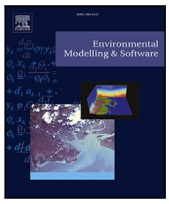
Environmental Modelling and Software 186 (2025) 106362

Contents lists available at ScienceDirect

Environmental Modelling and Software

journal homepage: www.elsevier.com/locate/envsoft



pyKasso: An open-source three-dimensional discrete karst network generator

François Miville [a], Philippe Renard [a,*], Chloé Fandel [b], Marco Filipponi [c]

[a] Centre for Hydrogeology and Geothermics, University of Neuchâtel, 11 Rue Emile Argand, Neuchatel, 2000, Switzerland
[b] Department of Geology, Carleton College, Anderson Hall 338, Northfield, MN 55057, USA
[c] National Cooperative for the Disposal of Radioactive Waste (NAGRA), Hardstrasse 73, Wettingen, 5430, Switzerland

ARTICLE INFO

Keywords:
Karst
Network
Stochastic
Geometry
Groundwater

ABSTRACT

Modeling groundwater flow using physically based models requires knowing the geometry of the karst conduit network. Often, this geometry is not accessible and unknown. It is therefore crucial to be able to model it. This paper presents pyKasso, an open-source Python package that generates those geometry based on a pseudo-genetic approach. The model accounts for multiple data sources: a 3D geologic model, the position of known inlets and outlets, the statistical distribution of fractures or inception features, and known base levels. This approach simplifies previously published work by considering a 3D anisotropic fast marching algorithm. The paper presents the structure of the code and explains in detail how it can be used from the most simple 2D situation to a complex 3D case.

1. Introduction

One of the most important features when modeling groundwater flow and solute transport in karst aquifers is to account for the presence of a network of karst conduits that are often well-connected over long distances and constitute fast pathways (Ford and Williams, 2007). Among the various existing approaches to simulate flow and solute transport in karst aquifers, this paper considers the Discrete Conduit Network (DCN) and Combined Discrete-Continuum (CDC) models (Kovács and Sauter, 2007; Hartmann et al., 2014). In these approaches, the flow equations are solved in karst conduits explicitly represented as a network of one-dimensional numerical elements. In the DCN approach, the matrix is not considered while in the CDC approach, the conduits are embedded in a (possibly fractured) matrix that is itself discretized. The matrix can be heterogeneous and the network of conduits can have properties varying in space. The DCN or CDC approaches are implemented in several numerical codes such as Modflow-CFP (Shoemaker et al., 2008; Reimann and Hill, 2009; Reimann et al., 2023), Modflow-USG and CLN (Kresic and Panday, 2018; Duran and Gill, 2021), Feflow (Diersch, 2013; Berglund et al., 2020), Groundwater (Cornaton, 2007; Vuilleumier et al., 2013), DisCo (de Rooij et al., 2013; Dall'Alba et al., 2023) or openKARST (Kordillaa et al., 2025). To be applicable, all the DCN and CDC models require a geometrical description of the network of conduits: a Discrete Karst conduit Network or DKN. It is analogous to the concept of Discrete Fracture Network (DFN) frequently used for fractured systems (Cacas et al., 1990; Erhel et al., 2009; Hyman et al., 2015). The DKN terminology is not yet widely adopted by the scientific literature: only the Ph.D. thesis of Giese (2017) and a few papers (Fernandez-Ibanez et al., 2019; Gouy et al., 2022, 2024) employ it. But as reviewed below, many papers have been published in the last 10 years which aim at generating DKNs. It is therefore time, in our opinion, to use the DKN terminology more widely and to distinguish it clearly from the DCN and CDC models. The DKN model refers to the generation and the geometry of the discrete conduits. It is different from the DCN or CDC models, which refer to how the flow equations are solved (Kovács and Sauter, 2007). The two models are linked because the DCN and CDC models require first a DKN model.

The construction of a DKN can be based on the knowledge of the actual geometry of the karstic conduits when this is available (Worthington, 2009; Vuilleumier et al., 2019). However, in many practical cases, the exact geometry of the network of conduits is unknown and therefore, a preliminary step is to build a model to simulate or generate the geometry of the conduits. Different techniques exist. They can be roughly classified into two main groups: process-imitating and structure-imitating models.

The *process-imitating* or *process-based* models represent the physics and chemistry of the formation of karst conduit networks (speleogenesis) using reactive transport models (Dreybrodt et al., 2005; de Rooij and Graham, 2017; Cooper and Covington, 2020). They model the long-term evolution of the karst conduit network and can generate

* Corresponding author.
E-mail addresses: francois.miville@ikmail.com (F. Miville), philippe.renard@unine.ch (P. Renard), cfandel@carleton.edu (C. Fandel), marco.filipponi@nagra.ch (M. Filipponi).

https://doi.org/10.1016/j.envsoft.2025.106362
Received 27 July 2024; Received in revised form 14 January 2025; Accepted 1 February 2025
Available online 12 February 2025

complex patterns. They are used to analyze and understand the coupled processes that occur during the formation of karst conduit networks. Their aim is generally not to produce a DKN for further studies.

The *structure-imitating* models do not solve explicitly those equations and instead, they aim to reproduce the structures of the conduit network using simplifying assumptions. This last group of methods is very diverse and corresponds to what can be considered the core of the DKN approach. It includes purely statistical methods, rule-based, or pseudo-genetic techniques: for example cellular automaton models (Jaquet et al., 2004; Fischer et al., 2016; Escobar et al., 2021), self-avoiding percolation invasion (Ronayne and Gorelick, 2006; Ronayne, 2013), spatial statistics methods (Pardo-Igúzquiza et al., 2012; Erzeybek et al., 2012; Viseur et al., 2015; Le Coz et al., 2017), truncation of Gaussian random fields (Rongier et al., 2014; Schiller and Renard, 2016), conceptual approaches such as KARSYS (Jeannin et al., 2013; Malard et al., 2015), simulation based on graph theory and shortest path (Collon-Drouaillet et al., 2012; Gouy et al., 2024), sub-networks of percolation clusters (Hendrick and Renard, 2016), computer graphics techniques (Paris et al., 2021), or dynamic graph dissolution (Kanfar and Mukerji, 2023). Among the pseudo-genetic methods, the Stochastic Karst conduit Simulation (SKS) method was introduced by Borghi et al. (2012); it is capable of generating an ensemble of DKN models to represent uncertainty. It integrates a 3D geologic model representing the geometry of the aquifer, a 3D model of the main faults, and a stochastic discrete fracture network model. Helped by a hierarchical fast-marching algorithm, it also identifies plausible paths through the fractured carbonate formation between known or simulated inlets and outlets. SKS assumes that the conduits are created by water flowing from inlets to outlets and dissolving the rock matrix to generate the network along the path. SKS is therefore designed to simulate karstic networks corresponding to epigenic speleogenesis (De Waele and Gutiérrez, 2022). The method was not designed for hypogene systems or pre-structuration by ghost rocks. SKS has been used for multiple case studies (Vuilleumier et al., 2013; Sivelle et al., 2020; Fandel et al., 2022; Banusch et al., 2022; Dall'Alba et al., 2023; Fandel et al., 2023). One limitation that was identified by Fandel et al. (2022) is that the simulated DKNs were not influenced by the geometry (slope and orientation) of the rock formations in which they were simulated. The networks were not following the successive syncline and anticline structures that were known to control the network in the study area of the Gottesacker system in Germany (Fandel et al., 2022). To overcome this limitation, the isotropic fast marching that was used in the original SKS algorithm has been replaced by an anisotropic version. This allowed the inclusion of preferential directional information to better orient the network. A similar idea has been proposed by Luo et al. (2021) who also used the anisotropic fast marching to represent the anisotropy of the underlying DFN. Using this approach they were able to generate DKNs influenced by the local orientation of the fracture networks.

This paper introduces a new open-source Python package that implements for the first time in 3D the work initiated by Fandel et al. (2022) in 2D. An important new contribution of the present paper is a new method for the simulation of the network in the vadose zone. In the original SKS algorithm, Borghi et al. (2012) had to split the simulation domain into two parts: the vadose zone and the phreatic zone, to ensure that the simulated conduits were sub-vertical in the vadose zone. The simulation of the conduits was done in two steps. We propose in this paper to use the anisotropic fast marching (Sethian, 2001; Mirebeau and Portegies, 2019) to orient the anisotropy in the vadose and phreatic zones differently. This allows computing the complete path from an inlet on the topographic surface to the outlet through both the vadose and phreatic zones in one single step. As it relies on the general principles of SKS, the code is designed to simulate epigenic karstic networks in carbonate formations. But, the code is pretty flexible since it can handle any 3D geological model and the user can change the parameters (density of fractures, number of inlets or outlets) to create networks displaying different degrees of karstification. Because of its flexible nature, it is also possible to model a single or multiple phases of karstification. Several previously published examples show a variety of situations that can be modeled with this approach. This paper presents this novel and simplified approach and demonstrates its efficiency. Furthermore, the paper provides a detailed description of the structure of the code in Section 3 and shows in Section 4 how it can be used to generate karst conduit network models of increasing complexity.

2. General principle of SKS

The Stochastic Karst conduit Simulation (SKS) approach is designed to integrate different sources of geologic and hydrogeologic data. A detailed description of the concept and algorithm is given in Borghi et al. (2012) for the original algorithm and in Fandel et al. (2022) for its anisotropic extension in 2D. This paper provides therefore only a rapid overview of the method to help the reader understand what are the required input and how the algorithm functions. The main inputs for SKS are:

- the location of the inlets (for example dolines) and outlets (springs) obtained from surface mapping or a model of their statistical distributions;
- a discretized 3D geologic model (voxet) providing a 3D description of the location of the various geologic formations and faults in the study area;
- the statistical input parameters of a discrete fracture network model representing the fractures in the study area or DFN simulations obtained from an external code;
- if available, a surface, or a constant altitude, representing the boundary between the vadose and phreatic zone.

The overall workflow is illustrated in Fig. 1. Based on the input data, SKS generates an initial 3D propagation field containing *cost* values stored in a 3D voxet. The *cost* is inversely related to karstifiability. The propagation field represents therefore the spatial distribution of the susceptibility of karstification in the domain. Different types of inception features can be given explicitly. Some rocks are for example more susceptible to karstification than others. The faults can be more permeable than the geologic layers and therefore more susceptible to karstification. The fractured areas have also more chances to be karstified than the non-fractured areas.

Once the propagation field is defined, it is used as input for the fast marching algorithm (FMA) (Sethian, 1996) that computes the propagation of a front that travels from the output locations through the domain until it covers it entirely. The result is a 3D field of travel times (Fig. 1).

Knowing the positions of some randomly selected inlets and one or more outlets (also possibly randomly selected), the result of the fast marching algorithm is used to find the fastest path (particle tracking, in Fig. 1) between these inlets and outlets by following the highest gradient of time.

Note that the FMA algorithm is applied from the output locations and not from the inlet locations because it is numerically the most efficient way to find the shortest path from all the considered inlets to the outputs in one single step.

A central idea in SKS is that the conduits are not formed simultaneously but iteratively, leading to a hierarchical structure. This is described in detail in Borghi et al. (2012) and has been extended by Fandel et al. (2023). The procedure is as follows. All inlets and outlets are added progressively. Only a part of the inlets and outlets are used at each iteration. The conduit geometries obtained in a given iteration are used to update the propagation field for the next iteration. This allows the creation of hierarchical networks as well as to account for several phases of karstification since at each iteration, the newly generated conduits tend to connect preferentially to already existing conduits.

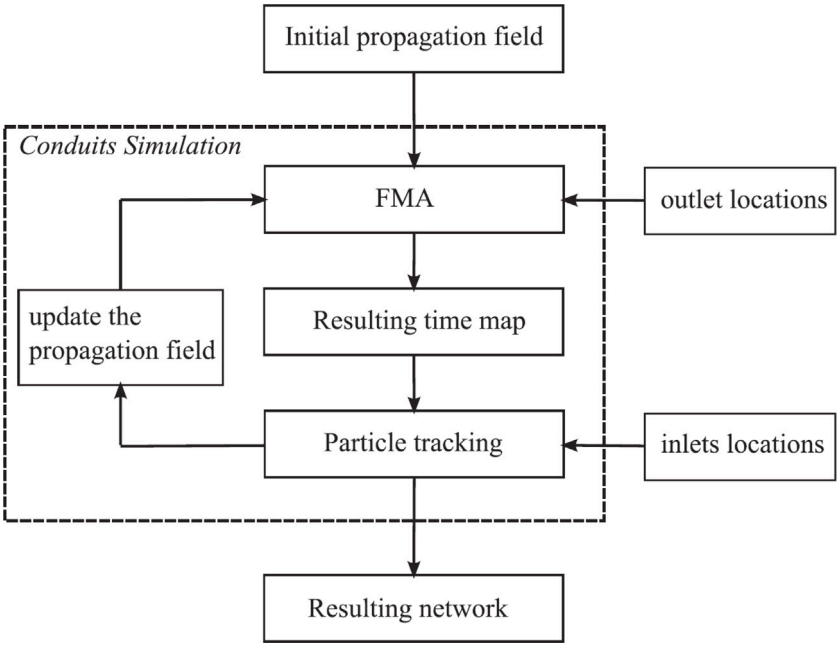


Fig. 1. Schematic representation of the main loop of SKS in which the conduits are generated iteratively (Borghi et al., 2012). FMA stands for Fast Marching Algorithm.

For each model run, SKS can generate different fracture network models, different locations for the unknown inlets or outlets, and different sequences of inlets/outlets during the simulation procedure. In this manner, the algorithm generates an ensemble of stochastic conduit networks.

The following sections of the paper present the implementation of SKS in the pyKasso package and explain how to use the code.

3. The pyKasso package

3.1. Design and structure

PyKasso is an object-oriented Python package. Fig. 2 illustrates the general organization of the package and the relationships between the main classes. PyKasso is composed of three main sub-packages: `model`, `analysis`, and `visualization` embedded in one central sub-package `core`. The **Application** class from the `core` sub-package acts as an interface from which all pyKasso functionalities can be accessed. Following a project-oriented approach, data and simulations are organized in separate folders. The **Project** class, from the `core` sub-package, contains all the relevant information concerning the current project and the corresponding simulations. The **Project** type object is also accessible from the other sub-packages. This design allows access to the sub-packages from one place and facilitates communication between them, the project, and the user. Practically, a project is defined by a project directory location and a grid defining the location, the extent, and the definition of the simulated system. Simulations will be stored in this dedicated folder.

Each sub-package plays a specific role. Their functionalities are accessible from a set of classes. The **SKS** class from the `model` sub-package is responsible for the karst conduit network modeling. It constructs the geologic model, pilots the fast-marching algorithms, and outputs karst conduit networks. It relies on the NumPy (Harris et al., 2020) and pandas (pandas development team, 2020) Python packages. The **Analyzer** class from the optional `analysis` sub-package allows to compute post-processing operations on the resulting karst conduit networks. Using the Karstnet and NetworkX (Hagberg et al., 2008) Python packages, it computes statistical metrics (Collon et al., 2017) from the network geometry to evaluate the plausibility of the results in comparison to real datasets. The **Visualizer** class from the optional `visualization` sub-package allows visualization of the resulting simulated karst conduit networks using visualization libraries such as Matplotlib (Hunter, 2007) and PyVista (Sullivan and Kaszynski, 2019).

These three tasks have been implemented in separate sub-packages to ensure better code maintenance, readability, and efficiency. Moreover, the package design is conceived such that the visualization and post-processing sub-packages are optional. The following section will mainly describe how the `model` sub-package works to model karst conduit networks.

3.2. Workflow

Fig. 3 presents the general workflow of pyKasso. From setting the model parameters to analyzing the results, it has been designed to be easy to use in Python `Notebooks` or scripts and to allow users to have maximum control over the simulation parameters.

After importing the pyKasso package, a new instance of the **Application** class from the `core` sub-package is created by calling the `pykasso` function. It returns an **Application** object that will be used to manage the project and the simulations.

```
# Import the pyKasso package
import pykasso as pk

# Create a pyKasso application
app = pk.pykasso()
```

To use the embedded sub-packages, a **Project** instance from the `core` sub-package must first be created or loaded within the application. The new **Project** is declared with the `new_project` method by providing a project name and a Python dictionary to describe the grid used to represent the studied system as input parameters. The name and the type of the parameters required to construct the grid are described in Table 1. The grid resolution must be fine enough to resolve the most important features controlling the geometry of the network. Still, it must be sufficiently coarse to allow fast computing time, limited memory storage, and simplifying the problem in a meaningful manner.

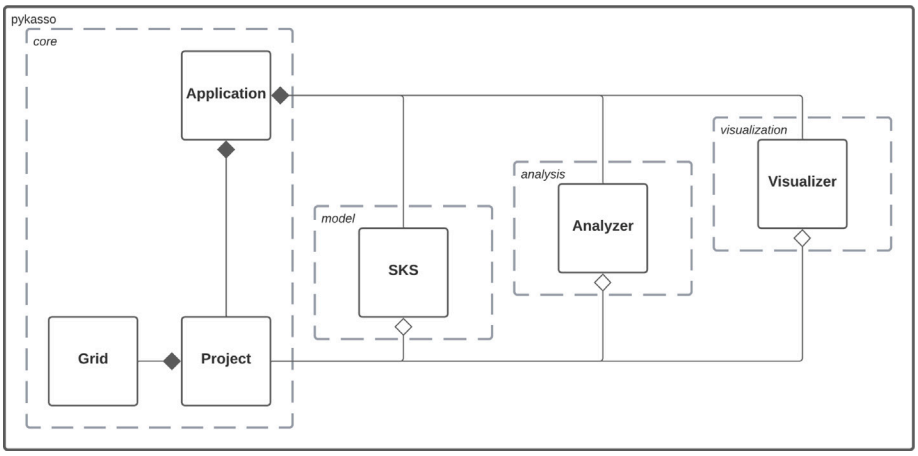


Fig. 2. Overview of the UML class diagram of the pyKasso package. **SKS**, **Analyser** and **Visualizer** are the most important classes. They are accessible via the sub-packages `model`, `analysis`, and `visualization`.

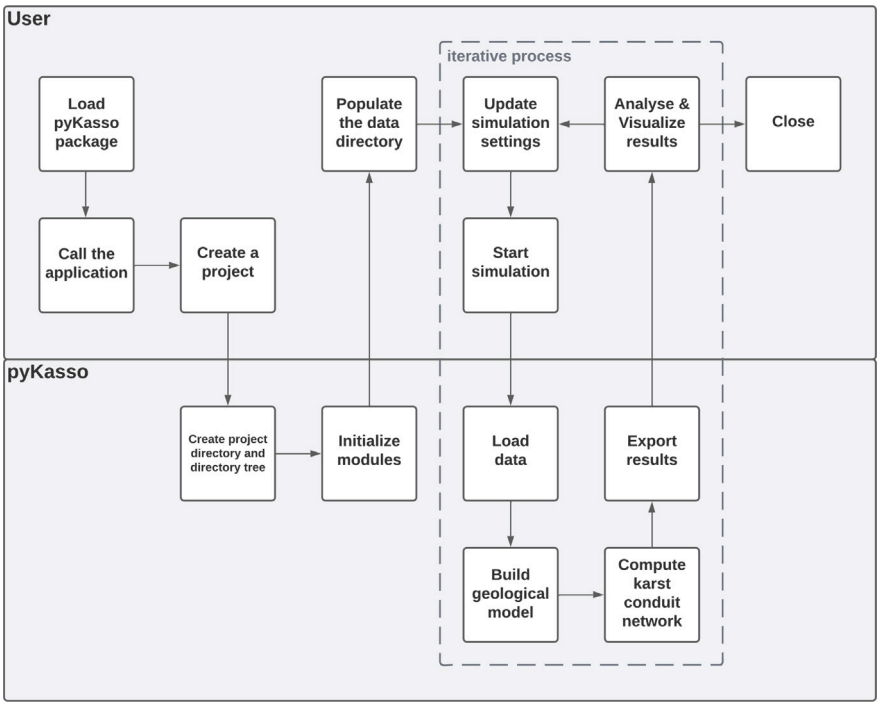


Fig. 3. General workflow for modeling karstic conduit networks.

```python
# Create a dictionnary storing the grid
# parameters for the ongoing project
grid_parameters = {
    'x0': 0, 'y0': 0, 'z0':  0,
    'nx': 50, 'ny': 50, 'nz': 10,
    'dx': 10, 'dy': 10, 'dz': 10}

# Declare the project parameters
app.new_project(name='Tsanfleuron',
                grid_parameters=
    grid_parameters)

# Display the project attributes
app.project
```

If the `project_name` does not point to an already-existing folder, then the `new_project` method will create a new directory located and named according to the provided argument. This operation also defines the `project` attribute in the **Application** instance, storing the project properties within the instance of the **Project** class. Finally, the following directory tree and files will be created within the project folder.

```
project_directory
├── inputs
│   └── parameters.yaml
├── outputs
│   └── simulations
├── project.yaml
└── project.log
```

The `inputs` subfolder should be used to store the input data files for the studied system, but this is not mandatory. The `outputs` subfolder contains all the model outputs. Because pyKasso can simulate karst conduit networks in three dimensions, memory usage can become a limiting factor. Memory consumption depends on the dimensions of the grid and can therefore be limited by the RAM capacity when multiple simulations are generated. To address this issue, the simulation results are stored on the disk in subfolders within the `outputs`

Table 1
Name, type, and description of pyKasso grid parameters.

Parameter	Data type	Description
x0	float	X-coordinate of center point of bottom left cell.
y0	float	Y-coordinate of center point of bottom left cell.
z0	float	Z-coordinate of center point of bottom left cell.
nx	int	Number of cells in the x-axis.
ny	int	Number of cells in the y-axis.
nz	int	Number of cells in the z-axis.
dx	float	Cell length in the x-axis.
dy	float	Cell length in the y-axis.
dz	float	Cell length in the z-axis.

Table 2
Name, type, and description of pyKasso project attributes.

Attribute	Data type	Description
name	str	Name of the project
description	str	Description of the project
creation_date	str	Date of creation of the project
pykasso_version	str	pyKasso version used during the creation of the project
grid	Grid	Grid instance used for the simulations
core	dict	Dictionary storing paths
n_simulations	int	Number of simulations already computed
simulations	list	Locations of the folders of the simulations

directory after computation. It is then possible to split the multiple simulations into several steps to overcome that limitation. The parameters of the **Project** instance, like the grid parameters, the number of simulations, or even the pyKasso package version are stored in a YAML data structure within the `project.yaml` file. Table 2 lists the names, types, and definitions of the attributes stored in the **Project** instance. The `project.log` file stores all the operations achieved by pyKasso once the project is created or loaded. If an issue occurs, pyKasso will keep a trace of it within this log file. The file is updated each time a new simulation is computed.

A previously created project can be reloaded with the `open_project` method.

```
# Load an already created project
# and create an instance of the Project class
app.open_project(name='paper')
```

Within the project directory, provided as an argument, pyKasso will automatically seek for the `project.yaml` file to load and set the project again.

Once the project is created or loaded, the three sub-packages can be used. The `analysis` and `visualization` sub-packages require that some simulations have been previously generated. A new karst conduit network can be simulated with the `generate` method from the **SKS** instance from the `model` sub-package stored in the `model` attribute.

```
# Generate one simulation with default parameters
app.model.generate()

# Retrieve number of simulations
app.project.n_simulation
> 1

# Retrieve localisation of simulations
app.project.simulations
> ['/path/to/simulation']
```

At that stage, the project contains one simulation located in the folder described in the `simulations` attribute.

By default, if the `generate` method is called without any parameters, pyKasso will load the default `parameters.yaml` parameter file located in the `inputs` subfolder. Otherwise, simulation parameters are provided through a custom Python dictionary or by pointing to a custom YAML file. Those parameters are defined in the next section.

Table 3
Name, type and description of `sks` key parameters.

Parameter	Data type	Description
seed	(int)	Seed used to initialize the pseudorandom number generator of numpy. Fixing a seed allows the results of a simulation to be reproduced. By default, a random value is drawn.
algorithm	(str)	Type of fast marching algorithm used during simulation. Valid values: 'Isotropic3', 'Riemann3'. By default, the 'Isotropic3' algorithm is selected.
costs	(dict)	Default travel costs dictionary used to build the travel cost map during the fast marching phase.

3.3. Defining the model parameters

The model parameters are defined via a Python dictionary. This structure offers a high degree of flexibility. This subsection explains the purpose of each key that the user can define in the model parameters dictionary and how they can be set. Each key corresponds to a specific feature modeled by pyKasso, usually with numpy arrays and pandas `dataframes`.

3.3.1. Sks

The optional `sks` key accepts as input value a dictionary that describes the general settings for the SKS model. Table 3 lists all the parameters available for this key. The `seed` parameter is an integer defining a random seed for the numpy Random Generator module. This parameter ensures the reproducibility of the results of a simulation. It is also possible to define sub-seeds for the generation of the fracturation and the inlet and outlet positions. If those seeds are not defined, this `seed` parameter acts as the master seed of the simulation and automatically generates the sub-seeds. The `algorithm` parameter indicates which algorithm will be considered during the fast-marching phase. The 'Isotropic3' option will use the isotropic algorithm while 'Riemann3' will use the anisotropic algorithm.

The `costs` parameter defines the fast marching costs. Table 11 in the appendix lists the keywords available and their associated default values.

If the `sks` key is not provided or remains empty, pyKasso will pick a random seed, select the 'Isotropic3' algorithm, and keep the default travel cost dictionary.

The following code shows two ways to define the model parameters and then run the DKN simulation with this `sks` parameters.

```
# Declare the model parameters
my_sks_parameters = {
    'sks' : {
        'algorithm' : 'Isotropic3',
        'cost' : {
            'faults' : 0.4,
            'fractures' : 0.2
        }
}
my_sks_parameters['sks']['seed'] = 182712
app.model.generate( model_parameters = my_sks_parameters )
```

The first lines (2 to 9) define a dictionary to store the model parameters in which the type of FMA algorithm is set to Isotropic3. The default cost values are modified for the faults and the fractures to indicate that the fractures will be more susceptible to karstification than the faults. In line 10, a different syntax is used to set the seed parameter for the generation of random numbers within the predefined dictionary, before launching the simulation in line 11.

Table 4
Name, type and description of `inlets` and `outlets` parameters.

Parameter	Data type	Description
`number`	(int)	Total number of inlets (or outlets) taken into account during simulation. If the `data` parameter is not provided, pyKasso will stochastically generate the required number of outlets. If the `data` parameter points to a list of points or to a file storing a list of points, pyKasso will pick the required number of points starting from the first element. If the list is too short, pyKasso will stochastically generate the missing points. If pyKasso needs to generate points, it will generate them accordingly to the `subdomain` and `geology` parameters.
`data`	(str, list)	Filename pointing to a list of 2D or 3D points, or list of 2D or 3D points representing outlet locations.
`shuffle`	(bool)	A boolean flag to shuffle the outlets order if set to `True`. By default, the value is set to `False`
`importance`	(list)	List describing how the outlets (or inlets) should be distributed during the hierarchical iterations of the DKN simulation. By default, the value is set to [1].
`subdomain`	(str)	Name of the subdomain where outlet generation will occur. Table 10 in the Appendix lists all the available options for subdomains. By default, the value is set to `'domain_surface'`.
`geology`	(list)	List of the geologic formations (codes) where outlet (or inlet) generation shall occur. The codes must match with declared geology parameters.
`seed`	(int)	Random seed for outlet (or inlet) generation.

3.3.2. Inlets and outlets

The `inlets` and `outlets` input parameters must be defined by the user. They are set by specifying a dictionary containing the list of inlets and outlets used during the network generation. The locations of these points can be set by giving a list of 2D or 3D positions or by indicating pyKasso to generate them stochastically. Both approaches can also be combined. Table 4 lists all the parameters available for those keys. If the parameters are not set, pyKasso will return an error.

As explained earlier, the conduits are generated iteratively in order to obtain a hierarchical structure.

The simulation is divided in a number n_i of iterations. n_i is automatically calculated by pyKasso. At each iteration, $i = 1, \dots, n_i$, a group of outlets is taken as starting points for the fast marching algorithm, and a group of inlets is taken as starting points for the particle tracking. The inlets will connect to one of the current outlets during that iteration. Each inlet is used (by default) only once. The user controls how pyKasso splits the inlets and outlets into groups and defines how they are treated by setting the `importance` parameters for the inlets and outlets. These parameters take a list of integer values as input.

For the outlets, the length of the importance list corresponds to the number of groups of outlets considered simultaneously during one iteration. The value of each element in the list corresponds to a weight. The weight divided by the sum of all the weights gives the proportion of outlets that will be included during one iteration. For instance, an `importance` factor of $[1, 2, 3]$ means that the outlets are subdivided into three groups. The first group contains $1/6^{th}$ of the outlets, the second group $2/6^{th} = 1/3^{rd}$ and the last group the remaining $3/6^{th}$ or half of the outlets.

For the inlets, the principle is the same. The inlets are split into groups using the `importance` parameter. The number of iterations n_i equals the product of the length of the `importance` parameter lists for the inlets and outlets.

Note that if the length of the `importance` list for the inlets contains only one value, but the number of outlet groups is 2, the code divides the inlets into two groups of inlets: one for each iteration.

Fig. 4 illustrates how the `importance` parameters influence the simulation.

Fig. 4a shows the case with `outlets.importance = [1]` and `inlets.importance = [1]`. Only one iteration is used to compute the fast marching from the 3 outlets. All the paths for the simulation of the conduits are computed independently from the inlets to the outlets. There is no hierarchy, every inlet will be connected to the closest outlet via a single direct path. Fig. 4b shows the case with `outlets.importance = [1]` and `inlets.importance = [1, 1]`. This parameterization corresponds to two iterations. In both iterations, all the outlets are taken (there is only one single group of outlets). During the first iteration, half of the inlets are used and will connect directly to the closest outlet. Then during the second iteration, the second half of the inlets are used and because the cost map has been modified after the first iteration, the trajectory of the new conduits starting from these second group of inlets, will be affected by the already existing conduits creating a tree-like structure. Fig. 4i shows the case with `outlets.importance = [1, 1, 1]` and `inlets.importance = [1, 1, 1]`. Now, the number of iterations is equal to 9. Each outlet is used in two iterations, and the inlets are divided into nine groups. By alternating the outlets, new types of structures including loops (or cycles) are generated. This example shows how the `importance` parameters can be used to control globally the structure of the simulated DKN.

3.3.3. Domain

By default the domain in which the DKN is computed is the full 3D `Grid` object defined earlier. However, in many situations, one has to restrict the domain's extension. This can be done by using the optional `domain` key. It takes a dictionary as an argument. The specific extension of the region where the karst conduits can be generated can be defined with a dictionary of four optional keys (see details in Table 9 in the Appendix):

- The `delimitation` key allows defining the lateral (horizontal) extension of the domain, corresponding to the maximum extension of the karst aquifer catchment area. It requires a list of 2D points defining a polygon. This data can be provided either via an external file or as a Python list of points.
- The `topography` key permits defining the upper boundary of the simulation domain. It usually corresponds to the digital elevation model (DEM).
- The `bedrock` key permits limiting the base of the study area. It usually corresponds to the top of the underlying insoluble basal bedrock or basement. For the sake of brevity, in the following and in the code, the word bedrock is used to denote this basal surface.
- The `water_table` key can be set to delineate the spatial extension of the boundary between the vadose zone in which the conduits are preferentially sub-vertical and the phreatic zone in which the conduits are preferentially sub-horizontal. It corresponds to a base level at a certain stage of karstification. It is possible for complex systems, to loop over different stages of karstification and change the paleo base levels to model these systems (see for example Banusch et al. (2022)).

The inputs for these three last attributes must represent a 2D surface. Such surface can be defined either from an external file (see Table 5 for the supported file formats) or directly from a custom 2D numpy array. Some examples of the usage of the domain are given in Examples 2 and 3 in Sections 4.2 and 4.3.

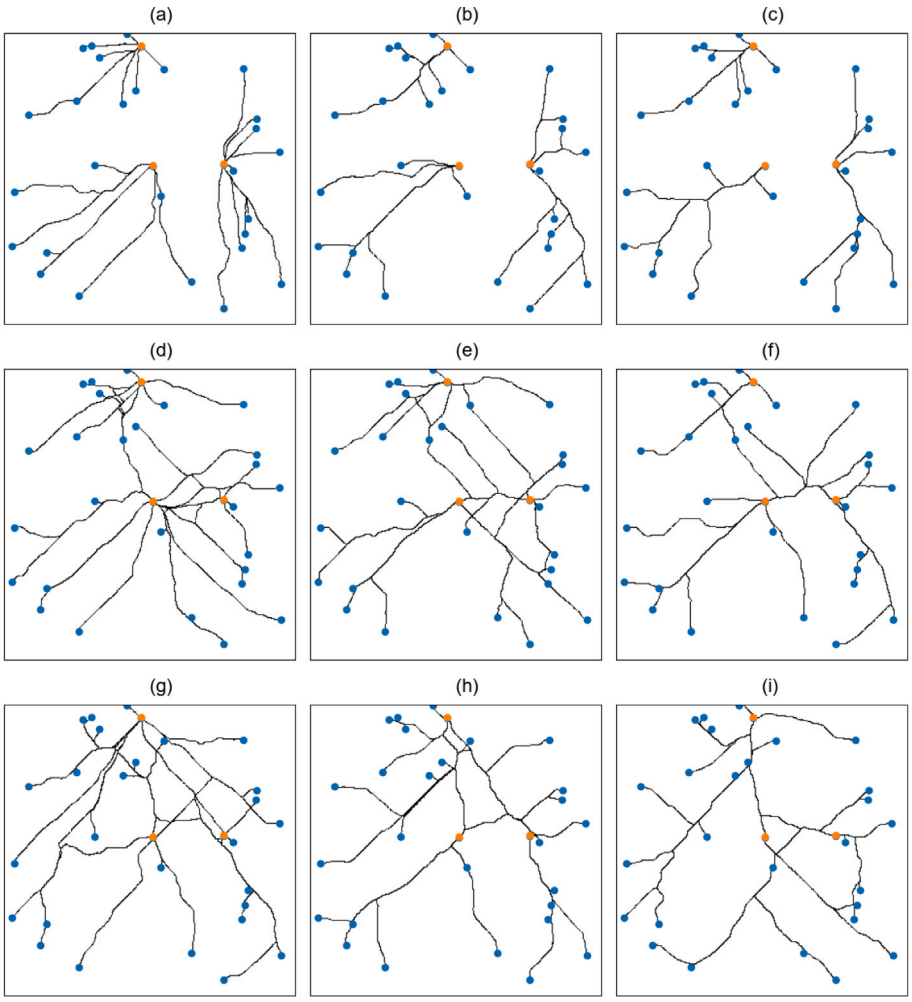


Fig. 4. Examples of simulations illustrating the sensitivity of the algorithm to the `importance` parameters. In all the simulations, the number and location of the outlets (orange dots) and inlets (blue dots) are identical. The only differences are the `importance` parameters. The `importance` parameter for the outlets is [1] for the first row, [1,1] for the second row, and [1,1,1] for the last row. The `importance` parameter for the inlets is [1] for the first column, [1,1] for the second and [1, 1, 1] for the last column. See the text for the discussion of the results.

Table 5
List of currently supported file formats.

Parameter	Type	Supported formats
`delimitation`	List	txt, csv

`topography`
`bedrock`
`water_table`
	2D array	txt, csv, gslib, vox, npy, tif, asc

`geology`
`structure`
`fractures`
	2D array
	3D array
		txt, csv, gslib, vox, npy

Formally, within pyKasso the domain is defined as an indicator matrix I:

$$I(i,j,k) = \begin{cases} 0 & outside\ the\ model \\ 1 & inside\ the\ model \end{cases} \quad (1)$$

with i, j and k the Grid indices.

If the `domain` key is not provided, pyKasso will ignore this feature. If the `domain` key is used and some of its parameters are not filled in, pyKasso will set those values to `None`.

3.3.4. Geology

Geology is used within pyKasso as the keyword to describe the geologic model. This model provides a three-dimensional representation of the geological units that are relevant for karstification.

Within the domain of interest, karst conduit networks are preferentially formed in geological units containing certain rock formations and lithologies that are susceptible to karstification. When this information is available, the user can pass it to the SKS algorithm by setting the components of the `geology` dictionary.

The codes describing the geological units are stored in a regular three-dimensional array G (voxet) having the dimension of the 3D Grid of the domain. For each cell of the voxet the geology is defined by a code g:

$$G(i,j,k) = g \quad (2)$$

where $g = 0, \ldots, N_g$ represents the units (or rock formations, or lithology) and N_g is the number of geological units in the model. The value 0 stands for the absence of rock and therefore cannot be used to describe a geological unit. When the geological model is not provided by the user, pyKasso assumes that the lithology is identical everywhere and assigns the default value from the fast marching costs dictionary.

Table 6 lists all the parameters that can be set in the `geology` dictionary. The `data` parameter can be set either to a predefined variable containing a 3D numpy array, or the name of an input file (see Example 3 in Section 4.3 and Table 5).

Table 6
Name, type and description of `geology` key parameters.

Parameter	Data type	Description
`data`	(str, np.ndarray)	Filename pointing to a 2D or 3D model, or a 2D or 3D numpy array representing the geologic model. Each geologic unit is identified by a code. If the geologic model is 2D and the grid is 3D, it will be replicated in the direction defined by the `axis` parameter.
`names`	(dict)	Dictionary assigning each geologic unit a name. By default, each identified unit is named `unit_n` with n the number identifying the geologic unit.
`costs`	(dict)	Dictionary assigning each geologic unit a specific travel cost for the fast marching algorithm. By default, each identified unit will get the value of the default travel cost set for geology.
`model`	(dict)	Dictionary assigning each geologic unit a Boolean value. If the value is `True`, the geologic unit will be taken into account during the simulation, otherwise not. By default, the value is set to `True`.
`axis`	(str)	Direction in which the model should be replicated (extended) if it is provided as 2D model. This is useful when only a geologic map, or a vertical section is given to build a simple 3D models assuming that the map or cross section does not change significantly in the perpendicular direction. This mode works only if the dimensions of the input 2D array match the corresponding dimensions of the side of the 3D Grid. Valid values: `'x'`, `'y'`, `'z'`. By default, the value is set to `'z'`.

For each code representing a rock unit and identified in the model, a name can be attributed with the help of a dictionary using the `names` key. Following the same principle, the `costs` key allows to attribute a specific fast marching cost to a given rock formation, and the `model` key allows the inclusion or exclusion of a rock unit in the simulation with Boolean values.

Note that a 2D geological model can also be provided to PyKasso as a 2D vertical cross-section or a 2D map. PyKasso can then extend this 2D model in a 3D grid but the user must specify the orientation of the 2D model using the `axis` key (see details in Table 6).

3.3.5. Faults and other geological features

In addition to the geological units describing the types of the rock formations, the user can provide a 3D model of known structures that can influence the karstification. These objects can be major faults crossing the domain, or any other important features such as bedding planes, unconformities, thrust systems, or alteration corridors that have been mapped and modeled in 3D. These features when they are susceptible to enhance the karstification are named inception features, or regional inception features if they have a significant spatial extent (Filipponi et al., 2009). Some of these features can also act as barriers to groundwater flow and influence the karstification process. In pyKasso, and for the sake of brevity, all these features are grouped within the optional `faults` dictionary (even if they are not actual faults). Following the approach previously described for the rock unit model, each of these features can be represented by an integer value stored on a 3D grid. If the parameter is not set, pyKasso uses an array full of zeros. Table 12 in the Appendix lists all the parameters that can be set in the `faults` dictionary.

3.3.6. Fractures

The `fractures` parameter is also optional. It allows for generating stochastic sets of fractures, joints, or bedding planes expected to influence the karstification process. Again, for the sake of brevity, the terminology fractures is used for all the planar features that pyKasso generates using a stochastic process. The parameter describes the orientation, distribution, length, and susceptibility to karstification of these local inception features. The fractures parameter takes a dictionary as input to describe the joint sets within the rock unis that should be taken into account by SKS. Following the principle used for rock formations and faults, each joint set is identified by an integer code. If the `fractures` parameter is not set by the user, pyKasso uses an array full of zeros and will not account for the joint set. The keys of the `fractures` dictionary (Table 13 in the Appendix) are similar to those introduced in the geology and faults subsections.

But an additional `generate` key is available for the fractures. It allows generating the stochastic Discrete Fracture Network (DFN) based on the given parameters. The `generate` key takes as input a dictionary with joint sets names as keys. Each key requires a dictionary with parameters defining the way pyKasso should simulate the fractures for that fracture set. Table 7 lists those parameters. A set of parameters controlling the dip, orientation, length, and density of the generated fractures is available.

As an alternative, it is also possible to import a DFN generated outside of pyKasso. The user must specify the file or numpy array containing the DFN using the `data` key.

Section 4.1 shows an example of the setup of a DFN with two families of fractures.

The DFN generator coded in pyKasso is rather simple and aims only at generating fractures that will influence the geometry of the DKN when no better DFN is available. The aim is not to model the complete fracture system that one can observe on a real site but to model the fractures that influence significantly the DKN.

In pyKasso, a joint set is defined by its density, orientation, dip, and length. The fracture set density parameter d_i corresponds to the number of fractures per unit area. Based on this parameter, the mean number of fractures λ_i of the i family is computed by multiplying the fracture density d_i by the surface of the domain S:

$$\lambda_i = d_i \cdot S \tag{3}$$

Then for each realization of a DFN, the number of fractures to simulate is drawn from a Poisson distribution:

$$p(k) = P(X = k) = \frac{\lambda_i^k}{k!} \cdot e^{-\lambda_i} \tag{4}$$

Without any constraints, the fracture locations are simply drawn from a uniform distribution that covers the extension of the simulation grid. Orientation, dip, and length are then randomly generated from the user-defined statistical distribution. Each parameter expects a list of two elements representing the range in which the simulated values should occur. The available statistical distributions are given in Table 8. If the selected statistical distribution requires additional arguments, these must be declared in the dictionary defining the fracture family.

The von Mises distribution (see Table 8) requires the parameters μ (the mean direction of the distribution) and κ (a shape parameter reflecting the measure of concentration). These are automatically calculated by pyKasso. The μ parameter is directly computed as the mean between the minimum angle θ_{min} and the maximum angle θ_{max}:

$$\mu = \frac{\theta_{min} + \theta_{max}}{2} \tag{5}$$

The κ parameter is calculated from the variance of the von Mises distribution (see Eq. (6)). The standard deviation σ is defined as the third of the difference between the maximum angle and the mean angle:

$$\sigma = \frac{\theta_{max} - \mu}{3} \tag{6}$$

This approach ensures that 99.7% of the angles are drawn between the user indicated range. The value of κ is then numerically solved from:

$$\sigma^2 = 1 - \frac{I_1(\kappa)}{I_0(\kappa)} \tag{7}$$

Table 7
Name, type and description of `generate` parameters from the `fractures` key.

Parameter	Data type	Description
`density`	(float)	Fracture density = number of fractures per unit area (m^2). It corresponds to the P20 parameter as defined by Dershowitz and Herda (1992).
`orientation`	(float, list)	Azimuth of the fracture strike orientation in degrees. The user can either give a constant value, that will be used for all the fractures, or a range of values given in a list [minimum, maximum].
`orientation_distribution`	(str)	Type of statistical distribution used to generate the fracture orientations. Table 8 lists the available options. By default, `'vonmises'`.
`dip`	(float, list)	Dip value or dip range in degrees. As for the orientation, the user can provide a constant value or a [min, max] range via a list.
`dip_distribution`	(str)	Type of statistical distribution used to generate the fracture dips. By default, `'vonmises'`.
`length`	(float, list)	Length of the fractures in the strike direction. The user can provide a constant value or a [min, max] range via a list.
`length_distribution`	(str)	Type of statistical distribution used to generate fracture lengths. By default, `'power'`.

Table 8
Statistical distributions available for the generation of the DFN.

Value	Type	Parameters	Probability distribution function
`uniform`	Uniform	No	$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \le x \le b, \\ 0 & \text{for } x < a \text{ or } x > b. \end{cases}$
`vonmises`	Von Mises	No	$f(x \mid \mu, \kappa) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$
`power`	Truncated power law	`alpha`	$f(x \mid x_{min}, x_{max}) = \frac{(\alpha-1)\cdot x^{-\alpha}}{x_{min}^{-\alpha+1} - x_{max}^{-\alpha+1}}$

Where $I_0(\kappa)$ and $I_1(\kappa)$ are the modified Bessel function of the first kind respectively of order 0 and order 1.

Once the DFN is simulated, it is rasterized on the Grid to obtain a matrix of indicator values indicating the presence of fractures in the grid cells following the approach used for the geologic formations or the faults. When several fractures cross the same cell, pyKasso keeps only the one with the lowest cost for the FMA.

3.4. Computing the karstic conduits

When the parameters are set, pyKasso can proceed to the computing of the karst conduit network. The algorithm follows the methodologies described in Borghi et al. (2012) and Fandel et al. (2022). This section shows how the main steps are implemented in pyKasso and the aspects that the user should know.

3.4.1. Conceptual model

Once all the parameters concerning the geologic features previously described have been defined, loaded, and modeled using numpy arrays, `Python` list, and `Python` dictionaries, they are all stacked in a *conceptual model*. This new object defines how each geologic feature is taken into account in the final simulation. Fig. 5 illustrates this operation. The figure shows that the rock types, faults, and fractures are given as independent arrays. The conceptual model is obtained by stacking these arrays. When different pieces of information are overlapping, the faults have the higher priority and will be kept preferentially, then the fractures, and finally the rock type.

To check how the conceptual model has been setup, the numpy array `conceptual_model`, stored in the `app.model` object, contains the result and can be visualized (Fig. 5). Furthermore, the Pandas table `conceptual_model_table`, stored in the same object and summarizing the main features taken into account during the simulation, can be accessed after the simulation.

3.4.2. Karst conduit network simulation

From the conceptual model, the main task is to compute the propagation field and run the fast marching algorithm (FMA) to obtain a time map from which the karst conduit network will be derived. To get a DKN having a hierarchical structure, the previous step is iteratively repeated. Inlets and outlets are distributed among the iterations using the `importance` and `per_outlet` parameters. The results of the first iterations are transferred to the successive ones.

The user can decide to use an isotropic or anisotropic propagation field, by selecting the type of algorithm in the sks parameters (see Table 3). For the isotropic case, the cost is given in all the cells of the grid as a local scalar. For the anisotropic case, the same scalar is used, but to obtain an anisotropy tensor, pyKasso needs, in addition, a local rotated coordinate system defined by three orthogonal unit vectors, and the local anisotropy ratio.

For each simulation, the following steps are iteratively executed:

1. **Define the scalar cost map:** This step follows what is described in detail in Borghi et al. (2012). The values of the conceptual model representing each feature are transformed into their respective FMA cost. The operation is only carried out during the first iteration. In subsequent iterations, the cost map is updated only to account for the karst conduits generated in the previous iteration. The resulting cost map is then appended to a list for checking, if needed.
2. **Define the anisotropy cost tensor:** If the anisotropy mode is selected, a propagation field containing spatially varying anisotropy tensors must be defined. This was presented in 2D in Fandel et al. (2023). It is extended here in 3D. The anisotropy tensors are defined using the spatially varying scalar cost map given above, anisotropy ratios, and orientation vectors that are different in different regions of the domain. This is different from what was proposed by Luo et al. (2021) who define the anisotropy tensor based on the equivalent permeability tensor computed from the DFN. Here, the tensor is defined differently. In the vadose zone (above the `water_level` surface and below the `topography`), the first principal direction of anisotropy is vertical. The cost in that direction is minimal. The two other directions are perpendicular to the first one and are therefore lying within the horizontal plane. The cost in these two directions is identical and higher than the one in the vertical direction. This leads to the formation of conduits that are preferentially vertical in this region. At the bottom of the vadose zone along the insoluble basal bedrock surface, the anisotropy tensor is oriented such that the minimal cost is given in the direction of the maximum slope of the bedrock. This is to favor the formation of conduits following the steepest slope at the interface between the karstifiable rock formation and the underlying not karstifiable bedrock. All the cells located two grid cells above the bedrock have this type of anisotropy. Then, the remaining cells in the phreatic zone are considered isotropic to let the karst conduit network develop following the geologic heterogeneity and be oriented toward the outlets.
3. **Fast marching algorithm:** Once the propagation field is defined, the time map is calculated by applying the fast marching

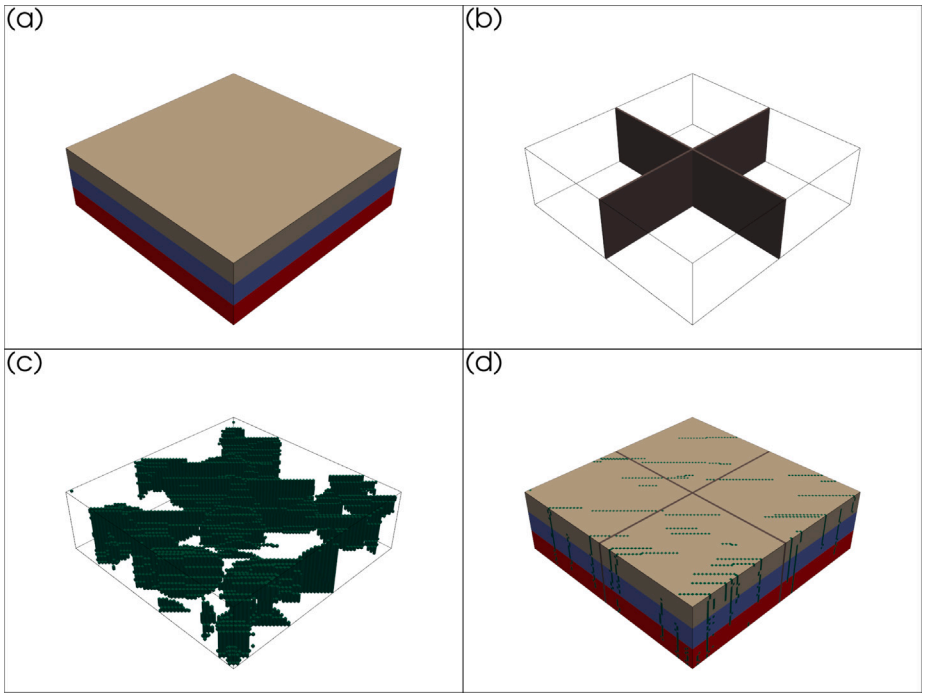


Fig. 5. Illustration of the construction of the conceptual model: (a) Geologic model of three uniform units stacked horizontally; (b) Fault model of two vertical faults crossing perpendicularly; (c) Discrete fracture model; (d) Conceptual model with the geology overlapped by the fractures, and then overlapped by the faults.

algorithm (Mirebeau and Portegies, 2019) first. The streamlines, or particle paths, are then computed from the current inlet to the current outlet using the gradient of the time map. During this stage, the streamlines are extracted and transformed to construct the karst conduit network of that iteration. The karst conduit network is then voxelized into the 3D grid to update the cost map for the next iteration or for further post-processing and visualization.

A complete simulation is calculated each time the `generate` method is called. To generate multiple simulations, the `generate` method can be integrated into a loop, with a parameter dictionary as an argument. To get results independent from the previous simulation, the settings of the dictionary must be edited after each simulation. For instance, this snippet showcases how to change the number of outlets within a `for` loop of three iterations:

```
outlets = [10, 20, 30]
for n in outlets:
    parameters['outlets']['number'] = n
    app.model.generate(parameters)
```

At the end of every simulation, pyKasso stores each feature in a dictionary and saves it in the `results.pickle` file. This file is stored in the root of the folder corresponding to the current simulation index. If the whole data from the simulation is exported, the results file can become very large. For example, for the Tsanfleuron case study (Section 4.3), the grid contains 7.6 million cells. The pickle file has a size of 1.8 Gb. But all this information is mainly useful for controlling and checking the results. If we are only interested in the final DKN in vectorial form, the file size for this example is around 200 Kb, which is very small. The **Project** object is also updated at the end of each simulation. The value of the `n_simulations` attribute is incremented, and the location of the folder of the simulation is appended in the `simulations` attribute. Finally, the `project.yaml` is exported with the new attribute values.

Ultimately, the memory used by the simulation is not flushed away. Because I/O operations are time-consuming in Python, a memoization step optimizes memory usage between two successive simulations. By comparing the current simulation parameters dictionary with the previous one, only the parts where modifications are required are updated.

4. Examples

This section presents three examples to illustrate how to define the parameters and run pyKasso for modeling DKNs. The first example is a very basic 2D case illustrating the minimal configuration that can be used. The second example is slightly more complex. It showcases the differences in results obtained in 3D with the isotropic and anisotropic fast marching options. The final example shows the application of pyKasso to a real field site: the Tsanfleuron aquifer in Switzerland.

4.1. Example 1: 2D simulation of karst conduits in a fractured karstifiable rock unit

For this example, the aquifer is assumed to be two-dimensional and homogeneous. There is one known outlet and one known inlet with both fixed positions. The fractures are simulated in a stochastic manner. The full code of this example is available on the pyKasso GitHub repository.

4.1.1. Setting up the parameters

After the creation of a new pyKasso application, a two-dimensional grid of 300 by 300 cells and the name of the new project `'example_01'` are defined using the following code snippet:

```
# Import the pyKasso package
import pykasso as pk

# Create a pyKasso application
app = pk.pykasso()

# Declare the grid parameters
```

```
grid_parameters = {
    'x0': 0,   'y0': 0,   'z0': 0,
    'nx': 300, 'ny': 300, 'nz': 1,
    'dx': 10,  'dy': 10,  'dz': 10,
}

# Declare the project parameters
app.new_project(name='example_01',
    grid_parameters=grid_parameters)
```

The smallest configuration that can be used in pyKasso consists of defining only the inlet and outlet options. If the other components of the geologic model are not defined, then pyKasso automatically uses the default values for them. The code below defines the dictionary of the model parameters. The inlet is located at the center of the left edge ($x = 0, y = 1500$) and the outlet at the center of the right edge of the domain ($x = 2990, y = 1500$). Since there is only one inlet and one outlet, the simulation will not be hierarchical and the lists of importance weights for the inlets and outlets contain only a 1 for both cases. For the fractures, the `generate` option calls the internal DFN generator. Two families of stochastic fractures are defined. The orientations are constant for each family with an azimuth of 135 degrees for the first one and 45 degrees for the second. They are both vertical. The lengths are also different for the two families. The second family has longer fractures (500 m) than the first one (300 m).

```
# Declare the model parameters
model_parameters = {
    'sks' : {},
    'outlets' : {
        'data' : [[2990, 1500]],
        'number' : 1,
    },
    'inlets' : {
        'data' : [[0, 1500]],
        'number': 1,
    }
    'fractures' : {
        'generate': {
            'family_01': {
                'density'     : 0.00005,
                'orientation' : 135,
                'dip'         : 90,
                'length'      : 300,
            },
            'family_02': {
                'density'     : 0.00005,
                'orientation' : 45,
                'dip'         : 90,
                'length'      : 500,
            }
        }
    }
}
```

4.1.2. Running the simulations

Once the dictionary containing the parameters is defined, the code below shows how to generate 100 stochastic simulations with a `for` loop. The computing time is about 1 min for generating a hundred simulations on a laptop (Intel i7 processor) running under Windows 10. At each iteration, the same global model parameters are used as arguments for the `generate` method. To ensure the reproducibility of the results, the seed is set for each simulation as a function of the iteration number. If we need to generate again one specific simulation, it is sufficient to call the `generate` method again with the same seed.

```
# Compute 100 simulations in a row
for i in range(100):
    model_parameters['sks']['seed'] = i
    app.model.generate(model_parameters=
    model_parameters)
```

4.1.3. Visualizing and analyzing the results

The `visualization` sub-package offers a set of methods to display the results. For this example, only the methods used to visualize 2D results are presented. Other methods will be discussed later.

The code snippet below illustrates how the `mpl_plot_2D` method can be used to generate two-dimensional maps of the different quantities used and computed by pyKasso. The `mpl_plot_2D` method plots the results of the last simulation in 2D. The `feature` argument allows to select which quantity must be plotted. Since this function is based on the `matplotlib imshow` function, all the usual options of `imshow` can be employed using the argument `imshow_option`. This is illustrated for example to select a binary colormap (`'cmap':'binary'`).

```
# Plot the last simulated karstic conduit
    network
app.visualizer.mpl_plot_2D(feature='karst',
                           imshow_options={'
    cmap': 'binary'},

    scatter_inlets_options={},

    scatter_outlets_options={})

# Plot the discrete fracture model
app.visualizer.mpl_plot_2D(feature='
    fractures',
                           imshow_options={'
    cmap': 'binary'})

# Plot the cost array
app.visualizer.mpl_plot_2D(feature='cost',
                           imshow_options={'
    cmap': 'binary'})

# Plot the travel time array
app.visualizer.mpl_plot_2D(feature='time',
                           imshow_options={'
    cmap': 'binary'})
```

Fig. 6 shows the resulting figures. Fig. 6a depicts the simulated karst conduit network. It was generated from the blue point representing the inlet and found its way through the fractured media to reach the outlet represented by the orange point. The conduit geometry is influenced by the fracture pattern (Fig. 6b), but also by the contrast of the travel cost of the matrix and the fractures (Fig. 6c). Here the contrast is equal to 2, but a smaller contrast would favor a conduit geometry that would be more straight between the inlet and outlet. A higher contrast would lead to a geometry even more influenced by the fractures (see Borghi et al. (2012) for some examples). Finally, the time map (Fig. 6d) is computed by the fast marching algorithm from the cost map and the location of the outlets. A gradient descent is applied on this map to determine the trajectory of the karst conduits. All these maps can be visualized by the user to check intermediate calculations if needed when constructing a DKN.

With the help of the `analysis` sub-package, one can also calculate the map of the probability of occurrence of a conduit. Since the conduits are rasterized on the grid and represented by an indicator function for each simulation (0 means no conduit, 1 means the presence of a conduit), it is sufficient to average all the maps of the rasterized karst conduit networks to compute this probability. To compute this average, the code snippet below shows that one can use the `compute_stats_on_networks` method of the `analysis` sub-package to compute the mean of the indicator functions. It is also possible to obtain other statistics by changing the argument with the name of an another function supported by numpy. For instance, with `numpy_algorithm='median'`, the methods will return the median array of all the simulations. Then the visualization is made with the `mpl_plot_array_2D`. The `mpl_plot_array_2D` method differs from the `mpl_plot_2D` method in that it is a static method. It can be used independently from the project to plot any array.

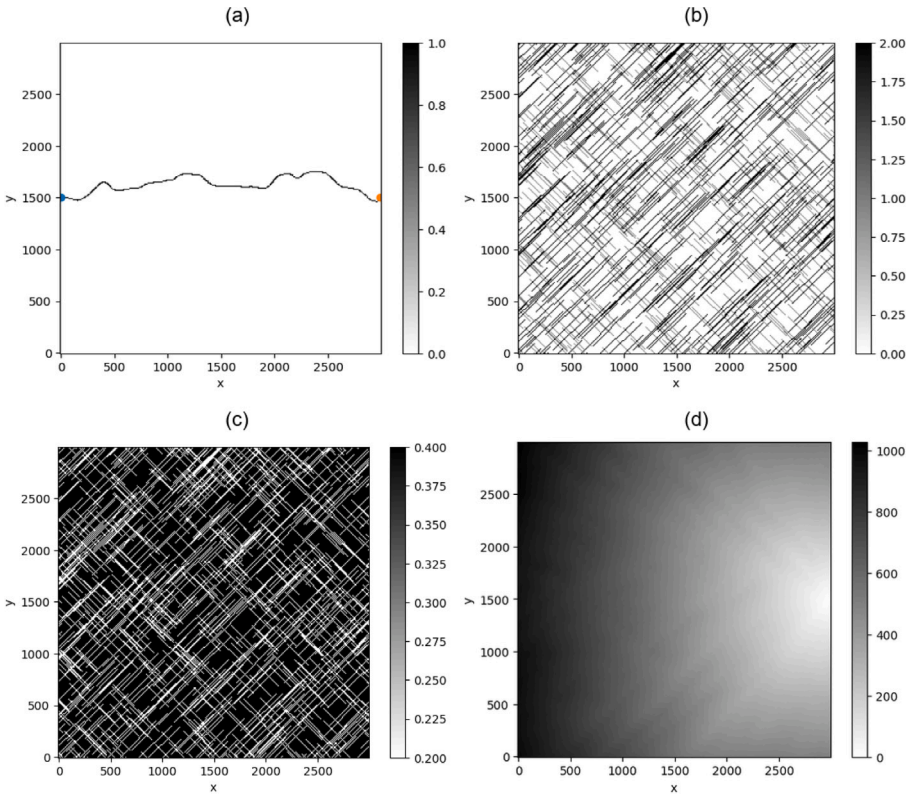


Fig. 6. Examples of outputs from the results of the last simulation: (a) Karst conduit network; (b) Generated fracture model; (c) Travel cost map; (d) Time map calculated by the fast marching algorithm.

```
# Compute the mean of all the simulated
    karstic networks
probability = app.analyzer.
    compute_stats_on_networks(
    numpy_algorithm='mean')
extent = app.project.grid.extent
imshow_options = {
    'extent' : extent,
    'cmap' : 'binary',
    'vmin' : 0, 'vmax' : 0.1}
app.visualizer.mpl_plot_array_2D(array=
    probability,

    imshow_options=imshow_options)
```

Fig. 7 shows such a result with 100 simulations. Far away from the known inlet and outlet, the probability becomes very rapidly small, but the width of the region in which the conduit could be observed is rather well defined by this map. These results depend on the assumptions made for the calculation (parameters of the DFN model and cost contrast).

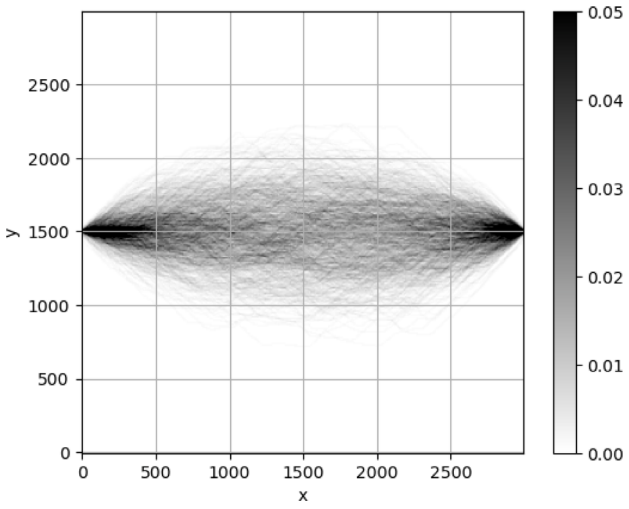


Fig. 7. Probability of occurrence of a conduit computed from 1000 simulations. The probability is the highest close to the inlet and outlet. It rapidly decreases but becomes negligible for y below 700 and higher than 2300.

4.1.4. Retrieving the results

The results of the simulations are accessible from the `model` sub-package. For each simulation, the results are stored in attributes within the **SKS** instance located in the `model` attribute of the `app` instance. According to the output settings, the results are also saved in the sub-directory corresponding to the current simulation.

All the input arrays used by the fast marching algorithm to compute the karst conduit network are stored in the `maps` attributes from the `model` sub-package. For instance, the array representing the DKN from iteration i can be retrieved with the `'karst'` key: `app.model.maps['karst'][i]`. All the other variables can be retrieved in the same way through the `maps` attribute.

```
app.model.maps.keys()
> ['outlets', 'nodes', 'cost', 'alpha', '
    beta', 'gradient', 'time', 'karst']
karst = app.model.maps['karst'][0]
time = app.model.maps['time'][0]
```

4.2. Example 2: 3D simulation of karst conduits including a water table

The second example illustrates how to set a basic three-dimensional model including a water table to separate the vadose and phreatic

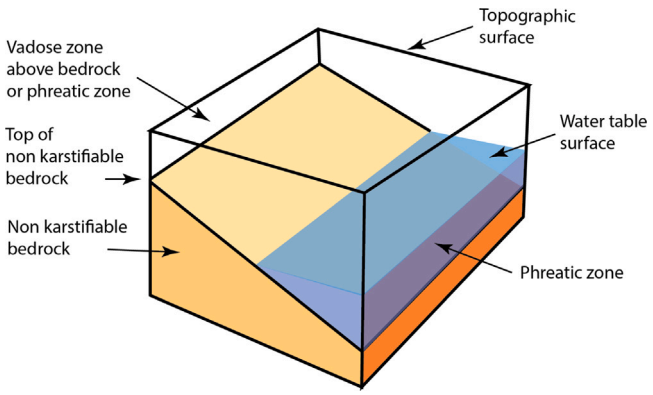


Fig. 8. Conceptual model for Example 2. The 3D domain includes an non-karstifiable basal bedrock and a karstifiable formation separated by an inclined surface sloping from 300 m of altitude on the west side to 100 m of altitude on the east side. A horizontal groundwater surface located at an altitude of 200 m separates the phreatic from the vadose zone within the karstifiable unit.

zones. This example highlights the differences in results obtained using the isotropic and the anisotropic fast marching algorithm. The full code is available on the pyKasso GitHub repository.

4.2.1. Setting up the parameters

As for the previous example, after the creation of a new pyKasso application, a three-dimensional grid and a name are declared. The only difference with the previous example is that the vertical extension of the grid is discretized in 50 horizontal layers ($nz = 50$).

```
# Define the grid parameters and the project
grid_parameters = {
    'x0': 0, 'y0': 0, 'z0': 0,
    'nx': 100, 'ny': 100, 'nz': 50,
    'dx': 10, 'dy': 10, 'dz': 10
}
app.new_project(name='example_02',
    grid_parameters=grid_parameters)
```

The conceptual model in this example is illustrated in Fig. 8. The 3D domain is a rectangular box. The top surface is the topography of the system. The domain is divided into two parts: the upper part corresponds to a rock formation susceptible to karstification, and the lower part is a non-karstifiable rock formation or insoluble basal bedrock. The boundary between these units is a perfect plane sloping toward the east from 300 m of altitude on the west side of the domain to 100 m on the east side. The boundary between the vadose and phreatic zone (water table) is a horizontal plane set at 200 m of altitude. This level is assumed to be fixed during the karstification phase, and corresponds to the final position of the groundwater base level. The water table boundary cuts the top of the bedrock right in the middle of its vertical and lateral extension. Ten inlets are randomly drawn at the surface of the model, while one outlet is purposely drawn within the phreatic zone at the surface of the insoluble basal bedrock.

A simple way to implement this conceptual model in pyKasso is to define the top bedrock and the water table surfaces using numpy arrays. The code snippet below shows how the two arrays can be created using standard numpy methods. The arrays must have the same horizontal extension as the 3D grid. They can also be imported from external data, or interpolated from field data with any technique before being used in pyKasso. The resulting `bedrock` and `water_table` arrays are then used directly in the model parameters as shown below. Another novelty in this code snippet is the generation of random inlets. In this situation, it is very simple, the user has just to provide the number of inlets. Since the coordinates are not given, they will be simulated randomly on the top surface of the domain. By providing the seed in the parameters and keeping it constant, it is possible to see the impact of the modification of the other parameters on the final results while keeping the locations of the simulated inlets fixed. Later, if needed, the seed can be changed to analyze the impact of this source of uncertainty on the results.

```
# Construct the bedrock elevation
bedrock = np.linspace(300, 100,
    grid_parameters['nx'])
bedrock = np.repeat(bedrock[:, np.newaxis],
    grid_parameters['ny'], axis=1)

# Construct the water table
grid_shape = app.project.grid.shape[:2]
water_table = np.full(grid_shape, 200)

# Declare the model parameters
model_parameters = {
    'sks' : {
        'seed' : 1
    },
    'domain' : {
        'bedrock' : bedrock,
        'water_table' : water_table
    },
    'outlets' : {
        'seed' : 1,
        'number' : 1,
        'subdomain': 'bedrock_phreatic',
    },
    'inlets' : {
        'seed' : 1,
        'number': 10,
    }
}
```

4.2.2. Running and visualizing the simulations

For this example, only two networks are simulated: one with the isotropic fast marching algorithm and one with the anisotropic version. As explained before, because the seed for generating the inlets is kept identical for the two simulations, their locations remain the same. The `pv_show` method (based on the `pyvista` package (Sullivan and Kaszynski, 2019)) is used to visualize the three-dimensional results.

```
# Compute 1 simulation with isotropic and 1
    simulation with anisotropic fast
    marching algorithm
for algorithm in ['Isotropic3', 'Riemann3']:
    model_parameters['sks']['algorithm'] =
    algorithm
    app.model.generate(model_parameters=
    model_parameters)

# Print out 3D results
app.visualizer.pv_show(simulations=[0, 1],
    features=['karst'])
```

Fig. 9 shows the results of these two simulations. In the isotropic case, pyKasso does not account for the vadose zone: the conduits follow the shortest path in the three-dimensional domain which are straight lines from the inlets to the outlet. In the anisotropic case, the figure shows three types of paths. In the vadose zone, the conduits run vertically downwards until they reach the bedrock surface or the water table. When they reach the bedrock, they follow the bedrock slope until they finally reach the water table. When the conduits enter the saturated zone below the water, the cost becomes isotropic and the conduits run straight to the outlet. The location of the spring is unusual, but it was chosen on purpose to illustrate the behavior of the anisotropic algorithm.

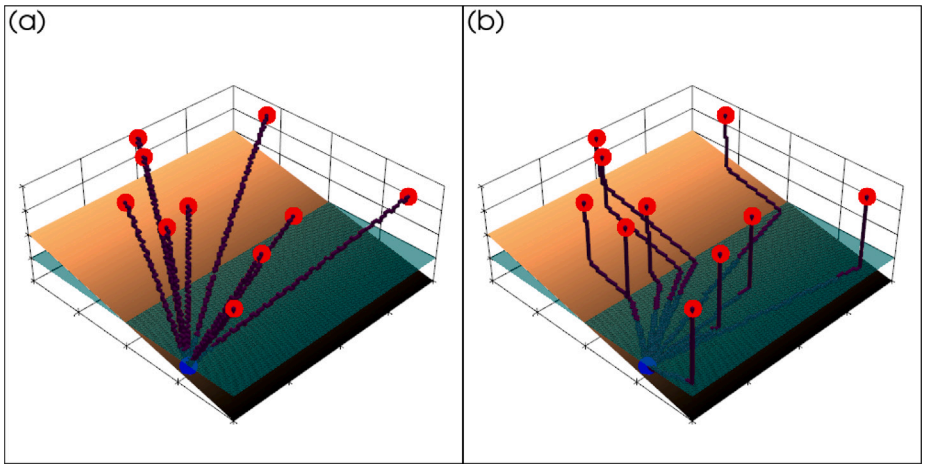


Fig. 9. Karst conduit networks simulated with the isotropic (a) and anisotropic (b) fast marching algorithms. The orange surface represents the top of the non-karstifiable bedrock and the blue transparent surface represents the water table. The complete code to reproduce this figure is available on the GitHub repository.

Among the results, the positions (x, y, z) of the inlets and outlets are stored in the pandas *DataFrame* `inlets` and `outlets`accessible from the `model` sub-package.

```
inlets = app.model.inlets
outlets = app.model.outlets
```

4.3. Example 3: the tsanfleuron aquifer

In this last example, a DKN model is constructed using a real data set. The following text explains how to import a geologic model, delimit a watershed, define the faults, and create a DFN based on statistical observations. We also show how to define the inlets and outlet parameters to produce the DKNs.

4.3.1. Geologic and hydrogeologic context

The study site is located in the Diablerets Massif (Schoeneich and Reynard, 2021) in the Swiss Alps. It is bounded on the West by the summits of Les Diablerets and Becca d'Audon Olderon and on the east side by the Sanetsch path. A karstic aquifer of Barremian-Aptian age (Urgonian formation) and belonging to the Helvetic nappes outcrops in most of the area. It is partly covered (and fed) on the upper part by the Tsanfleuron glacier (Neven et al., 2021) and is drained on its southeastern side by the Glarey spring (Gremaud et al., 2009; Gremaud and Goldscheider, 2010). However, even though cave exploration is active in the area, the main karst conduit network that brings water from the catchment to the Glarey Spring is still unknown.

The following will illustrate how pyKasso can be used to construct DKN models for this situation. The work relies on previous geologic and hydrogeologic studies (e.g. Gremaud et al., 2009; Gremaud and Goldscheider, 2010; Measday, 2021). The 3D geologic model, the inventory of possible inflow locations, main faults, and statistical distributions for the main fracture families were constructed and assembled by Measday (2021).

4.3.2. Setting up the parameters

The 3D grid is defined with the following parameters.

```
grid_parameters = {
    'x0': 2582750, 'y0': 1127767, 'z0': 1000,
    'nx': 394, 'ny': 193, 'nz': 100,
    'dx': 20, 'dy': 20, 'dz': 20
}
app.new_project(project_name='example_03', grid_parameters=grid_parameters)
```

The resolution is coarse (20 m x 20 m x 20 m) as compared to the expected dimensions of a single conduit (a few meters). However, it is sufficient to describe the structure of the system and the possible occurrence of a conduit within a cell. With around 7.6 million cells, this model is already rather large. It needs around 5.7 Gb of RAM and less than 4 min of computing time for generating one simulation on a laptop (Intel i7 processor) running under Windows 10. Note that the memory management is such that RAM usage does not increase with the number of simulations since the results are stored on the disk in a separate file after each simulation.

The geologic model was built (Measday, 2021) using geologic and tectonic maps, structural measurements (strike and dip), a set of geologic sections, and 3D interpretations. The model includes the main faults that can be identified on the topographic surface. The faults are interpolated in 3D as surfaces controlled by their traces on the surface and geologic sections, and by structural data. It is assumed that their tiplines have an elliptical shape (Calcagno et al., 2008). The 3D geologic model (rock formations and faults) is exported from GeoModeller as a regular grid in a .vox file. This is an ASCII file containing in each line the coordinates of the cells and an integer value representing the rock formation. To import that file, pyKasso provides a data reader object. The following snippet illustrates how the .vox file can be loaded by pyKasso.

```
# Create a data reader object
dr = pk.DataReader(grid=app.project.grid)

# Load geologic model
filename = input_dir + 'tsanfleuron_geology.vox'
df = dr.get_dataframe_from_file(filename)
geology = dr._get_data_from_vox_df(df)
```

The correspondence between the integer code and the names of each rock unit is given with a dictionary. If some keys are not given in the dictionary, pyKasso will use a default name based on the geologic key. The travel costs for the SKS algorithm of each rock unit are given with the same method, again if no value is provided, pyKasso will use the default travel cost for rock units (see Table 11). To declare which units should be considered or ignored during the simulation, another dictionary associating each rock unit key with a boolean value must be set. By default, pyKasso consider all the units. The code below shows how all these parameters were set and then stored in the dictionary `model_parameters`.

```
# Set names
geology_names = { -9999: 'Out', 0: None, 1: 'UHNappe', 2: 'Dogger',
```

```
    3: 'Malm', 4: 'Berriasian', 5: '
    LateBerriasianValanginian',
    6: 'Hauterivian', 7: 'EarlyBarremian',
    8: 'Urgonian',
    9: 'Eocene', 10: 'Wildhorn'}

# Set costs
geology_costs = {8: 0.4, 9: 0.4}

# Set model
geology_model = { -9999: False,
    0: False, 1: False, 2: False, 3: False,
    4: False, 5: False, 6: False, 7: False,
    8: True, 9: True, 10: False}

model_parameters = {}
model_parameters['geology'] = {
    'data' : geology,
    'names' : geology_names,
    'costs' : geology_costs,
    'model' : geology_model,
}
```

The same procedure applies for the faults. The grid containing the faults and all the relevant parameters are added to the `model_parameters` dictionary. Note that the code below does not define specific costs for specific faults. pyKasso will use the same default values for all of them. But, following the example given above for the rock units, it is also possible to indicate that different faults have different susceptibilities to karstification processes with different cost values.

```
# Load faults model
filename = input_dir + 'tsanfleuron_faults.
    vox'
faults = dr.get_data_from_file(filename)

# Set names
faults_ids = list(range(21))
faults_names_ = ['Chevauchement', 'NEES001',
    'NEES002', 'NES01',
                 'NES011', 'NES02', 'NES022',
    'NES03', 'NES032',
                 'NES04', 'NES042', 'NES043',
    'NES044', 'NES05',
                 'NES06', 'NES07', 'NOOSEE1',
    'NOOSEE2',
                 'NOOSEE3', 'NS2', 'OE1']
faults_names = {faults_id: faults_name for (
    faults_id, faults_name) in zip(
    faults_ids, faults_names_)}

model_parameters['faults'] = {
    'data' : faults,
    'names' : faults_names,
}
```

For the simulation of the DFN, the statistics were inferred from field observations. Four main fracture families have been identified for the Urgonian formation. These fractures are assumed to be sub-vertical. The distribution of their length is based on the measurement of trace lengths from aerial photographs and high-accuracy digital elevation models. The statistical parameters of these distributions were estimated by Measday (2021). Note, that small fractures (< 20 m) are discarded from the distributions. We only account for fractures longer than the resolution of the grid. In the `'settings'` parameter dictionary, each fracture family is declared by its name as key followed by a dictionary of settings controlling the fracture generation. By default, the von Mises statistical distribution is used for the orientation and dip distributions and the truncated power law for the length distribution (Table 8).

```
model_parameters['fractures']: {
    'settings': {
        'family_01': {
            'density'     : 4.5e-5,
            'orientation' : [32, 86],
            'dip'         : 90,
            'length'      : [20, 400],
            'alpha'       : 1.4,
        },
        'family_02': {
            'density'     : 1.2e-5,
            'orientation' : [146, 196],
            'dip'         : 90,
            'length'      : [20, 130],
            'alpha'       : 1.2,
        },
        'family_03': {
            'density'     : 4.5e-5,
            'orientation' : [98, 163],
            'dip'         : 90,
            'length'      : [20, 740],
            'alpha'       : 2.1,
        },
        'family_04': {
            'density'     : 2.8e-5,
            'orientation' : [72, 115],
            'dip'         : 90,
            'length'      : [20, 160],
            'alpha'       : 1.8,
        }
    }
}
```

The position of the outlet corresponds to the location of the Glarey spring, it is given in the text file `Outlet_ModeleG.txt`. The positions of the inlets correspond to a set of points where tracers have been injected and a connection with the inlet has been shown (Gremaud and Goldscheider, 2010). All these locations are provided in the file `Inlet_ModeleG.txt`. The water table is set arbitrarily as a uniform level of 1686 m slightly above the level of the Glarey spring. This is assumed to correspond to the base level of the system but it is highly uncertain since no direct observation of the boundary between the vadose and phreatic zones are available in the area.

Finally, the model parameters are completed by defining the random seed, selecting the `Riemann3` algorithm (see Section 3.3.1 for details) for the anisotropic fast marching, and defining the anisotropy ratio to 10%.

```
model_parameters.update({
    'outlets' : {
        'data' : [ [2589373., 1128641.] ],
        'number' : 1,
        'subdomain' : 'domain_surface',
    },
    'inlets' : {
        'data' : 'tsanfleuron_inlets.txt',
        'number': 22,
        'importance': [1, 1, 2],
    },
    'domain' : {
        'water_level' : water_table,
    },
    'sks' : {
        'seed' : 3333,
        'algorithm': 'Riemann3',
        'costs': {'ratio': 0.1}
    },
})
```

Ten simulations are generated with identical parameters.

```
# Compute 10 simulations
```

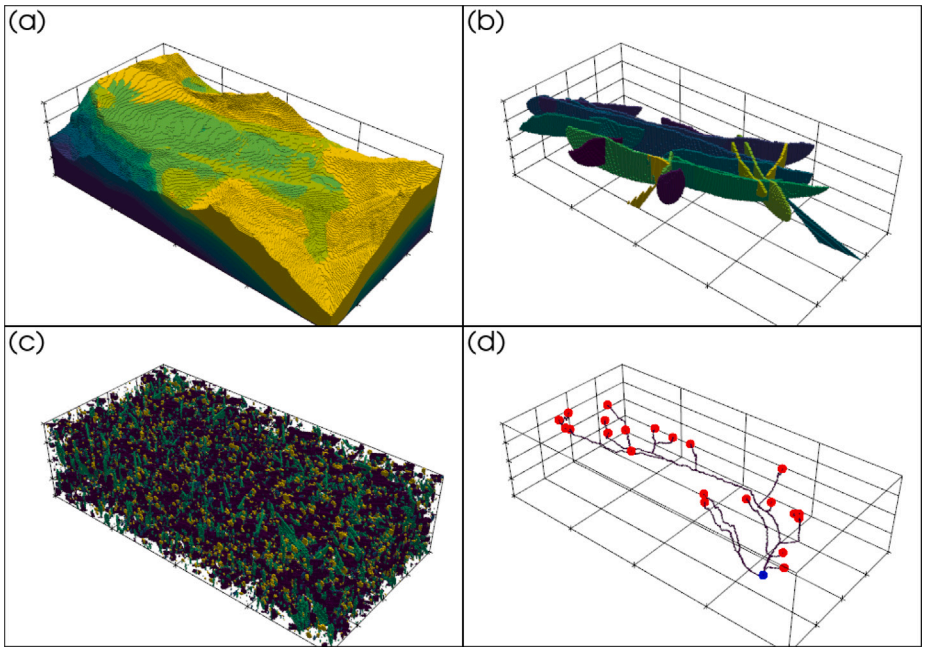


Fig. 10. Examples of outputs from the results of the last simulation: (a) Geologic model; (b) Faults model; (c) Generated fractures model of four different fracture families; (d) Karst conduit network.

```
for i in range(10):
    model_parameters['sks']['seed'] += 1
    app.model.generate(model_parameters=
    model_parameters)
```

For this model, the computing time is on the order of a few minutes per simulation. Once the simulations are finalized, all the geologic features can be retrieved by invoking their names as attributes from the `model` sub-package. The code snippet below shows for example how to obtain the numpy arrays describing the geology, the fractures, and the faults.

```
geology = app.model.geology.data_volume
fractures = app.model.fractures.data_volume
faults = app.model.faults.data_volume
```

All these features are stored as `GeologicFeature` objects containing the arrays and dictionaries describing the data. The array describing the corresponding features is stored in the `data_volume` attribute. Accessing the model attributes following this approach allows to obtain the input data or results of the last simulation. Figs. 10a, 10b and 10c show respectively the rock units, the faults, and the simulated fractures arrays.

Fig. 10d represents the DKN stored as a binary variable. But, before being voxelized, the fast marching algorithm returns the DKN as a set of nodes and edges representing the karst network as 3D lines. This data structure can also be retrieved from the `vectors` dictionary attribute with the following syntax.

```
nodes = app.model.vectors['nodes']
edges = app.model.vectors['edges']
```

The `nodes` dictionary associates a node key to a list of four elements $\{i_n : [x_{i_n}, y_{i_n}, z_{i_n}, \text{type}]\}$ describing the position of the node and its type (inlet, junction, or outlet). The `edges` dictionary associates an edge key with a list of two node keys $\{j : [i_n, i_m]\}$ from the `nodes` dictionary. This representation of the DKN corresponds to a graph in the mathematical sense. Such a graph can be exported and used to mesh the conduit network to solve flow and transport outside of pyKasso, for example with MODFLOW-CFPY (Reimann et al., 2023).

The graph can also be used to compute statistics describing the geometry and topology of the graph. Such statistics have been studied and described in detail by Collon et al. (2017) or Jouves et al. (2017). The `analysis` sub-package of pyKasso proposes several post-processing functions to compute those statistics and evaluate the plausibility of the simulated conduits. These functions call the Python package KarstNet[1] and compute five statistical indicators related to the geometry of the network such as the mean length of the branches, the coefficient of variation of those lengths, the mean tortuosity of the network, or the entropy of the orientations of the edges, as well as five topological indicators such as the average shortest path length or the central point dominance which provide information about the topological structure of the graph. The same statistics have previously been computed on a set of real karstic networks (Collon et al., 2017). If the computed values for a given simulated network lie in the range of the values computed for real networks, this is an indication that the simulated network is plausible.

To conduct this analysis, the user can apply the following code snippet.

```
df = app.analyzer.compute_metrics()
app.analyzer.compare_metrics(df)
```

The `compute_metrics` method is a wrapper for KarstNet. This method returns a pandas DataFrame storing the computed statistics for each simulation. The `compare_metrics` compare this DataFrame with the values computed on real networks. It highlights each value by coloring in green if the value for a simulated conduit is within the observed interval and in yellow if it is outside (Fig. 11).

5. Conclusions and discussion

This paper introduced the details of pyKasso, a Python open-source package for the generation of Discrete Karst conduit Networks (DKNs). It is based on an improved version of the Stochastic Karst conduit Simulation (SKS) algorithm (Borghi et al., 2012). As compared to previous implementations of SKS in 3D, the use of the anisotropic fast marching technique allows for directly generating the network while accounting for the presence of the phreatic and vadose zones in a

[1] https://github.com/karstnet/karstnet

	mean length	cv length	length entropy	tortuosity	orientation entropy	aspl	cpd	mean degree	cv degree	correlation vertex degree
0	73.325090	2.161135	0.172617	1.188073	0.926221	64.982313	0.426525	2.558352	0.287431	-0.458654
1	83.350260	2.071451	0.195627	1.183935	0.936492	60.685176	0.476157	2.548223	0.291998	-0.366845
2	112.241091	2.446064	0.170576	1.212199	0.971488	50.881359	0.457072	2.565657	0.303072	-0.370935
3	113.630698	2.410115	0.198984	1.178961	0.922808	51.505773	0.436575	2.535484	0.316596	-0.452190
4	99.945725	2.216915	0.260424	1.183249	0.937472	57.810042	0.410611	2.564417	0.298273	-0.366571
5	105.032556	2.327386	0.231935	1.191093	0.956700	61.155212	0.319264	2.552381	0.312594	-0.409848
6	97.381314	1.997789	0.271594	1.200758	0.945919	46.620201	0.445454	2.506667	0.303441	-0.310678
7	113.442757	2.255284	0.261312	1.250703	0.933291	53.324877	0.429322	2.503226	0.314486	-0.417596
8	111.053943	2.513841	0.179645	1.159893	0.943339	54.998630	0.414353	2.550459	0.307809	-0.466765
9	96.644635	2.323668	0.223345	1.144327	0.936863	49.188477	0.303406	2.538462	0.295795	-0.333193

Fig. 11. Output from the `compare_metrics` method.

karst aquifer. This significantly simplifies the implementation of the algorithm.

pyKasso has been designed to allow for easily modifying most of the parameters and importing data from various file formats to facilitate the user experience. Simple models can be constructed rapidly, and the sensitivity of pyKasso to the parameters can easily be tested. For more complex models, one can import components that are generated by many different other algorithms. For example, the paper has shown how a 3D geologic model constructed with GeoModeller can be imported, but it is also straightforward to instead use a model constructed with open source software such as GemPy (de la Varga et al., 2019) (for complex geologic situations) or ArchPy (Schorpp et al., 2022) (for simple situations), or any other commercial or open source geologic modeling tool. For the DFN simulations, pyKasso proposes a simple integrated tool but other more advanced packages such as DFNlab (https://fractorylab.org/dfnlab-software/) or DFNworks exist (Hyman et al., 2015) and can be employed to generate more realistic fracture networks if needed.

One of the limitations of pyKasso is that it has been designed to represent epigenic speleogenesis processes. Considering other types of pre-structuration of the network such as ghost-rocks or hypogene processes will require extending the pykasso framework. From a practical perspective, it means that some typical karstic structures such as deep sumps or highly labyrinthic networks may be difficult or impossible to simulate with the current version of the code.

Another, more technical, limitation is the use of a regular voxel grid to represent the 3D geology and its heterogeneity. This data structure has a fixed spatial resolution and cannot be refined in places where geometrical details may matter. This could be important in highly deformed and fractured areas. However, so far, this data structure has always been sufficient to represent the geologic situations that were needed. This method has been applied in numerous case studies and grid resolution has not been yet an issue. A related point is that the computing time may be large when the grid resolution is fine. Currently, the code is not compiled and not parallelized, this could be improved in the future if needed.

Finally, thanks to Python interoperability and its open-source philosophy, pyKasso is available on all major operating systems (Linux, MacOS, Windows). It can rather easily be extended and improved. Simulating variable conduit diameters along the DKN has still to be added. It should be straightforward using the method described by Frantz et al. (2021). Accounting for internal facies heterogeneity within some geological formations can also easily be done by having spatially variable costs correlated with a lithofacies model. PyKasso can also be used in conjunction with other software such as Modflow-CFP (Reimann et al., 2023), DisCo (de Rooij et al., 2013), or openKARST (Kordillaa et al., 2025) to simulate flow and transport in the karstic system or PEST (Doherty and Hunt, 2010) to identify model parameters in an inverse approach.

Data and code availability

Name of the software: pyKasso
Developer: François Miville, Chloé Fandel, Philippe Renard
Contact information: francois.miville@ikmail.com
Year first available: 2024 (for the 3D version), 2021 (for the 2D version)
Programing language: Python
Cost: free
License: GPL 3.0
Software availability: https://github.com/randlab/pykasso
Data for benchmarking: Available on the repository.
Size of Archive: 846 Mb (code + data)

CRediT authorship contribution statement

François Miville: Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization. **Philippe Renard:** Writing – review & editing, Writing – original draft, Supervision, Software, Project administration, Methodology, Funding acquisition, Conceptualization. **Chloé Fandel:** Writing – review & editing, Software, Methodology, Conceptualization. **Marco Filipponi:** Writing – review & editing, Supervision, Project administration, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work presented in this paper was funded by the National Cooperative for the Disposal of Radioactive Waste in Switzerland (NAGRA). P. Renard acknowledges funding by the European Union. His work is supported by ERC grant (KARST, 101071836). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Appendix

Tables 9 to 13 provide detailed explanations for the keywords and input parameters to control the definition of the modeling domain, the subdomains, the fast marching algorithm, the faults, and the fractures.

Data availability

The data and code are on github.

Table 9
Name, type and description of the key parameters used to define the modeling `domain`.

Parameter	Data type	Description
`delimitation`	(str, list)	Filename pointing to a list of points, or list of points modeling a polygon representing the horizontal delimitation of the model. The nodes outside the polygon will be not considered during the simulation. By default, the value is set to `None`.
`topography`	(str, np.ndarray)	Filename pointing to a 2D model, or 2D numpy array representing the digital elevation model. If a topographic surface is provided, the geologic model will be cut according to the surface values. All the nodes above the surface will be not considered during the simulation. By default, the value is set to `None`.
`bedrock`	(str, np.ndarray)	Filename pointing to a 2D model, or 2D numpy array representing the bedrock elevation model. If a bedrock surface is provided, the geologic model will be cut according to the surface values. All the nodes below the surface will be not considered during the simulation. By default, the value is set to `None`.
`water_level`	(str, np.ndarray)	Filename pointing to a 2D model, or 2D numpy array representing the water level elevation. If a piezometric surface is provided, the geologic model will consider a saturated and an unsaturated zone. All the nodes below the surface will be considered to belong to the saturated zone. By default, the value is set to `None`.

Table 10
Name, type and description of the key parameters used to define the subdomains.

Name	Description
domain	
`domain`	Cells located in the domain.
`domain_surface`	Cells located at the upper interface between the domain and the outside zone.
`domain_bottom`	Cells located at the lower interface between the domain and the outside zone.
borders	
`domain_borders`	Cells located at the interface between the domain and the outside zone.
`domain_borders_sides`	Cells located at the interface between the domain and the outside zone, but only in the x- and y-direction.
`domain_borders_surface`	Cells located at the upper interface between the domain and the outside zone, but only in the x- and y-direction.
`domain_borders_bottom`	Cells located at the lower interface between the domain and the outside zone, but only in the x- and y-direction.
vadose	
`vadose_zone`	Cells located in the vadose zone.
`vadose_borders`	Cells located in the vadose zone, at the interface between the vadose zone and the rest of the model, but only in the x- and y-direction.
phreatic	
`phreatic_zone`	Cells located in the phreatic zone.
`phreatic_surface`	Cells located in the phreatic zone, at the interface between the phreatic zone and the vadose zone.
`phreatic_borders_surface`	Cells located in the phreatic zone, at the interface between the phreatic zone and the vadose zone, as well as the outside zone.
bedrock	
`bedrock`	Cells located in the zone below the bedrock surface.
`bedrock_`	Cells located in the zone defined by the bedrock surface, with an additional two-cells layer in the upper z-direction.
`bedrock_vadose`	Two-cells layer located above the zone defined by the bedrock surface and intersected by the vadose zone.
`bedrock_phreatic`	Two-cells layer located above the zone defined by the bedrock surface and intersected by the phreatic zone.

Table 11
Key, associated default value, and description of the cost parameters controlling the fast marching algorithm.

Key	default value	Description
`out`	10	Travel cost of the cells outside the model boundaries, if defined.
`geology`	0.4	Travel cost of geologic model cells with no specified cost value.
`faults`	0.2	Travel cost of faults model cells with no specified cost value.
`fractures`	0.2	Travel cost of fracture model cells with no specified cost value.
`karst`	0.1	Travel cost of cells identified as containing a karstic conduit.
`ratio`	0.5	Ratio of cost applied during anisotropic fast marching defining the contrast in travel cost as a function of flow direction. It is defined as the travel cost parallel to gradient divided by the travel cost perpendicular to gradient. If the ratio equals 1, travel cost parallel and perpendicular to gradient are the same and describe an isotropic situation. If the ratio is lower than 1, travel cost is lower parallel to gradient and conduit paths will follow steepest gradient. If ratio is bigger than 1, travel cost is lower perpendicular to gradient and conduit paths will follow contours.

Table 12
Name, type and description of `faults` key parameters.

Parameter	Data type	Description
`data`	(str, np.ndarray)	Filename pointing to a 2D or 3D model, or a 2D or 3D numpy array representing the faults model. It can be a binary model representing all the faults, or a matrix where each fault is identified by an index. If the faults model is in 2D and the grid in 3D, it will be replicated in the direction definite by the `axis` parameter.
`names`	(dict)	Dictionary associating for each fault a name. By default, each identified fault is named `fault_n` with *n* the number identifying the fault.
`costs`	(dict)	Dictionary associating for each fault a specific cost for the fast marching algorithm. By default, each identified fault will get the value of the default travel cost set for faults.
`model`	(dict)	Dictionary associating for each fault a Boolean value. If the value is `True`, the specific fault will be taken in account during simulations, otherwise not. By default, the value is set to `True`.
`axis`	(str)	Direction in which the model should be replicated if it is in 2D. For this to work, the dimensions of the 2D array must match the corresponding side model dimensions. Valid values: `'x'`, `'y'`, `'z'`. By default, the value is set to `'z'`.

Table 13
Name, type and description of `fractures` key parameters.

Parameter	Data type	Description
`data`	(str, np.ndarray)	Filename pointing to a 2D or 3D model, or a 2D or 3D numpy array representing the fracture model. Each fracture family is identified by an index. If the fracture model is in 2D and the grid in 3D, it will be replicated in the direction defined by the `axis` parameter.
`names`	(dict)	Dictionary associating for each fracture family a name. By default, each identified fracture family is named `family_n` with *n* the number identifying the fracture family.
`costs`	(dict)	Dictionary associating for each fracture family a specific travel cost for the fast marching algorithm. By default, each identified unit will get the value of the default travel cost set for fractures.
`model`	(dict)	Dictionary associating for each fracture family a Boolean value. If the value is `True`, the specific fracture family will be taken in account during simulation, otherwise not. By default, the value is set to `True`.
`axis`	(str)	Direction in which the model should be replicated if it is in 2D. For this to work, the dimensions of the 2D array must match the corresponding side model dimensions. Valid values: `'x'`, `'y'`, `'z'`. By default, the value is set to `'z'`.
`generate`	(dict)	Dictionary containing the parameters to stochastically generate fractures family. Available parameters are described in Table 7.

References

Banusch, S., Somogyvári, M., Sauter, M., Renard, P., Engelhardt, I., 2022. Stochastic modeling approach to identify uncertainties of karst conduit networks in carbonate aquifers. Water Resour. Res. 58 (8), e2021WR031710.

Berglund, J.L., Toran, L., Herman, E.K., 2020. Can karst conduit models be calibrated? A dual approach using dye tracing and temperature. Groundwater 58 (6), 924–937.

Borghi, A., Renard, P., Jenni, S., 2012. A pseudo-genetic stochastic model to generate karstic networks. J. Hydrol. 414–415, 516–529. http://dx.doi.org/10.1016/j.jhydrol.2011.11.032.

Cacas, M.-C., Ledoux, E., de Marsily, G., Tillie, B., Barbreau, A., Durand, E., Feuga, B., Peaudecerf, P., 1990. Modeling fracture flow with a stochastic discrete fracture network: calibration and validation: 1. The flow model. Water Resour. Res. 26 (3), 479–489.

Calcagno, P., Chilès, J.-P., Courrioux, G., Guillen, A., 2008. Geological modelling from field data and geological knowledge: Part I. Modelling method coupling 3D potential-field interpolation and geological rules. Phys. Earth Planet. Inter. 171 (1–4), 147–157.

Collon, P., Bernasconi, D., Vuilleumier, C., Renard, P., 2017. Statistical metrics for the characterization of karst network geometry and topology. Geomorphology (ISSN: 0169-555X) 283, 122–142. http://dx.doi.org/10.1016/j.geomorph.2017.01.034.

Collon-Drouaillet, P., Henrion, V., Pellerin, J., 2012. An algorithm for 3D simulation of branchwork karst networks using Horton parameters and A⋆. Application to a synthetic case. Geol. Soc. Lond. Spec. Publ. 370 (1), 295–306.

Cooper, M.P., Covington, M.D., 2020. Modeling cave cross-section evolution including sediment transport and paragenesis. Earth Surf. Process. Landf. 45 (11), 2588–2602.

Cornaton, F., 2007. Groundwater: a 3-D Groundwater and Surface Water Flow, Mass Transport and Heat Transfer Finite Element Simulator, Reference Manual. Technical report, University of Neuchâtel, Neuchâtel, Switzerland.

Dall'Alba, V., Neven, A., de Rooij, R., Filipponi, M., Renard, P., 2023. Probabilistic estimation of tunnel inflow from a karstic conduit network. Eng. Geol. 312, 106950.

de la Varga, M., Schaaf, A., Wellmann, F., 2019. GemPy 1.0: open-source stochastic geological modeling and inversion. Geosci. Model. Dev. 12 (1), 1–32.

de Rooij, R., Graham, W., 2017. Generation of complex karstic conduit networks with a hydrochemical model. Water Resour. Res. 53 (8), 6993–7011. http://dx.doi.org/10.1002/2017wr020768.

de Rooij, R., Perrochet, P., Graham, W., 2013. From rainfall to spring discharge: Coupling conduit flow, subsurface matrix flow and surface flow in karst systems using a discrete–continuum model. Adv. Water Resour. 61, 29–41.

De Waele, J., Gutiérrez, F., 2022. Karst Hydrogeology, Geomorphology and Caves. John Wiley & Sons, p. 895.

Dershowitz, W.S., Herda, H.H., 1992. Interpretation of fracture spacing and intensity. In: ARMA US Rock Mechanics/Geomechanics Symposium. ARMA, pp. ARMA–92.

Diersch, H.-J.G., 2013. FEFLOW: Finite Element Modeling of Flow, Mass and Heat Transport in Porous and Fractured Media. Springer Science & Business Media.

Doherty, J.E., Hunt, R.J., 2010. Approaches to Highly Parameterized Inversion: a Guide to Using PEST for Groundwater-Model Calibration, vol. 2010, US Department of the Interior, US Geological Survey Reston, VA, USA.

Dreybrodt, W., Gabrovšek, F., Romanov, D., 2005. Processes of a Speleogenesis: A Modeling Approach, vol. 4, Založba ZRC.

Duran, L., Gill, L., 2021. Modeling spring flow of an irish karst catchment using Modflow-USG with CLN. J. Hydrol. 597, 125971.

Erhel, J., De Dreuzy, J.-R., Poirriez, B., 2009. Flow simulation in three-dimensional discrete fracture networks. SIAM J. Sci. Comput. 31 (4), 2688–2705.

Erzeybek, S., Srinivasan, S., Janson, X., 2012. Multiple-point statistics in a non-gridded domain: Application to karst/fracture network modeling. In: Geostatistics Oslo 2012. Springer, pp. 221–237.

Escobar, R.G., Roehl, D., Quadros, F.B., Cazarin, C.L., 2021. Stochastic modelling of karstic networks of Potiguar Basin, Brazil. Adv. Water Resour. 156, 104026.

Fandel, C., Ferré, T., Miville, F., Renard, P., Goldscheider, N., 2023. Improving understanding of groundwater flow in an alpine karst system by reconstructing its geologic history using conduit network model ensembles. Hydrol. Earth Syst. Sci. 27 (22), 4205–4215.

Fandel, C., Miville, F., Ferré, T., Goldscheider, N., Renard, P., 2022. The stochastic simulation of karst conduit network structure using anisotropic fast marching, and its application to a geologically complex alpine karst system. Hydrogeol. J. 30 (3), 927–946. http://dx.doi.org/10.1007/s10040-022-02464-x.

Fernandez-Ibanez, F., Moore, P., Jones, G., 2019. Quantitative assessment of karst pore volume in carbonate reservoirs using discrete karst networks. In: AAPG Search and Discovery (Abs.): AAPG Annual Convention and Exhibition.

Filipponi, M., Jeannin, P.-Y., Tacher, L., 2009. Evidence of inception horizons in karst conduit networks. Geomorphology 106 (1–2), 86–99.

Fischer, P., Jardani, A., Lecoq, N., 2016. Modeling of a karstic field by application of a cellular automata-based deterministic inversion (Lez Aquifer, France). In: AGU Fall Meeting 2016.

Ford, D., Williams, P., 2007. Karst Hydrogeology and Geomorphology. John Wiley and Sons, Ltd., Chichester, England, p. 562.

Frantz, Y., Collon, P., Renard, P., Viseur, S., 2021. Analysis and stochastic simulation of geometrical properties of conduits in karstic networks. Geomorphology 377, 107480.

Giese, M., 2017. Identification and Quantification of the Effects of Flow Regime and Matrix-Conduit Interaction in the Characterization of Karst Aquifers (Ph.D. thesis). Georg-August-Universität Göttingen.

Gouy, A., Collon, P., Bailly-Comte, V., Galin, E., Antoine, C., Thebault, B., Landrein, P., 2024. KarstNSim: A graph-based method for 3D geologically-driven simulation of karst networks. J. Hydrol. 130878.

Gouy, A., Collon, P., Bailly-Comte, V., Landrein, P., 2022. Discrete karst network simulations: application to the Barrois limestones. In: IAMG 21st Annual Conference.

Gremaud, V., Goldscheider, N., 2010. Geometry and drainage of a retreating glacier overlying and recharging a karst aquifer,Tsanfleuron-Sanetsch, Swiss Alps. Acta Carsologica 39 (2).

Gremaud, V., Goldscheider, N., Savoy, L., Favre, G., Masson, H., 2009. Geological structure, recharge processes and underground drainage of a glacierised karst aquifer system, Tsanfleuron-Sanetsch, Swiss Alps. Hydrogeol. J. 17 (8), 1833–1848.

Hagberg, A.A., Schult, D.A., Swart, P.J., 2008. Exploring network structure, dynamics, and function using networkx. In: Varoquaux, G., Vaught, T., Millman, J. (Eds.), Proceedings of the 7th Python in Science Conference. Pasadena, CA USA, pp. 11–15.

Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E., 2020. Array programming with NumPy. Nature 585 (7825), 357–362. http://dx.doi.org/10.1038/s41586-020-2649-2.

Hartmann, A., Goldscheider, N., Wagener, T., Lange, J., Weiler, M., 2014. Karst water resources in a changing world: Review of hydrological modeling approaches. Rev. Geophys. 52 (3), 218–242. http://dx.doi.org/10.1002/2013rg000443.

Hendrick, M., Renard, P., 2016. Fractal dimension, walk dimension and conductivity exponent of karst networks around tulum. Front. Phys. 4, http://dx.doi.org/10.3389/fphy.2016.00027.

Hunter, J.D., 2007. Matplotlib: A 2D graphics environment. Comput. Sci. Eng. 9 (3), 90–95. http://dx.doi.org/10.1109/MCSE.2007.55.

Hyman, J.D., Karra, S., Makedonska, N., Gable, C.W., Painter, S.L., Viswanathan, H.S., 2015. dfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. Comput. Geosci. 84, 10–19.

Jaquet, O., Siegel, P., Klubertanz, G., Benabderrhamane, H., 2004. Stochastic discrete model of karstic networks. Adv. Water Resour. 27 (7), 751–760. http://dx.doi.org/10.1016/j.advwatres.2004.03.007.

Jeannin, P.-Y., Eichenberger, U., Sinreich, M., Vouillamoz, J., Malard, A., Weber, E., 2013. KARSYS: a pragmatic approach to karst hydrogeological system conceptualisation. Assessment of groundwater reserves and resources in Switzerland. Environ. Earth Sci. 69, 999–1013.

Jouves, J., Viseur, S., Arfib, B., Baudement, C., Camus, H., Collon, P., Guglielmi, Y., 2017. Speleogenesis, geometry, and topology of caves: A quantitative study of 3D karst conduits. Geomorphology 298, 86–106.

Kanfar, R., Mukerji, T., 2023. Stochastic geomodeling of karst morphology by dynamic graph dissolution. Math. Geosci. 1–25.

Kordillaa, J., Dentz, M., Hidalgo, J.J., 2025. openKARST: A novel open-source flow simulator for karst systems. Comput. Geosci..

Kovács, A., Sauter, M., 2007. Modelling karst hydrodynamics, in: methods in karst hydrogeology. In: Goldscheider, N., Drew, D. (Eds.), Methods in Karst Hydrogeology. Taylor and Francis London, pp. 201–222.

Kresic, N., Panday, S., 2018. Numerical groundwater modelling in karst. Geol. Soc. Lond. Spec. Publ. 466 (1), 319–330.

Le Coz, M., Bodin, J., Renard, P., 2017. On the use of multiple-point statistics to improve groundwater flow modeling in karst aquifers: A case study from the hydrogeological experimental site of Poitiers, France. J. Hydrol. 545, 109–119. http://dx.doi.org/10.1016/j.jhydrol.2016.12.010.

Luo, L., Liang, X., Ma, B., Zhou, H., 2021. A karst networks generation model based on the anisotropic fast marching algorithm. J. Hydrol. (ISSN: 0022-1694) 600, 126507. http://dx.doi.org/10.1016/j.jhydrol.2021.126507.

Malard, A., Jeannin, P.-Y., Vouillamoz, J., Weber, E., 2015. An integrated approach for catchment delineation and conduit-network modeling in karst aquifers: application to a site in the Swiss tabular Jura. Hydrogeol. J. 7 (23), 1341–1357.

Measday, A., 2021. Modeling of the Tsanfleuron-Glarey Karst System (Master's thesis). Université de Neuchâtel, Switzerland.

Mirebeau, J.-M., Portegies, J., 2019. Hamiltonian fast marching: A numerical solver for anisotropic and non-holonomic eikonal PDEs. Image Process. Line 9, 47–93. http://dx.doi.org/10.5201/ipol.2019.227.

Neven, A., Dallalba, V., Juda, P., Straubhaar, J., Renard, P., 2021. Ice volume and basal topography estimation using geostatistical methods and ground-penetrating radar measurements: application to the Tsanfleuron and Scex Rouge glaciers, Swiss Alps. Cryosphere 15 (11), 5169–5186.

pandas development team, T., 2020. pandas-dev/pandas: Pandas. http://dx.doi.org/10.5281/zenodo.3509134.

Pardo-Igúzquiza, E., Dowd, P.A., Xu, C., Durán-Valsero, J.J., 2012. Stochastic simulation of karst conduit networks. Adv. Water Resour. 35, 141–150.

Paris, A., Guérin, E., Peytavie, A., Collon, P., Galin, E., 2021. Synthesizing geologically coherent cave networks. Comput. Graph. Forum 40 (7), 277–287.

Reimann, T., Hill, M.E., 2009. MODFLOW-CFP: A new conduit flow process for MODFLOW–2005. Groundwater 47 (3), 321–325.

Reimann, T., Rudolph, M.G., Grabow, L., Noffz, T., 2023. CFPy—A Python Package for Pre-and Postprocessing of the Conduit Flow Process of MODFLOW. Groundwater 61 (6), 887–894.

Ronayne, M.J., 2013. Influence of conduit network geometry on solute transport in karst aquifers with a permeable matrix. Adv. Water Resour. 56, 27–34.

Ronayne, M.J., Gorelick, S.M., 2006. Effective permeability of porous media containing branching channel networks. Phys. Rev. E 73 (2), http://dx.doi.org/10.1103/physreve.73.026305.

Rongier, G., Collon-Drouaillet, P., Filipponi, M., 2014. Simulation of 3D karst conduits with an object-distance based method integrating geological knowledge. Geomorphology 217, 152–164.

Schiller, A., Renard, P., 2016. An optical laser device for mapping 3D geometry of underwater karst structures: first tests in the Ox Bel'Ha system, Yucatan, Mexico. Boletín Geológico y Min. 127 (1), 99–110.

Schoeneich, P., Reynard, E., 2021. Structural landscapes and relative landforms of the diablerets massif. In: Reynard, E. (Ed.), Landscapes and Landforms of Switzerland. Springer, pp. 123–141.

Schorpp, L., Straubhaar, J., Renard, P., 2022. Automated Hierarchical 3D Modeling of Quaternary Aquifers: The ArchPy Approach. Front. Earth Sci. 10, 884075.

Sethian, J.A., 1996. A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. 93 (4), 1591–1595. http://dx.doi.org/10.1073/pnas.93.4.1591.

Sethian, J., 2001. Evolution, implementation, and application of level set and fast marching methods for advancing fronts. J. Comput. Phys. 169 (2), 503–555. http://dx.doi.org/10.1006/jcph.2000.6657.

Shoemaker, W.B., Kuniansky, E.L., Birk, S., Bauer, S., Swain, E.D., 2008. Documentation of a Conduit Flow Process (CFP) for MODFLOW-2005, vol. 6, US Department of the Interior, US Geological Survey Reston, Va.

Sivelle, V., Renard, P., Labat, D., 2020. Coupling SKS and SWMM to solve the inverse problem based on artificial tracer tests in karstic aquifers. Water 12 (4), 1139.

Sullivan, C.B., Kaszynski, A., 2019. PyVista: 3D plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK). J. Open Source Softw. 4 (37), 1450. http://dx.doi.org/10.21105/joss.01450.

Viseur, S., Jouves, J., Fournillon, A., Arfib, B., Guglielmi, Y., 2015. 3D stochastic simulation of caves: application to Saint-Sébastien case study (SE, France). Karstologia 64, 17–24.

Vuilleumier, C., Borghi, A., Renard, P., Ottowitz, D., Schiller, A., Supper, R., Cornaton, F., 2013. A method for the stochastic modeling of karstic systems accounting for geophysical data: an example of application in the region of Tulum, Yucatan Peninsula (Mexico). Hydrogeol. J. 21 (9), 523–544.

Vuilleumier, C., Jeannin, P.-Y., Perrochet, P., 2019. Physics-based fine-scale numerical model of a karst system (Milandre Cave, Switzerland). Hydrogeol. J. 27 (7), 2347–2363.

Worthington, S.R.H., 2009. Diagnostic hydrogeologic characteristics of a karst aquifer (Kentucky, USA). Hydrogeol. J. 17 (7), 1665.