Deriving Hydrological Response Units (HRUs) using a Web Processing Service implementation based on GRASS GIS

Christian Schwartze
Department of Geography – Chair of Geoinformatics, Geohydrology and Modelling
University Jena
christian.schwartze@uni-jena.de

Keywords: QGIS, GRASS, WPS, PyWPS, Web Processing Service, Python, HRU, Hydrological Response Units

Abstract

QGIS releases equal to or newer than 0.7 can easily connected to GRASS GIS by means of a toolbox that provides a wide range of standard GRASS modules you can launch – albeit only on data coming from GRASS. This QGIS plugin is expandable through XML configurations describing the assignment of options and inputs for a certain module. But how about embedding a precise workflow where the several processes don't consist of a single GRASS module by force? Especially for a sequence of dependent tasks it makes sense to merge relevant GRASS functionality into an own and encapsulated QGIS extension. Its architecture and development is tested and combined with the Web Processing Service (WPS) for remote execution using the concept of hydrological response units (HRUs) as an example. The results of this assay may be suitable for discussing and planning other wizard-like geoprocessing plugins in QGIS that also should make use of an additional GRASS server.

Brief background

Hydrological Response Units may be considered as spatial entities with the objective of applying them to the process of water modelling. The designation of such regions as assumed for the present work operates on physiographical characteristics of the catchment area [2] and aims at its partitioning into zones similar to each other – both topography and dynamic related. For further information such as various additions you may refer to e.g. [1] and [5]. Details and sub-steps of the derivation used by the planned tool are discussed in section 4.

Architecture

Due to the abundance of tasks a complete HRU derivation consists of, it was decided to split it into modules developed as processes for PyWPS 2.0.1 [8]. To meet the requirements of a client/server system, albeit in this case running all components on just one single machine (including WPS), a user-friendly client enabling the several tasks sequentially would be more than appropriate and has to be developed. In this context QGIS gets the vote. Not only on account of the python scripting support in QGIS, but also because of its very well GIS visualization capabilities equipped with basic, spatial tools. As PyWPS comes with native GRASS support, consequently all HRU relevant computation is done by GRASS, here version 6.2.2. By the way, the written plugin profits i.e. from the temporary GRASS sessions in PyWPS since only important main data are swapped out when a HRU task ends – no extra management of GRASS mapsets is needed. So in that case PyWPS serves as a kind of middleware between two GIS, or in other words, it separates processing from visualization in the HRU tool.

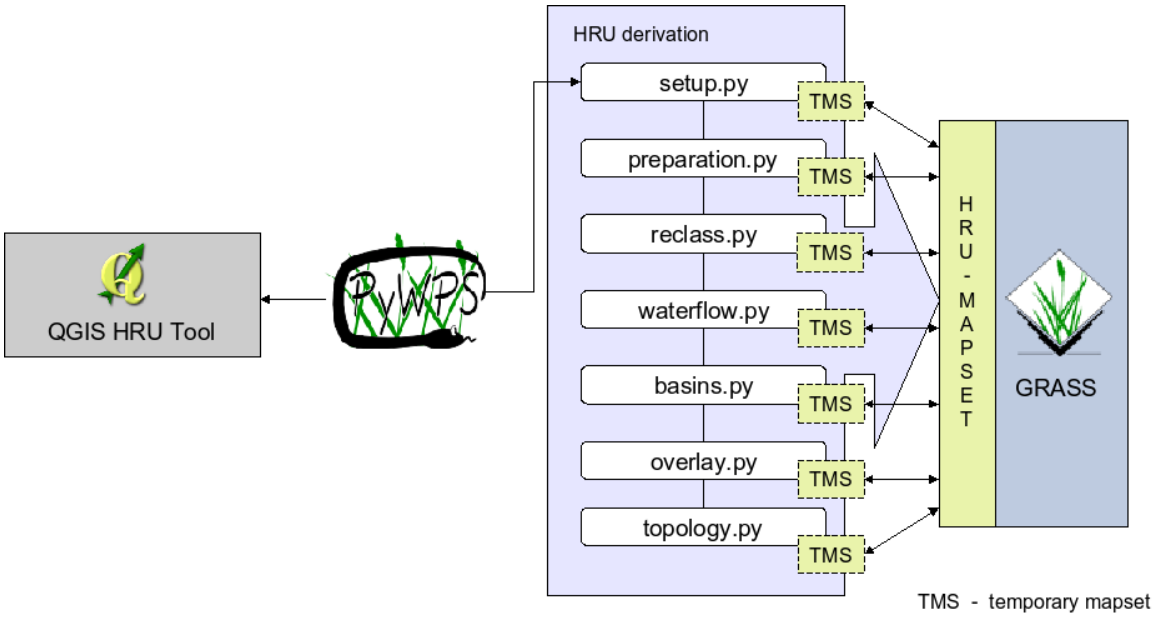

Figure 0: Architecture

Extending QGIS

In order to write a new extension for QGIS [6] you start work in an empty subfolder in `/python/plugins/` of your installation directory. The Plugin Manager gets its information about available python plugins from the primarily created `__init__.py` file – the starting point for all upcoming implementation code. More precisely, the first activation of the plugin by the installation routine results in a call of the *classFactory()* function that returns a plugin instance initiating the toolbar icon, menu entries and other plugin related control items.

The sample HRU plugin

Adaptability concerning the plugin options and functionality is mainly focused during the development. Later changes and improvements in the HRU derivation process should be easy to integrate. Hence, a module concept was designed and the phases of the current HRU work flow were mapped on ready-to-use components instantiated through Python classes. If you are willing to write some extension for the HRU derivation plugin you have to become acquainted with the abstract python class `HRUModule`. Therefore, an own module designed for the process chain has to be a subclass of `HRUModule` and has to implement four common functions:

- *SetInput()* specifies the layout of a tabbed widget and arranges the necessary input forms.
- *Validate()* addresses relevant module input parameters, checks and formats them to a valid PyWPS parameter string.
- *UpdateWizard()* manages the modules impact on any other tabbed widget within the plugin, e.g. enabling subsequent wizard tabs, filling out forms or predefining options in upcoming tasks.
- *UpdateMapView()* handles modifications that concern visualisation of map layers and linked legend entries in QGIS.

The individual processes were implemented according to the guidelines in [4]. Thus, the HRU derivation was divided into logical units which resulted in seven module classes. Once coded, you can integrate such modules using the statement

```
self.wizard.addTab(WaterFlowModule(), WaterFlowModule.MODULE_ICON, WaterFlowModule.MODULE_TAB_DEF)
```

that embeds a tab in the wizard whose initial state is enabled as long as an other module releases it. That is why the correct schedule of derivation is guaranteed, however a return to already performed steps is possible at any time. Especially for testing influence of various input parameters the backspaces are considered meaningful. In PyWPS [8] each process stores its assigned and calculated data in GRASS mapsets that do not outlive the end of the process. That means, a series of `n` PyWPS tasks is instantiated along with `n` temporary mapsets whose names follow the pattern `tmpmapset<x>`.

In spite of the alternative to handle all processes in only one but persistent and already existent GRASS location/mapset, the temporary version has been used. So each process implementation will end with lines containing some *g.copy* calls. The advantage is that any interim solution never belongs to user's location and is removed at the end of the WPS process. When it is triggered twice (or several times) the GRASS data would just be overwritten by the WPS process while copying it to the persistent mapset.

The workflow more detailed

All the processes explained in the following subsections have something in common: their results are relocated from a process-owned temporary mapset to a persistent mapset inside a predefined GRASS location. In process code stored (estimated) computing time information proves to be helpful for the user while he tracks the execution in the wizard (see the progress bar).

Preparation

The QGIS/GRASS based HRU derivation starts with an option dialogue where you have to specify essential data, including the digital elevation model (DEM), region characteristics (land use, soils and geology) as well as the locations of gauges. As the first noted are all raster maps, the latter one should be usually imported as a shapefile. To minimize every kind of computational effort in pending tasks users have to drag a bounding box keeping the rough catchment area in mind. The underlying WPS process produces a subimage of each stated data layer using GDAL/OGR and imports them to a GRASS location locally installed.

Yet another preprocessing task which is integrated into the wizard sequence as a separate module deals with the DEM to obtain a depressionless elevation model (see the actual but still disabled *Preparation* tab next to *Setup*, not explicitly focused in screenshot of figure 1). Means, another WPS process is triggered that not only runs *r.fill.dir* multiple times but also provides slope and aspect of the area.

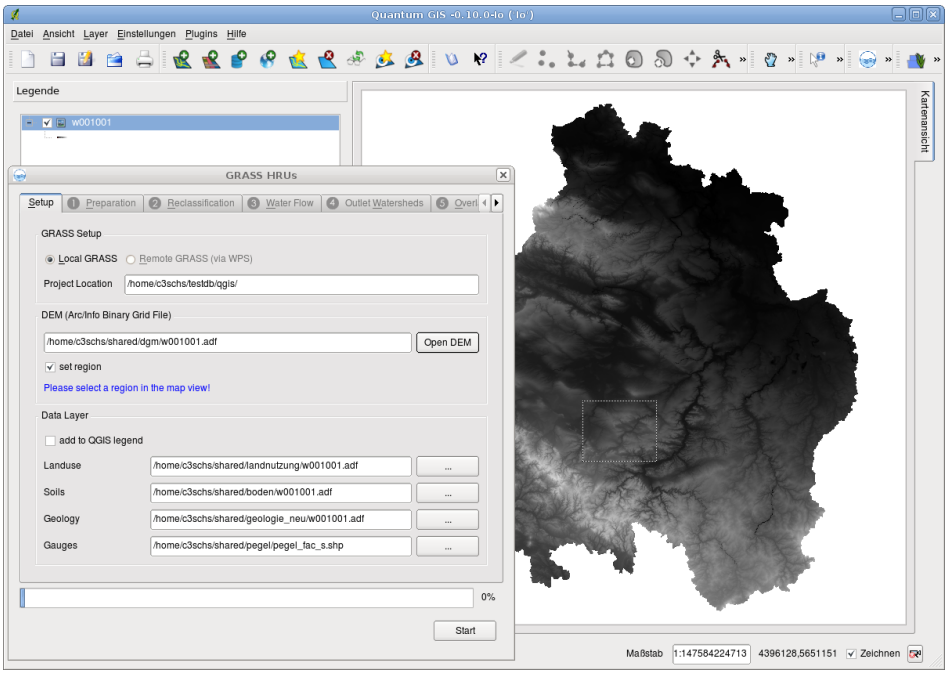

Figure 1: Setup module

Reclassification

As long as real-life surface values (gathered from whatever measuring method) represent slope, aspect and sinkless elevation data, an intersection between them is hard to handle. On that account the reclassification module expects rules defining classes of categories entered in three respective tables (figure 2). Recommended ranges may be accepted or changed. Internally, typical GRASS rule files are written and will serve as input for *r.reclass*.

Generation of waterflow related maps

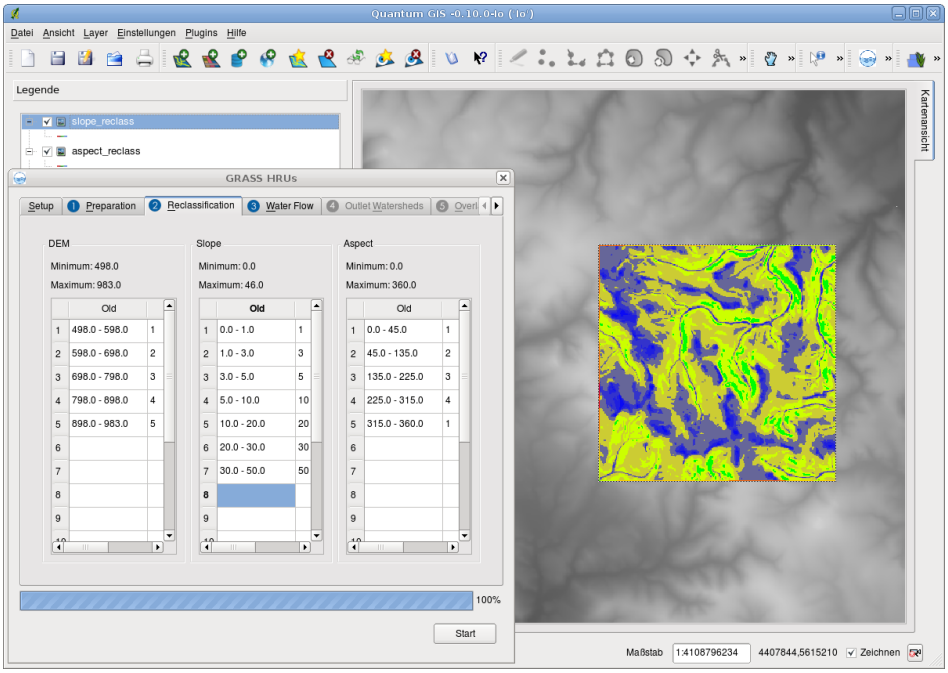

Figure 2: Reclassification module

Within the next step you have to make a set of water flow oriented maps available (figure 3). This includes the drainage direction, the accumulation and the location of watershed basins. An additional raster map has to point out the segmented stream network (so called "reaches"). There is one GRASS analysis tool that covers the computation of all desired maps in a single command – *r.watershed*. Unlike in many another WPS processes this almost elementary case leads to a quite concise task description in Python language.

Speaking about watershed basins means to distinguish between such type of basin derivation defined by *r.watershed* and such given through *r.water.outlet*. The latter GRASS module determines a basin as you pass a geographic coordinate, e.g. a gauge position. Using for instance *r.water.outlet* in a further WPS process and a well placed overlay statement inside the gauges iteration loop constitutes a solution for a gauge oriented basin map. In terms of accurate results you will probably have to move gauges onto reaches manually. But this can be done quickly since QGIS offers a vector data editing mode (figure 3, right).

Overlay strategy

The fifth step by the wizard (figure 5) serves as a special intersection operation between actual eight preset or calculated raster maps. Latter includes the reclassified DEM, slope and aspect data as well as soils, landuse and geology information. In addition, the watershed basin map and the basins relative to gauges in the catchment are required. The idea is shown in figure 4a and consists of following steps:

1. Load the gauge basin map from subsection 4.3 as a reference map for spatial extent of resulting HRU dataset and construct a map that masks out the relevant area
2. Join the mask and above-mentioned data layers separately using *r.patch* and apply *r.null*

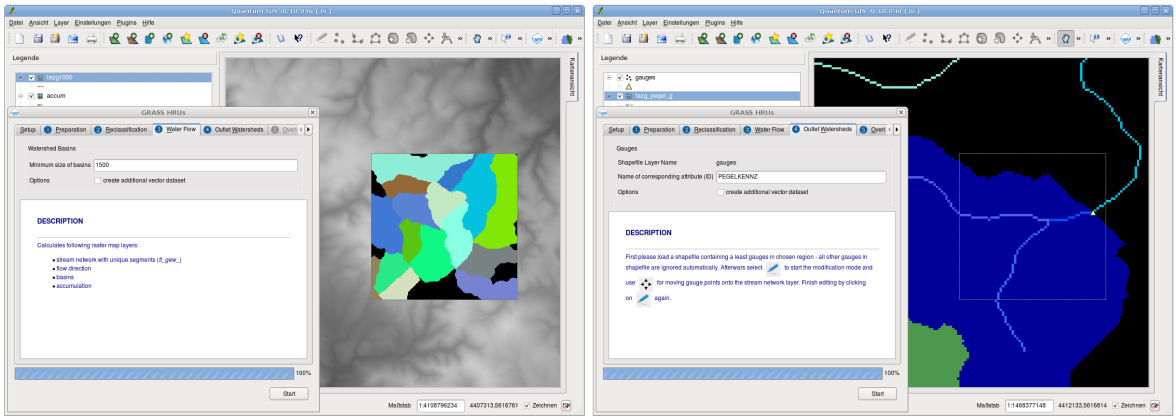

Figure 3: Water flow module

to redefine the null value in the new masked datasets

3. Merge the non zero data in the eight maps of (2) via *r.cross* to a single map
4. Make use of *r.clump* to relabel occurrences of non adjacent regions which still have the same category

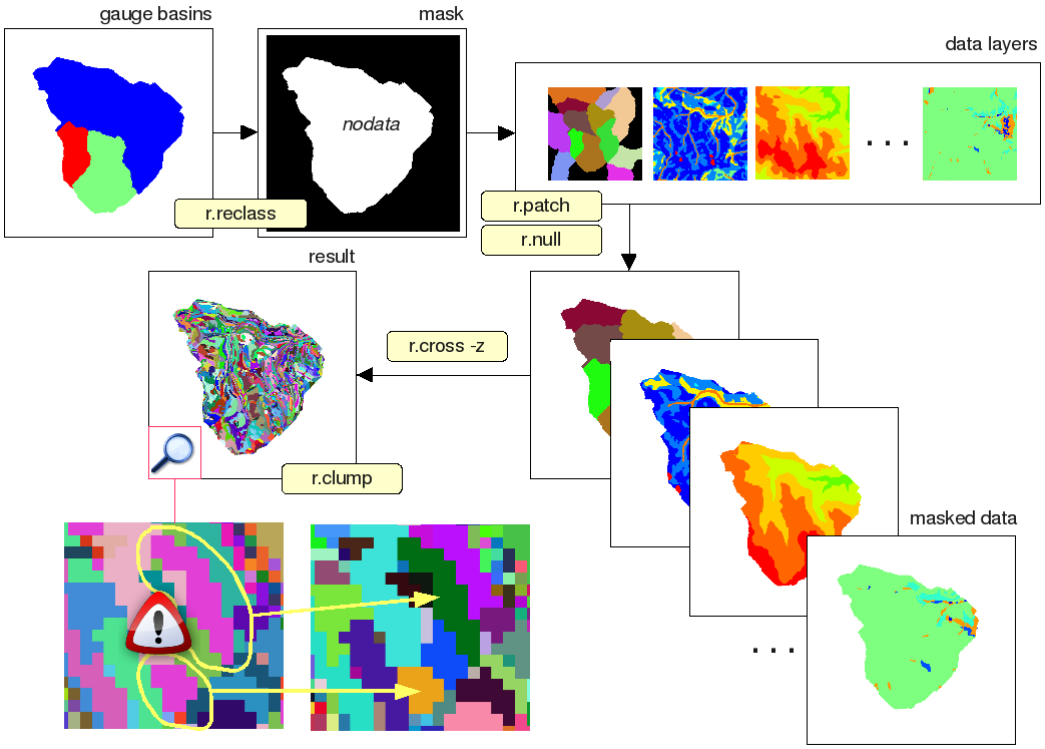

Figure 4a: Overlay method

This procedure does not yet result in final HRUs since so much spurious, midget areas may occur. Eliminating almost pixelsized intersection snippets and their reallocation is an essential part in the postprocessing. In the range of vector data *v.clean* with correct parameters hits

the spot. The same is true for *r.reclass.area* on raster maps but with the limitation that respective areas are filled with GRASS nodata cell value. Filling them taking nearby areas into account is one solution discussed in [3]. The next script operates in a similar way:

1. Detect areas which are smaller than a specific threshold, e.g. 28125m^2 (= 45 pixels, 25m resolution assumed)
2. While such areas exist do:
   (a) Get the one-pixel-wide boundary of each area and fill the interior with NULL
   (b) For every pixel onto the boundary do:
      i. Reassign the category value with largest occurrence in the 3x3 neighbourhood (corresponds to mode value)
      ii. Mark the left NULL values as removable, minimal areas (pink colored in 2 and 3, figure 4b)

As indicated in the output map (4, figure 4b) snippets are just not reallocated to one neighbour region but rather melt into adjacent areas proportionally. When the superior WPS process has done that kind of cleaning the HRUS obtain their final form. However, it raises the question as to whether the underlying data associated with each HRU is still significant. Due to the fact that the cleaning algorithm manipulates the original overlay map (see above) depending on the number of eliminating areas and their location to each other, any dominant characteristic (e.g. soil type) could be changed. For this reason a further script takes the regenerated and cleaned HRU map as a type of template. Based on it all data layers are checked to determine a potentially new raster category that accounts for a major portion within each HRU. This is done by calling *r.statistics* plus *mode* method as aggregation option.

At the end of the overlay section it appears to be appropriate to store these gained and probably new categories as labels to the HRU raster map. A piped combination of *r.stats*, some *awk* commands and *r.reclass* on the cleaned HRU data helps writing a vertical bar separated label entry that represents values for the linked data layers:

```
[...]
#var inputs: list of data layers (with new determined raster values)
inputs = inp_list.rstrip(",")
awk_cmd = "'{print $1,\" = \",$1,"
for i in range(1, len(inputs.split(","))+1):
    awk_cmd += "$"+str(i*2)+"\"|\""
awk_cmd += "}'"
g_cmd = "r.stats -l input=%s | " % inputs
g_cmd += "awk %s | " % awk_cmd
g_cmd += "r.reclass --o input=%s output=%s_result" % \
(os.getenv("GIS_OPT_INPUT"), os.getenv("GIS_OPT_INPUT"))
os.system(g_cmd)
[...]
```

Topological network

While the last preceding paragraph has created the prerequisites to feed physiographic properties into some model the next section focuses on how to include relations between HRUs. It aims at pointing out drainages from one HRU into others, furthermore into streams in-

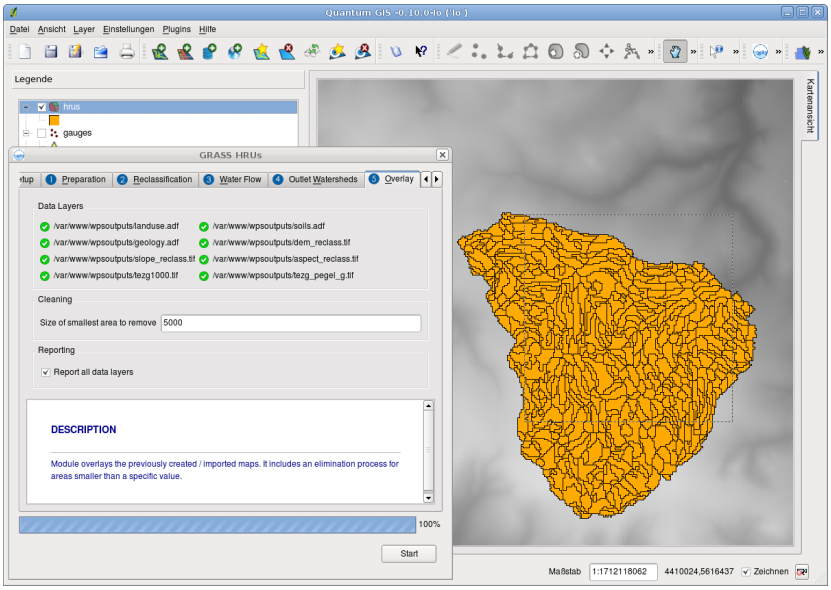

Figure 5: Overlay module

side catchment (routing). Therefore, the topological sequence acquisition is bipartite and exemplified by figure 6 where pink lines demonstrate HRU borders:

"HRU to HRU"

1. Respectively do a *r.mapcalc* to get
   (a) borderlines of the HRU map
   (b) drainage direction only on borderlines from (1) – see step 1, figure 6
   (c) drainage destination (ID of HRU) only on borderlines – see step 2, figure 6
   (d) accumulation data only on borderlines – see step 3. figure 6
2. Do a non null overlay only (*r.cross -z*) between HRU source map and (1.3) to hold the „HRU to HRU" relation as raster labels
3. Use (2) as base map in *r.statistics* to sum up accumulation data with regards to one and the same destination HRU – see step 4, figure 6
4. Finally overlay again (*r.cross*) to append the accumulation sums (3) to the „HRU to HRU" relation map (1.3)

As is evident, the operations take advantage of *r.cross* twice. Consequently, all required information about relations within the topological sequences is summarized in HRU raster labels up to sample `"category <from_hru>; category <to_hru>; <amount>"`. That proves true when you have a look into the GRASS category file (`/cats` subdirectory) of the result layer:

```
[...]
2:category 10; category 19; 53
3:category 10; category 20; 14
4:category 17; category 39; 141
[...]
```

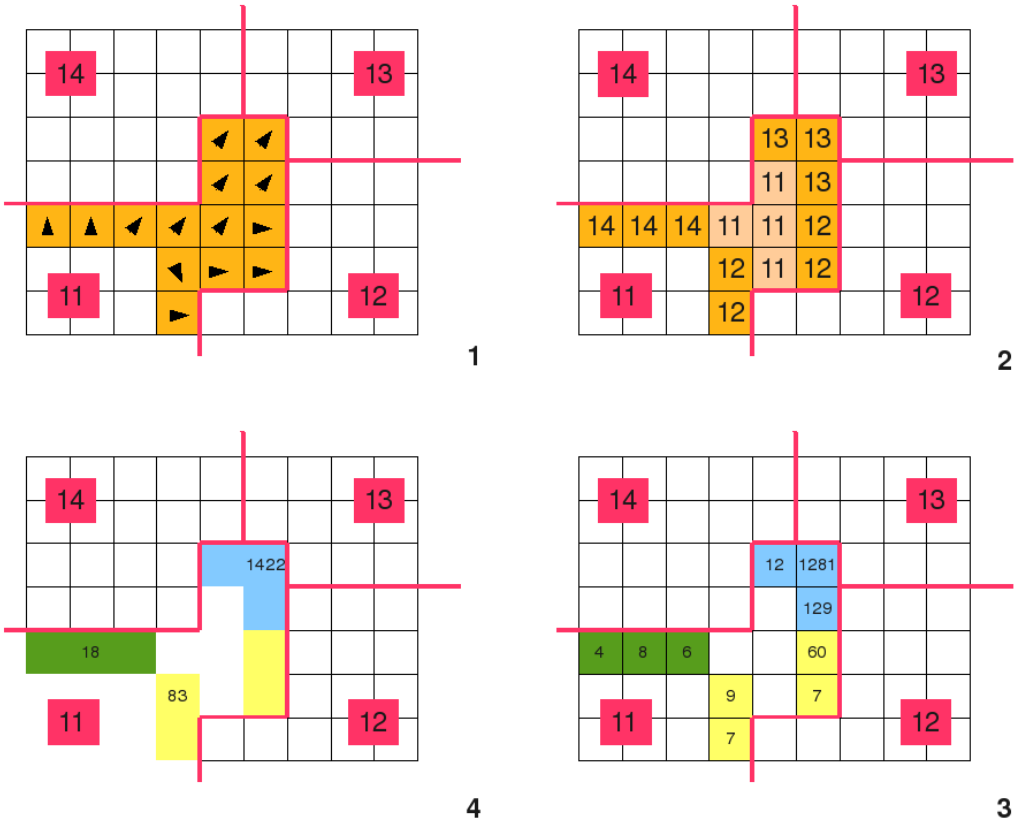

Figure 6: Relation *HRU to HRU*

According to the first two lines, HRU 10 drains into HRUs 19 and 20 to the value of respectively 53 and 14. Using this GRASS category file as an input for a small *awk* script topology information could be easily transformed to a more general format that joins one-to-many HRU relations into one output row:

```
[...]
10    19,53    20,14
17    39,141
[...]
```

As mentioned earlier the topology delineation is separated into two parts: One part was just discussed, the other one is still outstanding. Instead of draining into nearby HRUs it also would be thinkable that water flows directly into any reaches before. The fact implicates some changes in comparison to the prior approach (in figure 7 let's assume that blue lines illustrate the stream network):

"HRU to reach"

1. Do a *r.mapcalc* considering a stream buffer into account – with the objective to get the reaches in which stream neighbour cells flow (see figure 7)
2. Perform nearly the same operations like in *"HRU to HRU"* beginning with (1.4) but

ignore accumulation accurately located on streams

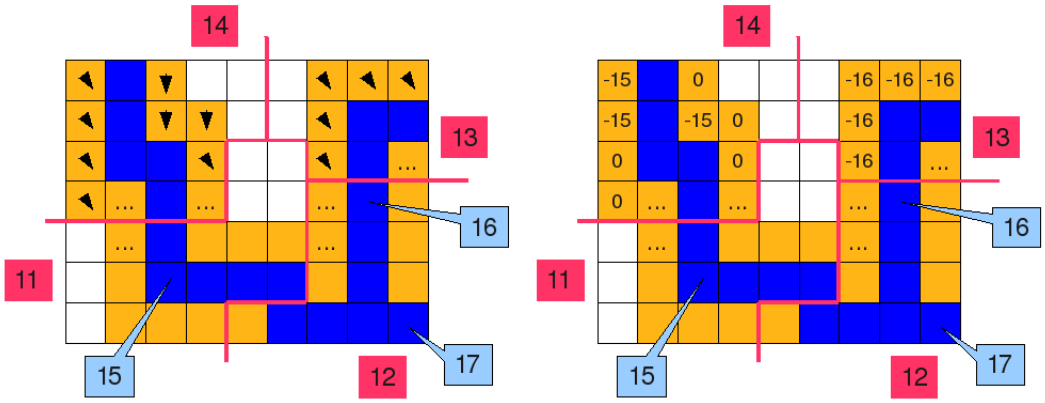

Figure 7: Relation *HRU to reach*

Since step 1 marks reaches as negative numbers to avoid confusions with HRU identifiers the process can carry on with parsing the category file as already done for *"HRU to HRU"*. Concluding work concatenates both into a final and all-embracing topology report. To this end, tools from UNIX command line are employed, for instance *sort* and *join*. Only on that condition meaningful weights (with regard to total flow-out of every HRU) are feasible with few *awk* instructions, i.e.:

```
AWK_calc_weights_in_topo = "'BEGIN {print \"#TOPOLOGY N:M * FORMAT: <Source-HRU> <Dest-HRU>| \
<Dest-Reach>;<Rate>[ <Dest-HRU>|<Dest-Reach>;<Rate> ...]\"} \
                           {for (i = 2; i <= NF; ++i) \
                                  {split($i,a,\",\"); \
                                   sum = sum + a[2]; } \
                            line = $1; \
                            printf line\" \"; \
                            for (i = 2; i <= NF; ++i) \
                                  {split($i,a,\",\"); \
                                   printf a[1]\";\"\"%.3f\"\" \", a[2]/sum; } \
                            sum=0; \
                            print \"\"; \
                            line = \"\";} \
                           END {}'"
```

The ultimate result looks like:

```
[...]
1542    1543;0.640    1655;0.175    1934;0.010    -14;0.004    -12;0.171
1543    875;0.955    1655;0.001    -12;0.044
1547    1165;0.382    1377;0.176    1468;0.029    1629;0.412
1568    1482;1.000
[...]
```

Conclusion

The duration of the whole derivation process in QGIS depends on the size of the selected subregion during initial wizard step (setup). The larger the chosen bounding box, the more noticeable the increase of computing time (see table 1). This is mainly attributed to the water flow oriented section of the wizard using *r.watershed* in the backend. At the expense of

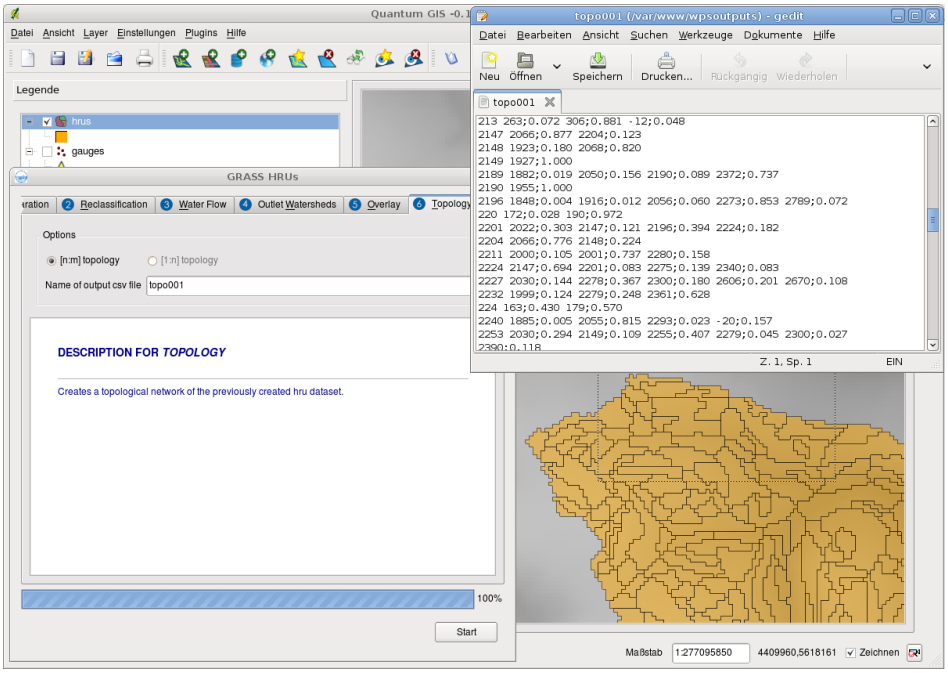

Figure 8: Topology module

execution time the GRASS module yields more accurate maps than *r.terraflow* [7], for which reason it was preferred. However, it remains to check whether [9] may considerably improves the performance of the watershed basin analysis. Or should process implementation changed by substitution with *r.terraflow*, provided that whose output raster maps are barely exact enough for the HRU derivation work? There is also a need for optimization regarding to that part of the overlay algorithm where resulting HRUs are relabeled after removing midget areas. Actually, a simple *r.reclass* statement does the job but not very fast which may affect the total computation time, too.

watercourse, gauge	**catchment size**	**number of HRUs**	**duration**
Erlbach, Thieschitz (Thuringia, GER)	105 km^2	2116	12 min
Hasel, Ellingshausen (Thuringia, GER)	340 km^2	6832	45 min
Gera, Erfurt-Möbisburg (Thuringia, GER)	850 km^2	16696	2.5 h

Table 1 – Performance of the HRU derivation in GRASS using the QGIS extension

References

1. Flügel, W.A. (1995): Delineating Hydrological Response Units by Geographical Information Systemanalyses for regional hydrological modelling using MMS/PRMS in the drainage basin of the river Bröl, Germany. In: Kalma, J.D. & Sivapalan, M. (1995): Scale Issues in Hydrological Modelling. 183-194

2. Leavesley, G.H.; Lichty, R.W.; Troutman, B.M.; Saindon, L.G. (1983): Precipitation-Runoff Modeling System; Users Manual, Denver

3. Neteler, M. and Mitášová H. (2008): Open Source GIS: A GRASS GIS Approach, Third Edition, Springer, ISBN 978-0-387-35767-6

4. Pfennig, B.; Fink M.; Krause P.; Müller Schmied H. (2006): Leitfaden für die Ableitung prozeßorientierter Modelleinheiten (HRU's) für die hydrologische Modellierung

5. Staudenrausch, H. (2001): Untersuchungen zur hydrologischen Topologie von Landschaftsobjekten für die distributive Flußeinzugsgebietsmodellierung. Dissertationsschrift. Jena

6. http://www.qgis.org/ – QuantumGIS

7. http://grass.itc.it/ – Geographic Resources Analysis Support System

8. http://pywps.wald.intevation.org/ – Python Web Processing Service

9. http://markus.metz.giswork.googlepages.com/r.watershed_fast_version.tar.gz – Metz, M. (2008): r.watershed.fast