
Hy2roresO Documentation

Release latest

Jan 18, 2019

1	Introduction	3
1.1	Strahler stream order	3
1.2	Shreve stream order	4
1.3	Horton stream order	4
2	Getting Started	7
2.1	A quick introduction	7
2.2	Installing the plugin	7
3	How to...?	9
3.1	Preparation of the process	9
3.2	Step 1 : essential parameters	9
3.3	Step 2 : optional parameters	10
3.4	Step 3 : process and output	11
3.5	End of the algorithm	12
4	Results	13
4.1	Simple network	13
4.2	Single island	15
4.3	Complex island	17
4.4	Whole network	19
4.5	Computation of the strokes	19
4.6	Comparison between other existing plugins for stream order computation	19
5	Approach & Strategy	21
5.1	General strategy	21
5.2	Input data	21
5.3	The algorithm	22
5.4	Update of the attribute table of the input layer	30
6	Documentation	31
6.1	Classes	31
6.2	Instanciations of the classes	32
6.3	Correct edges directions	33
6.4	Sources and sinks	34
6.5	Island detection	35
6.6	Orders	36

6.7	Write in table	39
6.8	Dialog messages	40
6.9	Save output	40
7	Perspectives	41
8	Team	43
9	Story	45
10	Acknowledgment	47
11	Contribute	49
	Bibliography	51

Hy2roresO is a QGIS plugin able to compute various hierarchisation stream orders, such as Strahler order, Horton order or Shrieve order, from an input shapefile.

The code is open source, and [available on GitHub](#).

The main documentation for the site is organized into a couple sections:

- *User Documentation*
- *Developer Documentation*

Information about development is also available:

- *About Hy2roresO*

The classification of a hydrographic network is a way of ranking all the branches of this network by assigning to each a whole value that characterizes its importance. Several different classifications have been developed, Strahler's classification in particular is very commonly used. Shreve and Horton's classification are also classifications that are frequently used. These three classifications are those calculated by processing Hy2roresO.

1.1 Strahler stream order

1.1.1 Presentation

The Strahler number is a numerical measure of a tree or a network's branching complexity. This number was first developed in hydrology by Robert E. Horton (1945) and Arthur Newell Strahler (1952, 1957). In this application, they are used to define stream size based on a hierarchy of tributaries.

In the application of the Strahler stream order to hydrology, each segment of a stream or river within a river network is treated as a node in a tree, with the next segment downstream as its parent. When two first-order streams come together, they form a second-order stream. When two second-order streams come together, they form a third-order stream. Streams of lower order joining a higher order stream do not change the order of the higher stream. Thus, if a first-order stream joins a second-order stream, it remains a second-order stream. It is not until a second-order stream combines with another second-order stream that it becomes a third-order stream.

The Strahler stream order of a sink is the highest one of the river, and does usually not exceed 10.

1.1.2 Hypotheses

Some hypotheses were made during the computing of the plugin. For more details about these hypotheses, please refer to the [Developer's Documentation](#).



Fig. 1: Example of application of the **Strahler stream order** on a network

1.2 Shreve stream order

1.2.1 Presentation

The Shreve stream order is another order used in hydrology with the similar aim of defining the stream size of a hydrologic network.

The Shreve system also gives the outermost tributaries the number “1”. At a confluence the numbers are added together, contrary to the Strahler stream order. This means that the Shreve stream order of a sink can be very high.

Shreve stream order is preferred in hydrodynamics: it sums the number of sources in each catchment above a stream gauge or outflow, and correlates roughly to the discharge volumes and pollution levels. Like the Strahler method, it is dependent on the precision of the sources included, but less dependent on map scale. It can be made relatively scale-independent by using suitable normalisation and is then largely independent of an exact knowledge of the upper and lower courses of an area.

1.2.2 Hypotheses

Some hypotheses were made during the computing of the plugin. For more details about these hypotheses, please refer to the [Developer’s Documentation](#).

1.3 Horton stream order

1.3.1 Presentation

The Horton stream order is the third most commonly used stream order in hydrology. It is based on a different idea which takes into account the strokes and the Strahler stream order. Horton’s stream order applies to the stream as a whole and not according to the edges of the network. The first step of its process is to define the strokes, which are sets of arcs that appear to be continuous (as in a straight line or curve). The second step is to define the Horton stream order for each stroke according to the strokes and the Strahler stream order. The Horton order of a stream is the maximum value of the Strahler order of the stroke the stream belongs to. The main stroke is first to be set its Horton stream order

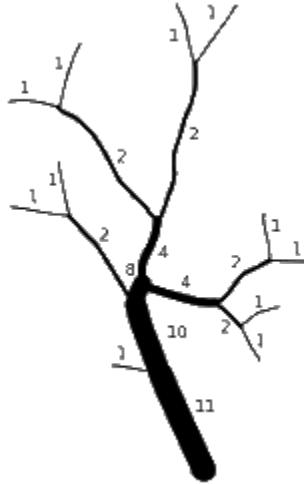


Fig. 2: Example of application of the **Shreve stream order** on a network

which corresponds to its Strahler stream order, and then each stroke is given a Horton stream order corresponding to its Strahler order.

1.3.2 Hypotheses

Some hypotheses were made during the computing of the plugin. For more details about these hypotheses, please refer to the [Developer's Documentation](#).

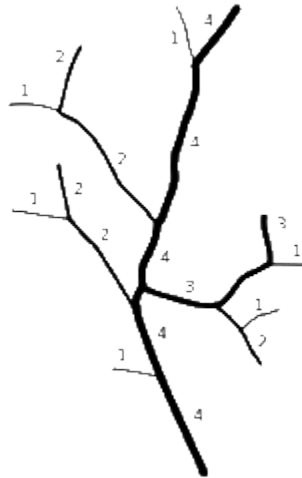


Fig. 3: Example of application of the **Horton stream order** on a network

This page will explain you how to get up and running with **Hy2roresO**.

If you already have installed the plugin, skip ahead to [the user guide of the plugin](#).

2.1 A quick introduction

This is a quick introduction about our plugin. Please refer to another page for more details.

2.2 Installing the plugin

2.2.1 From the repository

If you would like to get the latest code of the plugin, you can install it directly from the repository. The repository is in Github here. There are 2 ways that you can do generally:

- Download the zip from github here: [ZIP Master](#), extract the zip, and copy the extracted root directory into QGIS local plugins directory (on Linux it's `~/.local/share/QGIS/QGIS3/profiles/default/python/plugins` where `~` is `/home/USER`, on Windows it's `C:\Users\{username}\AppData\Roaming\QGIS\QGIS3/profiles/default/python/plugins`)
- Use git: clone the repository in that directory or clone in your preferred location and use symbolic link in local plugins directory.

2.2.2 From QGIS Plugin Manager

Unfortunately, you won't be able to download the plugin from the QGIS Plugin Manager.

The page [Contribute](#) has more information on getting in touch.


3.1 Preparation of the process

To use Hy2roresO properly, make sure you have opened the vector layer corresponding to the network you want to analyse. Your layer must not contain artificial networks (such as irrigation zones), and must not have duplicated geometries, for the algorithm to run properly. If you have duplicated geometries in your layer, please refer to the [Documentation](#) from QGIS to eliminate them.

The output of the plugin is the input layer with new fields corresponding to the different orders.

The fields will be named like the orders : “strahler”, “shreve” and “horton”. Fields “reversed” and “id_stroke” may also be added. Before running the plugin, **please make sure that the table of attributes of the input layer does not already have fields named like so**. They would be overwritten.

To open the Hy2roresO plugin, go to the *Extension* menu, then find Hy2roresO and open it. You can also find it thanks

to its icon : 

Note: You do not need to be in editing mode to use Hy2roresO, it occurs automatically by processing.

3.2 Step 1 : essential parameters

You will find yourself in front of a window :

The first parameters you must enter are :

- the layer on which you want to apply the algorithm
- the stream orders you want to get thanks to the plugin: Strahler, Shreve and Horton (you can see the description of each of these orders [here](#))

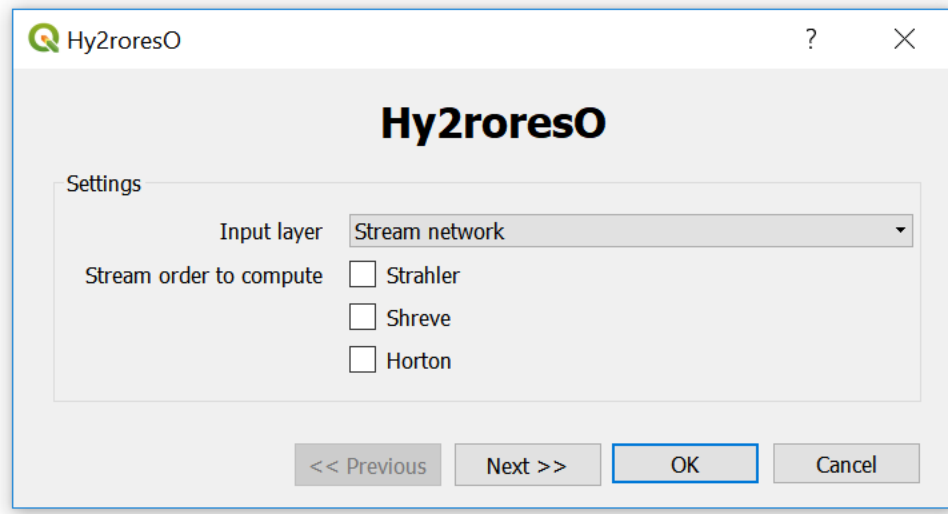


Fig. 1: First window of the plugin

Click on **Next** to go to the next parameters.

3.3 Step 2 : optional parameters

You can now enter more optional parameters to specify the names of the fields corresponding to the **name of the river**, the **initial altitude** and the **final altitude** of each section of the network.

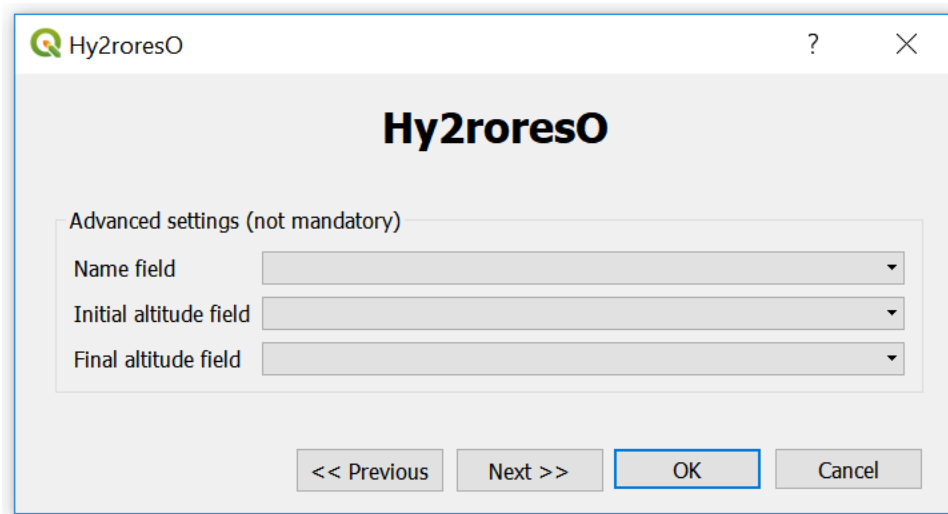


Fig. 2: Second window of the plugin

These parameters are optional : if they are not specified, the algorithm will still run, but may be less efficient because these parameters can be a key for a better hierarchisation.

Click on **Next** to get to the next step.

3.4 Step 3 : process and output

On this window you can :

- authorize the algorithm to reverse streams that may not be entered well and therefore cause some mistakes in the attribution of the orders; reversed streams are reversed for the computation but the input layer remains unchanged (checked initially)
- add a boolean field *reversed* to the layer you are applying the algorithm on, it indicates whether the streams were reversed during the algorithm (checked initially)
- save the output layer, and choose the path where you want to save this layer

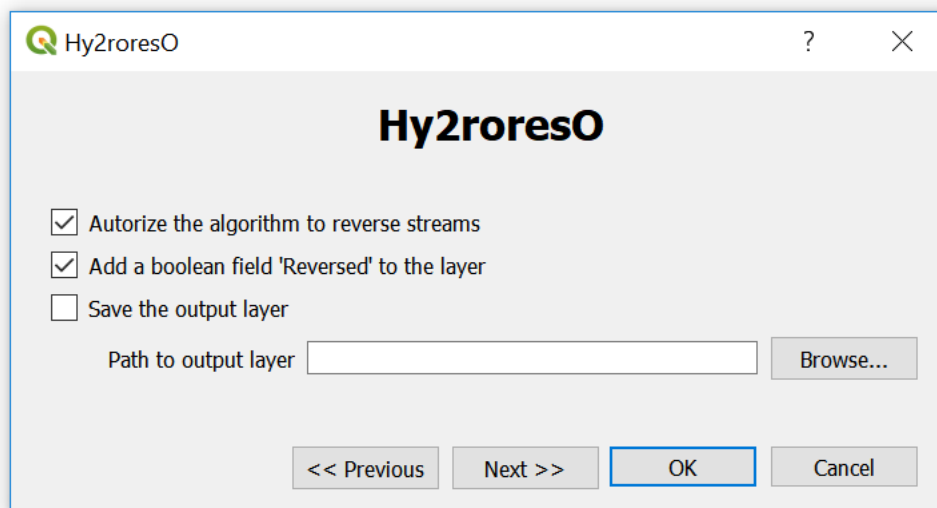


Fig. 3: Third window of the plugin

Once you have finished, you can click on **Next** for more information about the elaboration of the plugin.

You can finally click on **OK** to run the algorithm.

Note : You can click on OK right from the start once you have selected a layer and at least one order to calculate. The algorithm will run with the default settings. You can also come back to the previous step whenever you want by clicking on Previous.

3.4.1 During the algorithm

During the process of the algorithm, if you have chosen to authorize the algorithm to reverse some streams, you may find this type of window :

Streams that are suspected to be uncorrect are streams connected to a node that has several incoming edges but no outgoing edge, or several outgoing edges but no incoming edge; or streams whose initial altitude is lower than final altitude (if altitude fields are known).

You can reverse the feature which is being processed or not. You can also ask to reverse them all or to let them all at their initial state, knowing the number of streams that could be reversed.

Note : The algorithm does not modify the layer in itself by reversing some features, it is only for its good process!

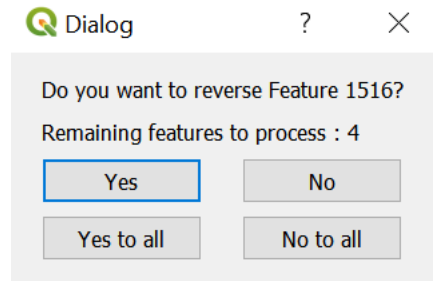


Fig. 4: Window asking if the user wants to reverse a feature

3.5 End of the algorithm

The algorithm is finished when you meet this final window :

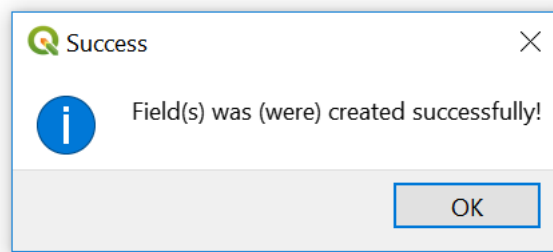


Fig. 5: Final window

Note: Do not panic if QGIS *does not respond* during the process, since two steps are particularly long : the island detection and the update of the layer with the writing of the new fields. The plugin is still running.

Hy2roresO handles a multitude of configurations in hydrological networks, which is a novelty comparing to former plugins from GRASS or QGIS which were less precise or even wrong at times.

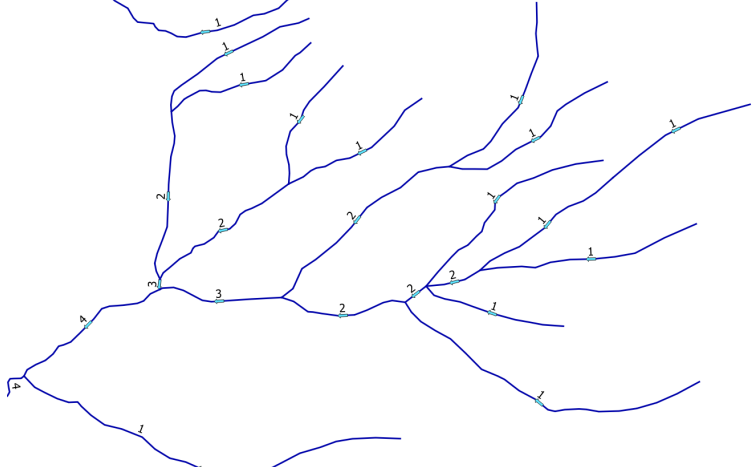
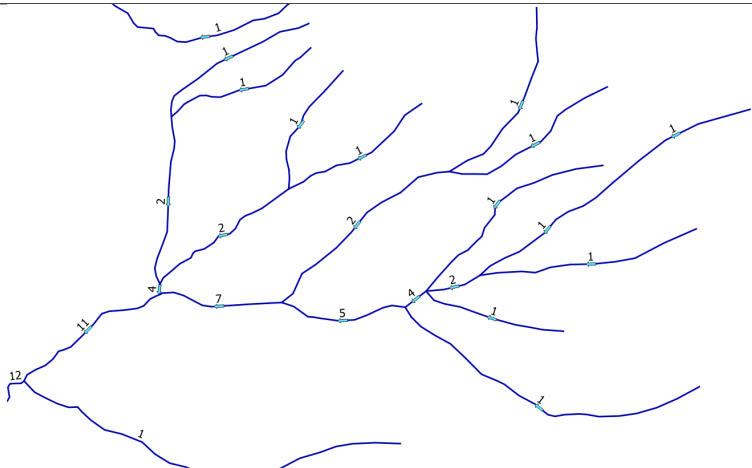
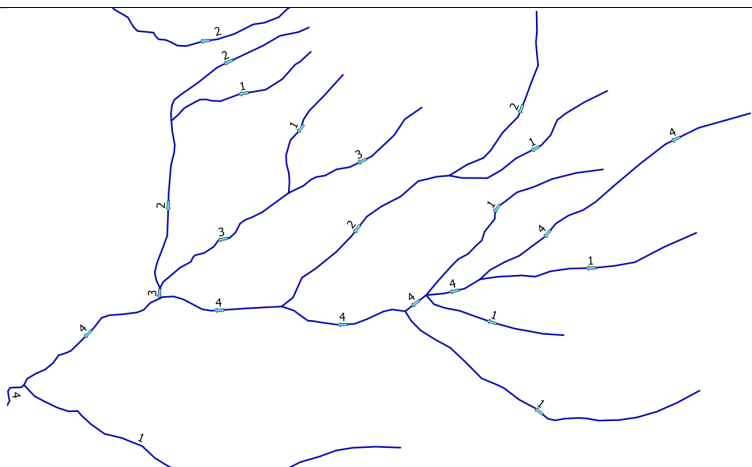
To begin with, the “Strahler” QGIS plugin, as its name recalls, handles only the Strahler stream orders of a network by selecting its sink, with several mistakes when processing in specific configurations (complex river networks). The GRASS plugin (`v.stream.order`) handles the Strahler and Shreve stream orders, however the results are not reliable when the process is applied on complex river networks (with islands in particular). There is also a section in the code intended to handle the Horton stream order, but it was not implemented.

Hy2roresO gives the input layer Strahler, Shreve and Horton stream orders and also takes into account the incorrect directions of the edges that the algorithm suspects to be erroneous and that the user confirms as wrong. It also proposes a strokes calculation of the stream network during Horton stream order computation. These two considerations are a major improvement in comparison to what have been available so far.

Let’s have a look at the results Hy2roresO creates with a few examples of various hydrological network configurations.

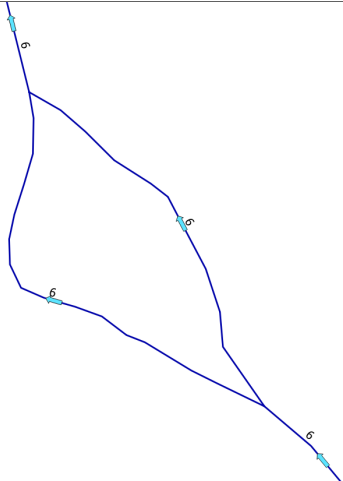
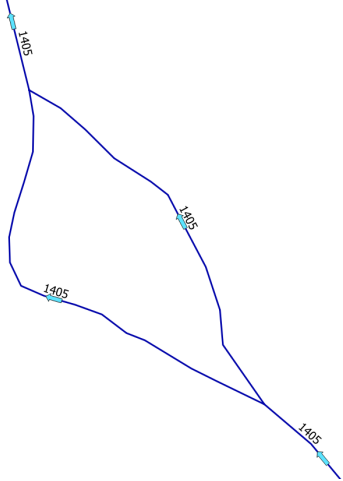
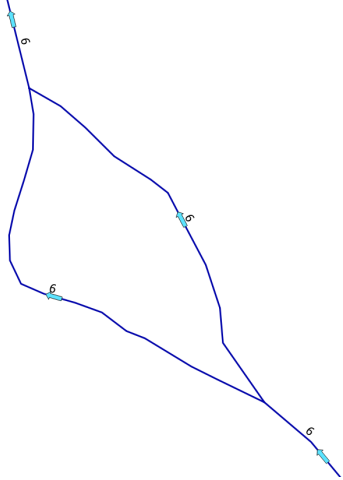
4.1 Simple network

In former plugins processing the different orders at stake, this configuration has never been a problem, and Hy2roresO also delivers a right result for these simple networks.

	<p>Strahler stream order on a simple network</p>
	<p>Shreve stream order on a simple network</p>
	<p>Horton stream order on a simple network</p>

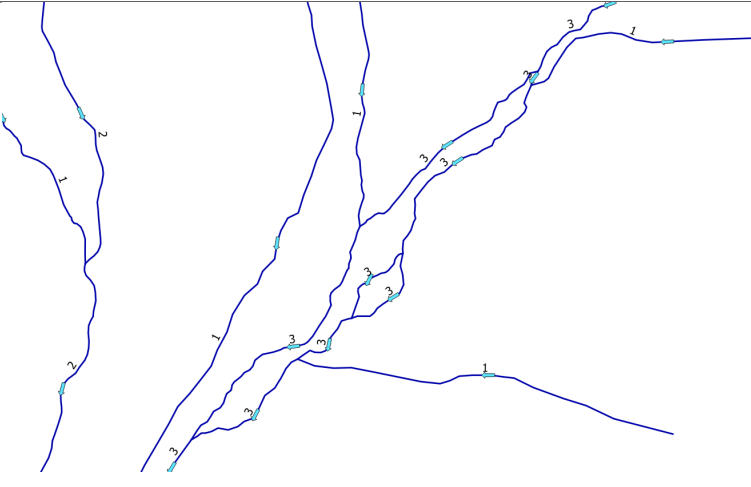
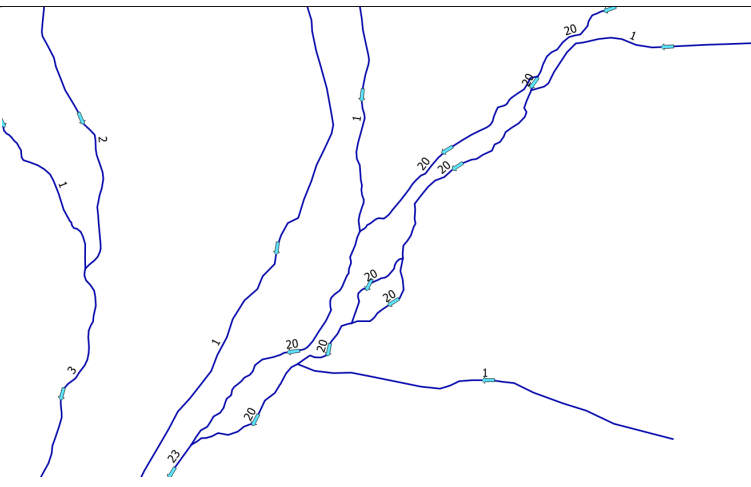

4.2 Single island

Hy2roresO handles a single island so as its outgoing edge has the same order as its incoming edge, which was not the case in former plugins. This way, the orders do not increment dramatically each time an island is met by the algorithm.

	<p>Strahler stream order on a single island</p>
	<p>Shreve stream order on a single island</p>
	<p>Horton stream order on a single island</p>

4.3 Complex island

A succession of adjacent islands is what we call a complex island. When processing previous plugins, this part has always been an issue. With Hy2roresO, it is handled correctly so as the orders do not increase dramatically and remain right passed the island.

	<p>Strahler stream order on a complex island</p>
	<p>Shreve stream order on a complex island</p>
	<p>Horton stream order on a complex island</p>

4.4 Whole network

Hy2roresO handles every edge of the network, while previous plugins struggled managing to process them all.

4.5 Computation of the strokes

Unlike the two other plugins, Hy2roresO computes the strokes, which allows to compute the Horton stream order on networks too. Thanks to this order, one can see the main branches of a network, as shown in this result obtained with Hy2roresO :



Fig. 1: Application of the Horton stream order on a network. The bolder the stroke, the higher the Horton order is.

4.6 Comparison between other existing plugins for stream order computation

There exist other plugins, on QGIS 2.18 and on GRASS that also compute stream orders. However, there are many differences between the three plugins. Here is a [PDF file](#) comparing them all.

5.1 General strategy

Hy2roresO is a QGIS plugin developed in Python 3.6 for QGIS 3.0. The implemented algorithm is iterative. The algorithm goes through the river network starting from the sources and down to the sinks. The orders computation relies on instances of classes specifically designed for the plugin. They will be detailed further in the documentation.

This section aims to present the main hypotheses we were led to make to enable the orders algorithm to work on complex networks that have singular configurations such as islands, when the theoretical algorithms of all three orders expect a network shaped as a binary tree (*see the [Introduction of the User documentation about the Strahler, Shreve and Horton algorithms](#)*). However, such networks are not the river structures that exist in reality. The goal of the hypotheses made for the implementation of our algorithm is to adapt the general spirit of each order algorithm defined only for binary trees to the more complex reality.

5.2 Input data

The data and chosen options the plugin needs as input are the orders to compute and the layer of the river network. The network layer must be a linear vectorial layer. The network **should not contain artificial zones** such as irrigation zones, since such configurations are not specifically handled by the algorithm and the result may be meaningless. Forks of streams that do not occur in an island (two or more streams exiting an island), immediately at the sources (multiple sources exiting a single node) or at the sinks of the network (deltas) may also introduce mistakes in the strokes computation, that might spread into the downstream network calculation afterwards (see the upstream length criteria for strokes computation below).

Be aware that **the layer must also not contain duplicated geometries**. Duplicated geometries are hard to notice since you cannot see them simply by displaying the layer, and they significantly alter the orders computation. Duplicated geometries are processed as two streams connected to the same nodes, but they do not make up an island (islands have a non-zero area). Thus duplicated geometries artificially increase the Strahler and Shreve orders at each node, completely distorting the results.

5.3 The algorithm

5.3.1 Classes

Instead of working directly on the features of the layer or using an external library, the plugin implements three classes instantiated using the layer features that make travelling through the network easy (and independent from external parts).

The algorithm sets 3 classes for each geometry type: *Edge* (lines of the networks), *Node* (connecting points) and *Island* (edges delimiting a surface, face of the network). Their attributes register their interconnection in the layer network.

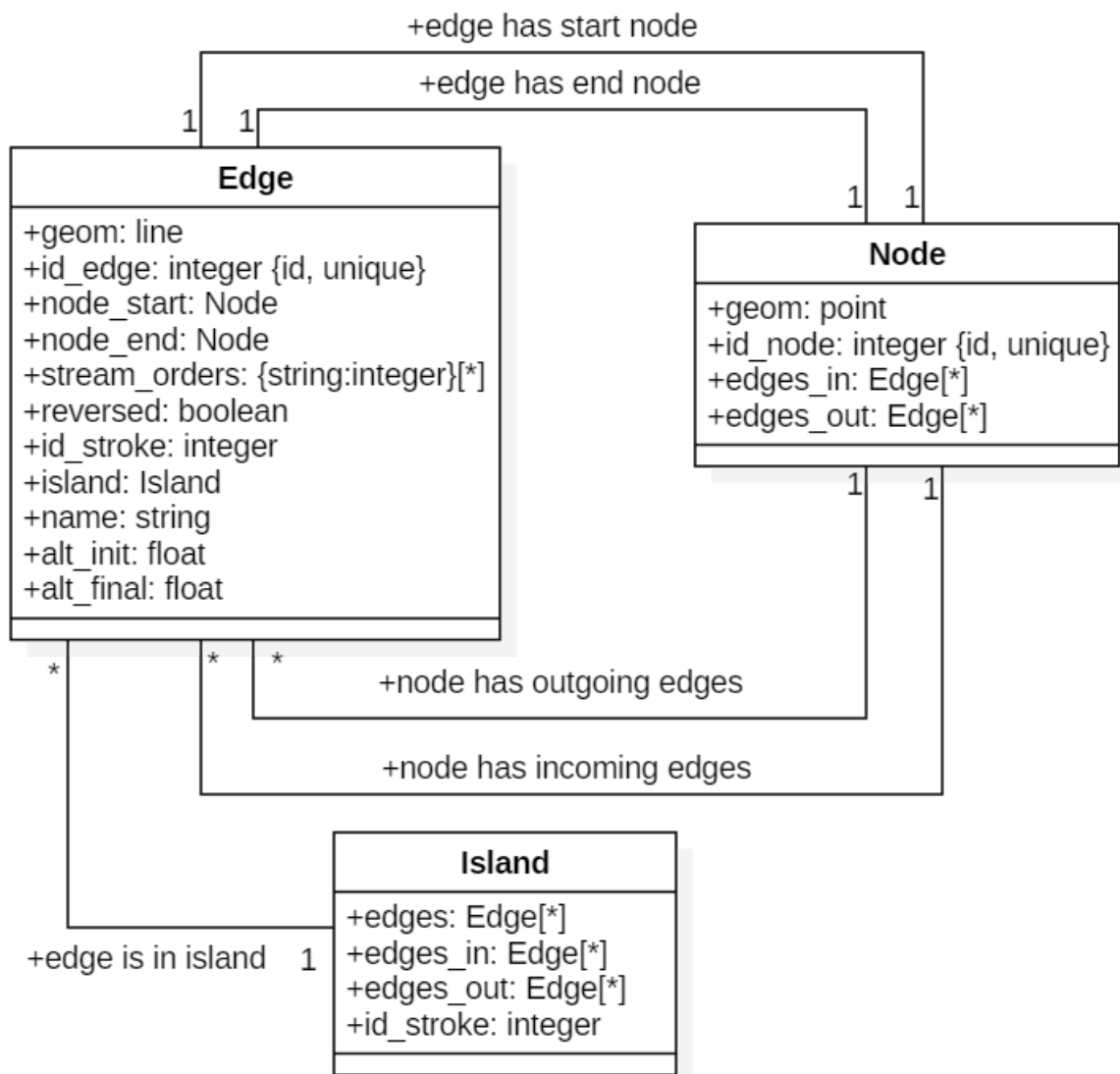


Fig. 1: Class diagram of the algorithm; the methods are not listed here

5.3.2 Initialization

At the beginning of the process, a method initializes each feature of the layer as an edge and its initial and final nodes, with all their attributes.

The objects instantiated are stored in two lists that are passed as arguments to all the other methods.

5.3.3 Correct stream direction

Methods were implemented to check and correct the stream direction. If the option in the launcher interface is checked, directions are tested.

The checking method is called if the checkbox in the plugin, that asks the user if he wants to be proposed some streams to reverse, is checked.

A stream direction is suspected to be incorrect if it meets one of the two following **criteria**:

- The altitudes are known (fields name filled by the user in the launcher) and **the initial altitude is lower than the final altitude** of the edge;
- A connected node is not a source or a sink (degree larger than 1) and has **only incoming edges or only outgoing edges**.

The suspicious edges are displayed thanks to a dialog box to the user, who can choose for each edge if they want to reverse it or not by knowing the direction of connected streams and the structure of the network. Obviously, amongst all the suspicious edges, only the edges approved by the user will be reversed for the orders computation.

If the edge is reversed, the information is stored as a boolean attribute of the object and a field can be added to the input layer to restore this information to the user (option in the launcher).

Note: Changing the direction of the stream will not change the geometry of the feature of the input layer. It will only change the attributes of the object instantiated from the layer feature, that only exist within the plugin.

5.3.4 Sources and sinks detection

The plugin detects the sources and sinks of the network. The user does not have to indicate them to the algorithm.

A source is a node that has no incoming edges. The outgoing edges of the sources are stored into a list that is passed as an argument to the method which implements orders computation. They initialize the iterative process of orders computation.

A sink is a node that has no outgoing edges. Their detection is not useful to the Hy2roresO algorithm.

Note: It is important that directions are corrected before this step, as missing a source will affect the whole branch connected to the source edge.

5.3.5 Island detection

Islands are the most frequent structures a real network may have that differ from and that will alter the orders. We call an island the structure induced by the split of a stream into two or more arms that join back downstream. If the regular algorithm is systematically applied as if the network was a binary tree, the streams that meet again at the end of the island will increase the order. This is an unwanted effect, as this increase is meaningless. It does not relate an upgrade in the hierarchy or a flow increase: no affluent actually meets the stream, the stream meets itself. Therefore, the order

should be the same as the upstream order. Thus islands need to be identified, or more accurately edges that delimit islands need to be identified, so that two edges that are actually part of an island do not induce an increase of the order when they meet. The regular algorithms do not apply to edges that belong to islands.

Note: All three orders under study are affected by islands, as Strahler and Shreve orders increase when rivers cross and Horton is based on the value of the Strahler order.

A great improvement proposed by Hy2roresO in comparison to plugins existing so far is the detection of islands, that enables specific process.

The edges that belong to islands are detected as such by the algorithm, and will be processed differently from the other edges when computing their orders.

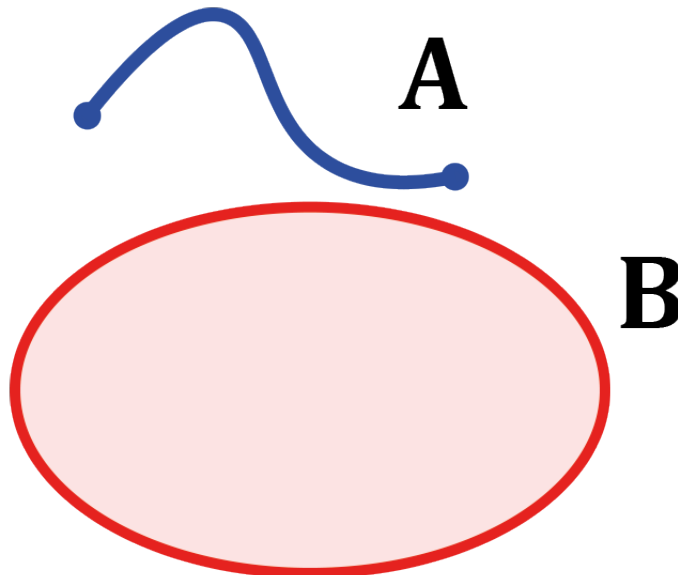
An island is a face of the network. The steps of island detections are the following:

- Polygonize the network (create the polygons that correspond to the faces of the graph). We re-used the code of the *Polygonize* QGIS tool found in the toolbox.

Note: Let's underline that underground features are not differentiated from features on other levels, and thus might induce faces that are not islands in reality. Once again, be aware of man-made structures in the network.

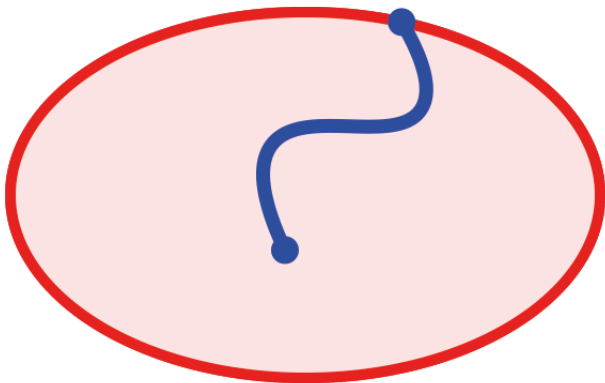
Single islands (one face of the graph) or complex islands (a succession of adjacent faces) can be processed similarly. Therefore edges are identified as belonging to one common island whether they delimit a single island or they belong to a complex island. Hence the following steps:

- Merge the polygons to transform adjacent single islands into one complex island (one bigger polygon).
- Detect the edges that belong to the islands. For this step we studied the topological relations between the edges and the islands. We defined our own topological request using a QGIS method *relate()* and DE-9IM matrices.

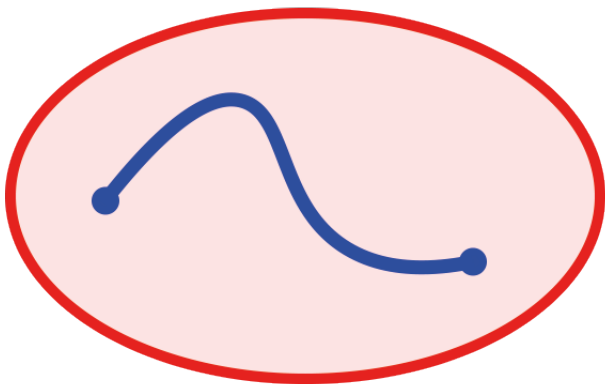


Then:

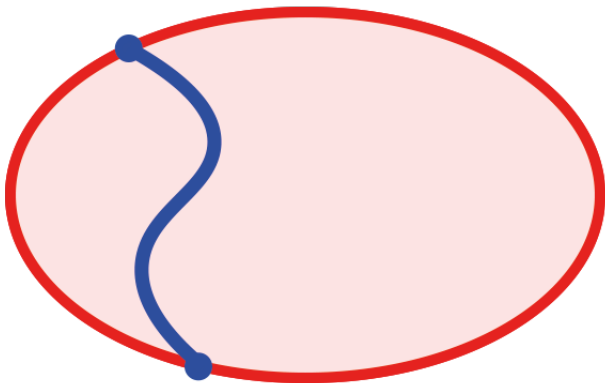
- Store the edges in a list of lists of the edges of each island.



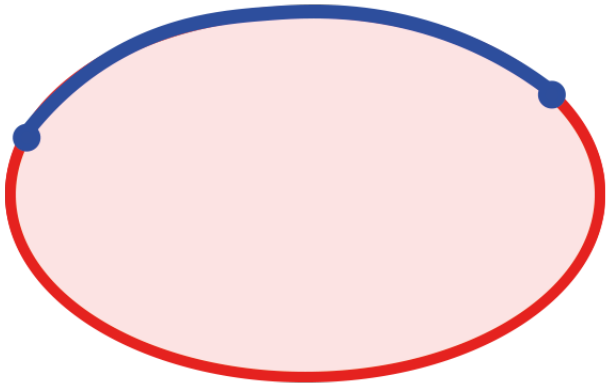
	Int(B)	Front(B)	Ext(B)
Int(A)	1	F	F
Front(A)	0	0	F
Ext(A)	2	1	2



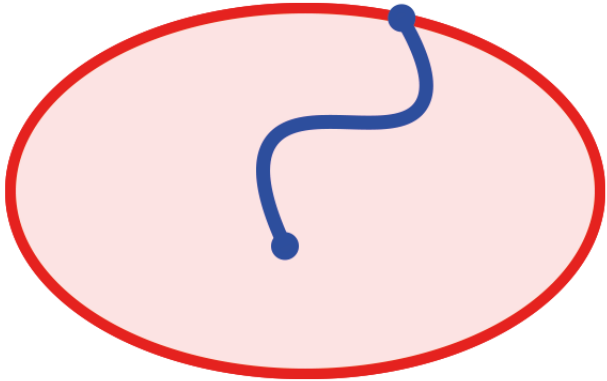
	Int(B)	Front(B)	Ext(B)
Int(A)	1	F	F
Front(A)	0	F	F
Ext(A)	2	1	2



	Int(B)	Front(B)	Ext(B)
Int(A)	1	F	F
Front(A)	F	0	F
Ext(A)	2	1	2



	Int(B)	Front(B)	Ext(B)
Int(A)	F	1	F
Front(A)	F	0	F
Ext(A)	2	1	2



	Int(B)	Front(B)	Ext(B)
Int(A)	1	F	F
Front(A)	0	0	F
Ext(A)	2	1	2

Fig. 2: Figures of DE-9IM used in the island detection algorithm.

- Instantiate Island objects from each list of edges corresponding to each (complex) island. The Island objects instantiated are stored as attributes of the Edge objects that belong to the islands. When computing the orders, testing whether this attribute is null or refers to an island tells if the edge belongs to an island and informs what process to apply on the edge.

Successive islands are yet another type of topological relation between islands, that also has to be detected. Successive islands are not adjacent, and are not separated by any edge (that does not belong to an island). Therefore successive islands do not have regular outgoing edges (except the last one of the series) and thus have to be processed all at once.

- Unlike complex islands, this structure can not be detected using merging. Another specific topological request is defined, still with the *relate()* function and a DE-9IM matrix.

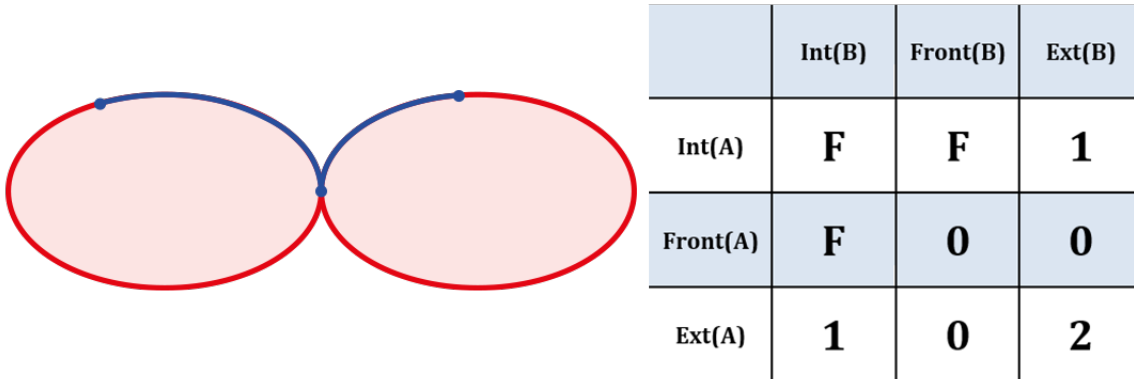


Fig. 3: Figure of DE-9IM used to detect successive islands.

- The lists of edges belonging to complex (or single) islands that are successive are concatenated, so that the orders computation method will read the edges as making up one island and the appropriate process will be applied to the whole island.

5.3.6 Orders computation

The user can choose to compute the Strahler order, the Shreve order and/or the Horton order in the launcher. The orders are defined in the user [documentation](#).

The algorithm computes the orders, store them as attributes of the Edge objects specifically instantiated and add a column for each chosen order to the input layer.

Computing meaningful orders requires to take the specificities of the network structure into consideration. Islands are processed specifically. We present in this section some hypotheses we made and the process we chose for cases handled distinctively.

Strahler, Shreve and Horton stream orders

The algorithm starts from the sources and travels through the river network down to the sinks.

The main steps of the algorithm are the following:

- The iterative process is initialized by setting the Strahler and Shreve orders of the source edges to 1. Each source edge also defines a new stroke (except sources that are in islands).
- For each edge, if all the incoming edges have already been processed, the edge can be processed.

- If the edge is not in an island, its orders are computed following the rules defined for each order. Its stroke is computed by selecting which of its upstream edges the edge continues the best. (*See more on the strokes below.*)
- If the edge is in an island, all the edges of the island the edge belongs to are processed. Then all the outgoing edges of the island are processed. (*See how below.*)
- The Horton order is computed after all the edges have been processed for Strahler order computation. Indeed the Horton order is based on the Strahler value and its computation needs all the Strahler orders to be computed and all the strokes to be built beforehand.

The algorithm runs while there are edges left to process, or until the number of edges to process does not decrease between two iterations (meaning that the edges left to process can not be processed). Edges cannot be processed if they form a loop, as each edge needs all the other edges of the loop to be processed first before they can be processed.

- Potential edges that form a loop are detected. The order computation of the loop is forced. All the edges of the loop are given the same order, which is the order computed standardly from the orders of all the incoming edges of the loop (that are not in the loop). The process is then executed again to compute the orders of the potential edges downstream from the loop that can finally be computed now that their incoming edges have been processed.

Criteria defining a stroke

In the code, its ID defines a stroke. Edges that belong to the same stroke share its ID as attribute.

Each source initiates a new stroke. Each source is given a unique stroke ID. As the algorithm travels through the network, each edge continues one of the upstream strokes. Algorithmically, it means that each edge takes as stroke ID the stroke ID of one of its incoming edges. While there is only one incoming edge, there is no ambiguity and the edges belong to the same stroke, and they are given the same stroke ID. When at a river crossing, there are several incoming edges. Only one stroke will continue downstream, the others stop there.

The upstream stroke that continues downstream from a river crossing can be theoretically chosen according to 4 criteria [TOUYA2007] :

- The name of the river remains the same along the stroke, up and down the river crossing.
- The stroke that has the highest flow is the main stroke, and is the one that continues.
- If the longest stroke upstream from the river crossing is at least 3 times longer than the other incoming strokes, it is the one that continues.
- The stroke that forms an angle with the downstream edge the closest to 180 degrees is the most continuous, and it is the one that continues downstream from the river crossing.

These criteria are in priority order: each criterion applies if the previous criteria are not met.

The algorithm actually takes into account the following criteria: - The names of the edges exist (name field given as input through the launcher), and **the name of the outgoing edge is exactly the same as one of its incoming edges.**

Note: As for now, there is no other test on the strings than strict equality. Therefore, any typing error, any upper/lower case difference, etc. will not allow to match the names. Tests on toponym similarity could improve this criterion (see [Perspectives](#)). Beware also that strings such as “NR” or “N/A” that indicate unknown toponyms will be detected as identical names. We chose not to implement a criterion to eliminate this case as writing conventions in the database may differ.

- One of the incoming strokes is **at least 3 times longer** than the other incoming strokes.
- The stroke that **forms an angle with the downstream edge that is the closest to 180 degrees.**

The flow criterion is pushed aside as such data is rarely available and if it is, it does not follow a regular writing convention (see [Perspectives](#)).

Note: There is no specific process implemented in case of a fork (ie several downstream edges) that is not an island in a network. Forks in islands (ie several edges exit the island) are processed (*see more about that below*). If there is a fork, edges downstream from the fork are processed individually as described above and they may continue the same stroke: the stroke forms a fork. This behavior is appropriate at a river delta. Deltas are thus correctly handled by default. However, one of the criteria is based on the length of the strokes. The lengths of the arms of a fork will add up as if the edges were continuous forming a single line, making the stroke that split in two (incorrectly) long, and thus making it artificially most likely to be chosen as the main stroke at each river crossing downstream from the fork.

Once the strokes are defined, it is possible to compute the Horton stream order, for which each stroke is given the maximum of the Strahler orders of the edges of the stroke.

Stream orders and strokes in islands

In islands, the Strahler order and Shreve order of each edge is the maximum of the orders of its incoming edges. It guarantees the order won't increase at each river crossing inside the island, and the order still gets larger if larger streams meet the island, which is intuitively expected by the user. **The Horton order of the edges in islands is the Horton order of the stroke of the island** (that is the stroke all the edges belong to).

All the edges in an island belong to the same stroke. This decision respects most aspects of a stroke. An island respects good continuity (in general) with one of its incoming edges and one of its outgoing edges. If you look at the network from afar, you will want to draw a line that goes through the island and connects its two ends. There is at first sight no reason why you should pick one edge of the island over the others (in general). This is particularly obvious for single islands, that have only one incoming edge and one outgoing edge. The edges of the two arms are not two rivers but two arms of the same river, therefore they are part of the same stroke. Another criterion in favour of this decision is that a stroke is supposed to start from a source and end either at a sink or at a river crossing. If only one edge of the island was chosen to define the stroke, the other edges would consequently define their own stroke that would not be connected to a source (in general).

There are two downsides to this. The first is that the strokes are supposed to be linear geometries in many situations they are used in. Islands break the continuous single line. The second downside is that the length of the stroke is not clearly defined anymore. Again, this could be a setback in many situations. It actually affects Hy2roresO. Indeed the strokes are defined using a criterion on the upstream length of the stroke (amongst other criteria, *more on stroke construction above*). Adding the lengths of all the strokes of the islands together is meaningless realistically. To overcome this issue, edges that belong to an island are stored separately from the rest of the network, and merged back with the main stroke after each edge has been processed and associated with a stroke, and before computing the Horton order (so that the edges of an island still belong to a stroke and can have an Horton order). **This means that the upstream length of a stroke calculated at a river crossing does not include the river length in islands.**

The stroke of the island edge is based on the incoming edges of the island (the edges that enter the island but that do not delimit the island nor are enclosed in the island). The determination of the stroke of the island edges is based on two criteria:

- If **one of the incoming edges splits in two entering the island**, it probably is the stream delimiting the island and thus the best continuity. If there is only one splitting edge, its stroke is the stroke of the island.
- Otherwise, the **longest upstream stroke** is the stroke of the island.

Note: An angle criterion would be a possible improvement. However, it requires to define the angle between a linear edge and the island surface. See more about that in the [Perspectives](#).

Stream orders and strokes exiting islands

The order of each outgoing edge of the island is computed standardly, taking the incoming edges of the island as incoming edges to compute the order. For instance if there are two edges entering an island whose Strahler orders equal 2 and 2, the Strahler order of the outgoing edge(s) will be 3. The orders of the actual incoming edges (that belong to the island) of the edge exiting the island are ignored. Conceptually, the island is thus similar to a node of the network. **What happens inside the island does not impact the rest of the network.** This is the reason why Hy2roresO is robust to islands when other algorithms are not.

When there is only one edge exiting the island, there is no fork in the network and the stroke of the outgoing edge is quite understandably the stroke of the island, as defined above.

However, there often is more than one edge exiting an island. As mentioned above, allowing forks in strokes has consequences on the length computation of the stroke used as a criterion to compute the strokes. As this situation is frequent and has impacts on the orders computation, Hy2roresO handles forks in islands.

To respect the characteristic that strokes start at a source and end either at a river crossing or at a sink, **all the arms of a fork belong to the same stroke.** In the algorithm, edges of each arm are stored separately. One (random) edge continues the island stroke, while others initiate new arms. Downstream from the fork, each arm is processed as a regular stroke. Its upstream length at a river crossing is the length of the stroke from the source to the fork (shared section), plus the length from the fork to the river crossing (arm length).

At the end of the orders computation, the arms of each forked stroke are merged back together to form one unique stroke. The Horton order can then be computed. It is the maximum of the Strahler orders of the edges of the global stroke, whatever the arm. Every arm will have the same Horton order, as they belong to the same stroke.

Note: As for now, the algorithm does not process forked arms (successive forks). Such “sub-arms” might be missed out when strokes are merged at the end, implying the Horton order could not be computed.

Once the orders and the stroke of all the edges exiting the island are computed, the edges downstream from the island can be processed. The island has been dealt with and the algorithm can continue on the rest of the network.

5.4 Update of the attribute table of the input layer

The last step of the algorithm is to update the input layer by adding new fields.

There is one written field for each computed stream order. Each field is named after the order: “**strahler**”, “**shreve**” or “**horton**”. The field “**id_stroke**” that indicates for each edge the ID of the stroke it belongs to is systematically added to the layer if the strokes have been computed, that is if the Horton order has been computed. An optional field “**reversed**” can also be added (if the option was checked in the launcher), which equals True if the edge was reversed for the orders computation and False if it was not.

Note: As for now, there is no test on the name of the column. Beware if there already is an existing field named as one of the fields to be created by Hy2roresO, as it will be overwritten.

6.1 Classes

class Hydroreso (*self*)

run_process (*self*)
Do the whole stuff

class Edge (*self*)

Class of an Edge of the river network. Instantiated with attributes of the features of the layer of the network.

__init__ (*self, geom, id_edge, node_start, node_end*)
Constructor of the class

Parameters

- **geom** – Geometry of the feature (line)
- **id_edge** – ID of the edge (same as the ID of the feature; integer)
- **node_start** – Start node of the edge
- **node_end** – End node of the edge

copy_edge (*self*)

Copy an Edge object. Create an Edge object that has the same attributes.

Returns Copy of the edge

Return type Edge object

class Node (*self*)

Class of a Node of the river network. Instantiated with attributes of the features of the layer of the network.

`__init__(self, geom, id_node)`

Constructor of the class

Parameters

- **geom** – Geometry of the feature (point)
- **id_node** – ID of the node (ID of one of its connected edges and number 1 or 2 concatenated)

`copy_node(self)`

Copy a Node object. Create a Node object that has the same attributes.

Returns Copy of the node

Return type Node object

`class Island(self)`

Class of an Island of the river network.

Instantiated with the edges of the island.

`__init__(self, island_edges)`

Constructor of the class

Parameters **island_edges** – Edges that make up the island (Edge objects)

Island_edges type

`copy_island(self)`

Copy an Island object.

Create an Island object that has the same attributes.

Returns Copy of the island

Return type Island object

`compute_edges_in_out(self)`

Compute the incoming and outgoing edges of the island.

Set attributes edges_in and edges_out from the edges of the island and their connections to the network.

`compute_edges_in(self)`

Compute the incoming edges of the island.

Set attribute edges_in from the edges of the island and their connections to the network.

`compute_edges_out(self)`

Compute the outgoing edges of the island.

Set attribute edges_out from the edges of the island and their connections to the network.

6.2 Instantiations of the classes

`create_edges_nodes(features, name_column, alt_init_column, alt_final_column)`

Instantiate all the Edge and Node objects that make up the river network.

The name of the river and the altitudes are attributes of the objects if the names of the columns are given in arguments.

Parameters

- **features** (*list of QgsFeatures objects*) – list of all the features of the river network layer
- **name_column** (*string*) – name of the column of the name of the river (selected by the user, empty string if not selected)
- **alt_init_column** (*string*) – name of the column of the initial altitude (selected by the user, empty string if not selected)
- **alt_final_column** (*string*) – name of the column of the final altitude (selected by the user, empty string if not selected)

Returns list of all the edges, list of all the nodes making up the river network

Return type list of Edge objects, list of Node objects

set_edges_connected_nodes (*nodes, edges*)

Fill the lists of incoming and outgoing edges of the input nodes (lists are attributes of Node objects).

The connection between nodes and edges is given by the start node and end node of each edge.

Parameters

- **nodes** (*list of Node objects*) – list of all the nodes making up the river network
- **edges** (*list of Edge objects*) – list of all the edges making up the river network

create_islands (*streams_in_islands*)

Instanciation of Island objects from the list of the edges that make up the island.

The instantiated objects are stored as attributes of the edges that belong to the island.

Parameters **streams_in_islands** (*list of lists of Edge objects*) – edges that belong to the island

6.3 Correct edges directions

test_direction (*edges, nodes*)

Test the direction of edges and return the list of abnormal edges (probable wrong direction).

Uses altitudes if known or studies links in graph if altitude is unknown.

Parameters

- **edges** (*list of Edge objects*) – list of all the edges making up the river network
- **nodes** (*list of Node objects*) – list of all the nodes making up the river network

Returns list of abnormal edges

Return type list of Edge objects

is_node_abnormal (*node*)

Test if a node is abnormal, ie if all its connected edges are in the same direction (all incoming or all outgoing edges) and the node is not a source nor a sink (it has more than one incoming or outgoing edge). A node that is not a source nor a sink should indeed have at least one incoming edge and one outgoing edge (unless it is a multiple source or sink).

Returns True if the node is regarded as abnormal.

Parameters **node** (*Node object*) – node to test

next_node_of_edge (*node*, *edge*)

Return the node of the edge that is not the input node.

Parameters

- **node** (*Node object*) – current node
- **edge** (*Edge object*) – current edge

Returns next node of the edge

Return type Node object

reverse (*edge*)

Reverse an Edge object. The method swaps the nodes of the edge, updates the incoming and outgoing edges lists of the nodes, reverses the geometry of the edge and updates the attribute `edge.reverse` to `True`. Only the object is altered, the input layer remains unchanged.

Parameters **edge** (*Edge object*) – edge to reverse

reverse_all_edges (*edges_to_reverse*)

Reverse edges of the input list (call `reverse(edge)` method).

Parameters **edges_to_reverse** (*list of Edge objects*) – list of edges to reverse

edges_to_features (*list_edges*, *input_layer*)

Transform a list of Edges objects into a list of the corresponding features of the layer.

Parameters

- **list_edges** (*list of Edge objects*) – list of the edges corresponding to the desired features
- **input_layer** (*QgsVectorLayer object*) – layer of the features (and the corresponding edges)

Returns list of features

Return type list of QgsFeatures objects

features_to_edges (*list_features*, *edges*)

Transform a list of QgsFeatures objects into a list of the corresponding Edge objects of the layer.

Parameters

- **list_features** (*list of QgsFeatures objects*) – list of the features corresponding to the desired edges
- **input_layer** (*QgsVectorLayer object*) – layer of the features (and the corresponding edges)

Returns list of edges

Return type list of Edge objects

6.4 Sources and sinks

find_sources_sinks (*edges*)

Find source edges and sink edges of the network.

A source edge is an edge exiting a node that is only connected to this edge. A sink edge is an edge entering a node that is only connected to this edge.

Parameters **edges** (*list of Edge objects*) – list of all the edges making up the river network

Returns list of source edges, list of sink edges

Return type list of Edge objects, list of Edge objects

6.5 Island detection

detect_islands (*stream_layer, edges*)

Detect islands in the network. Return a list of lists of the edges that make up each island.

Parameters

- **stream_layer** (*QgsVectorLayer object*) – layer of the river network
- **edges** (*list of Edge objects*) – list of all the edges that make up the river network

Returns list of lists of edges of the islands

Return type list of lists of Edge objects

polygonize (*input_layer, name="temp"*)

Island detection algorithm. If there is no island, return None.

Parameters

- **input_layer** (*QgsVectorLayer object*) – layer of the river network
- **name** (*string*) – name of the layer if displayed

Returns layer of faces of the network (islands, polygons)

Return type QgsVectorLayer object

create_layer_geom (*list_geom, crs, name="temp"*)

Create a Polygon layer with the input list of geometries (must be polygons).

Parameters

- **list_geom** (*list of QgsGeometry*) – list of polygons
- **crs** (*string (format Wkt)*) – the crs of the output layer
- **name** (*string*) – (optional) Name of the layer to display. Default = “temp”

Returns layer of polygons

Return type QgsVectorLayer object

iterator_to_list (*iterator*):

Transform the input iterator into a list.

Parameters **iterator** (*iterator*) – the iterator to convert

Returns the list of the values of the iterator

Return type list

aggregate (*listFeatures*)

Aggregate the geometries of the input list of features into one geometry.

Parameters **listFeatures** (*list of QgsFeatures objects*) – features to aggregate

Returns the aggregated geometry

Return type QgsGeometry object

multi_to_single (*geom*)

Transform the input multi-polygon into a list of single-polygons.

Parameters **geom** (*QgsGeometry object*) – multi-polygon

Returns list of the single geometries

Return type list of QgsGeometry objects

relate_stream_island (*stream_layer, island_layer*)

Return the streams inside or delimiting islands. The topology is defined by DE-9IM matrices.

Parameters

- **stream_layer** (*QgisVectorLayer object (lines)*) – the layer of the river network
- **island_layer** (*QgisVectorLayer object (polygons)*) – the layer of the islands

Returns list of lists of all the streams that make up the islands

Return type list of lists of QgisFeatures objects

merge_successive_islands_streams (*streams_in_island_list*)

Compute successive islands.

Successive islands are islands that are not adjacent, and there is no edge between them (that does not belong to an island). The topology is defined by a DE-9IM matrix. Successive islands are merged into one complex island: lists of edges of successive islands are concatenated into one list. Return the list of lists of features (edges) of the islands.

Parameters **streams_in_island_list** (*list of lists of QgisFeatures objects*) – list of lists of all the streams that make up the islands

Returns list of lists of all the streams that make up the islands, successive islands merged

Return type list of lists of QgisFeatures objects

merge_duplicate (*merged_streams_in_island_list*)

Merge lists that have at least one common element into one list.

Parameters **merged_streams_in_island_list** (*list of lists*) – list of lists to test and merge

Returns list of merged lists

Return type list of lists

6.6 Orders

compute_stroke (*dict_strokes, edge, list_incoming_edges*)

Compute the stroke of the input edge. Return the ID of the stroke.

Parameters

- **dict_strokes** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built {key= stroke ID: values= list of the edges of the stroke}

- **edge** (*Edge object*) – edge of which the stroke is computed
- **list_incoming_edges** (*list of Edge objects*) – list of the incoming edges of the input edge

Returns ID of the stroke of the input edge

Return type integer

compute_length (*stroke*)

Return the total length of a stroke (sum of the lengths of the geometries of the edges that make up the stroke).

Parameters **stroke** (*list of Edge objects*) – list of edges

compute_angle (**edge_in**, **edge_out**):

Compute the angle formed by edge_in and edge_out, edge_in entering the node edge_out exits.

Parameters

- **edge_in** (*Edge object*) – one side of the angle
- **edge_out** (*Edge object*) – one side of the angle

azimuth_angle (**node_start**, **node_end**):

Compute the azimuth of a line defined by its start node and its end node.

Parameters

- **node_start** (*QgsPointXY object*) – origin of the line
- **node_end** (*QgsPointXY object*) – end of the line

compute_stroke_of_island (*dict_strokes*, *island*, *incoming_edges_island*)

Compute the stroke of the island. Return the ID of the stroke.

Parameters

- **dict_strokes** (*dictionary {integer: list of Edge objects}*) – dictionary of the strokes already built {key= stroke ID: values= list of the edges of the stroke}
- **island** (*Island object*) – island of which the stroke is computed
- **incoming_edges_island** (*list of Edge objects*) – list of the incoming edges of the island

Returns ID of the stroke of the input edge

Return type integer

compute_stroke_outgoing_island (*dict_strokes*, *dict_forks*, *island_id_stroke*, *outgoing_edges_island*)

Compute the stroke of the outgoing edges of the island. Set the attribute id_stroke of the edges.

Parameters

- **dict_strokes** (*dictionary {integer: list of Edge objects}*) – dictionary of the strokes already built {key= stroke ID: values= list of the edges of the stroke}
- **dict_forks** (*dictionary {integer: list of Edge objects}*) – dictionary of the strokes already built that split {key= upstream stroke ID: values= list of stroke IDs after the stroke}
- **island_id_stroke** (*integer*) – stroke ID of the island
- **outgoing_edges_island** (*list of Edge objects*) – list of the outgoing edges of the island

is_upstream_processed (*incoming_edges, edges_to_process*)

Check if all incoming edges have been processed.

Return True if processed.

Parameters

- **incoming_edges** (*list of Edge objects*) – list of edges to check (incoming edges of a current edge)
- **edges_to_process** (*list of Edge objects*) – list of edges left to process

process_network (*edges, sources_edges, orders_to_compute, edges_to_process, dict_strokes, dict_strokes_in_island, dict_forks*)

Compute stream orders: Strahler, Shreve and / or Horton, according to the selection of the user.

The computed orders are attributes of the Edge objects.

Parameters

- **edges** (*list of Edge objects*) – list of all the edges making up the river network
- **sources_edges** (*list of Edge objects*) – list of all source edges of the river network
- **orders_to_compute** (*list of strings*) – list of the orders to compute (selected by the user)
- **edges_to_process** (*list of Edge objects*) – list of the edges left to process
- **dict_strokes** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built (except edges of islands) {key= stroke ID: values= list of the edges of the stroke}
- **dict_strokes_in_island** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built of edges in islands ; {key= stroke ID: values= list of the edges of the stroke}
- **dict_forks** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built that split {key= upstream stroke ID: values= list of stroke IDs after the stroke}

is_in_loop (*left_edge, edges_to_process*)

Test if an edge is connected to a loop in the network. Return the edges of the loop in a list (return an empty list if no loop was detected).

Parameters

- **left_edge** (*Edge object*) – edge to test (could not be processed by process_network)
- **edges_to_process** (*list of Edge objects*) – list of edges left to process

Returns list of the edges of the loop (or empty list if no loop)

Return type list of Edge objects

process_loop (*edges_in_loop, orders_to_compute, edges_to_process, dict_strokes_in_island*)

Process edges of a loop.

Their order and their stroke take the same value. The orders are computed with orders of the incoming edges of the edges of the loop that are known (regular Strahler or Shreve, only on already processed incoming edges). The stroke is the stroke of the island (any loop is an island).

Parameters

- **edges_in_loop** (*list of Edge objects*) – list of the edges of the loop

- **orders_to_compute** (*list of strings*) – list of the orders to compute (selected by the user)
- **edges_to_process** (*list of Edge objects*) – list of edges left to process
- **dict_strokes_in_island** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built of edges in islands ; {key= stroke ID: values= list of the edges of the stroke}

Returns indicate if the loop was successfully processed (can be processed only if incoming edges were already processed)

Return type boolean

merge_strokes (*dict_strokes, dict_strokes_in_island, dict_forks*)

Merge the strokes of the islands and of the forks with the main stroke.

Parameters

- **dict_strokes** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built (except edges of islands) ; {key= stroke ID: values= list of the edges of the stroke}
- **dict_strokes_in_island** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built of edges in islands ; {key= stroke ID: values= list of the edges of the stroke}
- **dict_forks** (*dictionary {integer:list of Edge objects}*) – dictionary of the strokes already built that split ; {key= upstream stroke ID: values= list of stroke IDs after the stroke}

compute_horton (*dict_strokes*)

Compute the Horton order using the input strokes.

The computed orders are attributes of the Edge objects.

Parameters dict_strokes (*dictionary {integer:list of Edge objects}*) – dictionary of all the strokes built, except edges in islands ; {key= stroke ID: values= list of the edges of the stroke}

6.7 Write in table

update_table (*input_layer, orders_to_compute, field_reverse, edges*)

Updates the table of the layer by adding a column named like the name of the order and filling it with the orders calculated before.

Updates the table with a field “reversed” if the user selected the option (True if the edge has been reversed for the computation of the orders).

Parameters

- **input_layer** (*QgsVectorLayer object*) – layer of the river network
 - **orders_to_compute** (*list of strings*) – list of the orders to compute (selected by the user)
 - **field_reverse** (*boolean*) – field reversed is added to the table (selected by the user)
 - **edges** (*list of Edge objects*) – list of all the edges making up the river network
-

6.8 Dialog messages

show_field_created_successfully()

Display a message box that indicates when the input layer has been updated.

show_message_no_stream_order_selected()

Display a message box that indicates when no stream order was checked for computation by the user.

6.9 Save output

save_output_layer() (*output*, *path_to_saving_location*)

Save the output layer

Parameters

- **output** (*QgsVectorLayer*) – output layer to be saved
- **path_to_saving_location** (*string*) – the path to the place where the layer has to be saved

Hy2roresO deals with a whole bunch of cases that can happen with natural hydrological networks. However, there still are some things that can be realized to improve the plugin.

- Strokes can be defined using the criterion that their name should remain the same after a river crossing. Rivers were indeed named according to rather intuitive criteria of continuity that strokes should meet. However, toponyms can be written in many different ways in the database. As for now, the algorithm only tests the strict equality of the strings. Therefore, any typing error, any upper/lower case difference, etc. will not allow to match the names. **Tests on toponym similarity** could improve this criterion. Methods of distance measurements between toponyms exist and could be implemented to improve the pairing of names. Beware also that strings such as “NR” or “N/A” that indicate unknown toponyms will be detected as identical names. We chose not to implement a criterion to eliminate this case as writing conventions in the database may differ. Frequently used conventions should be implemented for **better matching of the names and improved strokes computation**. (See [Approach & Strategy](#) .)
- Forks in the network are not managed by the algorithm. Forks induce an error on the length of the stroke and generate mistakes when computing the strokes.
- One thing that can be done is to create fictive network inside the islands, for example by using a **skeleton** of the geometry of the island. This could for example **improve the definition of strokes entering and exiting an island**, especially if the island is curved. It also would grant the strokes a linear geometry (without the forks inside the islands), which is commonly expected of a stroke.
- The method *interpolateAngle()* from the class **QgsGeometry** was briefly studied to better deal with curved islands by interpolating the angle between the island and its incoming edges, or the island and its outgoing edges. Maybe this could lead to a better process for the **strokes**. As for now, there is indeed no consideration of angle between the island and its connected edges to define the strokes, which is a major weakness. **The angle between an edge and an island** is a notion yet to define.
- The **flow** of each edge is theoretically the second criterion to determine strokes, however it is not handled in our algorithm since flow data is rarely available. Adding a condition on it to the code could be beneficial for a **better determination of the strokes**. However testing such a field automatically implies that the field must follow a given format, which is tricky to generalize and is hardly reliable due to the diversity of database specifications (especially if strings are allowed).



Fig. 1: Example of a skeleton built inside an island, from Voronoï polygons

- The plugin already tests if edges seem to be **directed correctly**, based on their altitudes and the direction of their connected edges. The detection algorithm would be improved by adding an **angle criterion** to the tests.
- The plugin does not test whether **a field already exists** in the attribute table of the input layer. If an existing field is named like the field to create, it will be overwritten. Warning the user and asking whether they want to overwrite the existing field or rename the field to create can be an improvement.

CHAPTER 8

Team

This plugin is the achievement of a project by Alice Gonnaud, Michaël Gaudin and Guillaume Vasseur, students from the National School of Geomatics (ENSG) in Marne-la-Vallée (Greater Paris, France).

CHAPTER 9

Story

This project was developed between February and May 2018 by three students of the National School of Geomatics (ENSG), under the mentoring of Cécile Duchêne.



Hy2roresO

Une extension **QGIS** pour le calcul d'ordres d'un réseau hydrographique

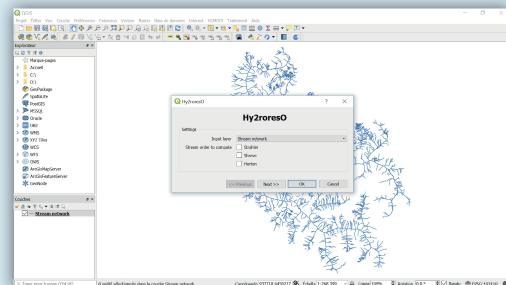
Contexte

Le calcul d'ordres hydrographiques est une méthode d'analyse spatiale permettant de hiérarchiser les cours d'eau. Il est notamment utilisé dans la détermination de l'espace occupé par un bassin versant, ou dans la surveillance des cours d'eau à risque. Peu de plugins prennent en compte la **complexité réelle** des configurations possibles dans un réseau hydrographique. L'idée derrière le projet **Hy2roresO** était de prendre en compte toutes ces données, du **réseau le plus simple** possible au **réseau anastomosé** en passant par les îles, tous ces facteurs ayant des influences fortes sur le calcul des ordres.

Comparatif avec l'existant

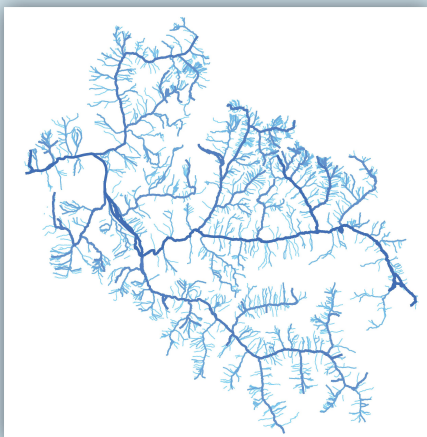
	Plugin GRASS	Plugin Strahler	Plugin Hy2roresO
Gestion des réseaux simples (arborescent)	✓	✓	✓
Gestion des îles	✗	✓	✓
Gestion des îles complexes	✗	✗	✓
Gestion des îles successives	✗	✗	✓
Retournements de tronçons possible	✓	✗	✓
Gestion des boucles	✗	✗	✓

Interface utilisateur



- ✓ Calcul des ordres de **Strahler**, **Shreve** et **Horton**
- ✓ Boîte de dialogue **pensée et orientée utilisateur**
- ✓ Prise en compte de **cas complexes**, plus réalistes

Cas d'étude



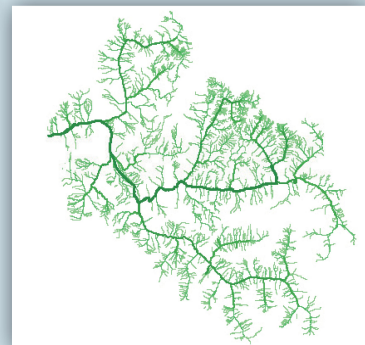
- ▶ **Cartographie d'un cours d'eau auquel on a appliqué une couleur par «stroke»**, ligne continue de tronçons permettant de calculer l'ordre de Horton. Les strokes sont la fusion des tronçons du tracé du fleuve en une ligne continue de l'exutoire à la source principale du fleuve. Les construire permet ainsi de reconstruire les fleuves principaux et leurs affluents, ce qui offre des applications bien plus larges que le calcul de Horton : généralisation de cartes, calcul d'indicateurs, etc.

- ▶ **Cartographie d'un cours d'eau auquel on a appliqué une symbologie en fonction de l'ordre de Horton.**

L'ordre de Horton permet de mettre en évidence les principaux cours d'eau, et de différencier le lit majeur d'un fleuve de ses affluents. Il met particulièrement en évidence les cours d'eau réguliers, en reliant une source à un exutoire.

- ▶ **Cartographie d'un cours d'eau auquel on a appliqué une symbologie en fonction de l'ordre de Strahler.**

L'ordre de Strahler met en valeur les cours d'eau importants, en augmentant au fur et à mesure que le cours d'eau rencontre des affluents, tout au long de son parcours de la source jusqu'à l'exutoire.



Retrouvez-nous
sur **GitHub**



Développé avec



3.0



3.6

Projet développement de 2^{ème} année du cycle
ingénieur mené entre février et mai 2018 par :

Michaël Gaudin
Alice Gonnaud
Guillaume Vasseur

Encadrés par **Cécile Duchêne**

ENSG
Géomatique

ÉCOLE NATIONALE
DES SCIENCES
GÉOGRAPHIQUES

Chapter 9. Story

CHAPTER 10

Acknowledgment

We would like to thank very much our contractor, Cécile Duchêne, for her time, her patience and her dedication to the accomplishment of this project.

We also thank Guillaume Touya, head of the COGIT Laboratory at the Institut National de l'Information Géographique et Forestière (IGN), for his precious advices on the implementation of the computation of stream orders algorithms.

Greetings to the Ecole Nationale des Sciences Géographiques for the working conditions provided.

Project realised under the « CHOUCAS : Intégration de données hétérogènes et raisonnement spatial pour l'aide à la localisation des victimes en montagne » (Integration of heterogeneous data and spatial reasoning to help locating victims in mountains) project. Project funded by the French National Research Agency (project n°ANR-16-CE23-0018)

Finally, a thought to Nicolas, Luc, Marion, Alice, Jocelyn, Matthieu, Hugo and most of the time Robin, who supported us during the whole development of this project.

CHAPTER 11

Contribute

If you want to contribute to our project to make our plugin even more efficient or if you have seen some bugs in the code, please tell us here : <https://github.com/mgaudin/Hy2roresO/pulls>

Bibliography

[TOUYA2007] <http://recherche.ign.fr/labos/cogit/publiCOGITDetail.php?idpubli=4181&portee=labo&id=1&classement=date&duree=100&nomcomplet=Touya%20Guillaume&annee=2007&principale=>

Symbols

`__init__()`, 31
`__init__()` (Edge method), 31
`__init__()` (Island method), 32

A

`aggregate()` (built-in function), 35

C

`compute_edges_in()` (Island method), 32
`compute_edges_in_out()` (Island method), 32
`compute_edges_out()` (Island method), 32
`compute_horton()` (built-in function), 39
`compute_length()` (built-in function), 37
`compute_stroke()` (built-in function), 36
`compute_stroke_of_island()` (built-in function), 37
`compute_stroke_outgoing_island()` (built-in function), 37
`copy_edge()` (Edge method), 31
`copy_island()` (Island method), 32
`copy_node()`, 32
`create_edges_nodes()` (built-in function), 32
`create_islands()` (built-in function), 33
`create_layer_geom()` (built-in function), 35

D

`detect_islands()` (built-in function), 35

E

Edge (built-in class), 31
`edges_to_features()` (built-in function), 34

F

`features_to_edges()` (built-in function), 34
`find_sources_sinks()` (built-in function), 34

H

Hydroreso (built-in class), 31

I

`is_in_loop()` (built-in function), 38
`is_node_abnormal()` (built-in function), 33
`is_upstream_processed()` (built-in function), 37
Island (built-in class), 32

M

`merge_duplicate()` (built-in function), 36
`merge_strokes()` (built-in function), 39
`merge_successive_islands_streams()` (built-in function), 36
`multi_to_single()` (built-in function), 36

N

`next_node_of_edge()` (built-in function), 33
Node (built-in class), 31

P

`polygonize()` (built-in function), 35
`process_loop()` (built-in function), 38
`process_network()` (built-in function), 38

R

`relate_stream_island()` (built-in function), 36
`reverse()` (built-in function), 34
`reverse_all_edges()` (built-in function), 34
`run_process()` (Hydroreso method), 31

S

`save_output_layer()` (built-in function), 40
`set_edges_connected_nodes()` (built-in function), 33
`show_field_created_successfully()` (built-in function), 40
`show_message_no_stream_order_selected()` (built-in function), 40

T

`test_direction()` (built-in function), 33

U

`update_table()` (built-in function), 39