration of SuperflexPy: creating a new model from existing components 13
45 2.3 Creating new model *components* with SuperflexPy 15

3 Examples of building hydrological models using SuperflexPy 16

3.1 Implementing SUPERFLEX configuration M4 16
3.2 Changing the equations of the fast reservoir in M4 18

3.2.1 General approach for creating a new reservoir with SuperflexPy 18

50 3.2.2 Simplified approach for creating a new reservoir element (from an existing element) 19

3.3 Implementing a distributed model 19

4 Implementation details of SuperflexPy 21

4.1 Parameters and states 21
4.2 Modular design following the Object-Oriented paradigm 22
55 4.3 Numerical solution of ODEs 22
4.4 Computational efficiency and language choice 23
4.5 Ability to represent multiple fluxes and states 24

5 Discussion 25

5.1 Balancing functionality, scope, and usability in a flexible model implementation 25
60 5.1.1 Structural flexibility 25

5.1.2 Spatial flexibility 26

5.1.3 Usability 27

5.1.4 Possibility of extension and customization 28

5.1.5 Computational efficiency 28

65 5.2 Current restrictions in model structure specification 29
5.3 Current usage and future developments 30

6 Summary and conclusions 31

Code availability 32
Author contributions 33
70 Competing interests 33
Acknowledgements 33
Financial support 33
References 33
Figures 38
75

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 forcing; 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 attempt to explicitly 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~~common in practice because they offer an attractive compromise in terms of predictive accuracy, computational demands, and amenability to interpretation. ~~Popular~~Common 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~~).~~ at the catchment outlet). Alternatively, distributed configurations can be used ~~when~~if the interest is in modeling hydrological behavior at ~~individual landscape sections~~internal locations (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., 2020), 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 (e.g., Boyle et al., 2001)~~,~~, and typically employs discretization based on principles of "hydrological similarity" (e.g., Sivapalan et al., 1987)~~, such as HRUs~~; HRU-based discretization is particularly common (e.g., Leavesley, 1984). In many ~~cases~~applications, 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~~motivated 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~~specific locations and/or 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~~notion of comparing ~~different~~multiple 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~~.~~ according to their intended scopes of application. For example, GEOframe-NewAge (Formetta et al., 2014), SUMMA (Clark et al., 2015), ~~RAVEN ,~~ and CHM (Marsh et al., 2020) focus on the realm of physically based models~~; the~~. The CAPTAIN toolbox (Young et al., 2009) is a general toolkit for time series analysis~~; machine~~. Machine learning frameworks such as scikit-learn (Pedregosa et al., 2011) ~~or~~and 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. Examples of such frameworks include FUSE (Clark et al., 2008)~~Many frameworks have been developed for such purpose, and offer different degrees of flexibility. For example, FUSE allows exchanging the components of 4 common models (Sacramento, PRMS, TOPMODEL and ARNO/VIC). SUPERFLEX~~, SUPERFLEX ~~(Fenicia et al., 2011) allows building model structures from generic elements (reservoirs, lag function and connections). CMF , represents the model as an abstract network of elements and can be adopted for conceptual models. PERSiST allows to create semi-distributed bucket-type models and is designed to be coupled with a water quality model. ECHSE is a framework for development of object-based conceptual hydrological model engines. MARRMoT 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 and R . SUPERFLEX, besides its original Fortran implementation~~ (Fenicia et al., 2011), CMF (Kraft et al., 2011)~~has also been coded in Matlab~~, PERSiST (Futter et al., 2014), ECHSE (Kneis, 2015), MARRMoT (Knoben et al., 2019), and RAVEN (Craig et al., 2020).

When discussing a mathematical model, it is relevant to distinguish its conceptual principles from its software implementation. In the hydrological literature, modelling concepts and their software implementation have been presented both jointly and separately. For example, the original FUSE publication (Clark et al., 2008) introduced the modelling concepts, while subsequent work (Vitolo et al., 2016) provided an R implementation. The original SUPERFLEX publications presented the modelling principles (Fenicia et al., 2011) and demonstrated its capabilities (Kavetski and Fenicia, 2011); while Fortran and Matlab implementations were developed as part of research work (e.g., 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~~, these implementations have not been published or made available as standalone products. In contrast, some models, (e.g., MARRMoT) have been presented with a publication describing both the theoretical principles and the software implementation.

A software implementation should fulfill the intended goals of the flexible framework, in particular supporting the envisaged flexibility in terms of processes representation, spatial distribution, numerical solution methods, etc. The software implementation should also be accessible to users in terms of ease of installation, operation, eventual extension, etc. Existing frameworks approach these conceptual and practical requirements with different priorities, e.g., focusing on selected modelling objectives (e.g., model mimicry) and/or limiting the range of applications (e.g., only to lumped setups), in order to simplify the model formulation and operation.

In terms of application scope of a flexible framework for conceptual hydrological modeling, we focus on 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~~In terms of software implementation, we consider the following 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~~to simulate 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 , 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~~These criteria are challenging or even impossible to meet simultaneously. Hence, implementing a flexible framework 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 defined above 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~~"~~", i.e., the degree of flexibility, 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~~select and combine individual model elements~~,~~ (e.g., reservoirs, lag functions, etc.), as well as to ~~change such flux expressions in multiple specific parts of the model~~build customized elements.

The development of the proposed framework capitalizes on the authors' collective experience in ~~using the earlier implementations~~hydrological model design and application. The original Fortran implementation of SUPERFLEX, hereafter referred to as SUPERFLEX-F90, has been used 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., 2020), 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). The earlier Flex framework was used in studies exploring the use of multivariate data to refine the model structure (e.g., Fenicia et al., 2006, 2008). The modelling framework FUSE was used for a range of experiments in process representation (e.g., Clark et al., 2011b), data analysis (e.g.,

Henn et al., 2018), and numerical solution (e.g., Clark and Kavetski, 2010; Kavetski and Clark, 2010). The SUMMA framework represented an application of flexible modelling principles to physically based modelling. These applications have highlighted the versatility of the SUPERFLEX principles, and of flexible modelling approaches in general, to solve increasingly complex modelling problems~~, and~~ – but have ~~led to insights into~~also highlighted implementation choices that limit the effectiveness and range of application of current software~~ design and configuration aspects not available in the earlier implementations.~~. This ~~study reports on these developments and offers an open source~~work provides a new implementation of SUPERFLEX ~~for use by the hydrological community~~that addresses many of these limitations.

1.3 Aims

This ~~work~~paper introduces SuperflexPy, ~~an~~which is a new open-source Python software ~~that implements the principles~~ implementation of the SUPERFLEX ~~framework~~principles for conceptual hydrological model development. Particular attention is given to the challenges of implementing a framework that achieves the flexibility envisaged by SUPERFLEX and flexible frameworks in general. 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~~and developing new ~~ones.~~models;
3. Provide a broad discussion of ~~how the~~the hydrological modelling software implementation challenges and of how 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.~~. Section 2 ~~presents~~ describes the SuperflexPy ~~to the hydrological community; Sect.~~architecture and building blocks, and provides a short demo (aims 1 and 2). Section 3 illustrates selected applications of the framework, including the setup of SUPERFLEX configurations used in earlier case studies~~, as well as how to~~ and the use SuperflexPy to create new *elements*~~; Sect.~~ (aim 2). Section 4 provides more technical SuperflexPy ~~implementation~~ details, useful for understanding the usage and general potential of the framework~~; Sect.~~ (aim 1). Section 5 discusses ~~the scope of SuperflexPy, its~~ SUPERFLEX design choices in the context of existing flexible frameworks, including current limitations~~,~~ and future developments~~.~~ (aim 3). Finally, ~~Sect.~~Section 6 ~~draws the~~ provides a brief overall summary and conclusions.

The examples presented in the paper are generally intended to provide the intuition and reasoning behind SuperflexPy. The model documentation provides detailed information and use instructions. The documentation is available and maintained online (refer to "code availability" section); references from the paper to the documentation point to the static PDF version provided as supplementary material to this paper.

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 to as "*components*". These *components* are shown in Figure 1 and described below. Further practical details are provided in Chapter 4 of the supplementary material:

1. ***Element*** (Figure 1a). This level represents the basic model building ~~blocks~~block 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 hydrological process or response mechanism within the catchment.

   The **reservoir** element is used to conceptualize processes involving the storage and release of water and other fluxes. It 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~~if necessary (e.g., to keep track of snow and liquid water separately). Mathematically, a multistate reservoir can be represented by a system of differential equations of the form of equations (1) and (2)~~represent transport)~~.

   The solution of equation (1) is usually obtained numerically using external numerical procedures referred to as "numerical approximators" (see Section 4.3).

The **lag function** element is used to represent delays in the transmission of the fluxes (e.g., routing). It is described mathematically by a convolution integral:,

$$\mathbf{Y}(t)=\mathbf{X}(t)*\mathbf{g}_{\mathrm{H}}(t;\boldsymbol{\theta})=\int_{0}^{T}\mathbf{X}(t-\tau)\,\mathbf{g}_{\mathrm{H}}(\tau;\boldsymbol{\theta})\,\mathrm{d}\tau \tag{3}$$

305 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:

310 There is a general mathematical correspondence between reservoirs and lag functions (e.g., Nash, 1957). SuperflexPy users can select the element specification best suited to their specific context.

The **connection** element is used to connect two or more *elements* whenever a direct connection is not possible. For example, connection elements are used when a flux needs to be split among

315 multiple *elements* downstream (splitter), or, vice versa, when multiple fluxes need to be aggregated (junction). A particular type of connection is represented by the "transparent" element, which simply outputs the same fluxes it receives as inputs, and is used to facilitate the connection between elements (see description of *unit* below).

All connection elements are stateless and can be represented mathematically as follows,

320
$$\mathbf{Y}(t)=\mathbf{g}_{\mathrm{C}}\left(\mathbf{X}(t);\boldsymbol{\theta}\right) \tag{4}$$

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. or an HRU in a distributed model. Multiple reservoir

325 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 The multiple *elements* within a *unit* are arranged in *layers*, with the only restriction being thatfollowing restrictions: (i) feedback loops *between* the *elements* are not allowed and (ii) *elements* can be

330 connected only if they belong to two consecutive *layers*. Fluxes between elements in nonconsecutive layers are passed using transparent elements. The concept of *layers* will be elaborated and illustrated in Section 5.1.1.; see also Section 4.2 of the supplementary material.

In technical terms, the ~~overall model~~ structure formed by the *elements* must be ~~an acyclic~~a directional acyclic graph~~. This~~ (DAG). The motivation and implications of these design ~~restriction is motivated by~~ choices on model generality and computational efficiency are elaborated in Sections 5.1.1 and 5.2~~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)~~or HRUs within the catchment. Each *unit* within a *node* is characterized by a weight~~ (e.g. representing~~, which typically represents its area fraction~~) specified by~~ or, more generally, its contribution to the ~~modeler~~total outflow of the *node*. The weights are used to combine the output fluxes from the *units* into the total output flux of the *node*. Another important attribute of a *node* is its “area”, which is used when multiple *nodes* are combined into a *network* (see below).
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~~routes the fluxes from upstream *nodes* (leaves of the tree) to the final downstream *node* (root of the tree). ~~The routing~~ Routing delays in the river network can be simulated ~~adding delays (lag)~~by feeding *node* outputs into lag function elements. The area of each *node* is used to determine its contribution to the ~~*nodes* outputs~~total outflow of the *network*. Only a single network can be used in a given SuperflexPy model.

~~This~~The hierarchical organization of SuperflexPy makes the effort required to configure ~~SuperflexPy~~it to a new problem proportional to the problem complexity. In particular, many common model setups can be constructed without necessarily using all levels listed above, thus reducing configuration effort. Some representative examples are given below:

- 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. Beven and Kirkby, 1979; Gao et al., 2014; Nijzink et al., 2016);
- Level 4 is needed only in models that generate predictions at interior points, such as SWAT (Arnold et al., 2012), GEOframe-NewAge (Formetta et al., 2014), and distributed SUPERFLEX applications (e.g. Fenicia et al., 2016; Dal Molin et al., ~~2020b~~2020).

Examples of SuperflexPy models implemented at Levels 2 and 4 are given later in ~~Sect.~~Section 3. Note that the association of specific SuperflexPy components to specific hydrological entities, e.g., the use of *units* for HRUs and *nodes* for sub-catchments, is not intended as a rigid prescription. Other association choices may be favored by the modeler depending on the required model structure and spatial connectivity.

The clarity of visual model representation is particularly important in flexible frameworks because they can generate many subtly different configurations (e.g., Bancheri et al., 2019). The model schematics in this paper indicate explicitly every element, including reservoirs, lag functions, and junctions (e.g., Figure 1).

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.~~Section 4.2.

All SuperflexPy *components* are characterized by states and/or parameters, which are controlled programmatically using dedicated methods (refer to Section 4.1).

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~~The catchment is represented using a *network*, the sub-catchments are represented using *nodes*, and the HRUs are represented using *units*. Two distinct HRU-specific model structures are specified, and are implemented using *elements*. The corresponding SuperflexPy code is shown in Figure 2. An extended version of this demo is provided in Section 6.5 of the supplementary material.

~~For~~In 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.~~Section 2.3. ~~Even more~~More

complex setups are described in ~~Sect.~~Section 3 and in the ~~online documentation (see code availability section).~~supplementary material.

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~~ approximator `ImplicitEulerPython` and root finder `PegasusPython` needed to solve the ODEs associated with the reservoir elements are also imported (~~more on this in Sect.~~see Section 4.3~~).~~ for details). The import operation is shown in ~~lines~~Lines 1-7.

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

~~In detail:~~

~~*Elements* are~~More specifically:

L1. An *element* is initialized by specifying its parameters, states, ~~identifier (`id`)~~ and, ~~when needed~~where relevant, the numerical solver (~~lines~~Lines 14-16). Each element is given an identifier (`id`) for subsequent use, as shown on Line 23.

~~L2. *Units* are~~A *unit* is initialized by specifying the *elements* that compose ~~them~~it and the identifier (~~lines~~Lines 19-20). As noted earlier in Section 2.1~~The~~, the connectivity between *elements* is defined by conceptualizing the *unit* as a succession of *layers* that contain the *elements*. ~~Further~~More complex examples ~~on this~~ are given in ~~Sect.~~Section 3.

~~L3.~~L2. 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~~operation is shown on ~~line~~Line 23 for the `LinearReservoir` element.

~~L4.~~L3. ~~*Nodes* are~~A *node* is initialized by specifying the *units* that compose ~~them~~it, their contribution (weight) to the *node* output, the influence area of the *node* (here, the area of the sub-catchment), and the identifier (~~lines~~Lines 26-28).

~~Within a given node, *units* operate independently from each other.~~

~~L5.~~L4. The *network* is initialized by specifying the *nodes* that compose it and their connectivity, called ~~`topography` (line~~`topology` (Line 31). The connectivity is defined indicating, for each *node*, the *node* downstream of it. A network identifier is not specified (as only a single network can be used).

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~~Line 39 (~~note that~~ variable time steps are also supported, see Section 4.5.1 of the ~~documentation~~supplementary material).

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

Note that all input quantities provided to SuperflexPy, including fluxes, time step length, parameters, states, areas, etc., must have consistent units. To reduce model code complexity and execution overhead, we take the perspective that unit checks represent pre-processing and are best handled by the user according to their own preferences and standards. Output fluxes have the same (assumed) units as input fluxes, e.g., if precipitation is in mm/h, then streamflow is also in mm/h, etc.

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*. To this end, SuperflexPy provides a library of built-in high-level *components* that can be extended to achieve the desired functionality. We anticipate that the SuperflexPy *components* most likely to require extension are the *elements*, where new ~~reservoir~~ constitutive functions may be required ~~for~~in reservoir elements and new ~~applications.~~weight functions may be required in lag function elements. 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~~takes advantage of 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 the new 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.~~implement a new model *component*.

A detailed example of ~~making use of~~ this ~~design~~procedure is given in ~~Sect.~~Section 3.2, which shows how to implement a reservoir with a new storage-discharge relationship. More examples are provided in Chapters 8 and 9 of the supplementary material.

3 Examples of building hydrological models using SuperflexPy

This section provides more detailed 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., 2020)~~. Further details and more examples are provided in the model documentation (see code availability section).~~.

Compared to the demo in Section 2.2, which was intended to give a general sense of model building with SuperflexPy, the examples in this section represent "realistic" applications of SuperflexPy, including setting up a spatially distributed model with multiple HRUs and more complex model structure. Further technical details and additional examples, including the implementation of popular conceptual models (e.g., GR4J, HYMOD), are provided in the supplementary material (chapters 8-11).

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. The numerical procedures are imported and initialized on Lines 1-2 and 7-8 respectively. Similar to the model described in ~~Sect.~~Section 2.2, the two model *elements* (UR and FR) are already implemented. Hence, the user only needs to import ~~(lines~~the elements (Lines 1-3) and initialize ~~(lines~~their parameters (Lines 7-13~~) the *elements* together with the numerical routines.~~). Next, the *unit* ~~that comprises the two reservoirs~~ is imported (~~line~~Line 4) and initialized ~~(line~~to contain the two reservoirs (Line 15). The ~~input data, namely precipitation and PET time series, are set on line 20. Input data~~model configuration 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~~then complete.

~~The model configuration is then complete – line 23 runs the model with given input data to produce the model outputs.~~The loading of input data from text file(s), databases, etc. is separate from the configuration of SuperflexPy, and can be carried out using any suitable Python library or function. In this example, we use Numpy to read time series of precipitation and PET from a text file, as shown in Lines 17-18. The corresponding SuperflexPy inputs are set using these Numpy arrays, as shown on Line 20. Further practical details on input-output are provided in Section 4.5.5 of the supplementary material.

The model can now be run with the given input data to produce the model outputs, as shown on Line 23. 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

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

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

An *element* with this storage-discharge relationship has not been implemented in SuperflexPy yet (as of version 1.2.~~0~~1). The following sections give two approaches for creating such an *element*.

3.2.1 ~~Standard~~General approach for creating a new reservoir with SuperflexPy

The ~~standard~~general 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~~.~~ (see Section 8.1 of the supplementary material for full details). The new class must override the following methods:

- `__init__`: constructor of the class. Its main purpose is to ~~call~~invoke the constructor of the parent class (~~lines~~Lines 5-6) and to point ~~at~~to the method used to calculate the fluxes, here, `_fluxes_function_python` (see also ~~Sect.~~Section 4.3, which ~~shows~~illustrates the efficiency 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~~Line 15) that is then used when setting up and solving the model equations;
- `get_output`: ~~calls~~invokes the functionalities implemented by the `ODEsElement` to solve the *element* equation over the entire simulation (all time steps). Lines ~~18-~~20-22 get the current state of the reservoir, ~~call~~invoke the ODE solver, and set the state to ~~the~~its final value. Lines ~~22-26~~24-28 get the output flux arrays from the numerical approximator (see ~~Sect.~~Section 4.3). Line ~~28~~30 returns a list with the ~~outflow~~output of the *element* (here, the streamflow);
- `_fluxes_function_python`: calculates the fluxes for a given state, inputs, and parameters. Line ~~34~~36 implements the vector version while ~~line 36~~Line 38 implements the scalar version. Both versions are needed by the numerical approximator (see Section 4.3~~.~~; further practical details are provided in Section 8.1 of the supplementary material).

The new *element* `NewFastReservoir` is now defined and can be used in the "new" version of M4, ~~as shown~~ in ~~Sect. for~~lieu of the ~~pre-existing~~previous *element* `PowerReservoir`. The Object-Oriented

features of Python are very useful here to enable the new class `NewFastReservoir` 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 *element* parameters and states, etc.

3.2.2 Simplified ~~method~~approach for creating a new reservoir element (from an existing element)

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 therefore achieved by ~~making~~inheriting this class ~~inherit~~ directly from class `PowerReservoir` ~~instead of~~rather than 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~~general implementation approach described in ~~Sect.~~Section 3.2.1.

3.3 Implementing a distributed model

This section illustrates the ~~use~~implementation of ~~SuperflexPy to implement a~~an HRU-based, distributed hydrological model~~. The example follows~~, intended to simulate streamflow in a nested catchment. This implementation requires the ~~procedure~~entire workflow illustrated in ~~Sect.~~Section 2.2~~, creating the more realistic~~. The example is provided by model M02, developed in Dal Molin et al. (2020) to provide streamflow predictions at 10 sub-catchments of the Thur catchment in Switzerland~~, see~~ (Figure 7a~~.~~).

Each sub-catchment receives its own forcing ~~(~~, namely 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~~ . This HRU model structure ~~of~~differs from model structure M4 (section 3.1~~with~~) in the following ~~additions: (1~~additional *elements*: (i) a "snow" reservoir, WR, which controls the partition of incoming precipitation between rainfall and snowfall based on temperature, (~~2~~ii)

a lag function between UR and FR, and (~~3~~iii) a “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~~Similar to the simpler previous ~~examples, both~~ example in Section 3.1~~terms of~~, this "lumped" model structure ~~used for the HRU types (~~is implemented as a *unit*~~) and~~. However, a key difference is that in ~~terms of introducing a spatial discretization.~~the previous example the *unit* represented the entire system, whereas here it is part of a more complex system.

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 HRU model structure ~~used to represent the HRUs~~ has *elements* operating in parallel and, therefore, requires the use of connections. Figure 9 shows how the HRU model structure is “translated” into ~~the~~a SuperflexPy ~~framework~~*unit*. 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 (~~elements, which ~~outputs~~output the same fluxes ~~it receives~~they receive as inputs~~)~~. Splitters and junctions are used to divide and merge the fluxes to implement the parallel flow paths.

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

1. The incoming precipitation is partitioned into rainfall and snowfall. This partitioning is done internally in the WR *element*. 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~~have been 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~~ Therefore, the transfer of ~~the~~ PET values to the UR~~, therefore,~~ element is ~~done~~implemented using ~~the system “~~a separate path composed by three *elements*, labelled "upper splitter~~-~~", "upper transparent~~-~~", and "upper junction~~”~~" (Figure 9~~)~~

~~that allows to bypass~~). This choice simplifies the interface of element WR at the ~~WR~~expense of a somewhat more complicated model structure with additional elements.

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 the SR.

The code to setup this model is ~~listed~~detailed in Figure 11. ~~As shown in~~Similar to the ~~simplified~~earlier example in ~~Sect.~~Section 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~~Line 49).
4. Lines 54-60: Connect the *nodes* using a *network*~~; the topological structure~~. The topology of the *network* is defined by ~~labeling~~indicating, for each node~~ with a unique identifier and specifying,~~ the downstream ~~*node*~~one.

The *network* runs the *nodes* from upstream to downstream, collects their outputs, and routes them to the outlet. Customized routing functions can be implemented, as shown in Section 9.1 of the supplementary material. 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 time series of output fluxes over the simulation period.

4 Implementation details of SuperflexPy

This section presents additional technical details of SuperflexPy~~.~~ needed to understand better some aspects of the functioning of the framework. A more detailed and practical description is provided in the ~~model documentation~~supplementary material.

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.~~Section 2.2).

The parameters can be either constant or variable in time. Constant parameters represent the most common ~~application~~set up of hydrological models. ~~Time-variant~~In conceptual hydrological modelling, time-varying 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., Kuczera et al., 2006; Reichert and Mieleitner, 2009; Renard et al., 2011); see also earlier work in data-based mechanistic modelling (e.g.~~ Reichert and Mieleitner, 2009~~., Young, 2000).

4.2 Modular design following the Object-Oriented paradigm

As noted in ~~Sect. ~~Section2.1, 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~~see example ~~was provided ~~in ~~Sect.~~Section 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~~The mass balance of reservoir elements ~~are~~is described using ordinary differential equations (ODEs), which are typically solved (approximately) using numerical time-stepping ~~approximations. There are many~~algorithms. Many such ~~approximations~~algorithms have been described in the numerical methods literature, e.g. Euler methods, Runge-Kutta methods, etc. (e.g., Butcher and Goodwin, 2008).

SuperflexPy separates the formulation of model equations from the solution of these equations. More specifically, flux equations are defined internally ~~in~~ as methods of the *elements* (as shown in ~~the example in Sect.~~Section 3.2), while the numerical ~~method~~algorithm is specified externally (to the *element*) by defining a so-called "numerical approximator". ~~This~~The numerical approximator is a procedure, which constructs a numerical approximation of the differential equation(s) controlling the element. If the numerical approximator implements an implicit time stepping scheme, it will generally require an auxiliary "root finder", which is a procedure that solves nonlinear algebraic equation(s). The separation of equations and solvers in the model specification enables the ~~modeller~~modeler to select the numerical method without making any changes to the governing model equations. Further details are provided in Section 5.1 of the supplementary material.

SuperflexPy provides two built-in numerical approximators, namely the fixed-step implicit and explicit Euler time stepping schemes (e.g., Clark and Kavetski, 2010). The implicit Euler equations are solved using the Pegasus root finder (Dowell and Jarratt, 1972)~~methods.~~. The user can implement additional ~~solvers~~numerical algorithms, either by coding them directly or by interfacing with external ~~ODEs~~code (e.g. ODE solvers ~~(e.g.~~ from SciPy). –As detailed next in ~~Sect.~~Section 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 and language choice

Computational efficiency is a key requirement of a practical modelling framework. ~~Conceptual~~Model calibration via parameter optimization is a common computationally demanding task required by most hydrological models, typically requiring hundreds or thousands of model runs. Moreover, conceptual hydrological models are often used in Monte Carlo uncertainty quantification, ~~which requires 100's – 1000's of model runs (~~with comparable or even larger computational cost (up to millions of model runs in some cases). ~~Model calibration is another common computationally demanding task required by most hydrological models.~~

The choice of programming ~~languages~~language inevitably requires ~~a~~ trade-~~off~~offs 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~~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~~compiles (at runtime) a ~~normal~~ Python method ~~to~~into machine code that interacts efficiently with ~~NumPy~~Numpy arrays.

The combined use of Numpy and Numba is ~~extremely~~particularly effective when solving ODEs, where the ~~method loops through a vector to perform~~numerical algorithm performs element-wise sequential operations. The built-in SuperflexPy 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.~~each time it is executed (run). 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~~multiple years of hourly data) and/or when the model needs to be run a large number of times. For example, as a broad illustration of runtimes on a standard laptop, calibration of a HYMOD-like SuperflexPy model to observed daily data ~~(~~, requiring 1000's of model runs~~)~~ each with 1000 time steps, takes a few seconds with the Numba implementation compared to a couple of minutes with native Python execution~~ (we~~. Note that here ~~report only~~we refer to the runtime of the SuperflexPy model itself, and exclude the runtime of the calibration tool procedures; more details on benchmarking are given in section 5.3 of the supplementary material. Examples of interoperability of SuperflexPy with external libraries for model calibration (e.g., SPOTPY, Houska et al., 2015)~~in the documentation).~~ are given in chapter 14 of the supplementary material.

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* can accommodate an arbitrarily large number of fluxes. The use of multiple fluxes has been already shown in the model structure described in section 3.3~~are designed to deal with an arbitrary large number of fluxes.~~, where the `upper_splitter` handles three different variables (precipitation, temperature, and PET). Additional examples are provided in the supplementary material (e.g., chapters 10, 11).

~~This generality~~The capability to simulate multiple fluxes and states is intended to support the extension of SuperflexPy to ~~model~~new modelling scenarios. Several such scenarios may be of interest, including the transport of chemical substances (e.g., Fenicia et al., 2010; Ammann et al., 2020)~~contained in~~, the interaction between frozen and liquid water in a snow *element* (e.g., Jansen et al., 2021), interactions in the saturated/unsaturated soil zones (e.g., Seibert et al., 2003)~~. 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~~, and so forth.

730 While the current examples in SuperflexPy do not include ~~transport processes. However~~all the cases listed above, the framework architecture ~~foresee this~~ anticipates the need for more general simulation functionality, and has been designed to ~~be readily extended~~support extension to accommodate such multi-state processes.

5 Discussion

735 **5.1 Balancing functionality, scope, and usability in a flexible model implementation**

A software implementation that maximizes flexibility and usability is challenging to achieve, because flexible modelling functionality may increase configuration effort and computational cost. Existing flexible frameworks have approached this tradeoff with different priorities, based on their respective modelling objectives and paradigms.

740 The following sections offer a brief discussion of the design choices made by SuperflexPy in the context of selected existing frameworks with a similar scope. The discussion makes use of Table 1 and Table 2 ~~For example,~~, which summarize key design choices related to usability and simulation capabilities respectively.

5.1.1 Structural flexibility

745 Structural flexibility refers to the flexibility in how *elements* ~~(e.g. a reservoir) can be created to implement the functionality required to simulate such fluxes (e.g.~~ can be connected to compose the structure of the model (i.e., of the *unit*, following SuperflexPy's terminology). This consideration applies both to lumped and distributed models; the flexibility in specifying the spatial organization of the model is considered separately in Section 5.1.2~~governing equations for substance transport, etc).~~.

750 Some flexible frameworks are implemented using a master structure that incorporates all supported model configurations. In these implementations, the user can choose the flux equation(s) (e.g., FUSE, SUPERFLEX-F90) and/or activate/deactivate specific elements (e.g., SUPERFLEX-F90), but cannot change the overall connectivity of model elements. To the extent that the master structure is sufficiently general, it may not unduly restrict the practical usage of the framework.

755 Other frameworks (e.g., MARRMoT) propose a collection of existing conceptual model structures ready to use, which have been implemented following the same design rules in order to allow for a fair comparison. Such frameworks are typically intended for model intercomparison studies.

The most general frameworks allow connecting the elements freely without constraints. A distinction can be made between frameworks that allow for mutual interactions between the elements (e.g., CMF) and frameworks that do not allow such interactions (e.g., ECHSE).

SupeflexPy adopts the latter philosophy, allowing to connect the *elements* freely within the *unit* but restricting mutual interactions, i.e., constraining the structure to be a DAG (see Section 5.2). Moreover, we have chosen to define the DAG as a succession of *layers*, listing the *elements* in order from upstream to downstream and allowing for parallel flow paths (e.g., see the model structure in Figure 9Summary and discussion

4.6 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). This "list" formulation has been selected in preference to other methods for defining a graph, e.g., connectivity matrix, adjacency list, etc., for the following reasons: (i) simplicity/scalability, as the list dimension scales linearly with the number of *elements*, in contrast to the connectivity matrix approach where this scaling is quadratic; (ii) arguably better readability, as the *elements* are listed in the order they appear in the DAG; and (iii) it guarantees a graph topology without loops. Note that other popular modelling water isotopes and/or other chemicalstools (e.g., neural networks) adopt this type of formulation.

5.1.2 Spatial flexibility

Most frameworks (e.g., CMF, ECHSE, SUPERFLEX-F90, etc.). In order to accommodate this range of potential model complexity in a flexible and practical way, we have organized the .) support multiple types of spatial discretization (e.g., lumped, HRUs, sub-catchments, grids, etc.). Some frameworks (e.g., FUSE, MARRMoT) support solely lumped models.

SuperflexPy ~~software into four~~uses 4 hierarchical levels~~, namely *element*, *unit*, *node*, *network*. As shown in Sect. and in the examples of Sect. , 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),~~ of *components,* intended to facilitate the formulation of models that range in spatial complexity from a simple lumped model, to a composition of lumped models~~, designed to provide~~ intended for prediction at a single ~~outlet~~location (e.g. a catchment with several HRUs~~, characterized by lumped models~~), and ~~eventually~~ultimately 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~~locations. The use of a hierarchical set of components could be contrasted to a framework based solely on the lowest level components, here, *elements*. The use of higher level components enables the modeler to capture explicitly the natural groupings in the catchment of interest, e.g., sub-catchments, HRUs, etc.

5.1.3 Usability

The ~~flexibility~~usability of a framework can be judged according to several aspects.

The first aspect is how a framework is operated. Some frameworks are standalone and operated through a graphical interface (e.g., PERSiST) or the command line interface (e.g., SUPERFLEX-F90). Other frameworks are designed as libraries that can be called from the user code in ~~customizing~~a specific programming language to initialize, configure and run the model (e.g., CMF, MARRMoT; SUPERFLEX-F90 also allows this option when using the source code from Fortran). SuperflexPy ~~elements is enhanced through its Object-Oriented~~ is implemented as a Python package. Models can be created using a Python script and interfaced easily with external libraries (examples are provided in chapter 14 of the supplementary material).

The second aspect is the scope of the framework. Most frameworks (e.g., SUPERFLEX-F90, ECHSE) adopt, by design, the philosophy of "one tool per problem" and limit their functionality to the simulation of hydrological processes. Other frameworks integrate tools for parameter calibration and sensitivity analysis, uncertainty quantification, pre- and post-processing tasks such as input unit checks/conversions, etc. (e.g., RAVEN, PERSiST). SuperflexPy adopts the first philosophy: it limits its functionality to hydrological simulation.

Finally, documentation is another key aspect in the usability of a framework. Virtually all considered frameworks provide such documentation to a varying degree of detail. SuperflexPy documentation is available online and explains in detail how to use and further develop the framework.

Figure 12~~. As shown in Section ,~~ shows the online software management tools that are used to develop and deploy SuperflexPy. The framework itself, including source code, documentation, examples, etc., is hosted on GitHub. Automated workflows are then used to create new releases (PyPI), get DOIs for the software releases (Zenodo), host the documentation (ReadTheDocs), and run the examples (Jupyter and Binder). From a general user perspective, this setup improves model accessibility and reproducibility. From a developer and contributor perspective, it reduces the effort needed to maintain and extend the framework.

5.1.4 Possibility of extension and customization

Most frameworks have open source code and permissive licenses, making it possible to modify and extend their codebase. Within this category, some frameworks are specifically intended to be customized (e.g., implementing new functionalities) as part of their regular usage without an expectation of "developer-level" skills (e.g., ECHSE). Other frameworks do not envisage customization in their primary scope, but can still be modified by modelers with appropriate programming expertise in consultation with available developer guides (e.g., RAVEN).

Some frameworks have not been released as open source, and the only way to access their codebase for customization and extension is by contacting their developers (e.g., SUPERFLEX-F90, PERSiST).

SuperflexPy is designed to facilitate extension and customization as part of its regular usage. New *components* can be ~~built~~created by ~~inheriting most methods from~~extending or modifying existing components, as demonstrated in Section 3.2~~ and specifying only the ~~.

5.1.5 Computational efficiency

The computational efficiency of a model code, i.e., the time required ~~new features of interest. The potential downsides of using a scripted language Python in ~~to run a simulation, depends primarily on two aspects, namely the programming language and the numerical algorithms.

In terms of programming languages, most frameworks have been implemented in C/C++ and Fortran, which enable very fast computation. These implementations can be either purely single-language (e.g., FUSE, RAVEN), or wrapped within a scripting language to provide a more suitable interface (e.g., CMF). Amongst the considered existing frameworks, only MARRMoT is implemented entirely in an interpreted language (Matlab/Octave).

In terms of numerical algorithms, a wide range of options are available for solving differential equations. Broadly speaking, time stepping algorithms can be classified as implicit or explicit, and may employ fixed or adaptive step size. The choice of algorithm and its settings brings tradeoffs between solution accuracy,

algorithm complexity and computational cost. In the context of model development and comparison, it is important to separate the specification of model equations from the choice of numerical solution and to use robust numerical methods to avoid spurious artefacts (e.g., Kavetski and Clark, 2010). The majority of frameworks implement this separation and provide a choice of built-in numerical algorithms.

SuperflexPy, while written entirely in Python (a nominally "slow" language), makes several implementation choices to reduce computational costs. These choices include the use of efficient numerical libraries (section 4.4) and the solution of the *elements* in succession (DAG, section 5.2). The choice of numerical algorithm for individual elements is left to the user (section 4.3~~computational speed are mitigated by the ability to use the Numba package.~~). The (recommended) built-in approximators include the implicit Euler scheme with fixed step size, which offers stability and smoothness benefits valuable in parameter estimation contexts.

~~4.7~~5.2 Current ~~limitations~~restrictions 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~~a *unit* must ~~represent an acyclic~~ form a directional acyclic graph~~,~~ (DAG), with no feedback loops from downstream to upstream *elements* (~~Sect.~~Section 2.1). This restriction enables the numerical solvers to proceed, at each time step, in a single pass from upstream to downstream *elements* 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~~where the fluxes ~~in these models~~from a given *element* typically ~~flow~~depend only on the state in that *element* and not on downstream *elements*. In such systems, flows occur only in one direction ~~(~~, e.g. in ~~the~~ model M4 the water flows ~~only~~ from UR to FR ~~and~~but not vice versa~~). An~~ . A counter-example where internal model feedbacks ~~may be~~are required is given by the bidirectional interaction between surface water and groundwater in the hyporheic zone, where the ~~direction of the~~exchange flux (or fluxes ~~changes depending~~) depends on ~~the state of the two components.~~both states. 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 SuperflexPy restriction on model feedbacks applies to interactions *between* elements, but not to ~~the internal structure~~interactions *within* an element.

The second restriction, which also applies at the *unit* level, derives from the decision to define the DAG as a succession of *layers* (section 5.1.1). This choice simplifies the model definition in typical use cases,

when there are many *elements* with relatively few connections (i.e., the DAG is "sparse" rather than "dense"). However, the definition of a DAG as a succession of *layers* requires the *elements* to be connected directly one to the other, without skipping *layers*. Hence the need for transparent elements, which output the inputs they receive and are used to fill the gaps that arise when two or more parallel flow paths have a different number of *elements*. An example of such model configuration is given in Figure 9 ~~regards~~, where a transparent *element* (labeled “lower_transparent”) is used to fill the gap in layer 7.

The third restriction is that the topology of a *network*~~, which~~ must represent a tree where any given *node* can connect and transfer fluxes ~~to~~ only to a single downstream *node* (~~Sect.~~Section 2.1). ~~This requirement is met by typical conceptual~~ This restriction has a similar motivation to the restriction of a *unit* structure to a DAG, and allows for a simple and efficient computational implementation, which starts from the headwater *nodes* and proceeds downstream one *node* at a time. Typical distributed conceptual models~~, as discussed in the introduction and~~ meet this restriction, for example as illustrated in ~~Sect.~~Section 3.3. However, fully integrated distributed models, such as Parflow ~~or~~and Mike~~-~~-She, do ~~consider~~include mutual dependencies between spatial elements, e.g., leading ~~for example~~ to 2D or 3D groundwater flows. Such configurations are considered beyond the scope of conceptual distributed models, and therefore are not currently ~~not~~ supported in SuperflexPy.

~~4.8~~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~~Numpy and Numba (~~Sect.~~Section 4.4). Installation can be done directly using the package installer for Python (pip) ~~without the need of installing~~and does not require (additional) external libraries. We stress that SuperflexPy is not a wrapper of earlier SUPERFLEX-F90 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. (2021) performed a “model mimicry” study where similarities and differences within the HBV “family” of models were investigated. SuperflexPy was used to construct a ~~compare a~~ set of HBV-like models~~, assessing, among other things,~~ and compare them in terms of the behavior of ~~single~~individual model components~~ and~~, the impact of numerical implementation, and so forth. A list of publications using SuperflexPy is maintained on the documentation website.

In terms of future developments, we hope that SuperflexPy offers the broader hydrological community a versatile new tool for research work and practical applications. Further SuperflexPy developments are likely to follow from such work and collaborations, including: (~~1~~i) expansion of the library of model

components beyond the ones here presented (as shown in the example in ~~Sect.~~Section 3.2), and (~~2~~ii) more fundamental ~~changes~~developments 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~~ – though such collections could be also produced. The SuperflexPy model library may grow as new users share their implementations with the community. In order to facilitate the use of ~~the code, the~~SuperflexPy, its code is accessible on GitHub with license LGPL-3.0 and distributed using the Python package installer PyPI (~~refer to~~see the code availability section at the end of ~~the~~this paper~~ on code availability).~~). The online documentation provides a guide ~~on how~~ for colleagues interested ~~colleagues can contribute~~in contributing to the framework~~.~~ (section 2.1 of the supplementary material).

~~5 Conclusions~~

6 Summary and conclusions

SuperflexPy is a new Python flexible modelling framework for building conceptual catchment-scale hydrological models ranging from lumped to distributed configurations. ~~The framework~~SuperflexPy 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~~defines its *components* (building blocks) at four hierarchical levels, namely *element*, *unit*, *node*, and *network*~~, which correspond to~~. These *components* support conceptual model setups of increasing levels of complexity, ~~namely~~including but not limited to: a single element model (e.g. a reservoir), a typical lumped model (e.g. a collection of interconnected reservoirs), a ~~collections of lumped models~~semi-distributed model designed to provide prediction at a single outlet, and a semi-distributed model designed to provide predictions at internal sub-catchments. ~~As the~~The construction of a model ~~that requires~~ from components up to a ~~certain~~given hierarchical level does not require specifying ~~the~~components at higher levels~~ above it,~~, which makes the model configuration effort ~~is~~ proportional to the complexity of the application~~.~~ and reduces configuration/computational overheads. The framework supports multiple states and fluxes in each *component*, ~~and foresees an~~which facilitates future extension to ~~water quality~~ applications where such functionality is needed.

~~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 (Fenicia et al., 2011)~~. In order to facilitate usability and further developments, we focused~~ that builds on the collective experience of the authors and their colleagues in hydrological model design and application. The paper discusses the key design choices made in SuperflexPy, with emphasis on ~~several aspects of the model code, including~~the ease of use and interfacing, availability, amenability of extensions, and computational efficiency.

~~We presented two examples illustrating common~~The use ~~cases~~ of the SuperflexPy framework~~, including~~ is illustrated using two examples that represent typical tasks in conceptual hydrological modelling: the implementation of a ~~simple~~ lumped ~~reservoir~~ model to simulate an entire catchment, and the implementation of a distributed model to ~~represent~~simulate a system of multiple sub-catchments ~~and HRUs. It is hoped that~~with spatially varying landscape characteristics. We hope 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~~source code of SuperflexPy~~ as a software project~~, together with documentation and examples, is ~~shown~~hosted in ~~. The~~the public GitHub repository ~~contains all the source code of the framework, as well as examples and documentation.~~https://github.com/dalmo1991/superflexPy. Github is used for issue-tracking. 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 ~~(for the~~ through Zenodo. The release associated with this paper~~) through Zenodo. Documentation (~~ represents version 1.2.1 and has DOI https://doi.org/10.5281/zenodo.4698469. Detailed documentation is available through Read the Docs at https://superflexpy.readthedocs.io~~) and examples are updated periodically, and made available through Read the Docs and Binder respectively~~. The supplementary material to this paper represents a snapshot of the documentation at the time reported on the front page.

SuperflexPy is implemented using Python 3.7 and depends on ~~NumPy~~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.

980 **Competing interests**

The authors declare that they have no conflict of interest.

Acknowledgements

We thank Associate Editor Andrew Wickert, Philip Kraft, Riccardo Rigon, and two anonymous reviewers for their thoughtful and constructive feedback on our manuscript. We are grateful to James Craig, Martyn
985 Futter, David Kneis, Wouter Knoben, and Philip Kraft for providing fast and informative responses that helped us construct Tables 1 and 2.

Financial support

This research has been supported by the Schweizerischer Nationalfonds zur Förderung der Wissenschaftlichen Forschung (grant no. 200021_169003).

990 **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
995 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.

Arnold, J. G., Moriasi, D. N., Gassman, P. W., Abbaspour, K. C., White, M. J., Srinivasan, R., Santhi, C., Harmel, R. D., van Griensven, A., Van Liew, M. W., Kannan, N., and Jha, M. K.: SWAT: Model Use, Calibration, and Validation, Transactions of the ASABE, 55, 1491-1508, https://doi.org/10.13031/2013.42256, 2012.

1000 Bancheri, M., Serafin, F., and Rigon, R.: The Representation of Hydrological Dynamical Systems Using Extended Petri Nets (EPN), Water Resources Research, 55, 8895-8921, https://doi.org/10.1029/2019WR025099, 2019.

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,
1005 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,
1010 10.5194/hess-4-203-2000, 2000.

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

Boyle, D. P., Gupta, H. V., Sorooshian, S., Koren, V., Zhang, Z., and Smith, M.: Toward improved streamflow forecasts: value of semidistributed modeling, Water Resources Research, 37, 2749-2759, 10.1029/2000wr000207, 2001.

Butcher, J. C., and Goodwin, N.: Numerical methods for ordinary differential equations, Wiley Online Library, 2008.

1015 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., and Kavetski, D.: Ancient numerical daemons of conceptual hydrological modeling: 1. Fidelity and efficiency of 1020 time stepping schemes, Water Resources Research, 46, https://doi.org/10.1029/2009WR008894, 2010.

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 1025 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.

1030 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., 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, 1035 10.5194/hess-24-1319-2020, 2020.

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.

Dowell, M., and Jarratt, P.: The “Pegasus” method for computing the root of an equation, BIT Numerical Mathematics, 12, 1040 503-508, 10.1007/BF01932959, 1972.

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., Savenije, H. H. G., Matgen, P., and Pfister, L.: Is the groundwater reservoir linear? Learning from data in hydrological modelling, Hydrol. Earth Syst. Sci., 10, 139-150, 10.5194/hess-10-139-2006, 2006.

1045 Fenicia, F., Savenije, H. H. G., Matgen, P., and Pfister, L.: Understanding catchment behavior through stepwise model concept improvement, Water Resources Research, 44, https://doi.org/10.1029/2006WR005563, 2008.

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.

1050 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, 1055 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 1060 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-
1065 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.

1070 Henn, B., Clark, M. P., Kavetski, D., Newman, A. J., Hughes, M., McGurk, B., and Lundquist, J. D.: Spatiotemporal patterns of precipitation inferred from streamflow observations across the Sierra Nevada mountain range, Journal of Hydrology, 556, 993-1012, https://doi.org/10.1016/j.jhydrol.2016.08.009, 2018.

Houska, T., Kraft, P., Chamorro-Chavez, A., and Breuer, L.: SPOTting Model Parameters Using a Ready-Made Python Package, PLoS One, 10, e0145180-e0145180, 10.1371/journal.pone.0145180, 2015.

1075 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.

1080 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. 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?, Water Resources Research, n/a, e2020WR029143, https://doi.org/10.1029/2020WR029143, 2021.

1085 Kavetski, D., and Clark, M. P.: Ancient numerical daemons of conceptual hydrological modeling: 2. Impact of time stepping schemes on model analysis and prediction, Water Resources Research, 46, 10.1029/2009wr008896, 2010.

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.

1090 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.

1095 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
1100 Catchment Models, Environ Modell Softw, 26, 828-830, https://doi.org/10.1016/j.envsoft.2010.12.009, 2011.

Kuczera, G., Kavetski, D., Franks, S., and Thyer, M.: Towards a Bayesian total error analysis of conceptual rainfall-runoff models: Characterising model error using storm-dependent parameters, Journal of Hydrology, 331, 161-177, https://doi.org/10.1016/j.jhydrol.2006.05.010, 2006.

Lam, S. K., Pitrou, A., and Seibert, S.: Numba: a LLVM-based Python JIT compiler, Proceedings of the Second Workshop on
1105 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
1110 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-
1115 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.

1120 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,
1125 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.

1130 Nash, J.: The form of the instantaneous unit hydrograph, International Association of Scientific Hydrology, Publ, 3, 114-121, 1957.

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.

1135 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.

1140 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
1145 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.

Renard, B., Kavetski, D., Leblois, E., Thyer, M., Kuczera, G., and Franks, S. W.: Toward a reliable decomposition of predictive uncertainty in hydrological modeling: Characterizing rainfall errors using conditional simulation, Water Resources Research,
1150 47, https://doi.org/10.1029/2011WR010643, 2011.

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.

1155 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
1160 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
1165 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.

1170 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.

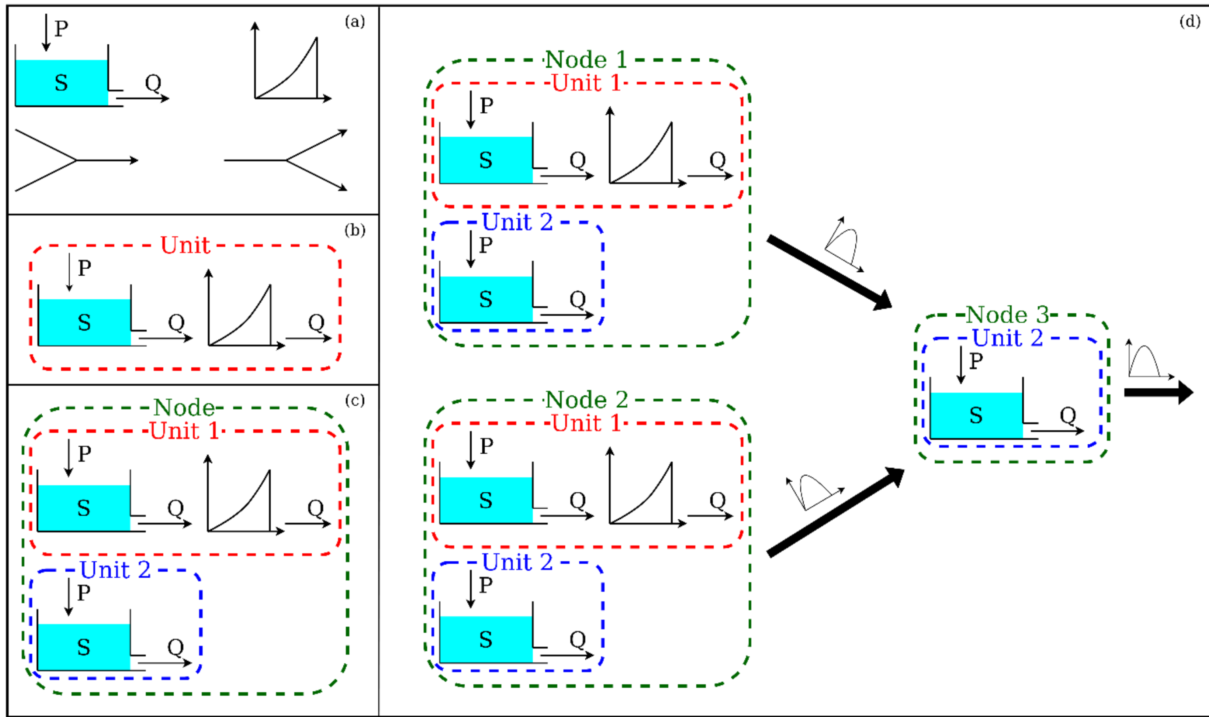
Wrede, S., Fenicia, F., Martínez-Carreras, N., Juilleret, J., Hissler, C., Krein, A., Savenije, H. H. G., Uhlenbrook, S., Kavetski, D., and Pfister, L.: Towards more systematic perceptual model development: a case study using 3 Luxembourgish catchments, Hydrological Processes, 29, 2731-2750, 10.1002/hyp.10393,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

1175 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.: Stochastic, dynamic modelling and signal processing: time variable and state dependent parameter estimation, Nonlinear and nonstationary signal processing, 74-114, 2000.

Young, P. C., Tych, W., and Taylor, C. J.: The Captain Toolbox for Matlab, IFAC Proceedings Volumes, 42, 758-763,
1180 https://doi.org/10.3182/20090706-3-FR-2004.00126, 2009.

Figures



1185

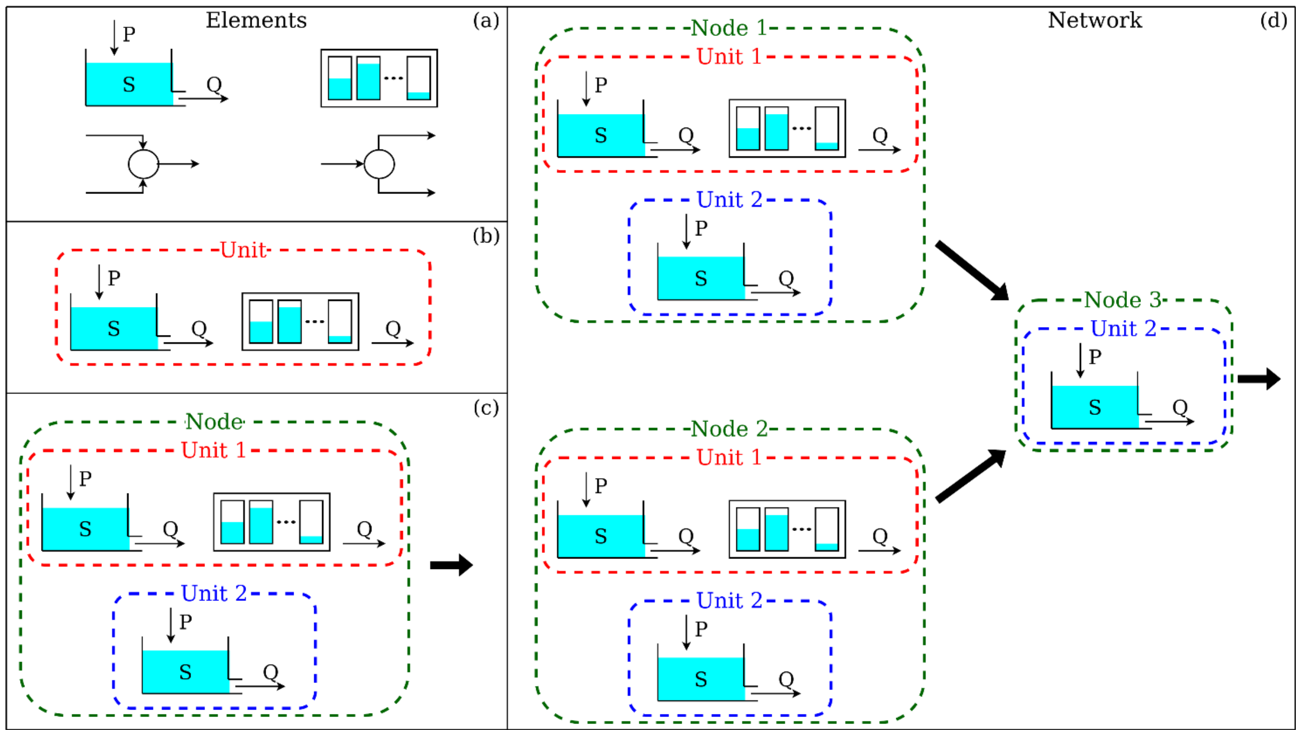


Figure 1. ~~Four basic *components*~~The four hierarchical levels of SuperflexPy~~.~~ and their respective *components*. (a) *Elements* (e.g. reservoirs, lags, connections) are used to represent ~~specific~~individual hydrological processes / catchment response mechanisms; (b) *Units* connect multiple 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 distributed setups.

```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()
```

```
 1 from superflexpy.implementation.elements.hymod import LinearReservoir
 2 from superflexpy.implementation.elements.thur_model_hess import HalfTriangularLag
 3 from superflexpy.framework.unit import Unit
 4 from superflexpy.framework.node import Node
 5 from superflexpy.framework.network import Network
 6 from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
 7 from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
 8
 9 # Initialize computational tools
10 root_finder = PegasusPython()
11 numerical_approximator = ImplicitEulerPython(root_finder=root_finder)
12
13 # Initialize the elements
14 linear_reservoir = LinearReservoir(parameters={'k': 0.1}, states={'S0': 10.0},
15                                    approximation=numerical_approximator, id='LR')
16 lag = HalfTriangularLag(parameters={'lag-time': 3.5}, states={'lag': None}, id='LAG')
17
18 # Initialize the units
19 unit1 = Unit(layers=[[linear_reservoir], [lag]], id='U1')
20 unit2 = Unit(layers=[[linear_reservoir]], id='U2')
21
22 # Change parameters
23 unit2.set_parameters({'U2_LR_k': 0.2})
24
25 # Initialize the nodes
26 node1 = Node(units=[unit1, unit2], weights=[0.7, 0.3], area=5.0, id='N1')
27 node2 = Node(units=[unit1, unit2], weights=[0.9, 0.1], area=2.0, id='N2')
28 node3 = Node(units=[unit2], weights=[1.0], area=1.0, id='N3')
29
30 # Initialize the network
31 net = Network(nodes=[node1, node2, node3], topology={'N1': 'N3', 'N2': 'N3', 'N3': None})
32
33 # Assign the inputs to the nodes (assume P1, P2, P3 have been read)
34 node1.set_input([P1])
35 node2.set_input([P2])
36 node3.set_input([P3])
37
38 # Set the timestep
39 net.set_timestep(1.0)
40
41 # Run the model
42 net.get_output()
```

Figure 2. ~~Code~~SuperflexPy code implementing the simple illustrative model in Figure 1d.

1200

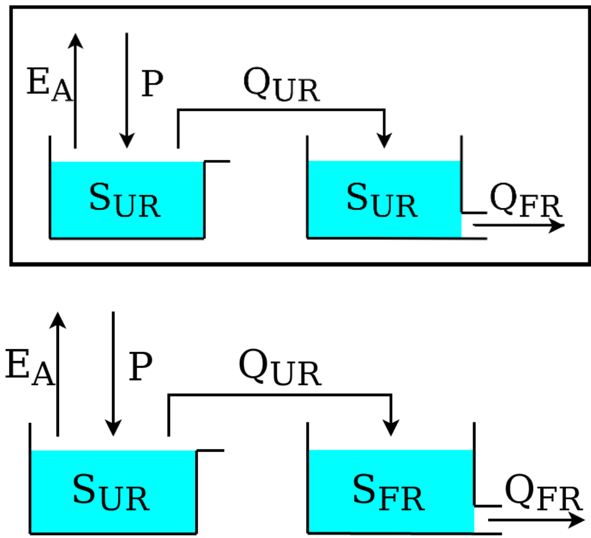


Figure 3. Schematic of model M4 used in the original SUPERFLEX case studies of Kavetski and Fenicia (2011).

1205

```
 1 from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
 2 from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
 3 from superflexpy.implementation.elements.hbv import UnsaturatedReservoir, PowerReservoir
 4 from superflexpy.framework.unit import Unit
 5 import numpy as np
 6
 7 root_finder = PegasusPython()
 8 numeric_approximator = ImplicitEulerPython(root_finder=root_finder)
 9
10 ur = UnsaturatedReservoir(parameters={'Smax': 50.0, 'Ce': 1.0, 'm': 0.01, 'beta': 2.0},
11                           states={'S0': 25.0}, approximation=numeric_approximator, id='UR')
12 fr = PowerReservoir(parameters={'k': 0.1, 'alpha': 1.0}, states={'S0': 10.0},
13                     approximation=numeric_approximator, id='FR')
14
15 model = Unit(layers=[[ur], [fr]], id='M4')
16
17 P = np.loadtxt('precipitation.txt')
18 EP = np.loadtxt('evap_pot.txt')
19
20 model.set_input([P, EP])
21 model.set_timestep(1.0)
22
23 output = model.get_output()
```


```
 1 from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
 2 from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
 3 from superflexpy.implementation.elements.hbv import UnsaturatedReservoir, PowerReservoir
 4 from superflexpy.framework.unit import Unit
 5 import numpy as np
 6
 7 root_finder = PegasusPython()
 8 numeric_approximator = ImplicitEulerPython(root_finder=root_finder)
 9
10 ur = UnsaturatedReservoir(parameters={'Smax': 50.0, 'Ce': 1.0, 'm': 0.01, 'beta': 2.0},
11                           states={'S0': 25.0}, approximation=numeric_approximator, id='UR')
12 fr = PowerReservoir(parameters={'k': 0.1, 'alpha': 1.0}, states={'S0': 10.0},
13                     approximation=numeric_approximator, id='FR')
14
15 model = Unit(layers=[[ur], [fr]], id='M4')
16
17 P = np.loadtxt('precipitation.txt')
18 EP = np.loadtxt('evap_pot.txt')
19
20 model.set_input([P, EP])
21 model.set_timestep(1.0)
22
23 output = model.get_output()
```


Figure 4. ~~Code~~SuperflexPy code implementing model M4 in Figure 3.

1210

```python
1 class NewFastReservoir(ODEsElement):
2 
3     def __init__(self, parameters, states, approximation, id):
4 
5         ODEsElement.__init__(self, parameters=parameters, states=states,
6                              approximation=approximation, id=id)
7 
8         self._fluxes_python = [self._fluxes_function_python]
9         self._fluxes = [self._fluxes_function_python]
10 
11     def set_input(self, input):
12 
13         self.input = {'P': input[0]}
14 
15     def get_output(self, solve=True):
16 
17         if solve:
18             self._solver_states = [self._states[self._prefix_states + 'S0']]
19             self._solve_differential_equation()
20             self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})
21 
22         fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
23                                           S=self.state_array,
24                                           S0=self._solver_states,
25                                           **self.input,
26                                           **{k[len(self._prefix_parameters):]: self._parameters[k] for k in
   self._parameters})
27 
28         return [- fluxes[0][1]]
29 
30     @staticmethod
31     def _fluxes_function_python(S, S0, ind, P, k, alpha, b):
32 
33         if ind is None:
34             return ([P, -(k * S**alpha)/(S + b)], 0.0, S0 + P)
35         else:
36             return ([P[ind], -(k[ind] * S**alpha[ind])/(S + b[ind])], 0.0, S0 + P[ind])
```

```
 1 class NewFastReservoir(ODEsElement):
 2 
 3     def __init__(self, parameters, states, approximation, id):
 4 
 5         ODEsElement.__init__(self, parameters=parameters, states=states,
 6                              approximation=approximation, id=id)
 7 
 8         # _fluxes_python is used to calculate the fluxes doing vector operations
 9         self._fluxes_python = [self._fluxes_function_python]
10         # _fluxes is used to solve the ODE and it is specific to the architecture of the numerical_approximator
11         self._fluxes = [self._fluxes_function_python]
12 
13     def set_input(self, input):
14 
15         self.input = {'P': input[0]}
16 
17     def get_output(self, solve=True):
18 
19         if solve:
20             self._solver_states = [self._states[self._prefix_states + 'S0']]
21             self._solve_differential_equation()
22             self.set_states({self._prefix_states + 'S0': self.state_array[-1, 0]})
23 
24         fluxes = self._num_app.get_fluxes(fluxes=self._fluxes_python,
25                                           S=self.state_array,
26                                           S0=self._solver_states,
27                                           **self.input,
28                                           **{k[len(self._prefix_parameters):]: self._parameters[k] for k in self._parameters})
29 
30         return [- fluxes[0][1]]
31 
32     @staticmethod
33     def _fluxes_function_python(S, S0, ind, P, k, alpha, b):
34 
35         if ind is None:
36             return ([P, -(k * S**alpha)/(S + b)], 0.0, S0 + P)
37         else:
38             return ([P[ind], -(k[ind] * S**alpha[ind])/(S + b[ind])], 0.0, S0 + P[ind])
39 
```

Figure 5. ~~Standard method~~General approach for implementing a new reservoir element `NewFastReservoir` by extending the class `ODEsElement` (Section 3.2.1~~.~~).

1215

```
1 class NewFastReservoir(PowerReservoir):
2
3     @staticmethod
4     def _fluxes_function_python(S, S0, ind, P, k, alpha, b):
5
6         if ind is None:
7             return ([P, -(k * S**alpha)/(S + b)], 0.0, S0 + P)
8         else:
9             return ([P[ind], -(k[ind] * S**alpha[ind])/(S + b[ind])], 0.0, S0 + P[ind])
```


```
1 class NewFastReservoir(PowerReservoir):
2
3     @staticmethod
4     def _fluxes_function_python(S, S0, ind, P, k, alpha, b):
5
6         if ind is None:
7             return ([P, -(k * S**alpha)/(S + b)], 0.0, S0 + P)
8         else:
9             return ([P[ind], -(k[ind] * S**alpha[ind])/(S + b[ind])], 0.0, S0 + P[ind])
```


1220 **Figure 6.** Simplified ~~code~~approach for implementing the `NewFastReservoir` by inheriting directly from class `PowerReservoir` (Section 3.2.2~~.~~).

1225

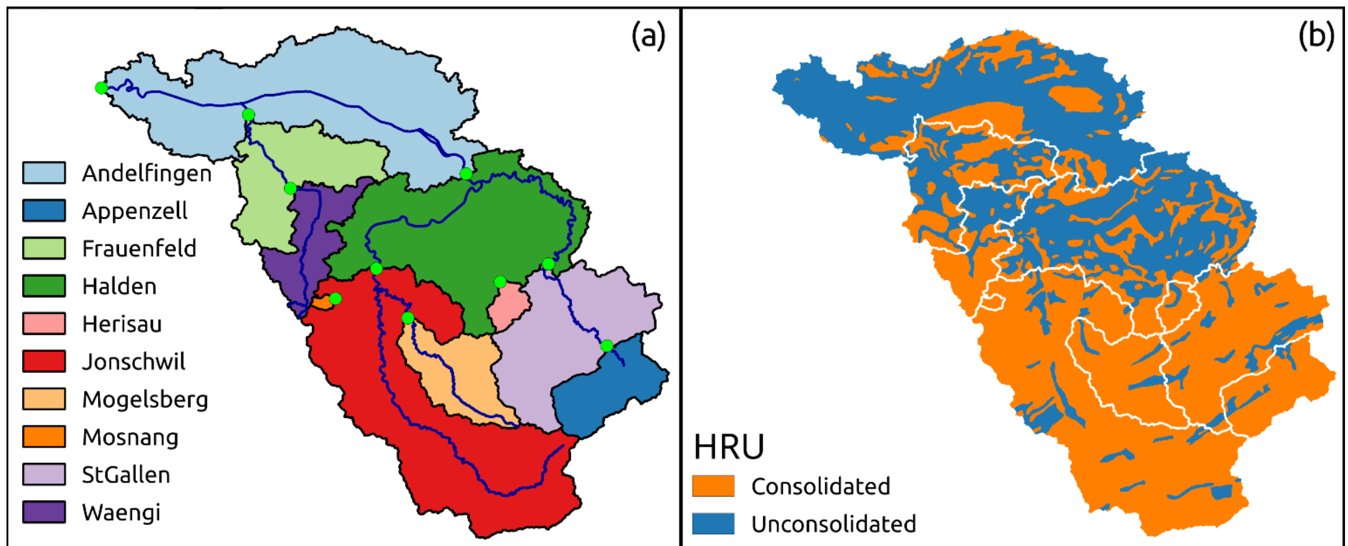
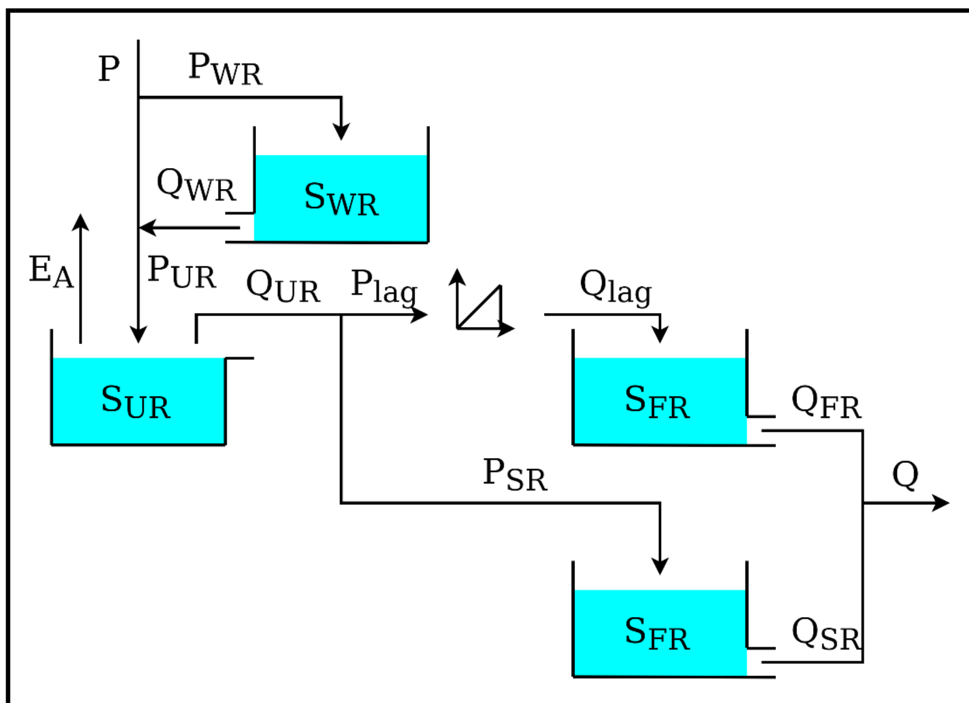


Figure 7. Illustration of catchment discretization used for a distributed application of SuperflexPy~~: (a) subdivision of~~ in the Thur catchment: (a) discretization into sub-catchments and (b) discretization into
1230 hydrological response units (HRUs) as presented in model M02 in Dal Molin et al. (2020). The panels of ~~the~~this figure were originally published in figures 1a and 6 of Dal Molin et al. (2020). The HRU model structure is shown in **Figure 8**~~.~~.



1235

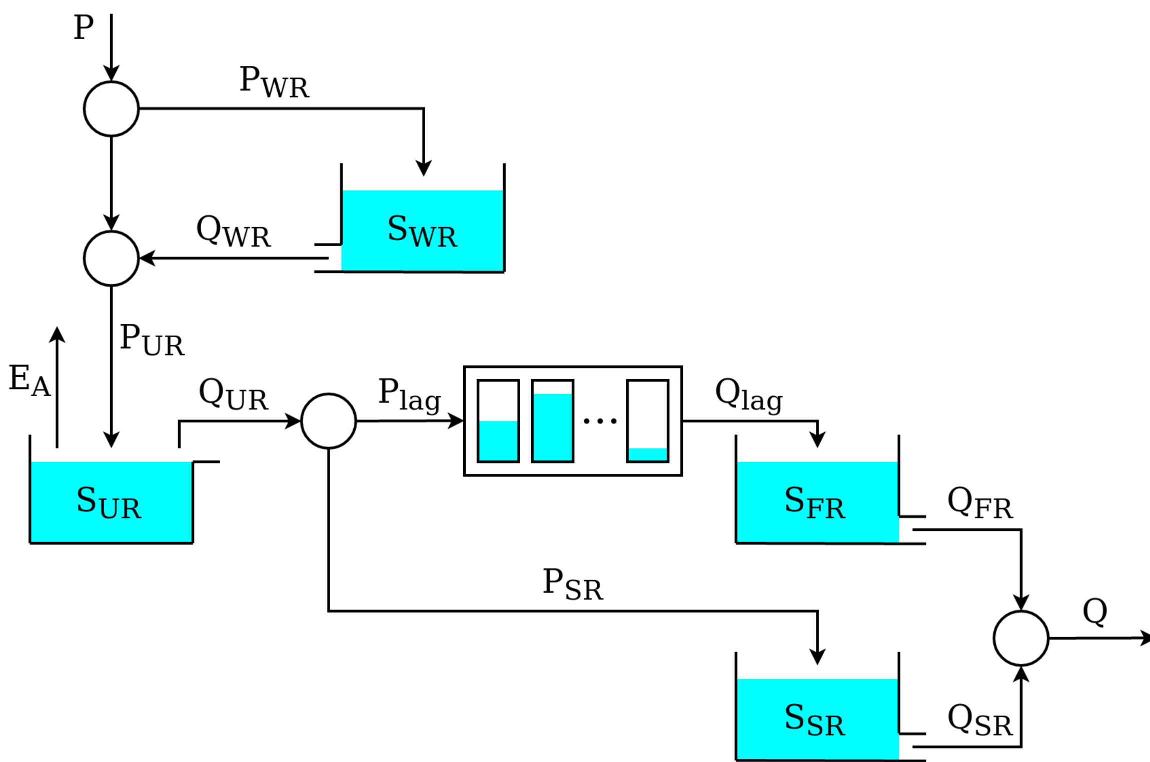
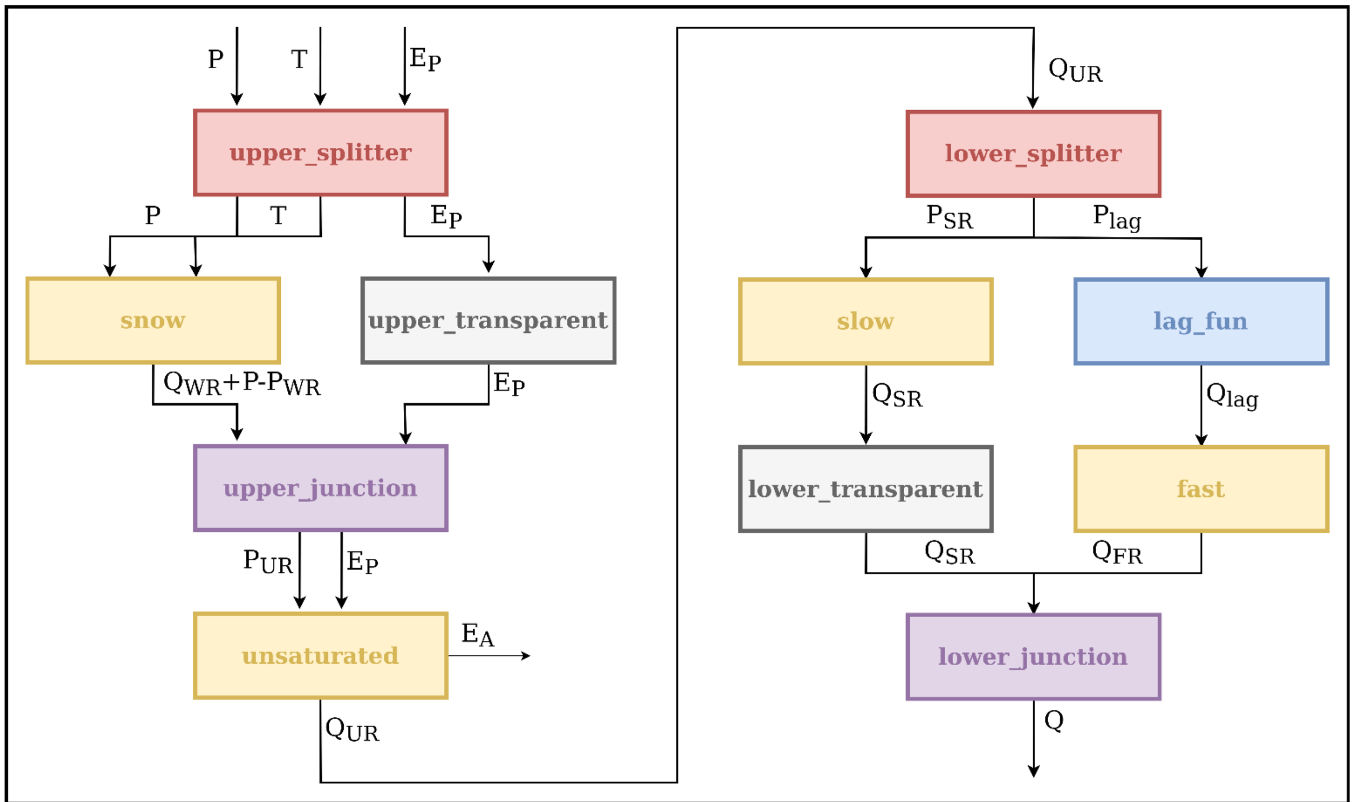


Figure 8. Model structure used to represent the HRUs in model M02 in Dal Molin et al. (2020). Refer to **Figure 7**. for the corresponding HRU discretization of the Thur catchment.

1240



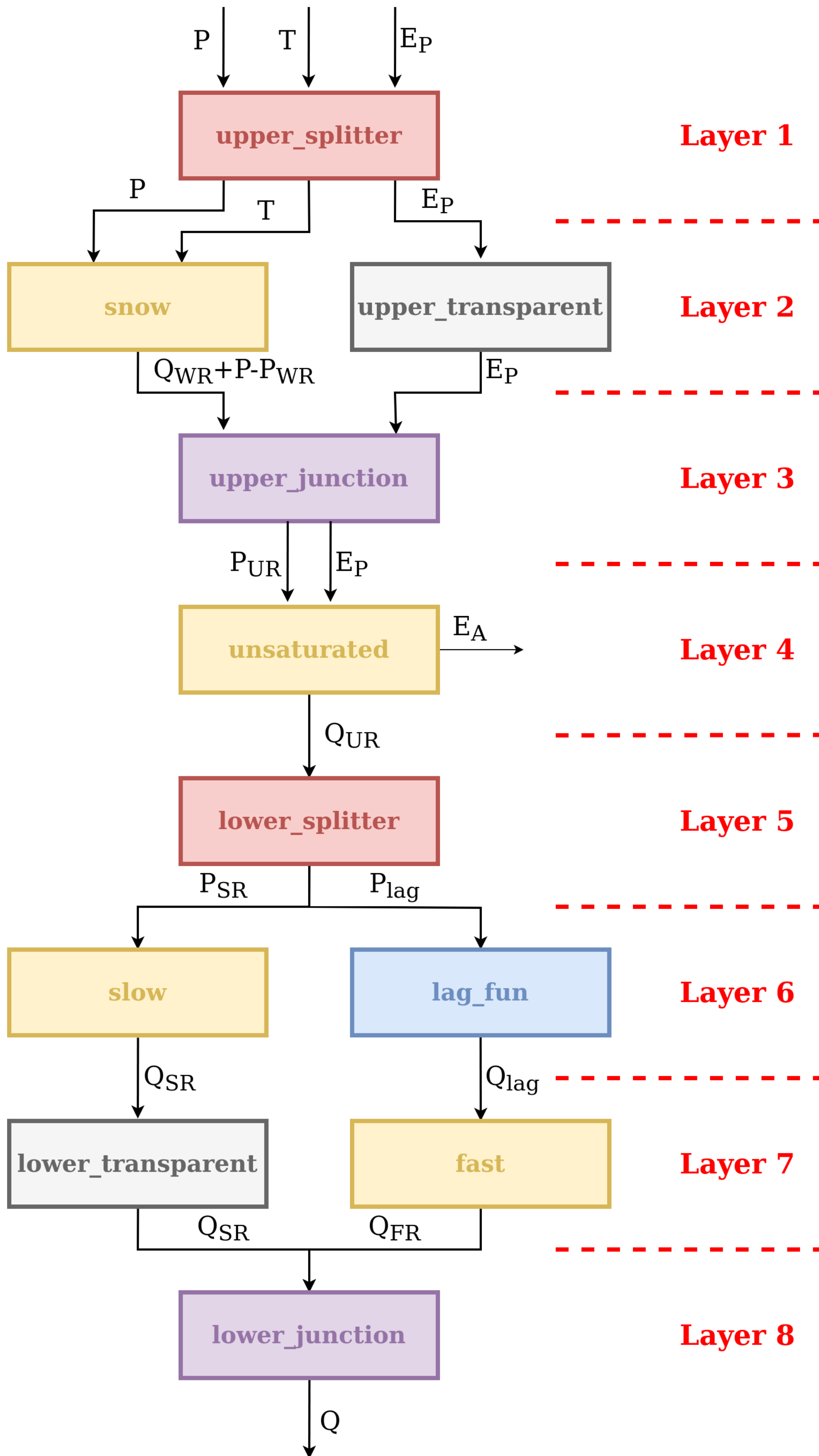
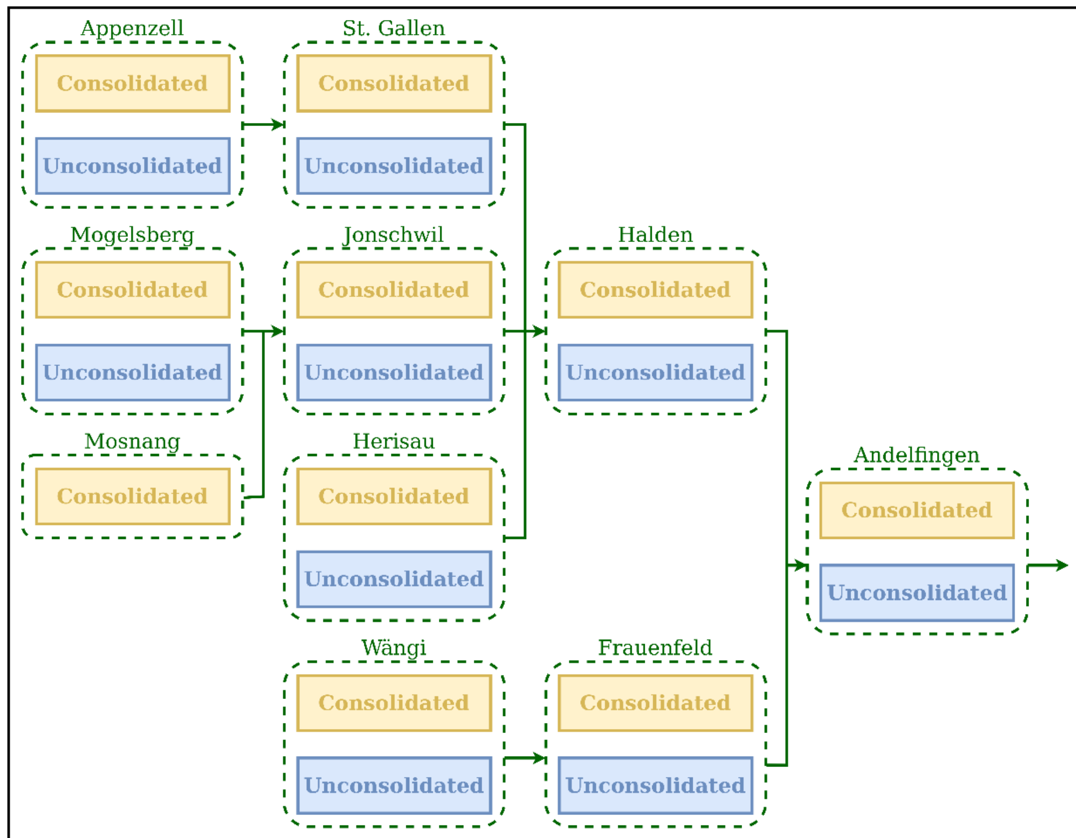


Figure 9. ~~Translation to~~ SuperflexPy representation of the model structure M02 ~~presented~~ in Figure 8.

1245



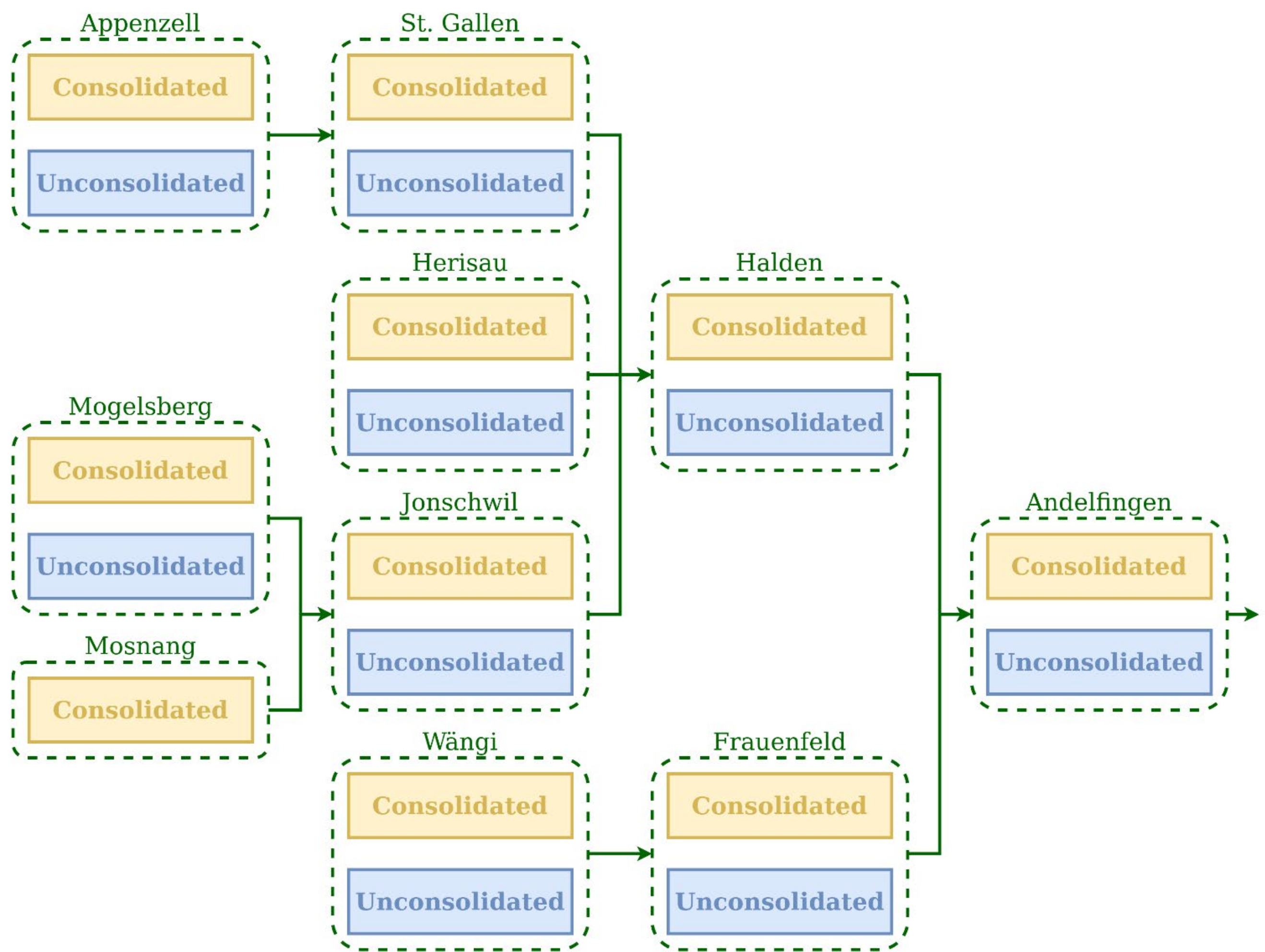


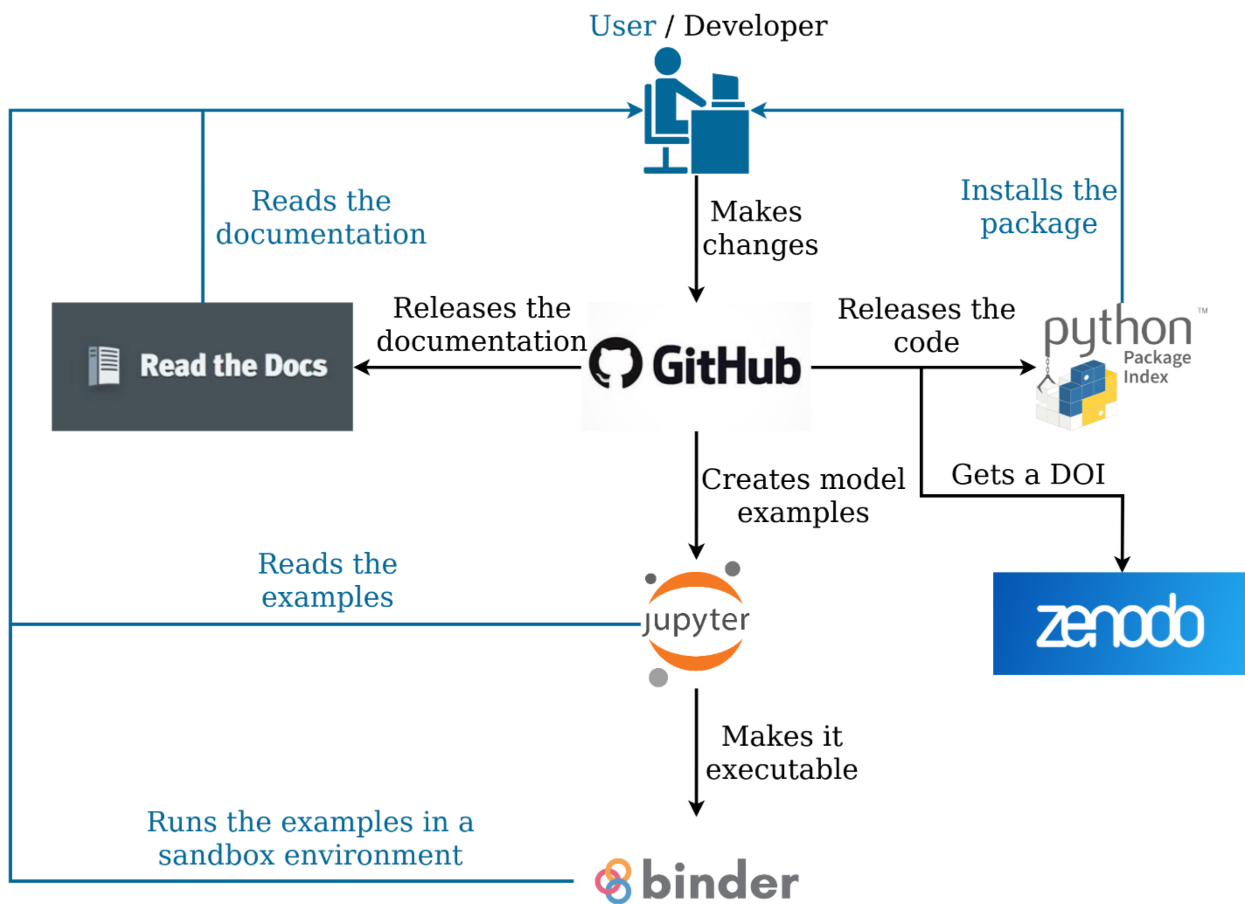
Figure 10. Spatial organization of the SuperflexPy model configuration used to simulate water fluxes in the Thur catchment (M02 inDal Molin et al., 2020). The *units*, used to represent the HRUs, are shown using the blue and yellow boxes. The nodes, used to represent the sub-catchments, are shown using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

1250

```python
1 from superflexpy.implementation.computation.pegasus_root_finding import PegasusPython
2 from superflexpy.implementation.computation.implicit_euler import ImplicitEulerPython
3 from superflexpy.implementation.elements.thur_model_hess import SnowReservoir, UnsaturatedReservoir,
  HalfTriangularLag, PowerReservoir
4 from superflexpy.implementation.elements.structure_elements import Transparent, Junction, Splitter
5 from superflexpy.framework.unit import Unit
6 from superflexpy.framework.node import Node
7 from superflexpy.framework.network import Network
8
9 # Initialize the elements
10 solver = PegasusPython()
11 approximator = ImplicitEulerPython(root_finder=solver)
12
13 upper_splitter = Splitter(direction=[[0, 1, None], [2, None, None]],
14                           weight=[[1.0, 1.0, 0.0], [0.0, 0.0, 1.0]],
15                           id='upper-splitter')
16 snow = SnowReservoir(parameters={'t0': 0.0, 'k': 0.01, 'm': 2.0}, states={'S0': 0.0},
17                      approximation=approximator, id='snow')
18 upper_transparent = Transparent(id='upper-transparent')
19 upper_junction = Junction(direction=[[0, None], [None, 0]], id='upper-junction')
20 unsaturated = UnsaturatedReservoir(parameters={'Smax': 50.0, 'Ce': 1.0, 'm': 0.01, 'beta': 2.0},
21                                    states={'S0': 10.0}, approximation=approximator, id='unsaturated')
22 lower_splitter = Splitter(direction=[[0], [0]], weight=[[0.3],  [0.7]], id='lower-splitter')
23 lag_fun = HalfTriangularLag(parameters={'lag-time': 2.0}, states={'lag': None}, id='lag-fun')
24 fast = PowerReservoir (parameters={'k': 0.01, 'alpha': 3.0}, states={'S0': 0.0},
25                      approximation=approximator, id='fast')
26 slow = PowerReservoir (parameters={'k': 1e-4, 'alpha': 1.0}, states={'S0': 0.0},
27                      approximation=approximator, id='slow')
28 lower_transparent = Transparent(id='lower-transparent')
29 lower_junction = Junction(direction=[[0, 0]], id='lower-junction')
30
31 # Initialize the HRUs
32 consolidated = Unit(layers=[[upper_splitter], [snow, upper_transparent], [upper_junction],
33                             [unsaturated],  [lower_splitter],  [slow, lag_fun],
34                             [lower_transparent, fast], [lower_junction]],
35                     id='consolidated')
36 unconsolidated = Unit(layers=[[upper_splitter], [snow, upper_transparent], [upper_junction],
37                               [unsaturated],  [lower_splitter],  [slow, lag_fun],
38                               [lower_transparent, fast], [lower_junction]],
39                       id='unconsolidated')
40
41 # Create the catchments
42 andelfingen = Node(units=[consolidated, unsaturated], weights=[0.24, 0.76], area=403.3, id='andelfingen')
43 appenzell = Node(units=[consolidated, unsaturated], weights=[0.92, 0.08], area=74.4, id='appenzell')
44 frauenfeld = Node(units=[consolidated, unsaturated], weights=[0.49, 0.51], area=134.4, id='frauenfeld')
45 halden = Node(units=[consolidated, unsaturated], weights=[0.34, 0.66], area=314.3, id='halden')
46 herisau = Node(units=[consolidated, unsaturated], weights=[0.88, 0.12], area=16.7, id='herisau')
47 jonschwil = Node(units=[consolidated, unsaturated], weights=[0.9, 0.1], area=401.6, id='jonschwil')
48 mogelsberg = Node(units=[consolidated, unsaturated], weights=[0.92, 0.08], area=88.1, id='mogelsberg')
49 mosnang = Node(units=[consolidated], weights=[1.0], area=3.1, id='mosnang')
50 stgallen = Node(units=[consolidated, unsaturated], weights=[0.87, 0.13], area=186.6, id='stgallen')
51 waengi = Node(units=[consolidated, unsaturated], weights=[0.63, 0.37], area=78.9, id='waengi')
52
53 # Create the network
54 thur_catchment = Network(nodes=[andelfingen, appenzell, frauenfeld, halden, herisau,
55                                 jonschwil, mogelsberg, mosnang, stgallen, waengi],
56                          topography={'andelfingen': None, 'appenzell': 'stgallen',
57                                      'frauenfeld': 'andelfingen', 'halden': 'andelfingen',
58                                      'herisau': 'halden', 'jonschwil': 'halden',
59                                      'mogelsberg': 'jonschwil', 'mosnang': 'jonschwil',
60                                      'stgallen': 'halden',  'waengi': 'frauenfeld'})
```

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

1255 **Figure 11.** ~~Code~~SuperflexPy code implementing the distributed model in Figure 9 and Figure 10.



1260 **Figure 12.** Organization of the SuperflexPy project, indicating the online software management tools used to develop the source code and documentation, release product versions with associated DOIs, and provide general open access to all project components. Typical workflow paths for users and developers are shown, respectively, in the blue and black lines and font.

Table 1. Summary of usability characteristics of SuperflexPy in the context of selected flexible frameworks for conceptual hydrological modeling.*

	Availability	**Distribution and installation**	**Documentation**	**Interface and setup**	**I/O format for settings and data**	**Possibility of customization**	**Built-in calibration and uncertainty analysis**
SuperflexPy	Open source	Python package	Available	Python package. Python script to setup	Direct I/O with Python. No binding to particular formats	Possible with moderate programming expertise	Not present
FUSE (Fortran) (2008)	Exe or code, by request from authors	Standalone exe/code	Comments in code (limited)	Executable with/without GUI, or Fortran subs. Setup files	Structured text files	Possible but not supported systematically	Some versions are coupled with optimization and MCMC sampling tools
SUPERFLEX-F90 (2011)	Exe or code, by request from authors	Standalone exe	Comments in code (limited)	CLI or DLL or Fortran subs. Setup files	Structured text files	Possible but not supported systematically	Not present
CMF (2011)	Open source	Python package. Code compilation for enhancements	Available	Python package. Python script to setup. GUI only for lumped models	Direct I/O with Python. No binding to particular formats	Customization using C++. Possibility with Python under development	No. Developers recommend to use the SPOTpy package from the same group
PERSiST (2014)	Exe/webapp after registration	Standalone executable or webapp	Exists. Not public at the moment	Desktop app or webapp. Setup files or GUI	Structured text files and XMLs	Possible but not supported systematically	Incorporates MCMC toolkit
ECHSE (2015)	Open source	R package to generate C code that has to be compiled	Available	CLI. Setup through text file or CLI	Delimited text files	Possible with moderate programming expertise	Not present
MARRMoT (2019)	Open source	Matlab/Octave package	Available	Collection of scripts and functions. Setup with script.	Direct I/O with Matlab/Octave. No binding to particular formats	Possible with moderate programming expertise	Not present

RAVEN (2020)	Open source	Standalone executable. May require NetCDF	Available	Executable without GUI. Setup files	Structured text files	Possible but requires developer-level expertise. Instructions in the documentation	DDS optimization. Reports model performance metrics usable by external software

*This information was collated based on published information. A brief informal review was provided by the framework authors.

Abbreviations: exe = binary executable, subs = subroutines, GUI = graphical user interface, CLI = command line interface, DLL = dynamic link library, MCMC = Monte Carlo Markov Chain, DDS = Dynamic Dimensioned Search

Table 2

1270 ~~Figure~~. Summary of simulation capabilities of SuperflexPy in the context of selected flexible frameworks for conceptual hydrological modeling.*

	Structural flexibility	Spatial flexibility	Hydrological processes	Numerical solution options	Pre and post processing	Programming language
SuperflexPy	Components can be connected freely	Lumped; semi-distributed	Water fluxes; Designed to handle multiple fluxes	Fixed step implicit and explicit Euler. Possibility to use custom solvers	Not available	Python
FUSE (Fortran) (2008)	Master structure; components selected for each model decision	Lumped	Water fluxes	Implicit, semi-implicit, explicit schemes; fixed and adaptive step solvers	Not available	Fortran
SUPERFLEX-F90 (2011)	Master structure; components can be turned on/off	Lumped; semi-distributed	Water fluxes; transport processes	Fixed step implicit and explicit Euler	Not available	Fortran
CMF (2011)	Components can be connected freely	Lumped; semi-distributed; fully-distributed	Water fluxes; transport processes	Implicit and explicit schemes; single or multistep solvers	Calculation methods for PET	Python wrapping of C++ code
PERSiST (2014)	Components can be connected freely	Semi-distributed	Water fluxes; designed to be coupled with transport models (INCA)	Implemented as a series of first order difference equations	PET calculated internally	C++
ECHSE (2015)	Components can be connected freely	Lumped; semi-distributed; grids	Water fluxes; transport processes	To be implemented by user when defining the components	Not available	C++; R package to generate C++ code
MARRMoT (2019)	Library of model structures. Possibility to combine different components	Lumped	Water fluxes	Fixed step implicit and explicit Euler. Possibility to use custom solvers	Not available	Matlab/Octave
RAVEN (2020)	Components can be connected freely	Grids; subbasin/HRUs; triangulated irregular network	Water fluxes; transport processes	Ordered series, Euler, and predictor/corrector global methods; local methods at process level	Calculation and interpolation (spatial and temporal) of derived fluxes and other variables	C++

~~: Organization of the SuperflexPy project.~~
*This information was collated based on published information. A brief informal review was provided by the framework authors.