RichDEM Documentation

Release 0.0.03

Richard Barnes

Sep 19, 2020

Contents

CHAPTER 1

RichDEM

RichDEM is a set of digital elevation model (DEM) hydrologic analysis tools. RichDEM uses parallel processing and state of the art algorithms to quickly process even very large DEMs.

RichDEM offers a variety of flow metrics, such as D8 and D∞. It can flood or breach depressions. It can calculate flow accumulation, slops, curvatures, &c.

RichDEM is available as a performant C++ library, a low-dependency Python package, and a set of command-line tools.

Please cite RichDEM (see below).

1.1 Documentation

Documentation is available at richdem.readthedocs.io. The documentation is auto-generated from the many `README.md` files throughout the codebase and the extensive comments in the source code.

CHAPTER 2

Design Philosophy

The design of RichDEM is guided by these principles:

- **Algorithms will be well-tested.** Every algorithm is verified by a rigorous testing procedure. See below.
- **Algorithms will be fast, without compromising safety and accuracy.** The algorithms used in RichDEM are state of the art, permitting analyses that would take days on other systems to be performed in hours, or even minutes.
- **Algorithms will be available as libraries, whenever possible.** RichDEM is designed as a set of header-only C++ libraries, making it easy to include in your projects and easy to incorporate into other programming languages. RichDEM also includes apps, which are simple wrappers around the algorithms, and a limited, but growing, set of algorithms which may have special requirements, like MPI, that make them unsuitable as libraries. These are available as programs.
- **Programs will have a command-line interface, not a GUI.** Command-line interfaces are simple to use and offer extreme flexibility for both users and programmers. They are available on every type of operating system. RichDEM does not officially support any GUI. Per the above, encapsulating RichDEM in a high-level interface of your own is not difficult.
- **Algorithms and programs will be portable.** Linux, Mac, and Windows should all be supported.
- **The code will be beautiful.** RichDEM's code utilizes sensible variable names and reasonable abstractions to make it easy to understand, use, and design algorithms. The code contains extensive internal documentation which is DOxygen compatible.
- **Programs and algorithms will provide useful feedback.** Progress bars will appear if desired and the output will be optimized for machine parsing.
- **Analyses will be reproducible.** Every time you run a RichDEM command that command is logged and timestamped in the output data, along with the version of the program you created the output with. Additionally, a history of all previous manipulations to the data is kept. Use `rd_view_processing_history` to see this.**

CHAPTER 3

Parsable Output

Every line of output from RichDEM begins with one of the following characters, making it easy to parse with a machine.

Tag	Meaning
A	Algorithm name
a	Analysis command: the command line used to run the program
c	Configuration information: program version, input files, and command line options, &c.
C	Citation for algorithm
d	Debugging info
E	Indicates an error condition
i	I/O: Amount of data loaded from disk
m	Miscallaneous counts
n	I/O: Amount of data transferred through a network
p	Progress information: inform the user to keep calm because we're carrying on.
r	Amount of RAM used
t	Timing information: How long stuff took
W	Indicates a warning

All output data shall have the form:

<INDICATOR TAG> <MESSAGE/MEASUREMENT NAME> [= <VALUE> [UNIT]]

The amount of whitespace may very for aesthetic purposes.

CHAPTER 4

Citing RichDEM

As of 883ea734e957, David A. Wheeler's SLOCCount estimates the value of RichDEM at $240,481 and 1.78 person-years of development effort. This value is yours to use, but citations are encouraged as they provide justification of continued development.

General usage of the library can be cited as:

> Barnes, Richard. 2016. RichDEM: Terrain Analysis Software. http://github.com/r-barnes/richdem

An example BibTeX entry is:

> **@manual{RichDEM,** title = {RichDEM: Terrain Analysis Software}, author = {Richard Barnes}, year = {2016}, url = {http://github.com/r-barnes/richdem}
>
> }

This information will be updated as versioned releases become available.

Although I have written all of the code in this library, some of the algorithms were discovered or invented by others, and they deserve credit for their good work. Citations to particular algorithms will be printed whenever an app, program, or library function is run. Such citations are prefixed by the character `C` and look like:

> C Barnes, R., Lehman, C., Mulla, D., 2014. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. Computers & Geosciences 62, 117–127. doi:10.1016/j.cageo.2013.04.024

A typical academic citation might read as follows:

> > We performed hydrological corrections on our DEM using the Zhou (2016) algorithm implemented in RichDEM (Barnes 2016).

CHAPTER 5

Ways To Use It

5.1 Python package from PyPI

Get the package with:

```
pip3 install richdem
```

And use:

```
import richdem
```

The command:

help(richdem)

provides all the relevant documentation.

5.2 Python package from source

Enter the `wrappers/pyrichdem` directory and run:

```
python3 setup.py install --user
```

5.3 As A Command-line Tool

To get the command-line tools, install the Python package with:

```
pip3 install richdem
```

The command-line tools are all named `rd_*`, so typing `rd_` on your command- line and hitting tab a few times should give you the full list of what's available.

5.4 As A Library

As an overview, upon compilation, point your library search path to the `richdem/include` directory. Include various files using, e.g.

```
#include "richdem/common/Array2D.hpp"
```

All files include extensive documentation. At this stage the location of certain functions may be subject to change. This will be noted in the `NEWS` file. (TODO)

More concretely, there are a number of compilation options to consider. With the GCC compiler the flag `FLAG` is activated by passing the `-DFLAG` command-line argument.

- `NOPROGRESS` turns off progress bars
- `RICHDEM_DEBUG` turns on line numbers and filenames in RichDEM's output
- `RICHDEM_LOGGING` turns on outputs such as notices about memory allocation, intermediate products, various progress indicators, and so on. Enabling this requires inclusion of the `richdem.cpp` file.
- `RICHDEM_GIT_HASH`. this should be set to the git hash of the code you checked out. A value of `RICHDEM_GIT_HASH=$(git rev-parse HEAD)` is usually good.
- `RICHDEM_COMPILE_TIME`. Date and time the code was compiled. A value of `RICHDEM_COMPILE_TIME=$(date -u +'%Y-%m-%d %H:%M:%S UTC')` is usually good.
- `USEGDAL`. Indicates that GDAL functionality should be included in the library. This allows reading/writing rasters from various file types. It also complicates compilation slightly, as discussed below.
- `NDEBUG` turns off a bunch of range-checking stuff included in the standard library. Increases speed slightly, butm akes debugging crashes and such more difficult.

Setting up compilation works like this:

```
CXXFLAGS="--std=c++11 -g -O3 -Wall -Wno-unknown-pragmas -Irichdem/include"
CXXFLAGS="$CXXFLAGS -DRICHDEM_LOGGING"
```

C++11 or higher is necessary to compile. Include other RichDEM flags as desired. Note that the `-g` flag doesn't slow things down, though it does increase the size of your executable somewhat. It's inclusion is always recommended for anything other than distributed production code because it makes debugging much easier. The `-O3` flag should be replaced by an optimization level or set of your choice. `-Wno-unknown-pragmas` hides warning messages from OpenMP if you choose not to compile with it. `-Wall` produces many helpful warning messages; compiling without `-Wall` is foolish. `-Irichdem/include` connects your code with RichDEM.

If you plan to use GDAL, include the following:

```
GDAL_LIBS="`gdal-config --libs`"
GDAL_CFLAGS="`gdal-config --cflags` -DUSEGDAL"
LIBS="$GDAL_LIBS"
CXXFLAGS="$CXXFLAGS $GDAL_CFLAGS"
```

If you plan to use RichDEM's parallel features include the following:

LIBS="$LIBS -fopenmp"

Finally, put it all together:

```
g++ $CXXFLAGS -o my_program.exe my_program.cpp $LIBS
```

5.5 As A Handy Collection of Tools

Running `make` in the `apps` directory will produce a large number of useful scripts which are essentially wrappers around standard uses of the RichDEM libraries. The apps/README.md file and the apps themselves contain documentation explaining what they all do.

5.6 For Processing Large Datasets

The `programs` directory contains several programs which have not been converted to libraries. This is usually because their functionality is specific and they are unlikely to be useful as a library. Each directory contains a makefile and a readme explaining the purpose of the program.

CHAPTER 6

Concepts

6.1 Gridded Data

RichDEM assumes that data is provided in the form of a rectangular grid of cells with some *width* and *height*. Furthermore, the data comprising this grid must be laid out in a flat array such that the value of any cell `(x,y)` can be accessed via the equation `y*width+x`. This is known as **row-major** ordering.

Data can be passed to RichDEM in one of three ways.

1. Data can be loaded via GDAL. GDAL handles the heavy lifting of ensuring that data is loaded in a form which complies with the above assumptions.
2. Data can be manually added to a `richdem::Array2D` object (C++) or a `richdem.rdarray` object (Python).
3. A `richdem::Array2D` (C++) or `richdem.rdarray` (Python) object can be used to wrap existing row-major memory. This capability allows RichDEM to easily integrate with existing code.

6.2 Metadata

A RichDEM array is accompanied by several kinds of metadata. These can be loaded by GDAL or specified manually.

- A **NoData value**, as discussed below.
- A **projection**. This is, typically, a PROJ4 or WKT string that identifies the projection the data maps to. The choice of projection does not affect RichDEM's operations.
- A **geotransform**. This is a six element array which determines where in a projection a RichDEM array's data is located, as well as the cell sizes. Further details are below. This setting does affect how RichDEM processes data.
- A **metadata** entry which contains arbitrary metadata strings such as `PROCESSING_HISTORY` (see below).

6.3 Geotransform

A **geotransform** is a six element array which determines where in a projection a RichDEM array's data is located, as well as the cell sizes.

Typically, the geotransform is an affine transform consisting of six coefficients which map pixel/line coordinates into a georeferenced space using the following relationship:

Xgeo = GT(0) + Xpixel*GT(1) + Yline*GT(2) Ygeo = GT(3) + Xpixel*GT(4) + Yline*GT(5)

In case of north up images, the `GT(2)` and `GT(4)` coefficients are zero, and the `GT(1)` is pixel width, and `GT(5)` is pixel height. The `(GT(0),GT(3))` position is the top left corner of the top left pixel of the raster.

Note that the pixel/line coordinates in the above are from `(0.0,0.0)` at the top left corner of the top left pixel to (width_in_pixels,height_in_pixels) at the bottom right corner of the bottom right pixel. The pixel/line location of the center of the top left pixel would therefore be `(0.5,0.5)`

(Text drawn from GDAL documentation.)

6.4 NoData values

RichDEM recognizes cells with a *NoData* value and treats them in special ways. The *NoData* value is a number, such as `-9999` which represents a cell that isn't part of a data set.

In depression-filling and the determination of flow directions, *NoData* cells are treated as having a lower elevation than any other cell, which enforces drainage to the edge of the DEM.

Because *NoData* values fundamental affect operations, RichDEM requires that you specify what *NoData* value it should use.

It is important that you choose a value that doesn't correspond to any of your actual data values. In 1-byte/8-bit DEMs this may not be possible since there are only 256 distinct values available. In this case, you should cast your data to a 2-byte/16-bit form and choose a *NoData* value that is not in the range `0-255`.

Other terrain analysis programs may use a binary masking array to indicate which cells are *NoData*. RichDEM does not do this because usually a minority of cells are *NoData* and a separate array increases the amount of memory used and reduces cache utilization, both of which may reduce performance.

Language	Set	Get
Python	`rda.no_data=-1`	`rda.no_data`
C++	`rda.setNoData(-1)`	`rda.noData( )`

6.5 Processing History

RichDEM automagically keeps track of what operations are performed on your data. This means that your outputs will contain an exact record of what operations were used to produce them. This helps ensure reproducibility. And, remember, good science is reproducible science.

In the following, a series of operations is performed and the Processing History is then examined.

```
import richdem as rd
dem   = rd.LoadGDAL("../data/beauford.tif")
rd.FillDepressions(dem, epsilon=False, in_place=True)
accum = rd.FlowAccumulation(dem, method='D8')
```

(continues on next page)

(continued from previous page)

```
rd.SaveGDAL('/z/out.tif', accum)
print(accum.metadata['PROCESSING_HISTORY'])
```

The processing history of a saved dataset can be viewed using a few different commands:

```
gdalinfo /z/out.tif
rd_info  /z/out.tif
```

The processing history appears as follows.

```
2017-12-20 17:55:19.892388 UTC | RichDEM (Python 0.0.4) (hash=e02d5e2, hashdate=2017-
↪12-19 23:52:52 -0600) | LoadGDAL(filename=../data/beauford.tif, no_data=-9999.0)
2017-12-20 17:55:19.900234 UTC | RichDEM (Python 0.0.4) (hash=e02d5e2, hashdate=2017-
↪12-19 23:52:52 -0600) | FillDepressions(dem, epsilon=False)
2017-12-20 17:55:20.514098 UTC | RichDEM (Python 0.0.4) (hash=e02d5e2, hashdate=2017-
↪12-19 23:52:52 -0600) | FlowAccumulation(dem, method=D8)
```

Note that the **first column** is the time at which the operation was performed, the **second column** is the program which performed the operation, and the **third column** is the command which was run.

Language	Command
Python	`rda.metadata`
C++	`rda.metadata`

6.6 In-Place Operations

To save memory RichDEM performs some operations, such as depression-filling, in place. This means that the data is modified and the original data will be lost unless it has been copied.

For instance, in Python `FillDepressions` has two distinct forms:

```
#In-place filling, no return value
rd.FillDepressions(dem, in_place=True)
#Fill a copy
dem_filled = rd.FillDepressions(dem, in_place=False)
```

whereas in C++, a copy must be made:

```
#In-place filling, no return value
richdem::FillDepressions(dem)

#Fill a copy
auto demcopy = dem; //TODO: Make sure this syntax is right
richdem::FillDepressions(demcopy)
```

6.7 Topology

RichDEM offers two topologies, though not all functions differentiate between them. Thse are:

- **D8: The cells are arranged in a regular, rectilinear grid. Each cell** connects with each of its neighbouring cells.

- **D4: The cells are arranged in a regular, rectilnear grid. Each cell** connects with the cells to its north, south, east, and west (the cells up, down, left, and right of it).

In C++, the foregoing topologies are accessed via the `Topology` enumeration, similar to the following:

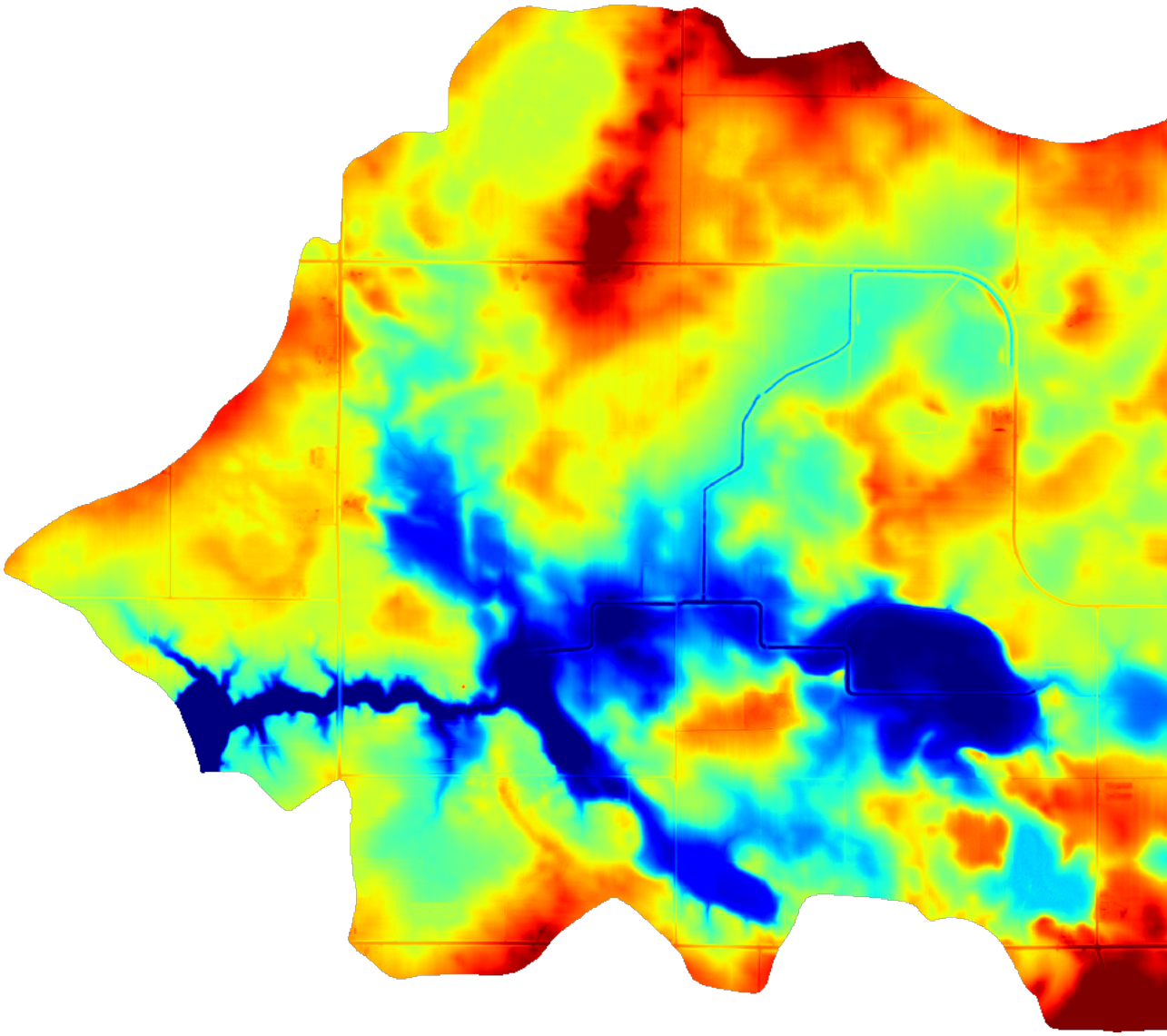
```
FillDepressions<Topology::D8>(dem);
FillDepressions<Topology::D4>(dem);
```

CHAPTER 7

Example DEMs

Beauford Watershed, Minnesota, USA is frqeuently used as an example dataset.

```
import richdem as rd
import numpy as np

beau = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)

rd.rdShow(beau, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```



CHAPTER 8

Loading Data

8.1 Python

Data can be loaded in several ways.

To load from disk, if GDAL is available on your system, almost any form of raster data can be easily loaded, like so:

8.1.1 GDAL

```
import richdem as rd
beau = rd.LoadGDAL("beauford.tif")
```

8.1.2 NumPy

Data can also be loaded from a NumPy array:

```
import numpy as np
import richdem as rd

npa = np.random.random(size=(50,50))
rda = rd.rdarray(npa, no_data=-9999)
```

Note that !‘rd.rdarray()‘ creates a *view* of the data stored in !‘npa‘. Modifying `rda` will modify `npa`. This prevents unwanted memory from being used. If you instead want `rda` to be a new copy of the data, use:

```
rda = rd.rdarray(a, no_data=-9999)
```

8.1.3 Saved NumPy Arrays

It is possible to save, and load, data to and from a NumPy array like so:

```
import numpy as np
import richdem as rd

npa = np.random.random(size=(50,50))
rda = rd.rdarray(npa, no_data=-9999)
np.save('out.npy', rda)
loaded = rd.rdarray(np.load('out.npy'), no_data=-9999)
```

This can be done in a compressed format like so:

```
np.savez('rda', rda=rda)
np.load('rda.npz')['rda']
```

Note that there is not yet a way to save the metadata of an rdarray. (TODO)

8.2 C++

TODO

CHAPTER 9

Depression-Filling

Depressions, otherwise known as pits, are areas of a landscape wherein flow ultimately terminates without reaching an ocean or the edge of a digital elevation model.

9.1 Depressions, Pits, and Sinks

Depressions have been called by a variety of names. To clarify this mess, Lindsay (2016) provides a typology. This typology is followed here.

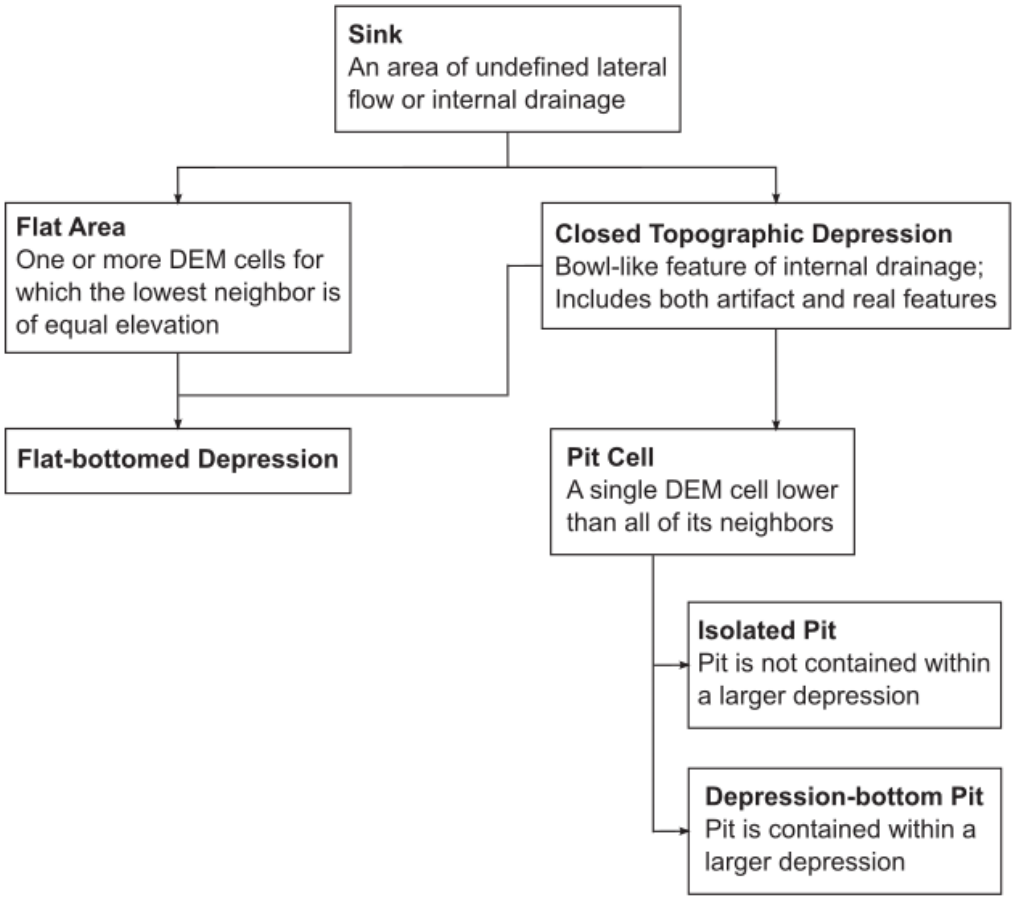


Figure 1. A typology of features found within DEMs that interrupt modelled flow paths and require flow enforcement

9.2 Original DEM

For reference, the original DEM appears as follows:

```
import richdem as rd
import numpy as np

beau    = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)
beaufig = rd.rdShow(beau, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```

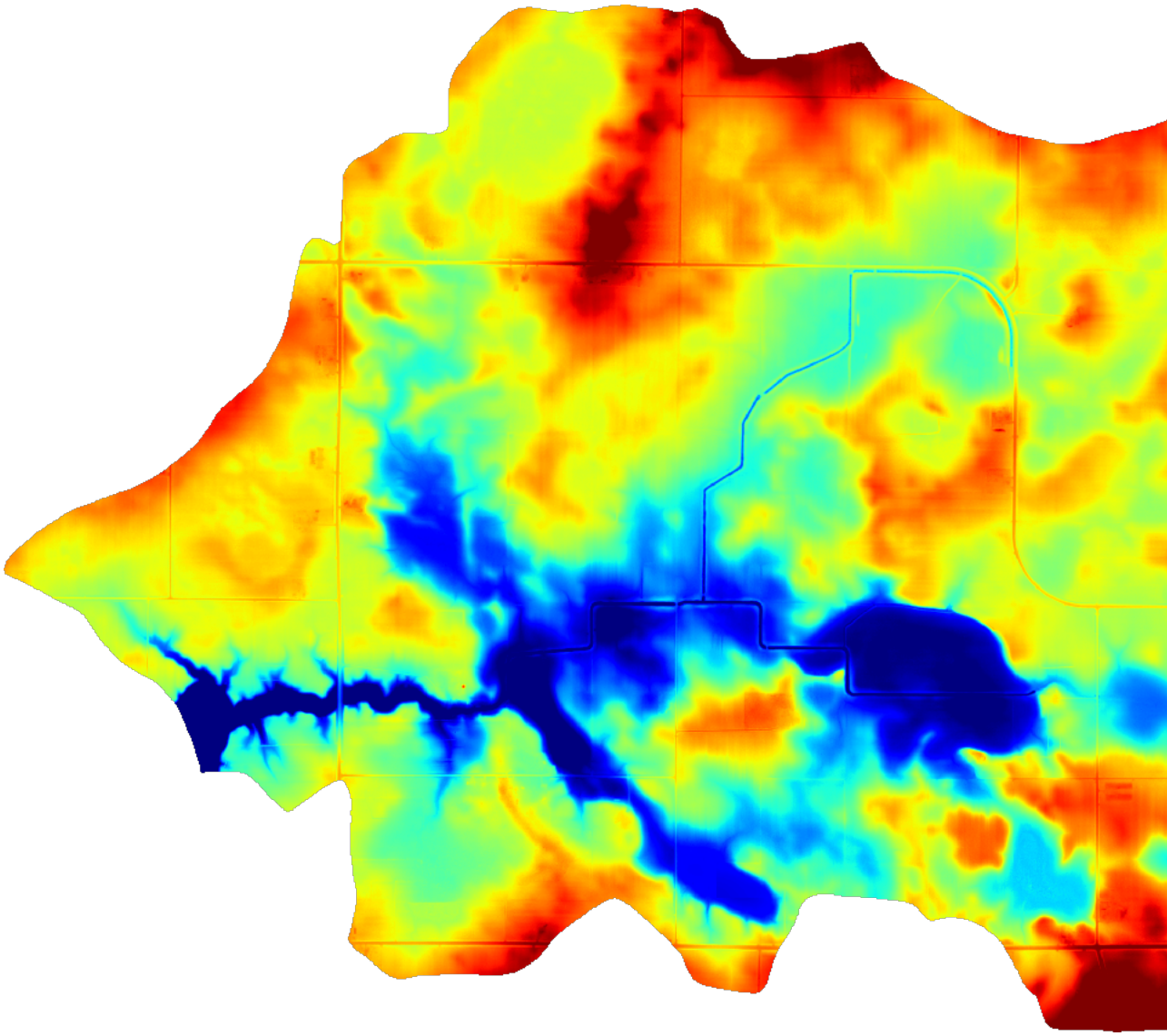
9.3 Complete Filling

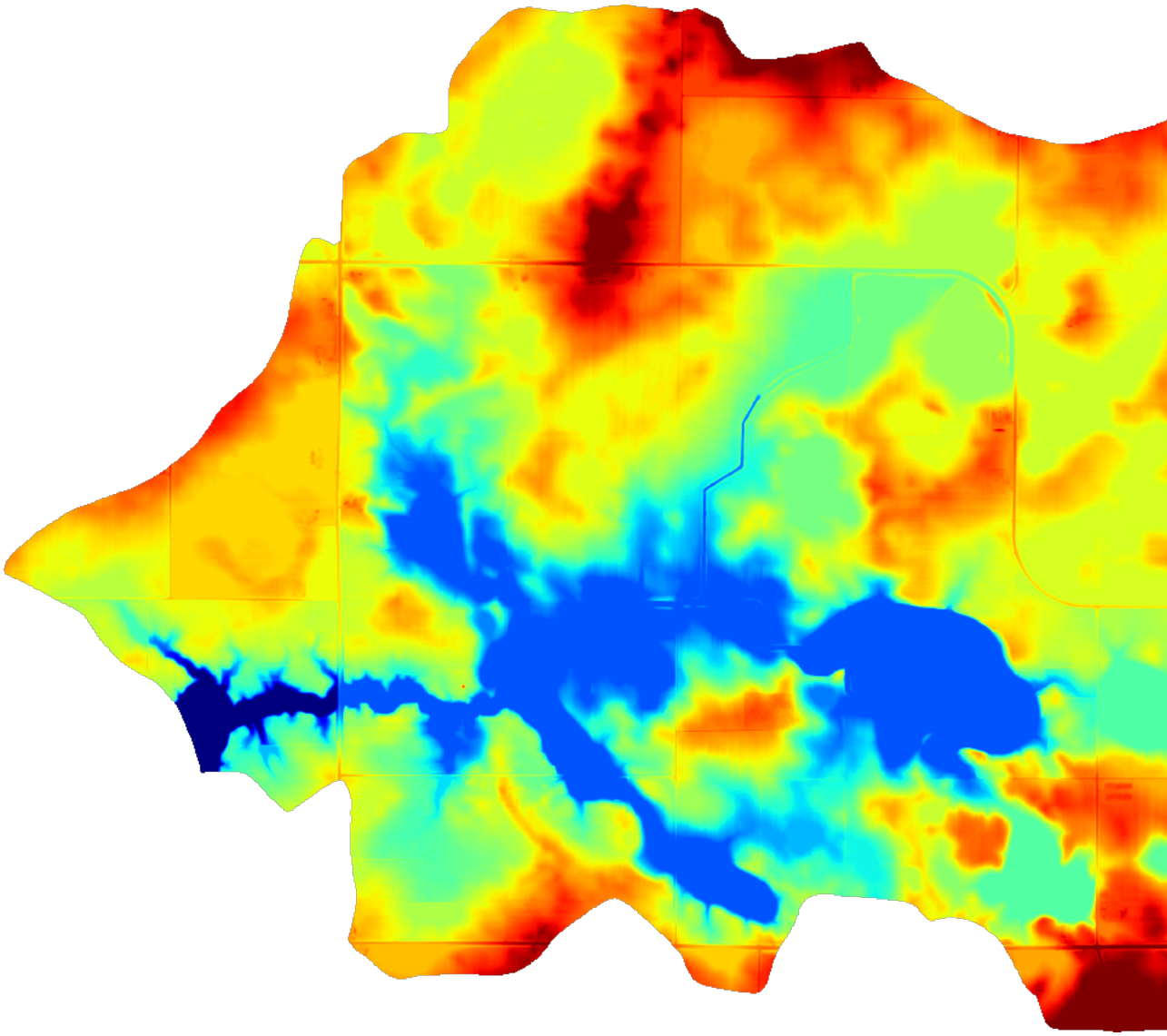
Depression-filling is often used to fill in all the depressions in a DEM to the level of their lowest outlet or spill-point.

The result looks as follows:

```
beau_filled    = rd.FillDepressions(beau, in_place=False)
beaufig_filled = rd.rdShow(beau_filled, ignore_colours=[0], axes=False, cmap='jet',␣
↪vmin=beaufig['vmin'], vmax=beaufig['vmax'], figsize=(8,5.5))
```

We can visualize the difference between the two like so:





```
beau_diff     = beau_filled - beau
beaufig_diff = rd.rdShow(beau_diff, ignore_colours=[0], axes=False, cmap='jet',␣
↪figsize=(8,5.5))
```

Complete Filling is available via the following commands:

Language	Command
Python	`richdem.FillDepressions`
C++	`richdem::FillDepression<Topology>`

Pros	Cons
• *Fast* • Simple	• Leaves flat regions behind • May modify large portions of a DEM

9.4 Epsilon Filling

A downside of complete filling is that it replaces depressions with a perfectly flat region with no local gradients. One way to deal with this is to ensure that every cell in the region is raised some small amount, ϵ, above cells which are closer to a depression's spill point.

This must be done carefully. In floating-point DEMs, the value ϵ is non-constant and must be chosen using the !'std::nextafter' function. If a depression is too large, the imposed gradient may result in the interior of the depression being raised above the surrounding landscape. Using `double` instead of `float` reduces the potential for problems at a cost of twice the space used. If a problem does arise, RichDEM provides a warning.

We can visualize the difference between the epsilon-filled DEM and the original DEM like so:

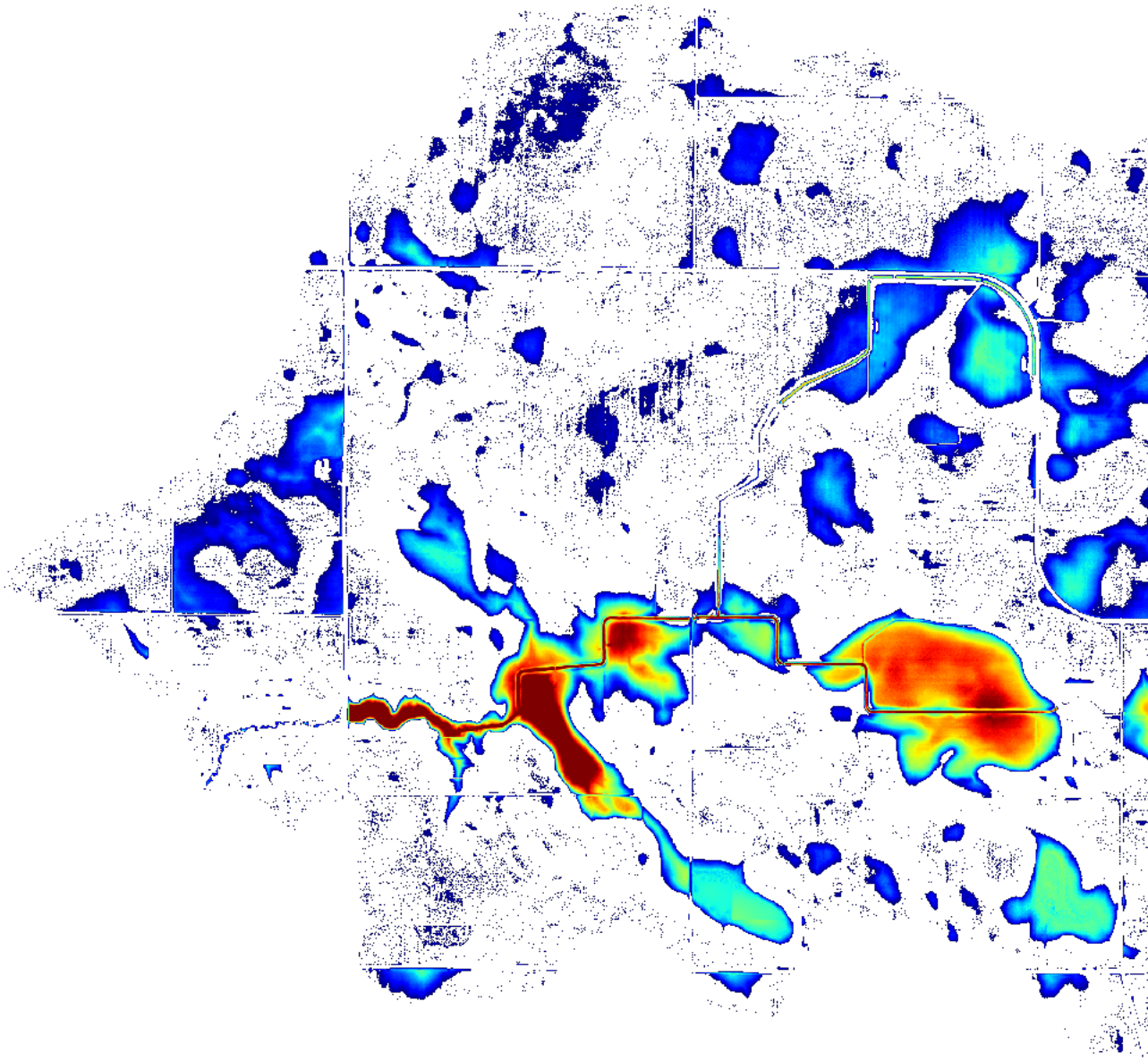
```
beau_epsilon          = rd.FillDepressions(beau, epsilon=True, in_place=False)
beau_eps_diff         = beau_epsilon - beau
beaufig_eps_diff      = rd.rdShow(beau_eps_diff, ignore_colours=[0], axes=False, cmap=
↪'jet', figsize=(8,5.5))
```

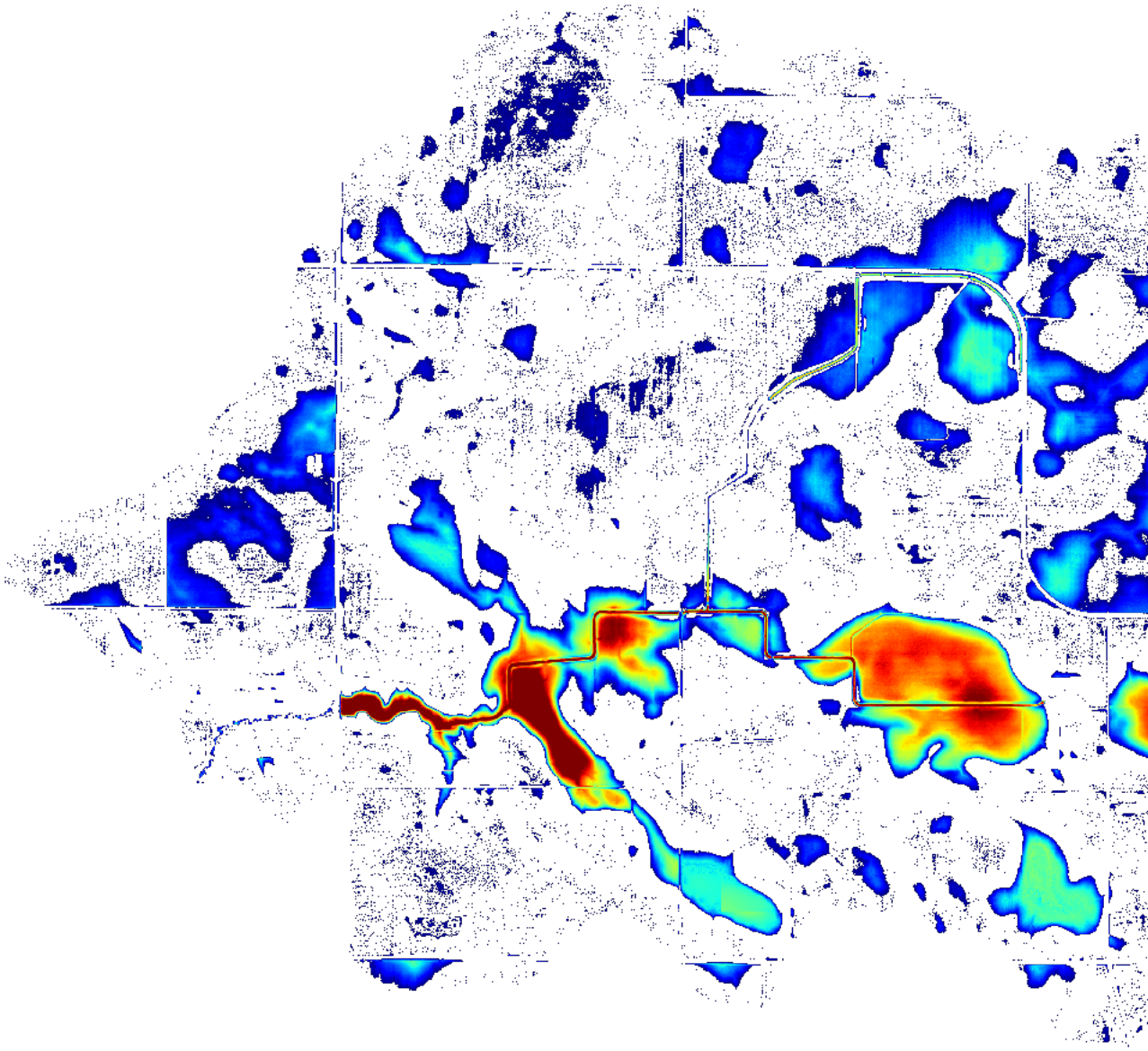
We can visualize the difference between the epsilon-filled DEM and the completely-filled DEM as follows. Note that elevation increases with distance from the depression's outlet: this is the effect of the epsilon.

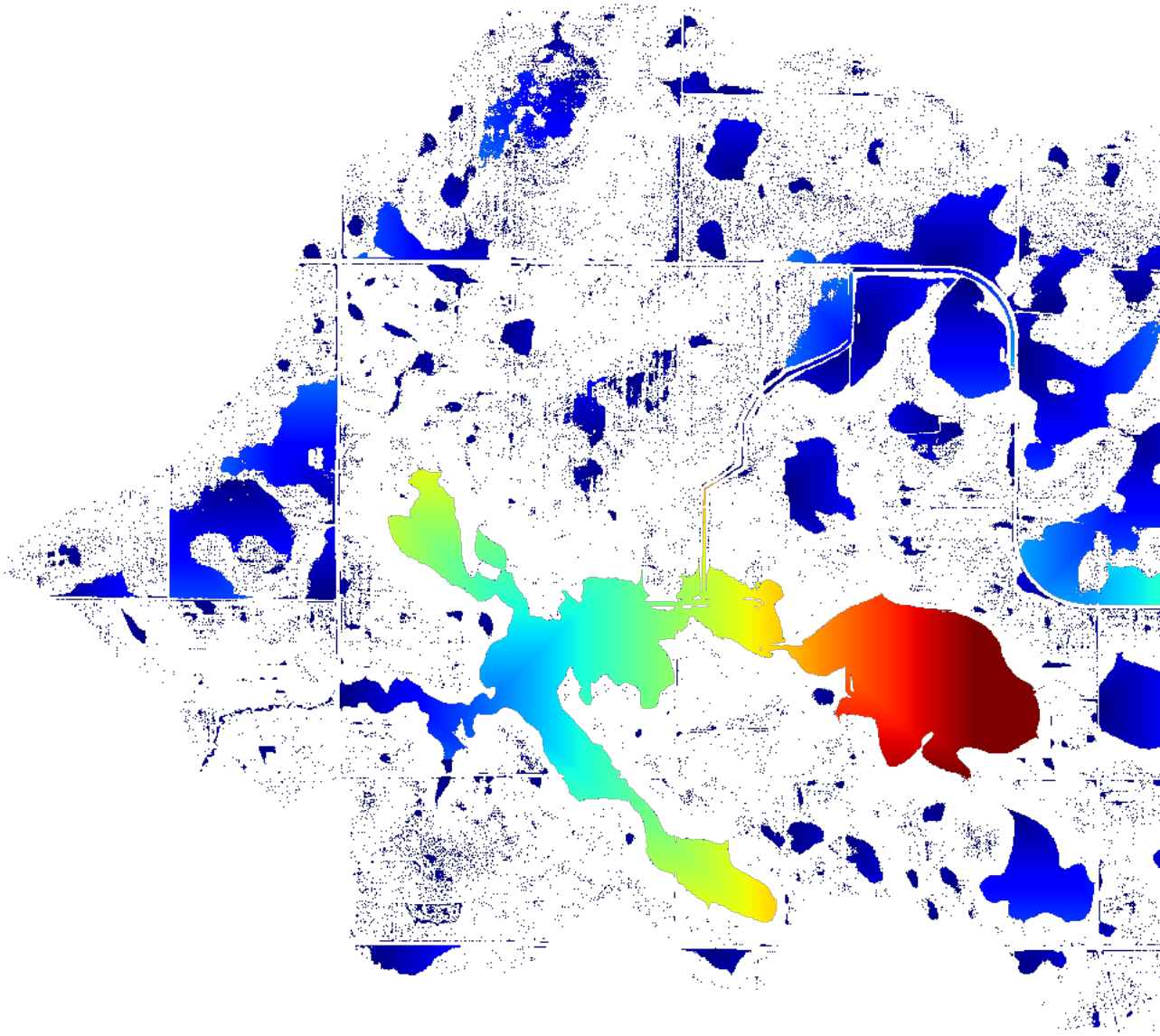
```
beau_diffeps_diff     = beau_epsilon - beau_filled
beaufig_diffeps_diff = rd.rdShow(beau_diffeps_diff, ignore_colours=[0], axes=False,␣
↪cmap='jet', figsize=(8,5.5))
```

Language	Command
Python	`richdem.FillDepressions`
C++	*`richdem::FillDepressionsEpsilon<Topology>()`*

Pros	Cons
• All cells drain	• Not as fast as simple depression filling • May modify large portions of a DEM • May create elevated regions • Success may depend on data type







CHAPTER 10

Depression-Breaching

Depressions, otherwise known as pits, are areas of a landscape wherein flow ultimately terminates without reaching an ocean or the edge of a digital elevation model.

10.1 Depressions, Pits, and Sinks

Depressions have been called by a variety of names. To clarify this mess, Lindsay (2016) provides a typology. This typology is followed here.

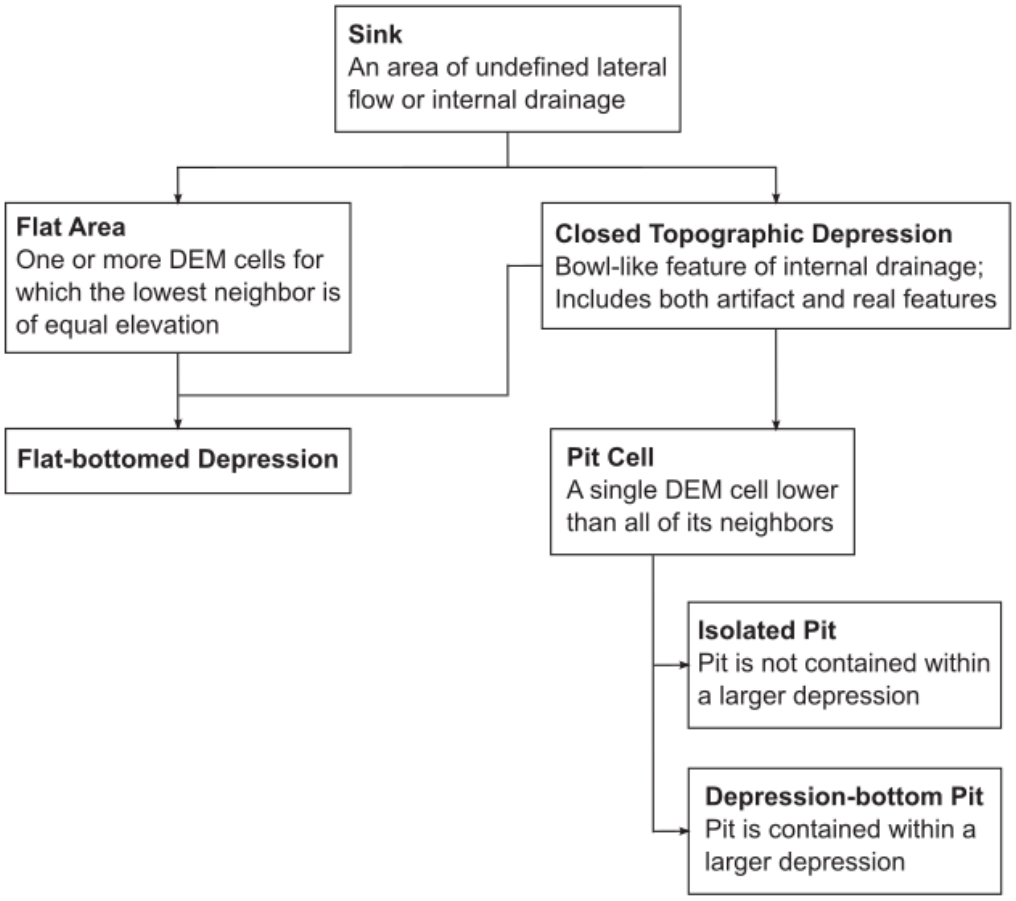


Figure 1. A typology of features found within DEMs that interrupt modelled flow paths and require flow enforcement

10.2 Original DEM

For reference, the original DEM appears as follows:

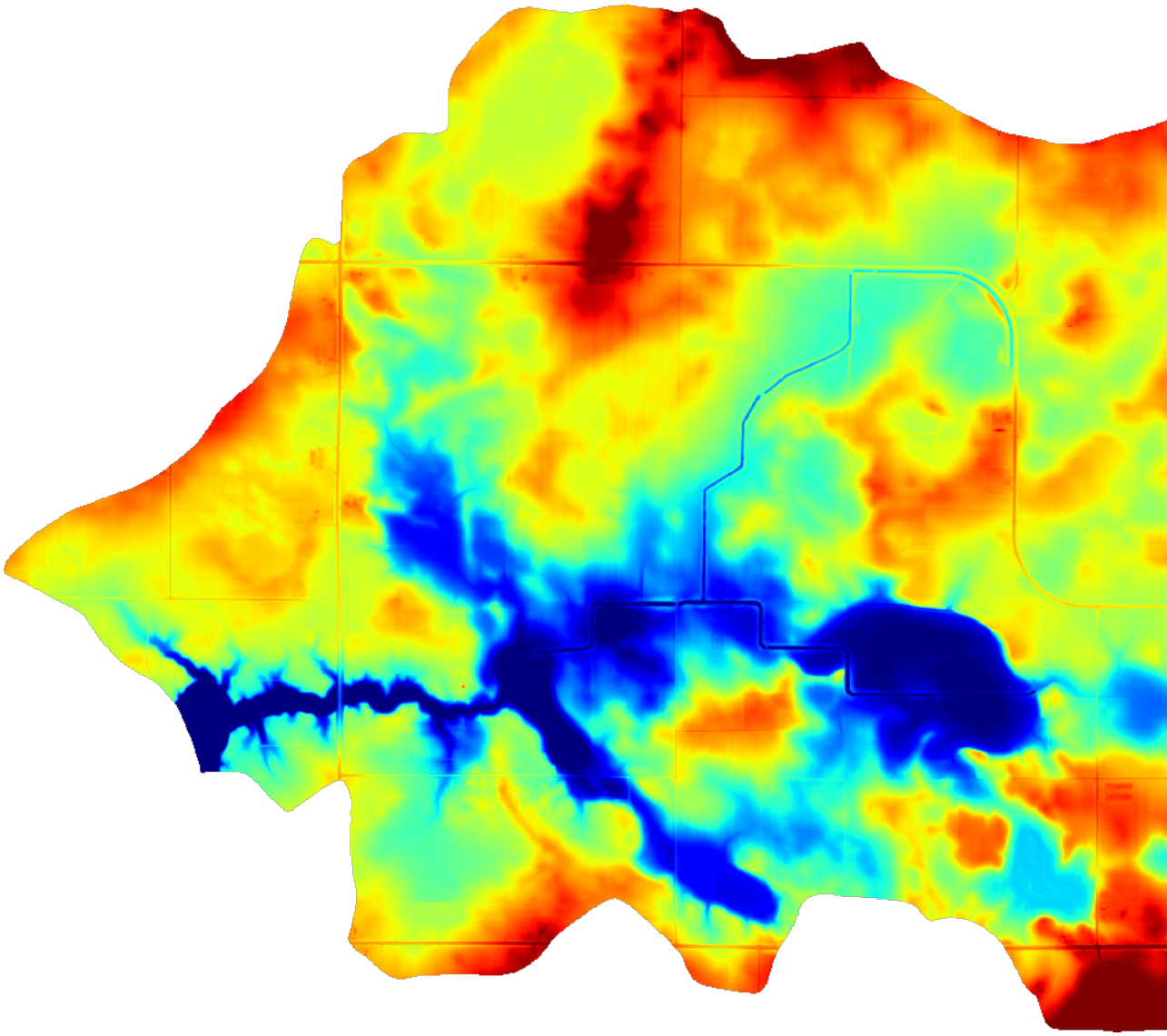
```
import richdem as rd
import numpy as np

beau    = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)
beaufig = rd.rdShow(beau, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```

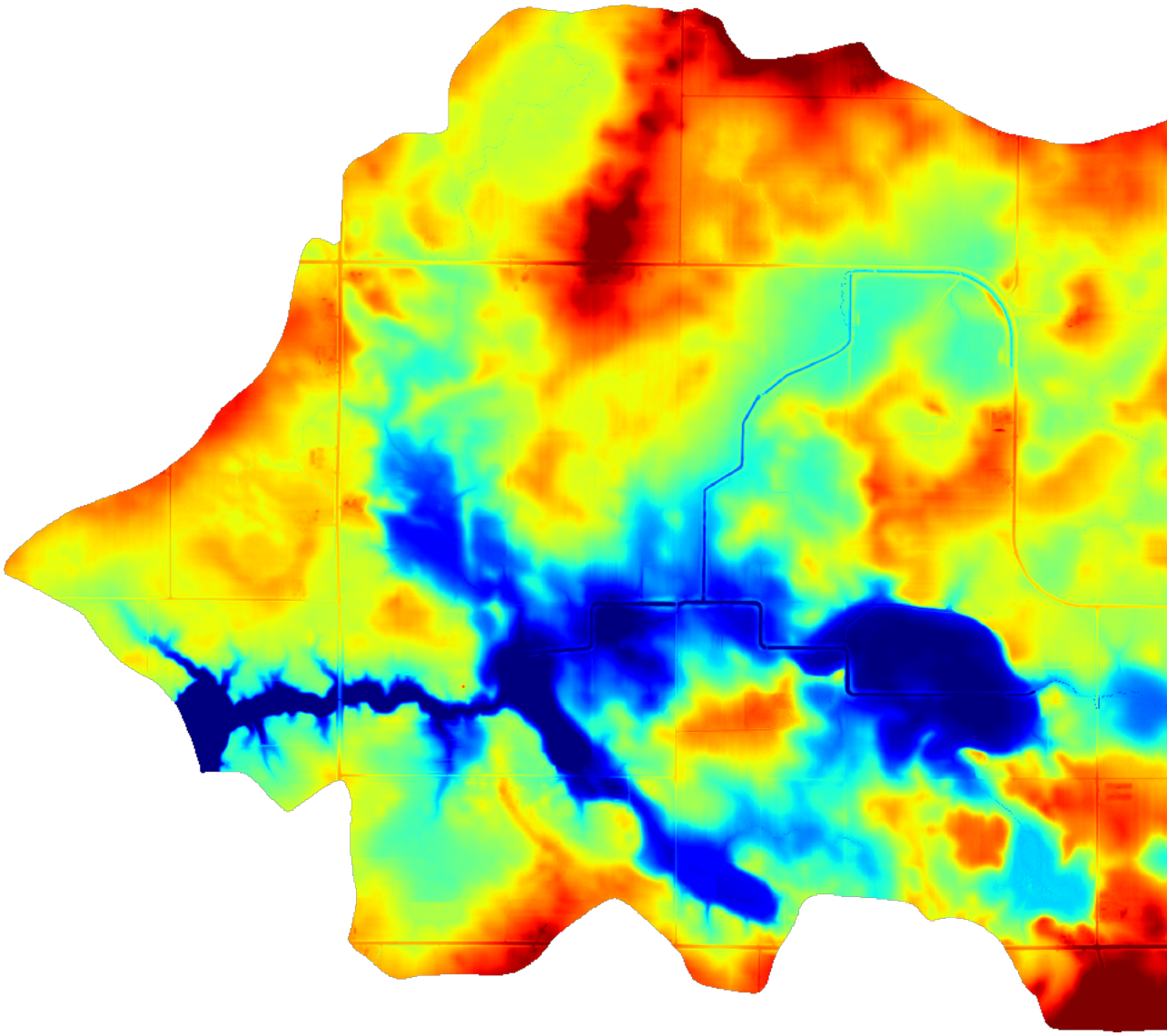
10.3 Complete Breaching

Depression-breaching is used to dig channels from the pit cells of a DEM to the nearest cells (in priority-flood sense) outside of the depression containing the pit. This resolves the depression as all cells in the depression now have a drainage path to the edge of the DEM.

The result looks as follows:

```
beau_breached    = rd.BreachDepressions(beau, in_place=False)
beaufig_breached = rd.rdShow(beau_breached, ignore_colours=[0], axes=False, cmap='jet
↪', vmin=beaufig['vmin'], vmax=beaufig['vmax'], figsize=(8,5.5))
```





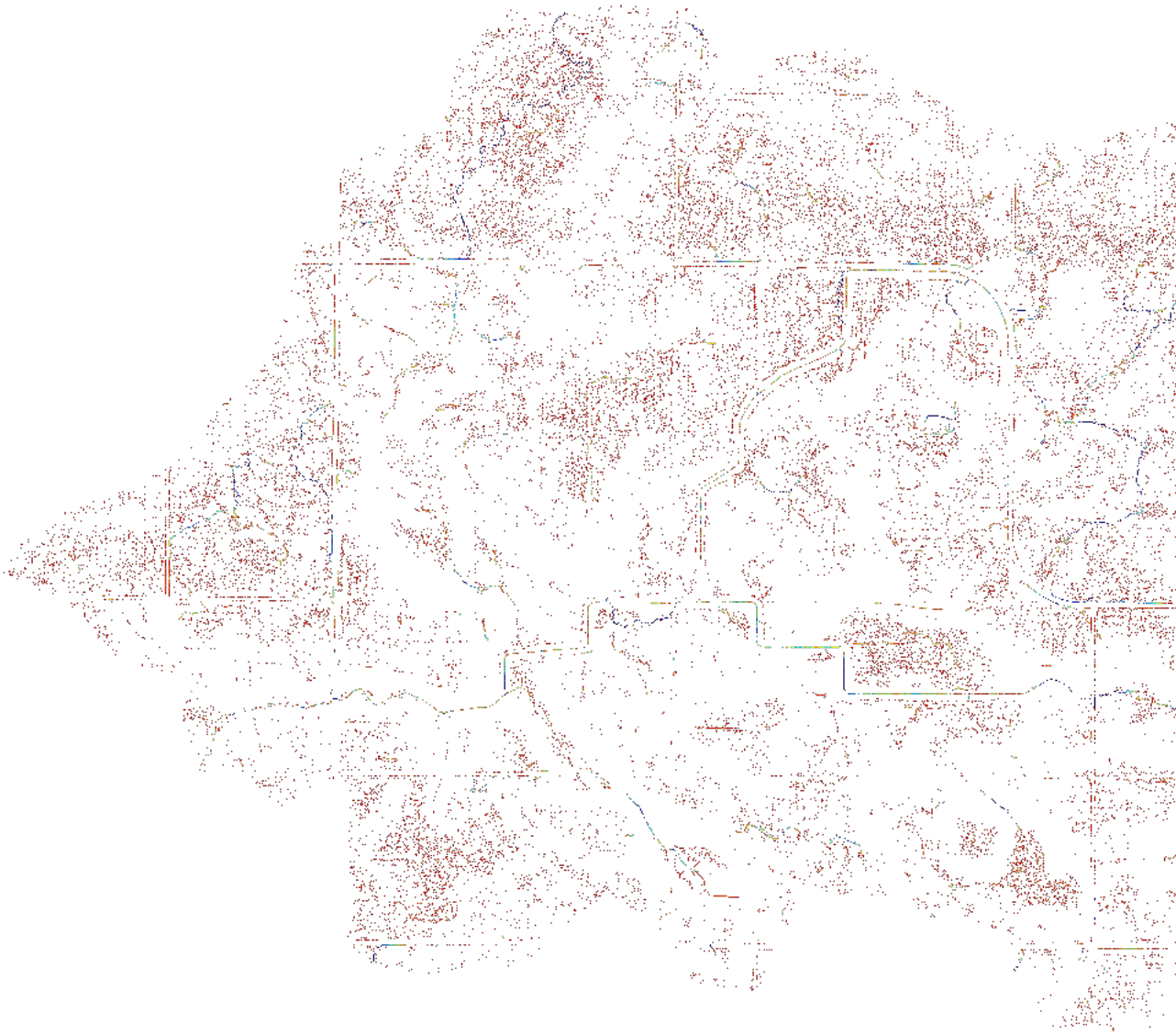
We can visualize the difference between the two like so:

```
beau_diff    = beau_breached - beau
beaufig_diff = rd.rdShow(beau_diff, ignore_colours=[0], axes=False, cmap='jet',␣
↪figsize=(8,5.5))
```

Complete Breaching is available via the following commands:

Language	Command
Python	`richdem.BreachDepressions`
C++	*`richdem::BreachDepressions<Topology>()`*

Pros	Cons
• Minimal Modifcations to DEM • Simple	• Slightly slower



CHAPTER 11

Flat Resolution

The problem with doing complete filling on flats is that the resulting DEM contains mathematically flat areas with no local gradient. This makes it impossible to determine flow directions for these areas.

Imposing an epsilon gradient during depression-filling is one solution as discussed in Epsilon Filling (*Epsilon Filling*); however, the gradient produced may appear unaesthetic because drainage takes a least-distance route to the flat's edges.

We stress the word **aesthetic** here since, after depression-filling, no information about local gradients remains from the original DEM, so, in a sense, all reconstructed drainage patterns are equally silly. Sometimes, though, this is the best you can do.

This pages discusses alternatives.

11.1 Barnes (2014) Flat Resolution

> Barnes, R., Lehman, C., Mulla, D., 2014a. An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Computers & Geosciences 62, 128–135. doi:10.1016/j.cageo.2013.01.009

The Barnes (2014) flat resolution algorithm routes flow both towards the edges of flats and away from the high areas surrounding them. The result is an aesthetically pleasing drainge pattern.

It can do so either by adjust the elevations of the DEM's cells or by adjusting flow metrics derived from the DEM.

11.1.1 Elevation Adjustment

In this method, the elevation of a DEM can be adjusted so that every cell in the region is raised some small amount, ϵ, above cells which are closer to a depression's spill point and farther from its surrounding high areas.

This must be done carefully. In floating-point DEMs, the value ϵ is non-constant and must be chosen using the !'std::nextafter' function. If a depression is too large, the imposed gradient may result in the interior of the depression being raised above the surrounding landscape. Using `double` instead of `float` reduces the potential for problems at a cost of twice the space used. If a problem does arise, RichDEM provides a warning.

Recall from Epsilon Filling (*Epsilon Filling*) that an epsilon gradient imposed during depression-filling results in an elevation adjustment that looks like this:

```
import richdem as rd
import numpy as np

#Load dataset
beau        = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)

#Fill the depression entirely
beau_filled = rd.FillDepressions(beau, epsilon=False, in_place=False)

#Construct the epsilon drainage surface via filling
beau_eps    = rd.FillDepressions(beau, epsilon=True, in_place=False)

diff    = beau_eps - beau_filled
rd.rdShow(diff, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```

In contrast, the Barnes (2014) convergent elevation adjustment looks like this:

```
#Resolve flats by imposing a convergent epsilon gradient
beau_flat_eps = rd.ResolveFlats(beau_filled, in_place=False)

diff    = beau_flat_eps - beau_filled
rd.rdShow(diff, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```

The difference versus the depression-filling epsilon adjustment appears as follows. Note the deep V-shaped notches in the flats indicating the increased convergence of the Barnes (2014) method.

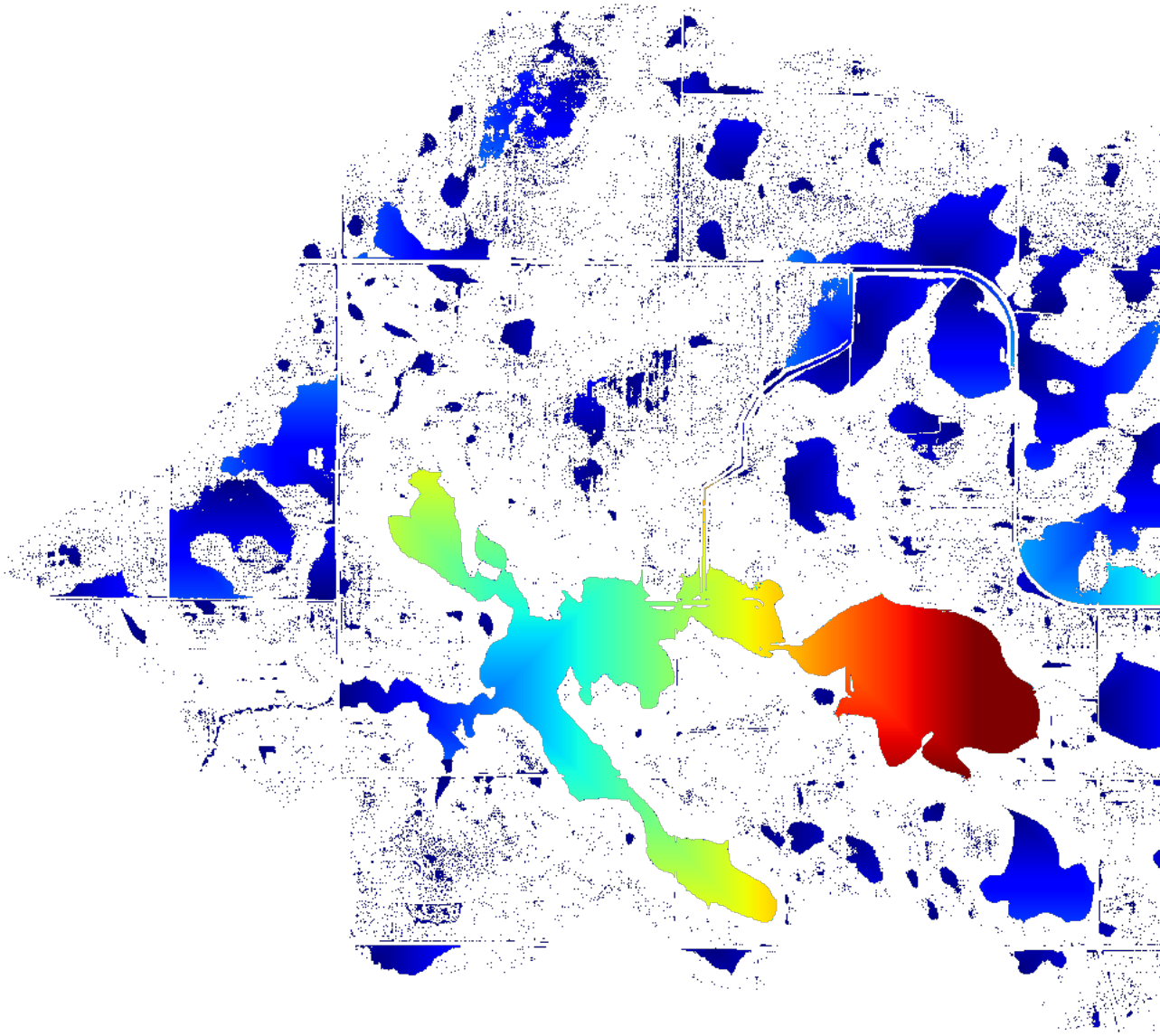
```
diff    = beau_flat_eps - beau_eps
rd.rdShow(diff, ignore_colours=[0], axes=False, cmap='jet', figsize=(8,5.5))
```

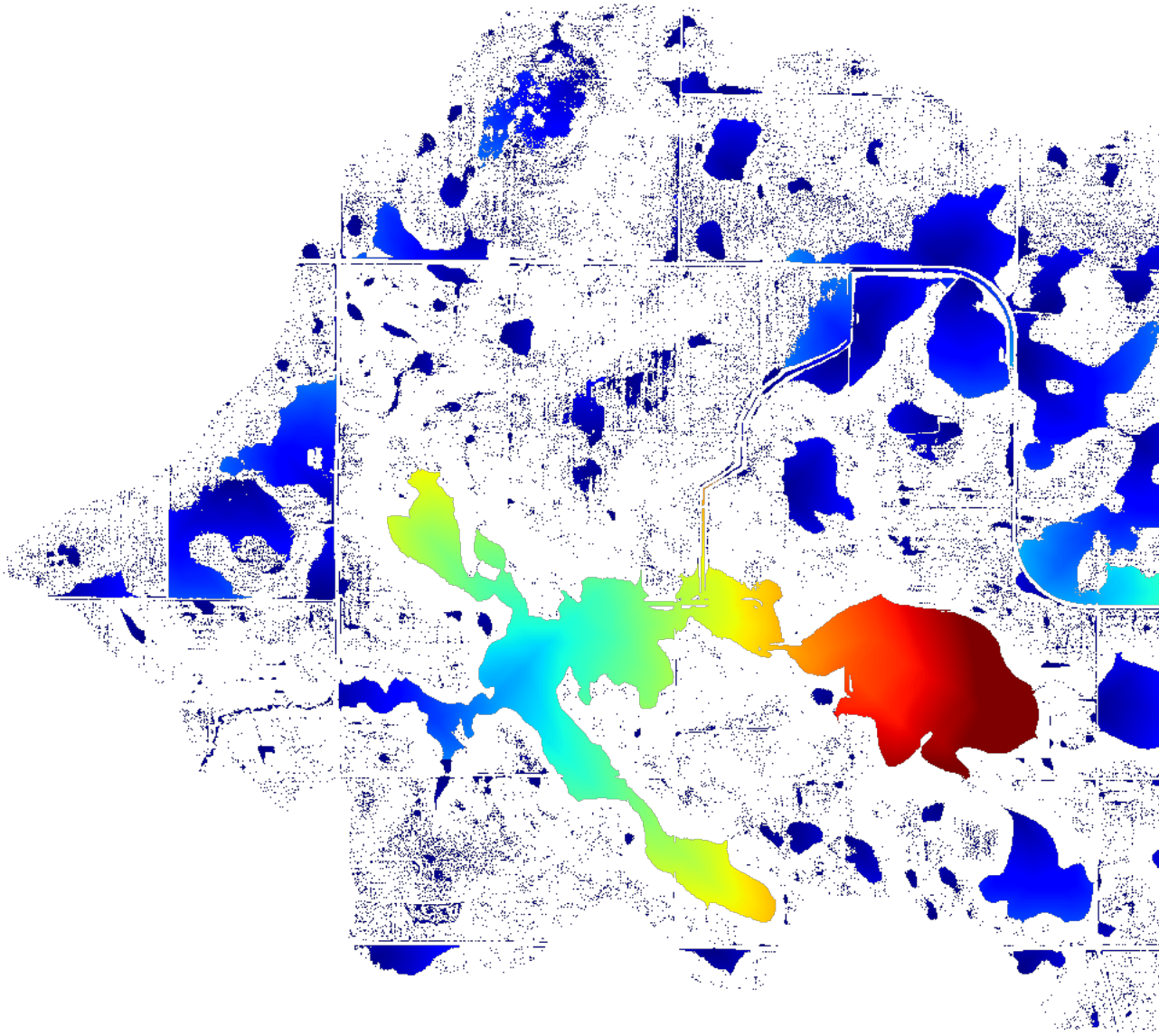
Language	Command
Python	`richdem.rdResolveFlats`
C++	*`richdem::ResolveFlatsEpsilon()`*

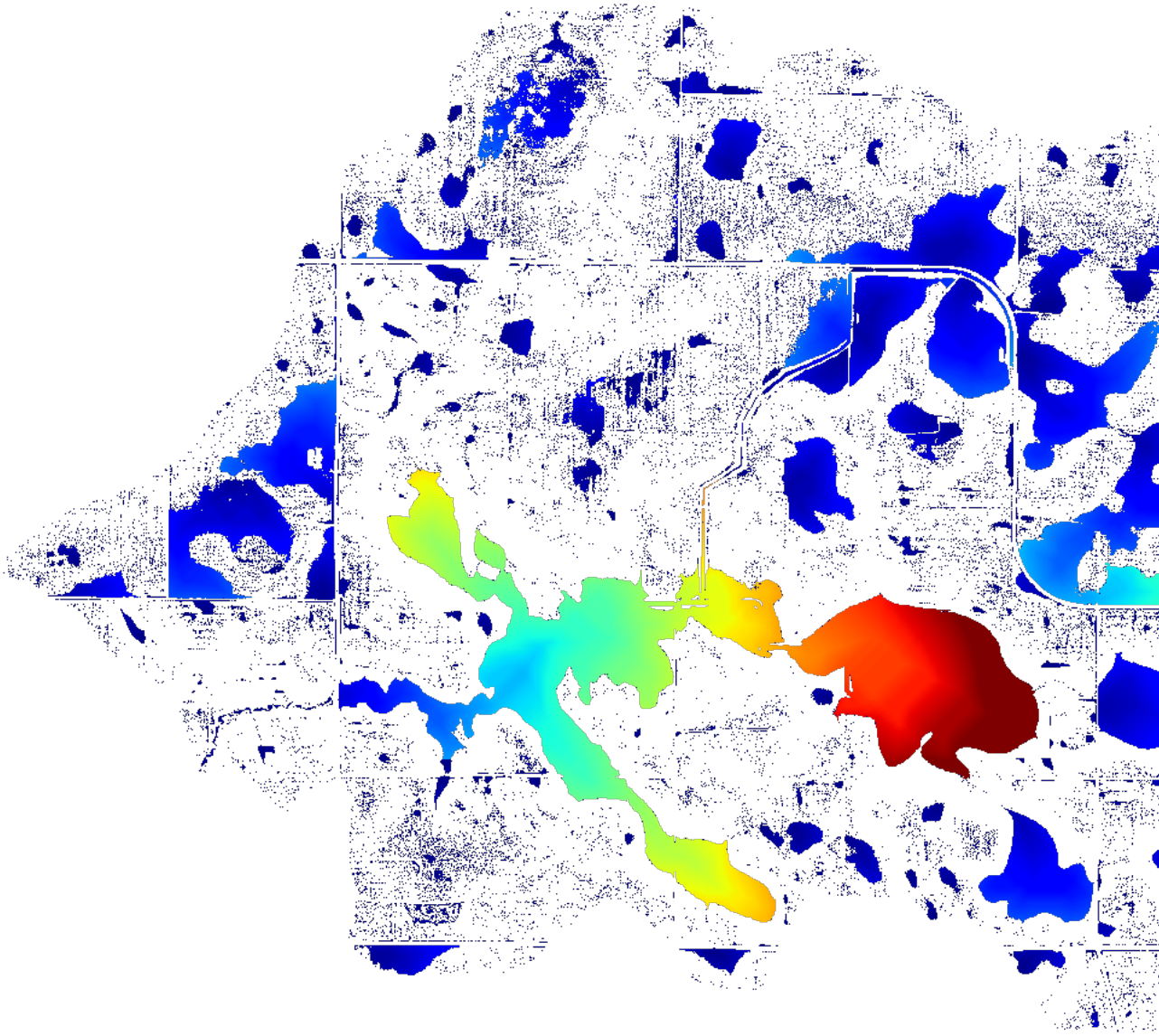
Pros	Cons
• All cells drain	• Not as fast as simple depression filling • May modify large portions of a DEM • May create elevated regions • Success may depend on data type

11.2 Flow Metric Adjustment

TODO







CHAPTER 12

Flow Metrics

A flow metric is a rule which apportions the flow passing through a cell into one or more of its neighbours.

The problem of how to best do this has been considered many times, since it is difficult to discretize flow onto a grid, and a number of solutions have been presented. Rather than choosing one, RichDEM instead incorporates many and leaves it to the user to decide which is appropriate.

Below, the various flow metrics included in RichDEM are discussed.

Wherever possible, algorithms are named according to the named according to their creators as well as by the name the authors gave the algorithm. For instance, `FM_Rho8` and `FM_FairfieldLeymarieD8` refer to the same function.

All flow metric functions are prefixed with !'FM_'.

Note that, in some cases, it is difficult or impossible to include a flow metric because the authors have included insufficient detail in their manuscript and have not provided source code. In these cases, the flow metric will either be absent or a "best effort" attempt has been made at implementation.

12.1 Flow Coordinate System

Internally, RichDEM refers to flow directions using a neighbourhood that appears as follows:

```
234
105
876
```

Neighbouring cells are accessed by looping through the indices 1 through 8 (inclusive) of the `dx[]` and `dy[]` arrays.

12.2 Convergent and Divergent Metrics

The greatest difference between flow metrics is in whether they are convergent or divergent. In a convergent method rivers only ever join: they never diverge or bifurcate. This means that landscape structures such as braided rivers cannot be adequately represented by a convergent method.

In a divergent method rivers may join and split, so braided rivers can be represented.

In general, convergent methods are simpler and therefore faster to use. There is a large diversity of divergent methods.

12.3 Note on the examples

Epsilon depression-filling replaces a depression with a predictable, convergent flow pattern. Beauford watershed has a number of depressions, as is evident in the example images below. A flow metric should not necessarily be judged by its behaviour within a filled depression. For convenience, a zoomed view of a non- depression area is shown and, at the end of this chapter, the views are compared.

12.4 D8 (O'Callaghan and Mark, 1984)

> O'Callaghan, J.F., Mark, D.M., 1984. The Extraction of Drainage Networks from Digital Elevation Data. Computer vision, graphics, and image processing 28, 323–344.

The D8 method assigns flow from a focal cell to one and only one of its 8 neighbouring cells. The chosen neighbour is the one accessed via the steepest slope. When such a neighbour does not exist, no flow direction is assigned. When two or more neighbours have the same slope, the chosen neighbour is the first one considered by the algorithm.

This is a convergent, deterministic flow method.

```
import richdem as rd
import numpy as np

dem = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)

rd.FillDepressions(dem, epsilon=True, in_place=True)
accum_d8 = rd.FlowAccumulation(dem, method='D8')
d8_fig = rd.rdShow(accum_d8, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.
↪5), axes=False, cmap='jet')
```

Language	Command
C++	`richdem::FM_OCallaghanD8` or *richdem::FM_D8()*

12.5 D4 (O'Callaghan and Mark, 1984)

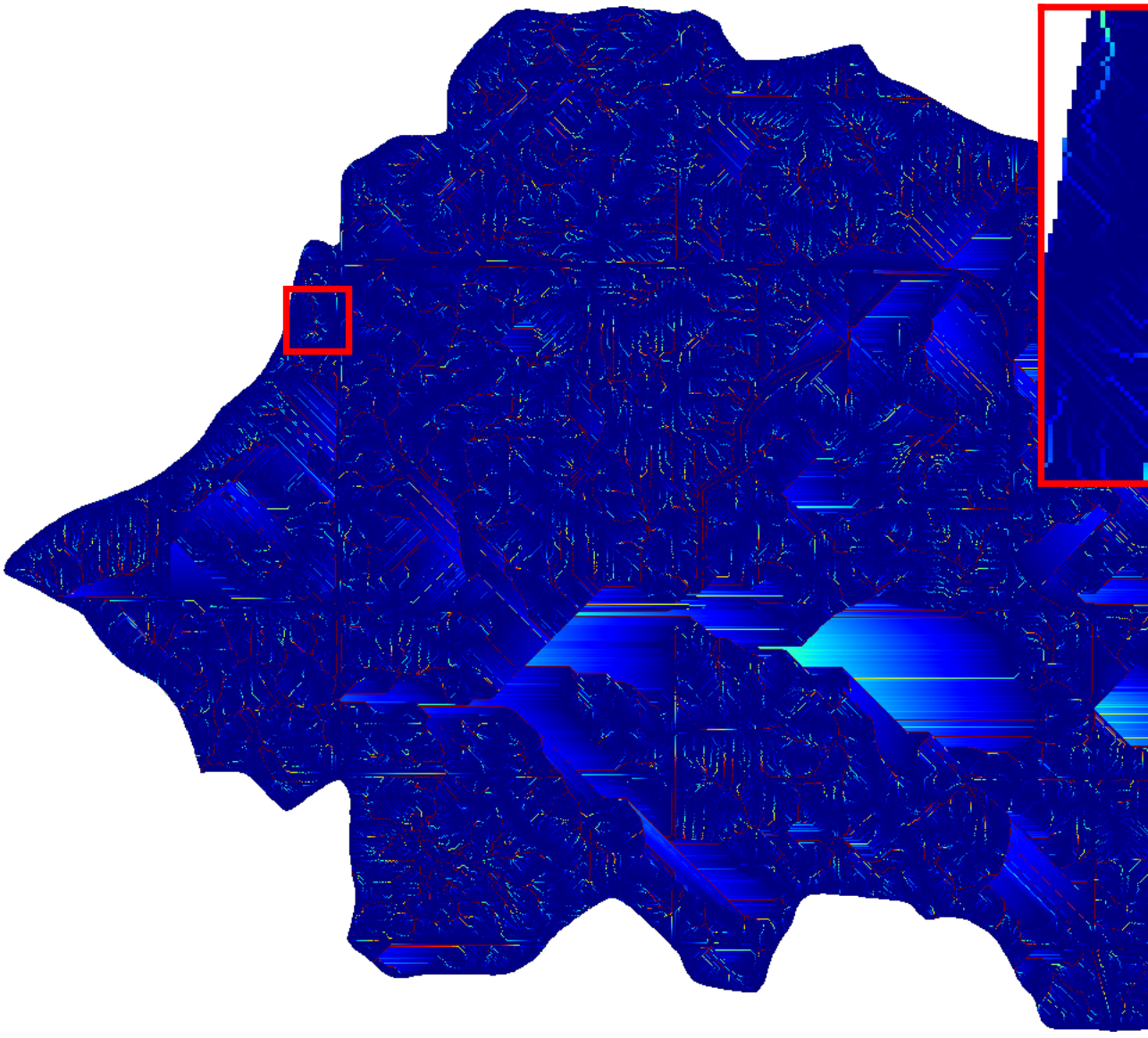
> O'Callaghan, J.F., Mark, D.M., 1984. The Extraction of Drainage Networks from Digital Elevation Data. Computer vision, graphics, and image processing 28, 323–344.

The D4 method assigns flow from a focal cell to one and only one of its 4 north, south, east, or west neighbouring cells. The chosen neighbour is the one accessed via the steepest slope. When such a neighbour does not exist, no flow direction is assigned. When two or more neighbours have the same slope, the chosen neighbour is the first one considered by the algorithm.

This is a convergent, deterministic flow method.

```
import richdem as rd
import numpy as np

dem = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)
```

(continues on next page)



(continued from previous page)

```
rd.FillDepressions(dem, epsilon=True, in_place=True)
accum_d4 = rd.FlowAccumulation(dem, method='D4')
d8_fig = rd.rdShow(accum_d4, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.
↪5), axes=False, cmap='jet')
```

Language	Command
C++	`richdem::FM_OCallaghanD4` or *`richdem::FM_D4()`*

12.6 Rho8 (Fairfield and Leymarie, 1991)

> Fairfield, J., Leymarie, P., 1991. Drainage networks from grid digital elevation models. Water resources research 27, 709–717.

The Rho8 method apportions flow from a focal cell to one and only one of its 8 neighbouring cells. To do so, the slope to each neighbouring cell is calculated and a neighbouring cell is selected randomly with a probability weighted by the slope.

This is a convergent, stochastic flow method.

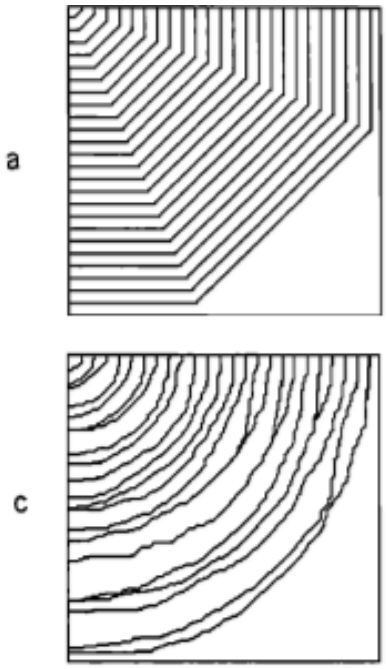


Fig. 2. Flow lines produced on a helical surface $z(x, y) = a \tan(y/x)$. Flow lines should be arcs of circles. (*a*) Method D8. (*b*) Method Rho4. (*c*) Method Rho8.

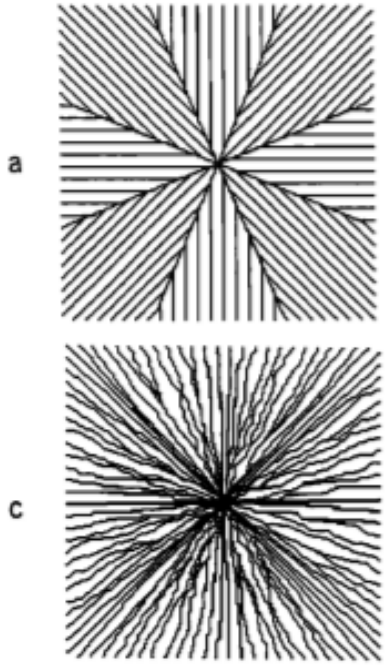
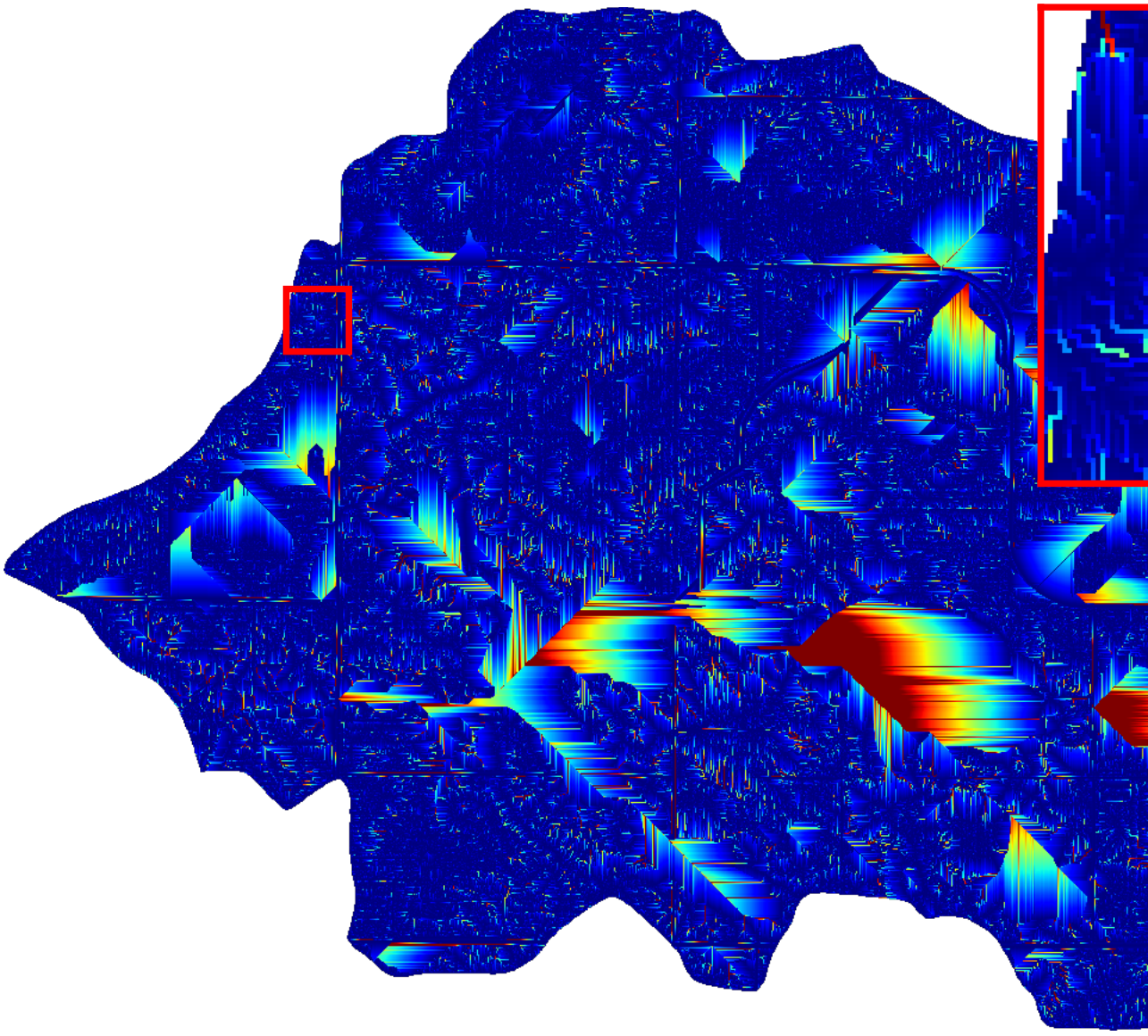
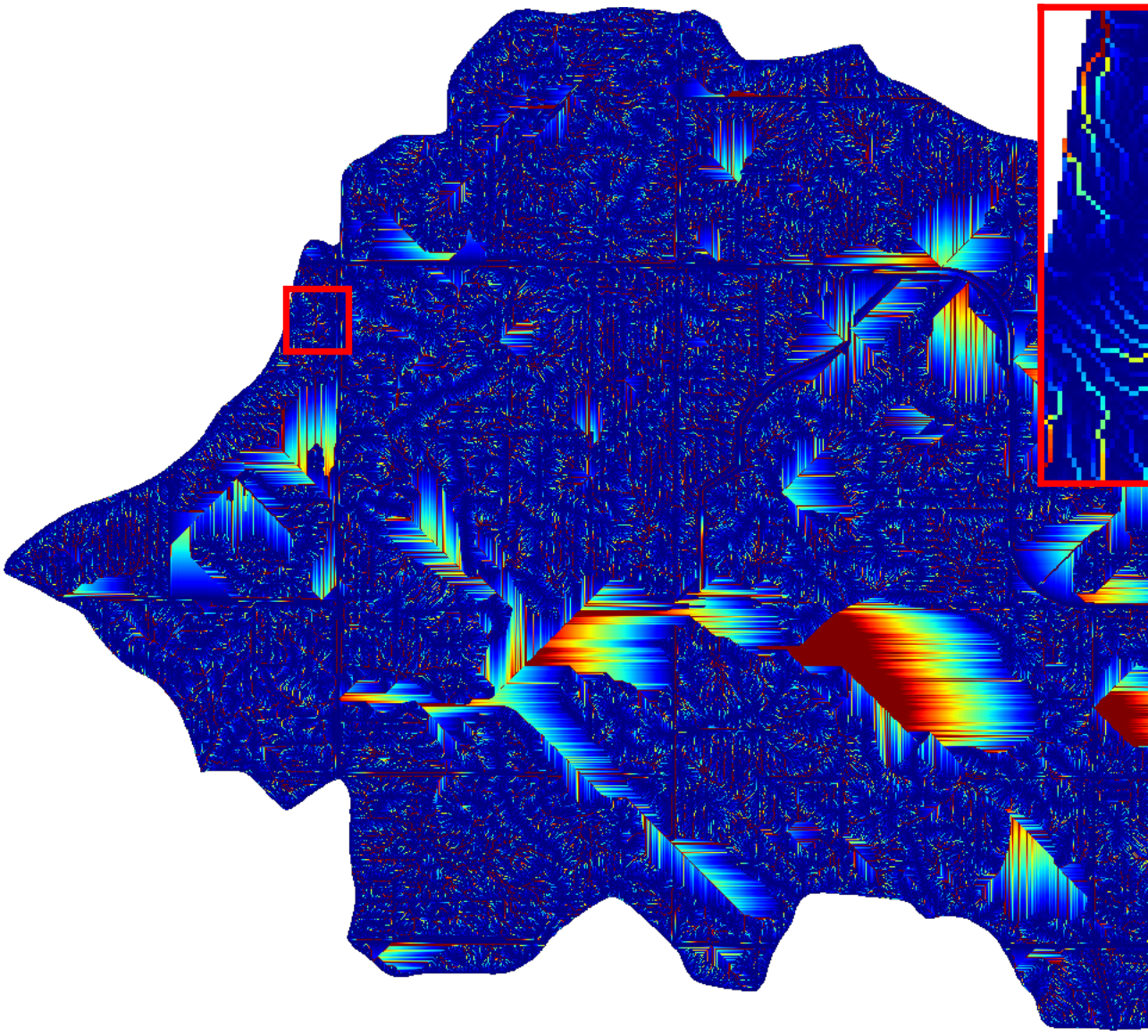


Fig. 3. Flow lines produced on a conical surface $z(x, y) = \text{sqrt}(x^2 + y^2)$. Flow lines should be straight lines through the center. (*a*) Method D8. (*b*) Method Rho4. (*c*) Method Rho8.

```
accum_rho8 = rd.FlowAccumulation(dem, method='Rho8')
rd.rdShow(accum_rho8, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),␣
↪axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```

Language	Command
C++	*`richdem::FM_Rho8()`* or `richdem::FM_FairfieldLeymarieD8`





12.7 Rho4 (Fairfield and Leymarie, 1991)

> Fairfield, J., Leymarie, P., 1991. Drainage networks from grid digital elevation models. Water resources research 27, 709–717.

The Rho4 method apportions flow from a focal cell to one and only one of its 8 neighbouring cells. To do so, the slope to each neighbouring cell is calculated and a neighbouring cell is selected randomly with a probability weighted by the slope.

This is a convergent, stochastic flow method.

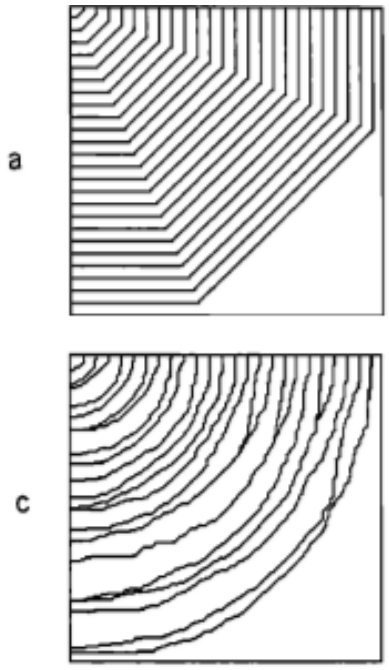


Fig. 2. Flow lines produced on a helical surface $z(x, y) = a \tan(y/x)$. Flow lines should be arcs of circles. (*a*) Method D8. (*b*) Method Rho4. (*c*) Method Rho8.

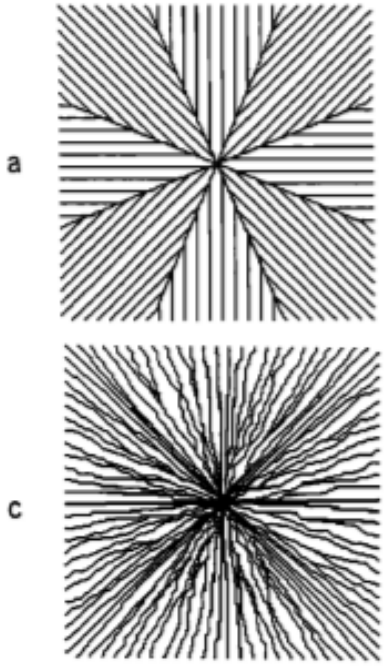


Fig. 3. Flow lines produced on a conical surface $z(x, y) = \text{sqrt}(x^2 + y^2)$. Flow lines should be straight lines through the center. (*a*) Method D8. (*b*) Method Rho4. (*c*) Method Rho8.

```
accum_rho4 = rd.FlowAccumulation(dem, method='Rho4')
rd.rdShow(accum_rho4, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),␣
↪axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```

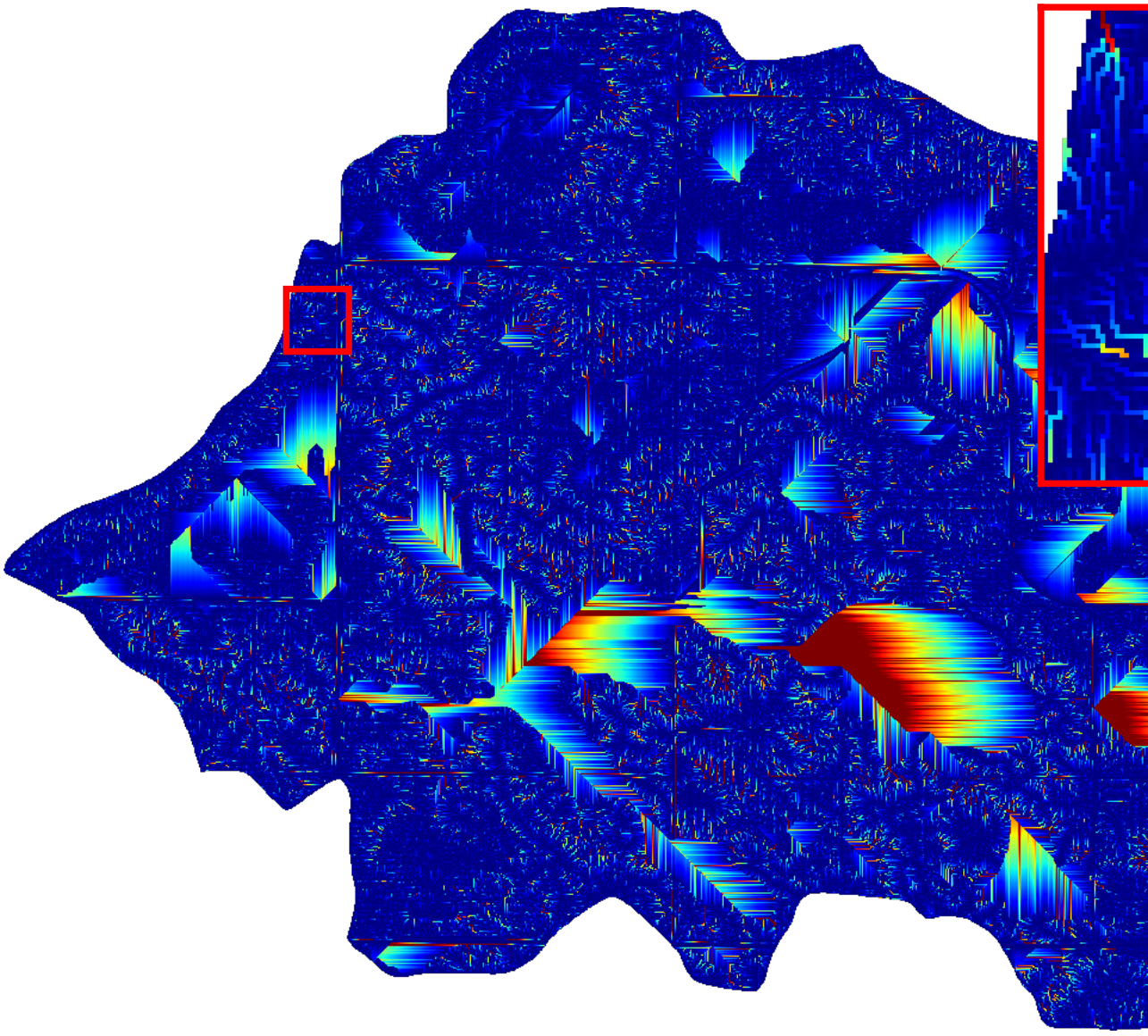
Language	Command
C++	*richdem::FM_Rho4()* or `richdem::FM_FairfieldLeymarieD4`

12.8 Quinn (1991)

> Quinn, P., Beven, K., Chevallier, P., Planchon, O., 1991. The Prediction Of Hillslope Flow Paths For Distributed Hydrological Modelling Using Digital Terrain Models. Hydrological Processes 5, 59–79.

The Quinn (1991) method apportions flow from a focal cell to one or more, and possibly all, of its 8 neighbouring cells. To do so, the amount of flow apportioned to each neighbour is a function $\tan(\beta)^1$ of the slope β to that neighbour. This is a special case of the Holmgren (1994) method.

This is a divergent, deterministic flow method.



```
accum_quinn = rd.FlowAccumulation(dem, method='Quinn')
rd.rdShow(accum_quinn, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),␣
↪axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```

Language	Command
C++	*richdem::FM_Quinn()*

12.9 Freeman (1991)

> Freeman, T.G., 1991. Calculating catchment area with divergent flow based on a regular grid. Computers & Geosciences 17, 413–422.

The Freeman (1991) method apportions flow from a focal cell to one or more, and possibly all, of its 8 neighbouring cells. To do so, the amount of flow apportioned to each neighbour is a function of the slope to that neighbour and a tuning parameter p. In particular, the fraction f_i of flow apportioned to neighbour i is

$$f_i = \frac{\max(0, \beta_i^p)}{\sum_{j \in N} \max(0, \beta_j^p)}$$

Freeman recommends choosing $p \approx 1.1$.

This is a divergent, deterministic flow method.

```
accum_freeman = rd.FlowAccumulation(dem, method='Freeman', exponent=1.1)
rd.rdShow(accum_freeman, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),␣
↪axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```

Language	Command
C++	*richdem::FM_Freeman()*

12.10 Holmgren (1994)

> Holmgren, P., 1994. Multiple flow direction algorithms for runoff modelling in grid based elevation models: an empirical evaluation. Hydrological processes 8, 327–334.

The Holmgren (1994) method apportions flow from a focal cell to one or more, and possibly all, of its 8 neighbouring cells. To do so, the amount of flow apportioned to each neighbour is a function of the slope that neighbour and a user-specified exponent x. In particular, the fraction f_i of flow apportioned to neighbour i is

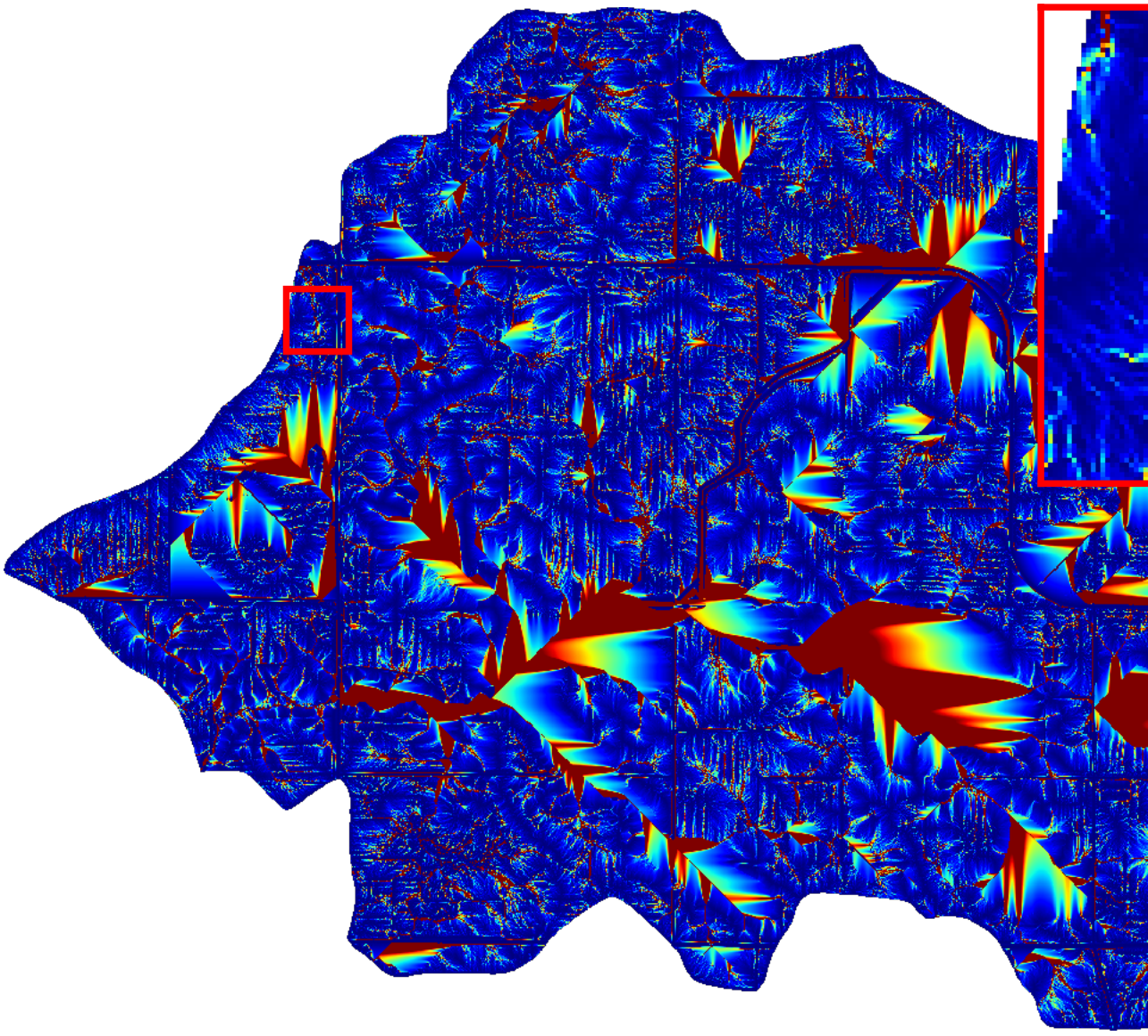
$$f_i = \frac{(\tan \beta_i)^x}{\sum_{j \in N} (tan \beta_j)^x} \forall \tan \beta > 0$$

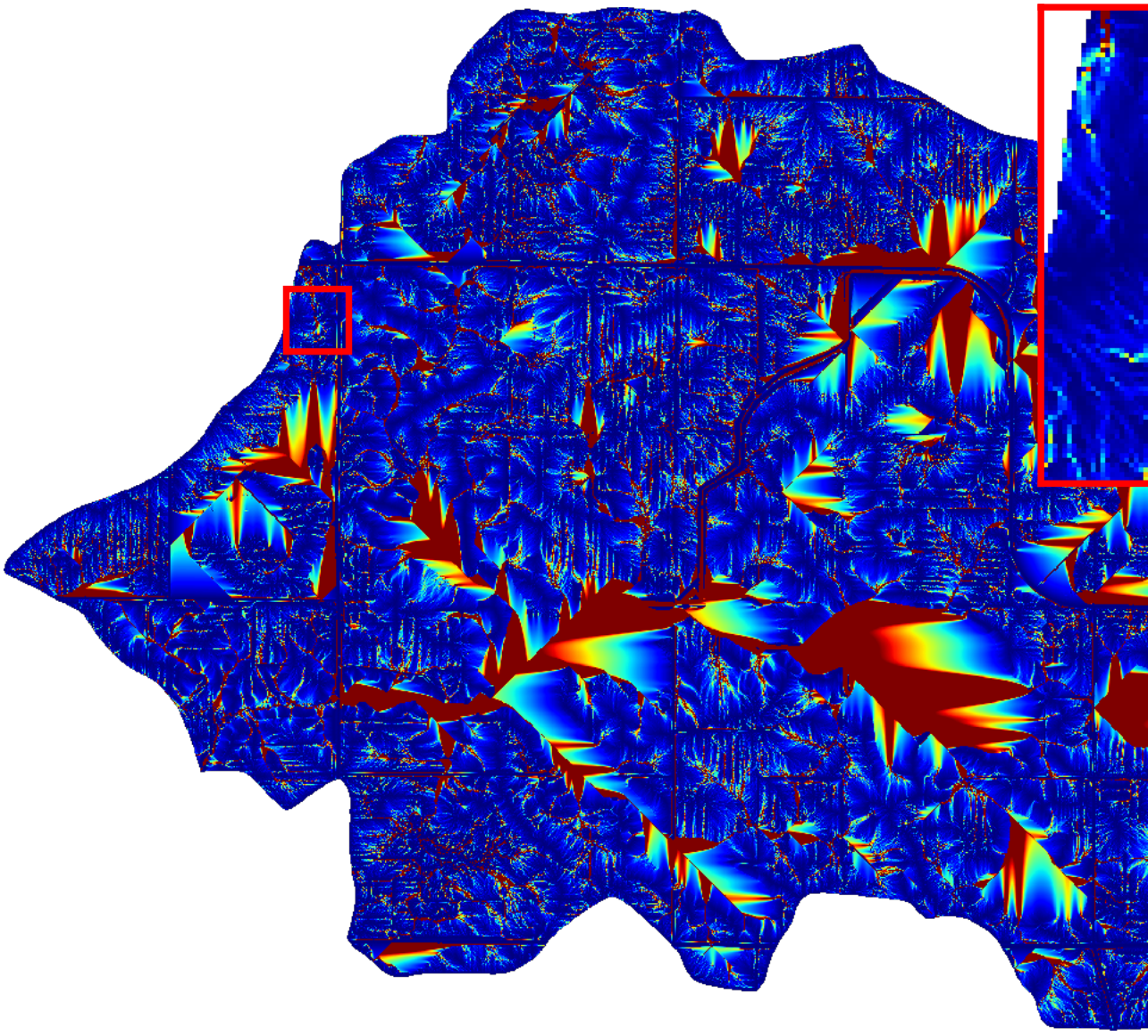
This is a generalization of the Quinn (1991) method in which the exponent is 1. As $x \to \infty$, this method approximates the D8 method.

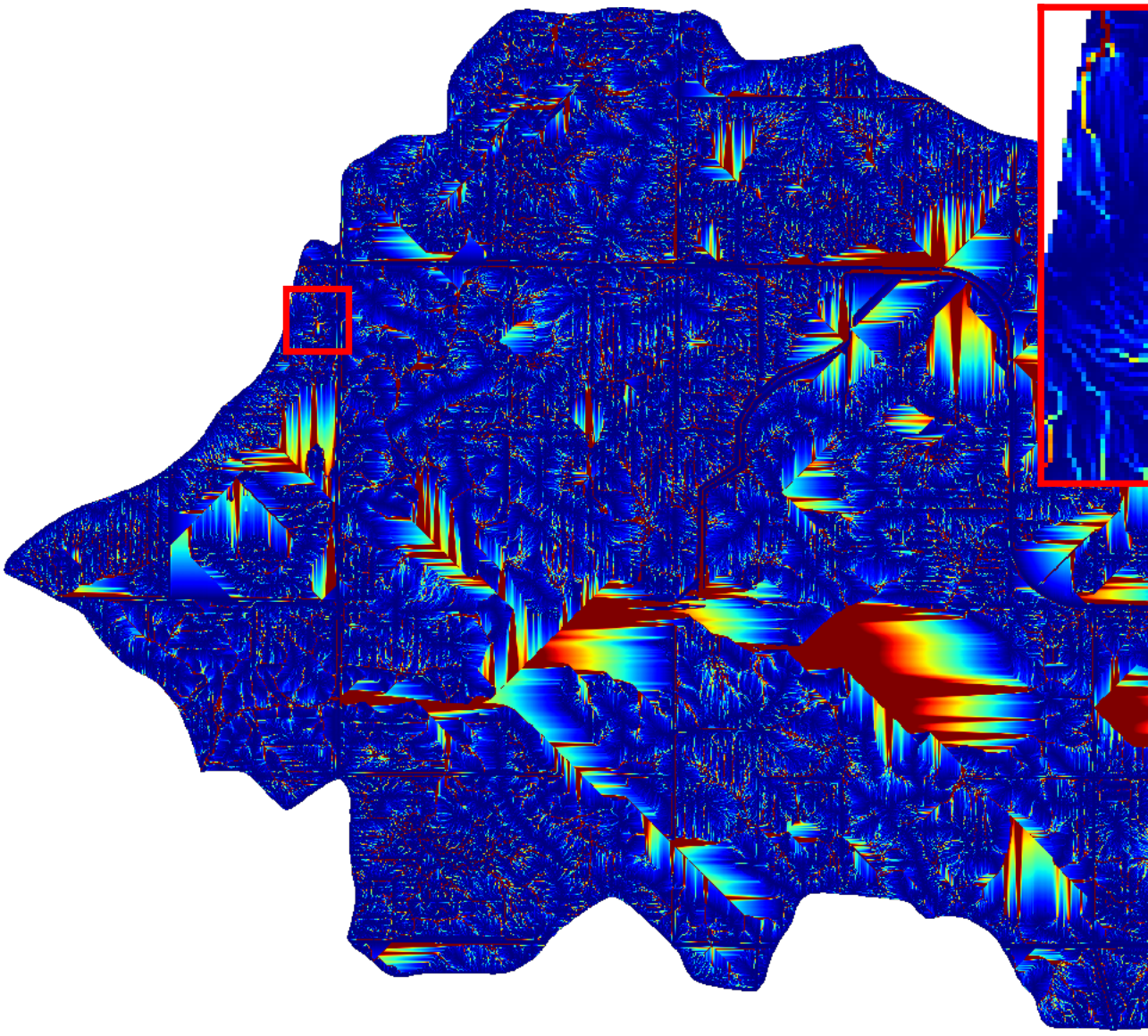
Holmgren recommends choosing $x \in [4, 6]$.

This is a divergent, deterministic flow method.

```
accum_holmgren = rd.FlowAccumulation(dem, method='Holmgren', exponent=5)
rd.rdShow(accum_holmgren, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),
↪ axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```







Language	Command
C++	`richdem::FM_Holmgren()`

12.11 D∞ (Tarboton, 1997)

Tarboton, D.G., 1997. A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water resources research 33, 309–319.

The D∞ method apportions flow from a focal cell between one or two adjacent neighbours of its 8 neighbouring cells. To do so, a line of steepest descent is calculated by doing localized surface fitting between the focal cell and adjacent pairs of its neighbouring cell. This line often falls between two neighbours.

This is a divergent, deterministic flow method.

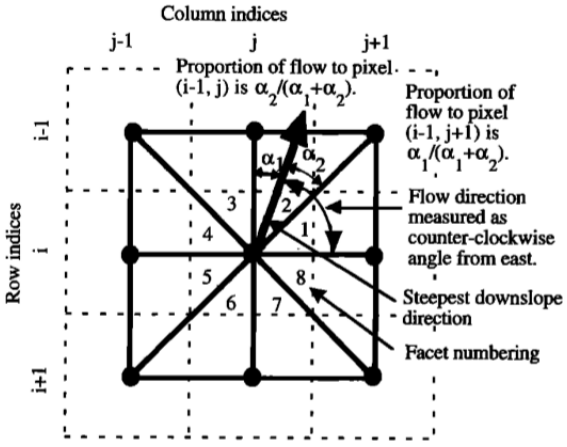


Figure 2. Flow direction defined as steepest downward slope on planar triangular facets on a block-centered grid.

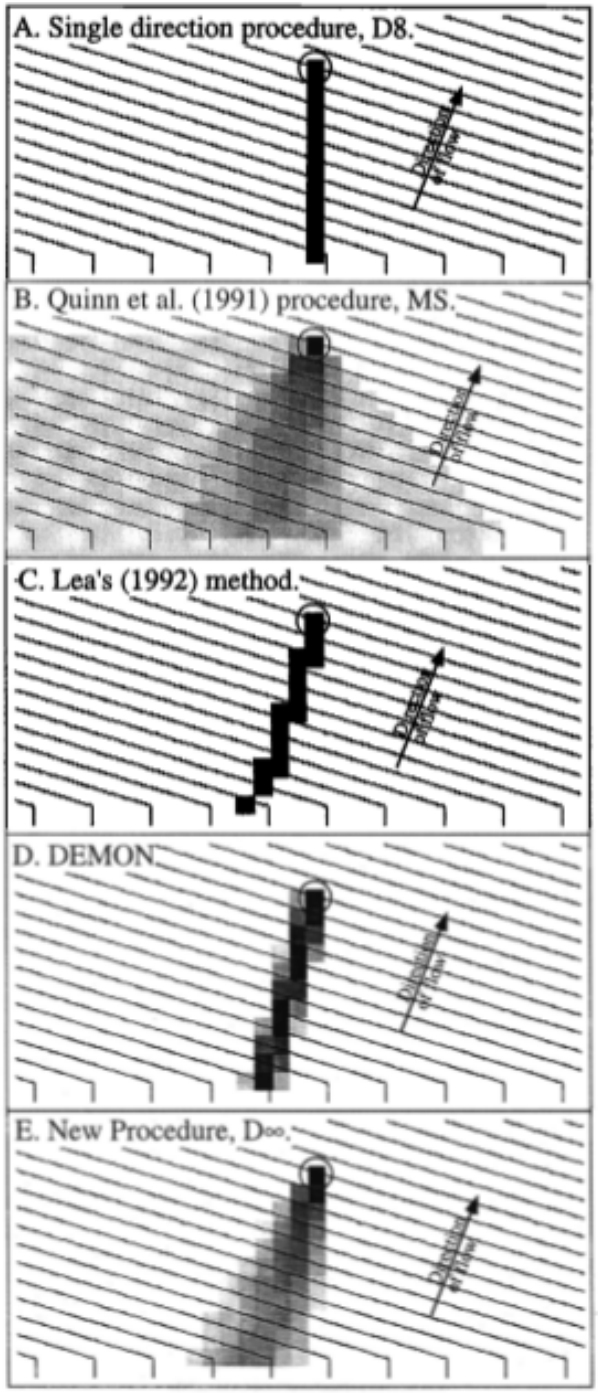
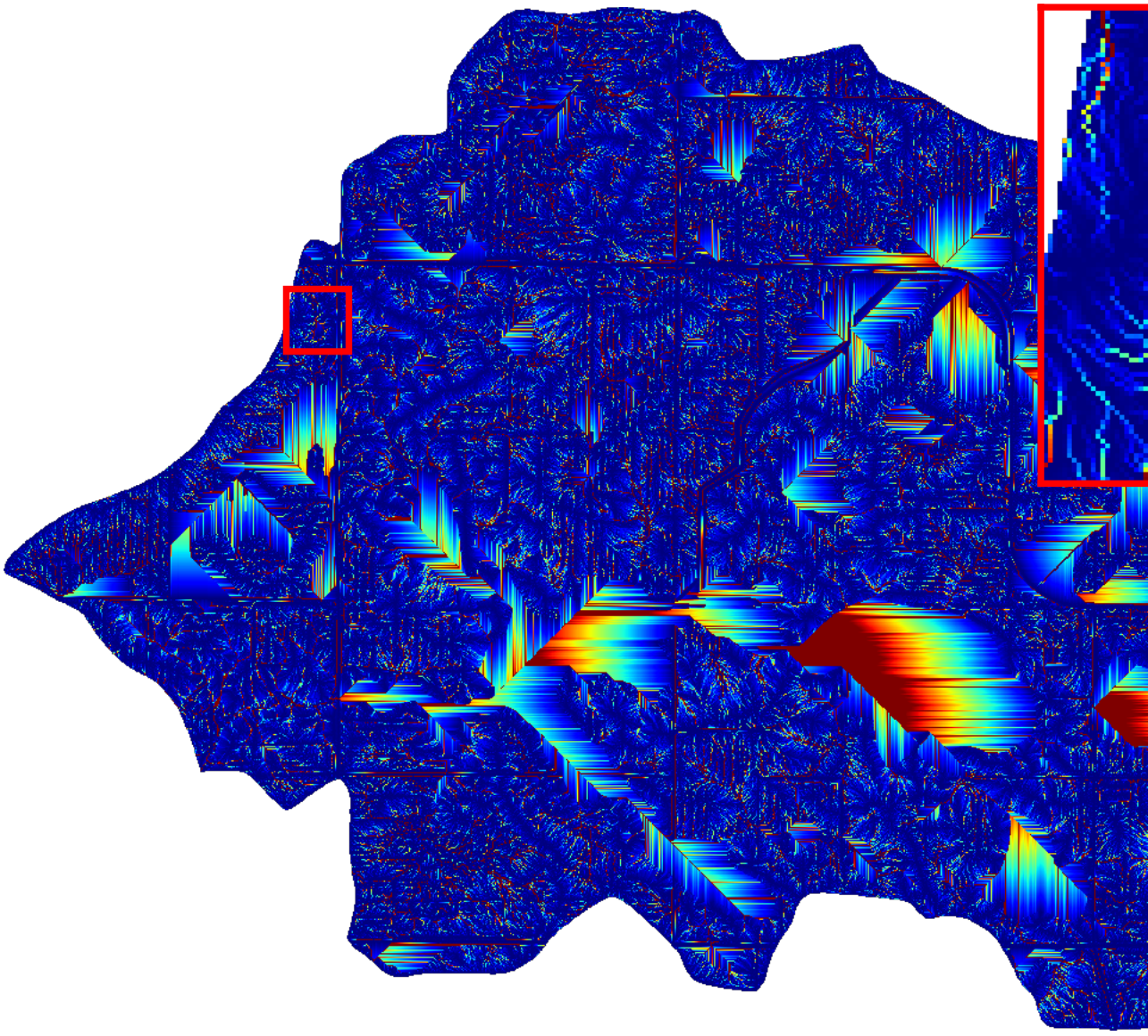


Figure 7. Dependence maps for planar surface. Gray scale (0 white, 1 black) indicates the fraction of each pixel upslope to the circled pixel.

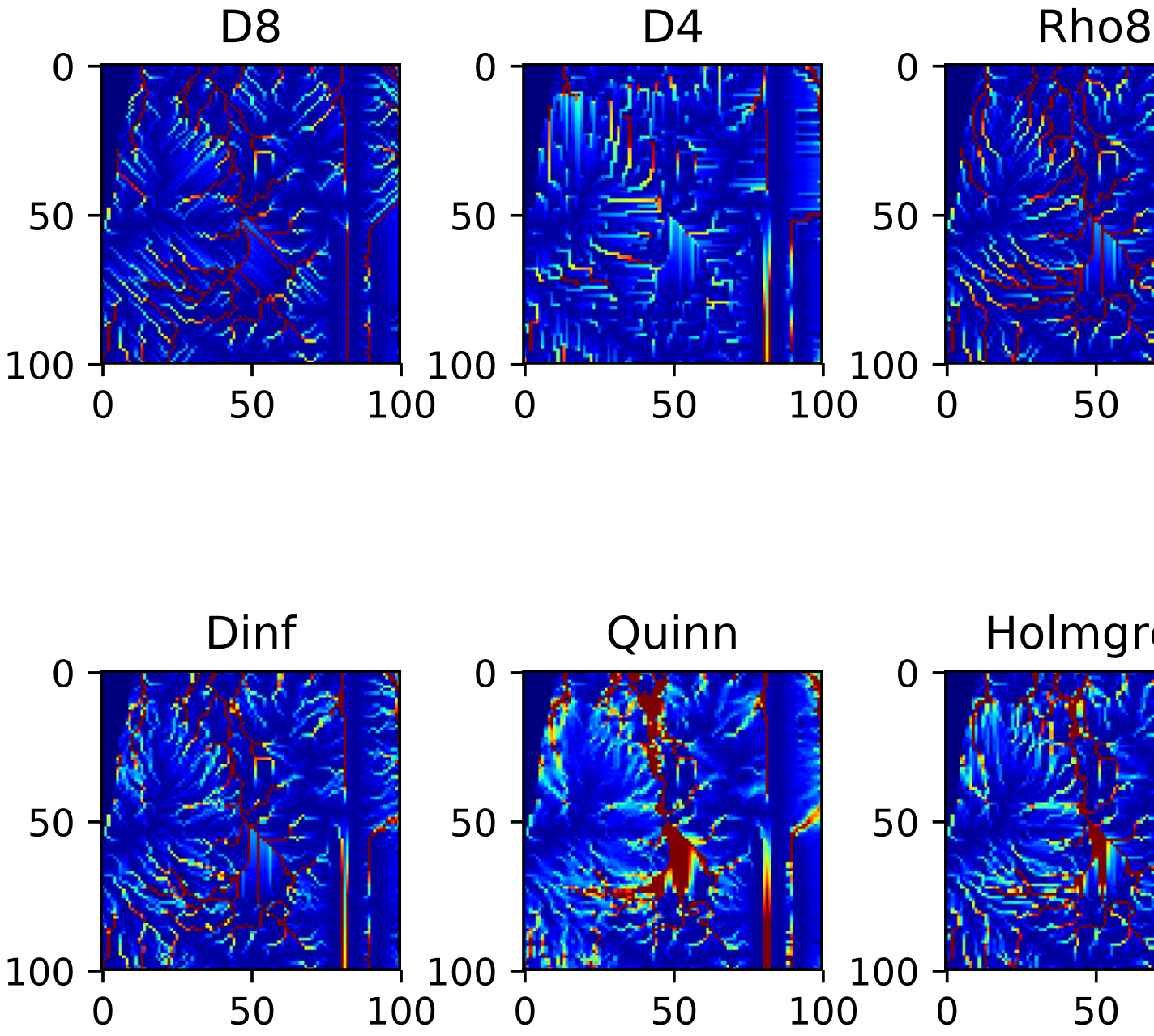
```
accum_dinf = rd.FlowAccumulation(dem, method='Dinf')
rd.rdShow(accum_dinf, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),␣
↪axes=False, cmap='jet', vmin=d8_fig['vmin'], vmax=d8_fig['vmax'])
```

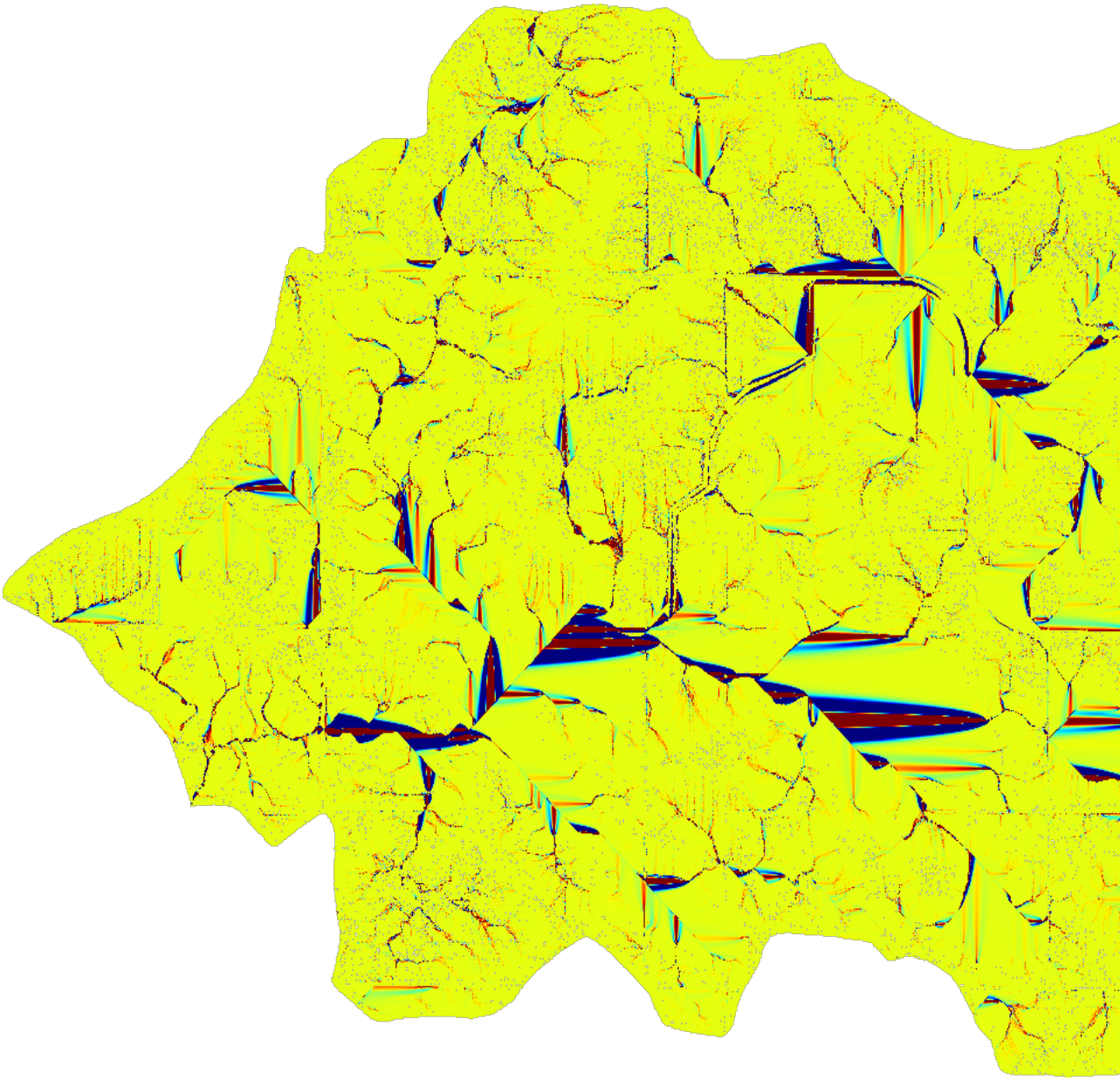
Language	Command
C++	*richdem::FM_Tarboton()* or *richdem::FM_Dinfinity()*



12.12 Side-by-Side Comparisons of Flow Metrics

Note that Quinn (1991) and Freeman (1991) produce rather similar results; nonetheless, they are different:


D8
D4
Rho8
Dinf
Quinn
0
50
100



CHAPTER 13

Accessing Flow Proportions Directly

In higher-level languages the foregoing flow proportions can be access via the flow proportions command, such as follows:

```
bprops = rd.FlowProportions(dem=beau, method='D8')
```

This command returns a matrix with the same width and height as the input, but an extra dimension which assigns each `(x,y)` cell 9 single-precision floating- point values.

The zeroth of these values is used for storing status information about the cell as a whole. If the 0th value of the area is `0` then the cell produces flow; if it is `-1`, then the cell produces no flow; if it is `-2`, then the cell is a NoData cell. The following eight values indicate the proportion of the cells flow directed to the neighbour corresponding to the index of that value where the neighbours are defined as in *Flow Coordinate System*.

For instance, the values:

```
0 0.25 0.25 0.25 0.25 0 0 0 0
```

direct 25% of a cell's flow to the northwest, north, northeast, and east.

These values can be manipulated and used to generate custom flow accumulations.

CHAPTER 14

Flow Accumulation

Each cell in a DEM can be modeled as generating a certain amount of flow. This flow is apportioned to downstream cells according to a chosen *flow metric*. The flow accumulation matrix, then, is one in which every cell's value is the summation of the flow it generates and all the flow which ultimately passes through it from upstream.

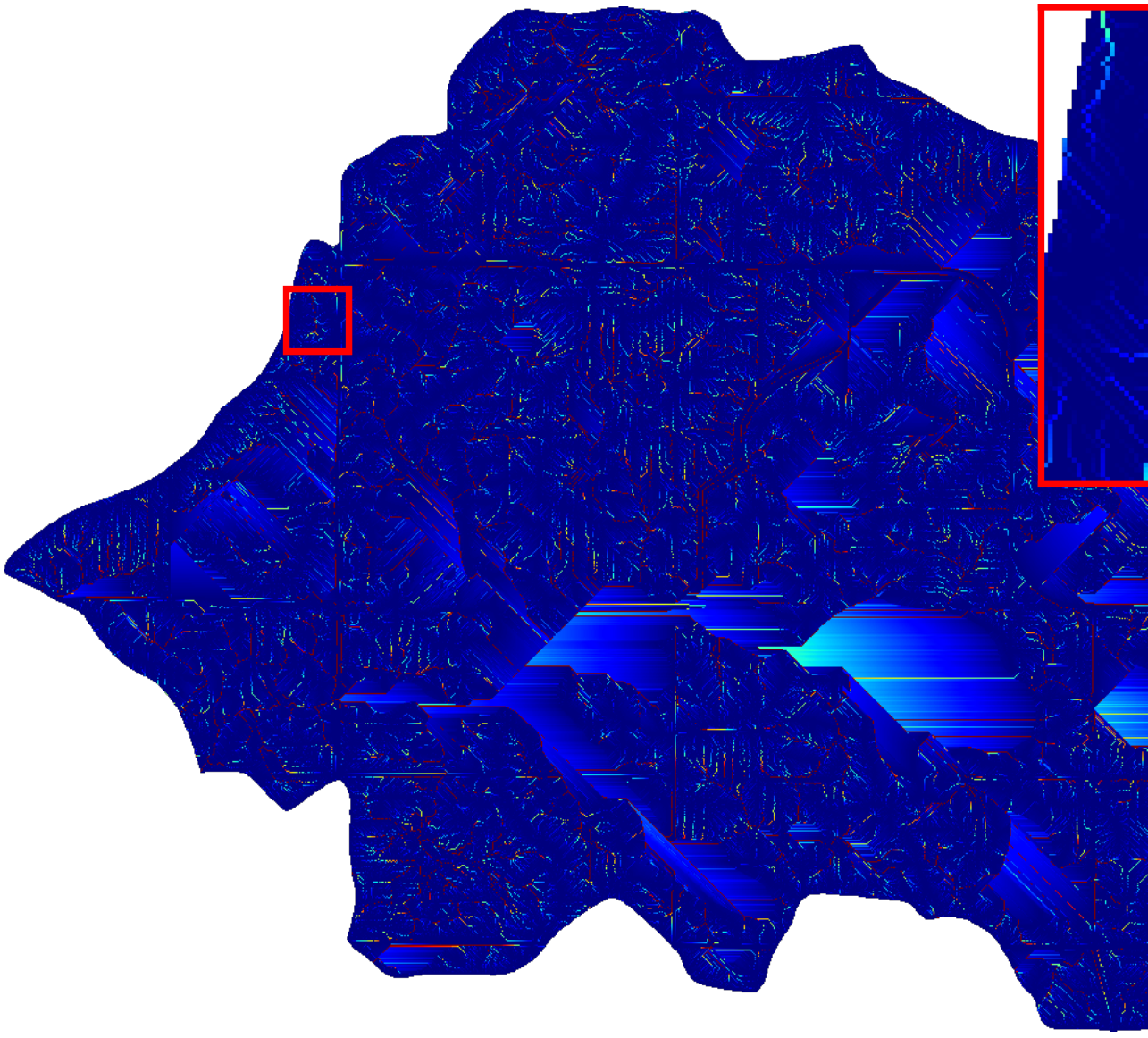
Flow Metrics shows the results of running a variety of flow metrics where each cell is modeled as producing 1 flow unit. This is shown again below for the D8 metric:

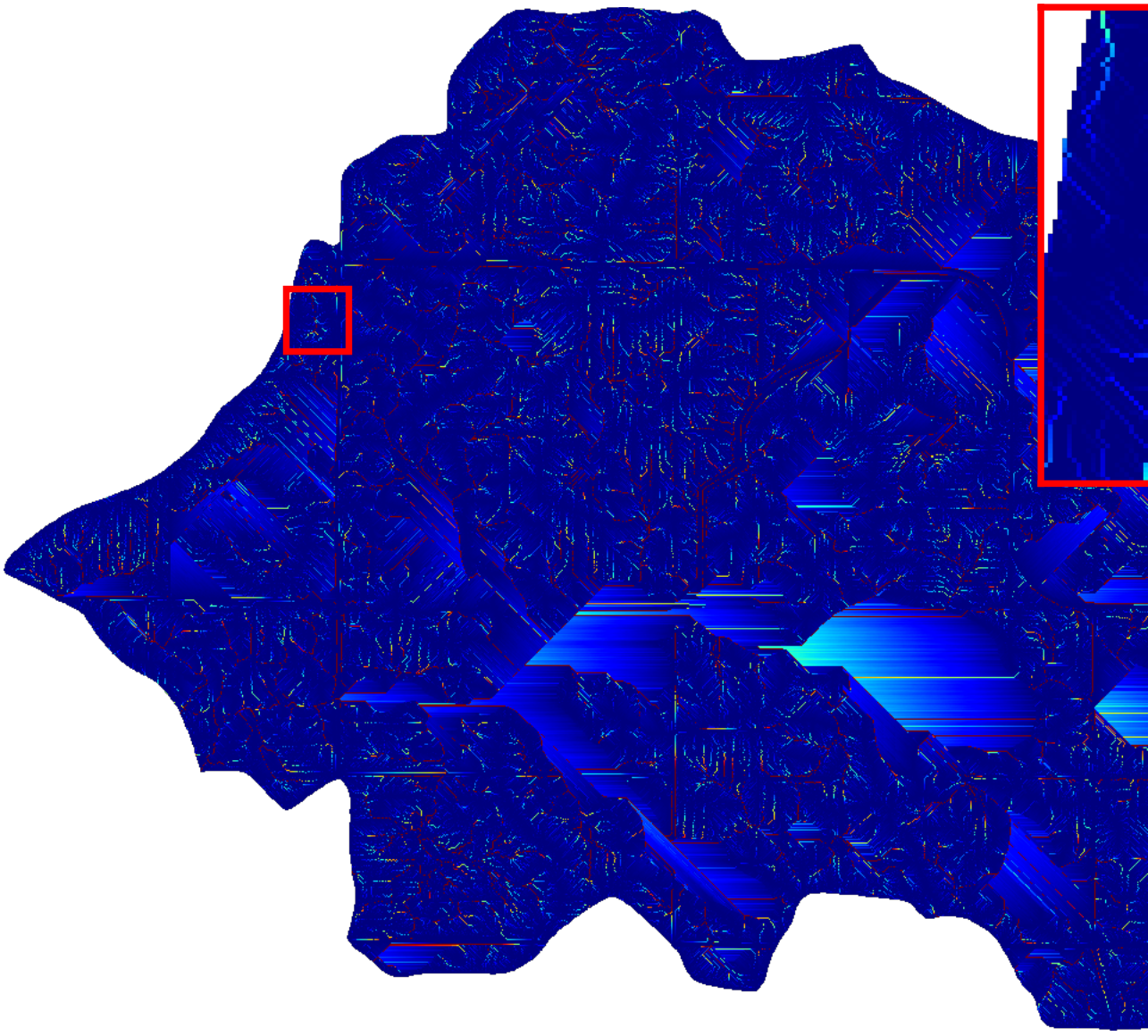
```
import richdem as rd
import numpy as np

dem = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)

#Fill depressions with epsilon gradient to ensure drainage
rd.FillDepressions(dem, epsilon=True, in_place=True)

#Get flow accumulation with no explicit weighting. The default will be 1.
accum_d8 = rd.FlowAccumulation(dem, method='D8')
d8_fig = rd.rdShow(accum_d8, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.
↪5), axes=False, cmap='jet')
```

But the amount of flow produced per cell can also be varied. For example, the amount of flow generated could be uniform random:

```
#Generate a random flow field
accum  = np.random.random(size=dem.shape)

#Modify the flow field into a flow accumulation field in place. A view of
#the modified data is returned as a metadata-enriched rdarray.
accum  = rd.FlowAccumulation(dem, method='D8', weights=accum, in_place=True)

d8_fig = rd.rdShow(accum, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),
↪ axes=False, cmap='jet')
```

Or flow generation could be concentrated to an area, as if there were a localized rain event:





```
#Make a circular region of flow generation

#Create coordinate grids
yy, xx = np.mgrid[:dem.shape[0], :dem.shape[1]]
#Find squared distance from center of grid
circle = (xx - dem.shape[1]/2) ** 2 + (yy - dem.shape[0]/2) ** 2
#Take only those cells within a radius
circle = (circle < 200**2).astype('float64')

#Don't modify the original accumulation data. Return a new matrix with flow
#accumulation values.
accum  = rd.FlowAccumulation(dem, method='D8', weights=circle, in_place=False)

d8_fig = rd.rdShow(accum, ignore_colours=[0], figsize=(8,5.5), axes=False, cmap='jet')
```

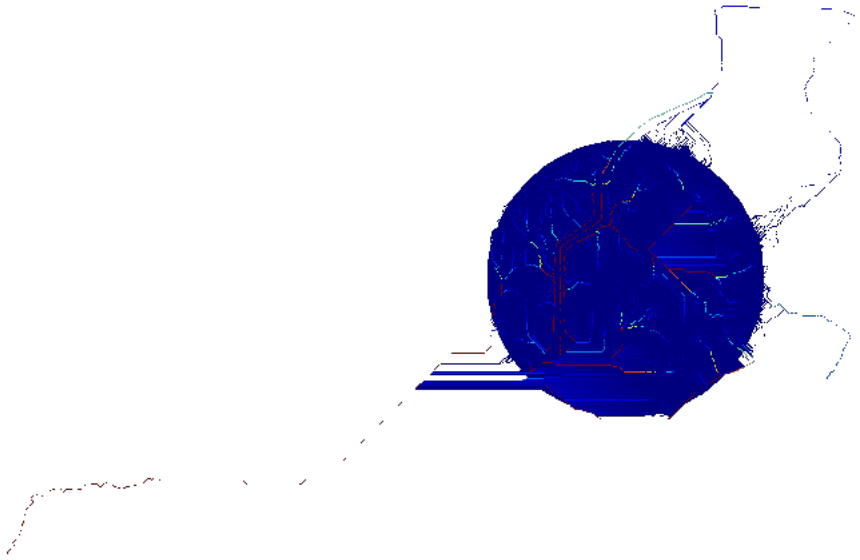
Or flow generation could be concentrated to part of a region, as though a mountain range were affecting weather:

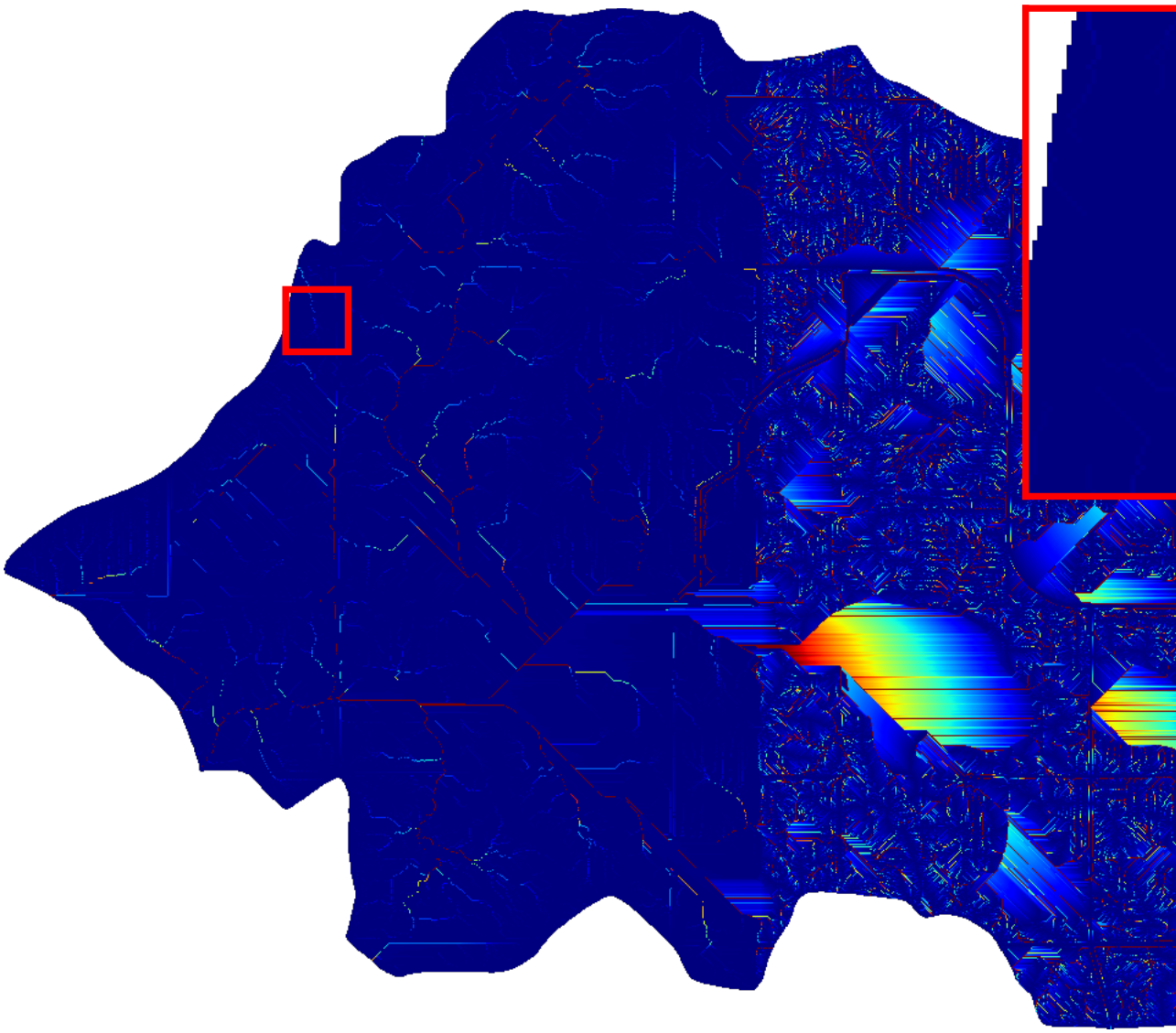
```
#Create coordinate grids
yy, xx = np.mgrid[:dem.shape[0], :dem.shape[1]]

#Create nominal weights
accum = rd.rdarray(np.ones(shape=dem.shape).astype('float64'), no_data=-1)

#Increase weights on right-hand side of field
accum[xx>dem.shape[1]/2] *= 50

#Don't modify the original accumulation data. Return a new matrix with flow
#accumulation values.
rd.FlowAccumulation(dem, method='D8', weights=accum, in_place=True)

d8_fig = rd.rdShow(accum, zxmin=450, zxmax=550, zymin=550, zymax=450, figsize=(8,5.5),
↪ axes=False, cmap='jet')
```

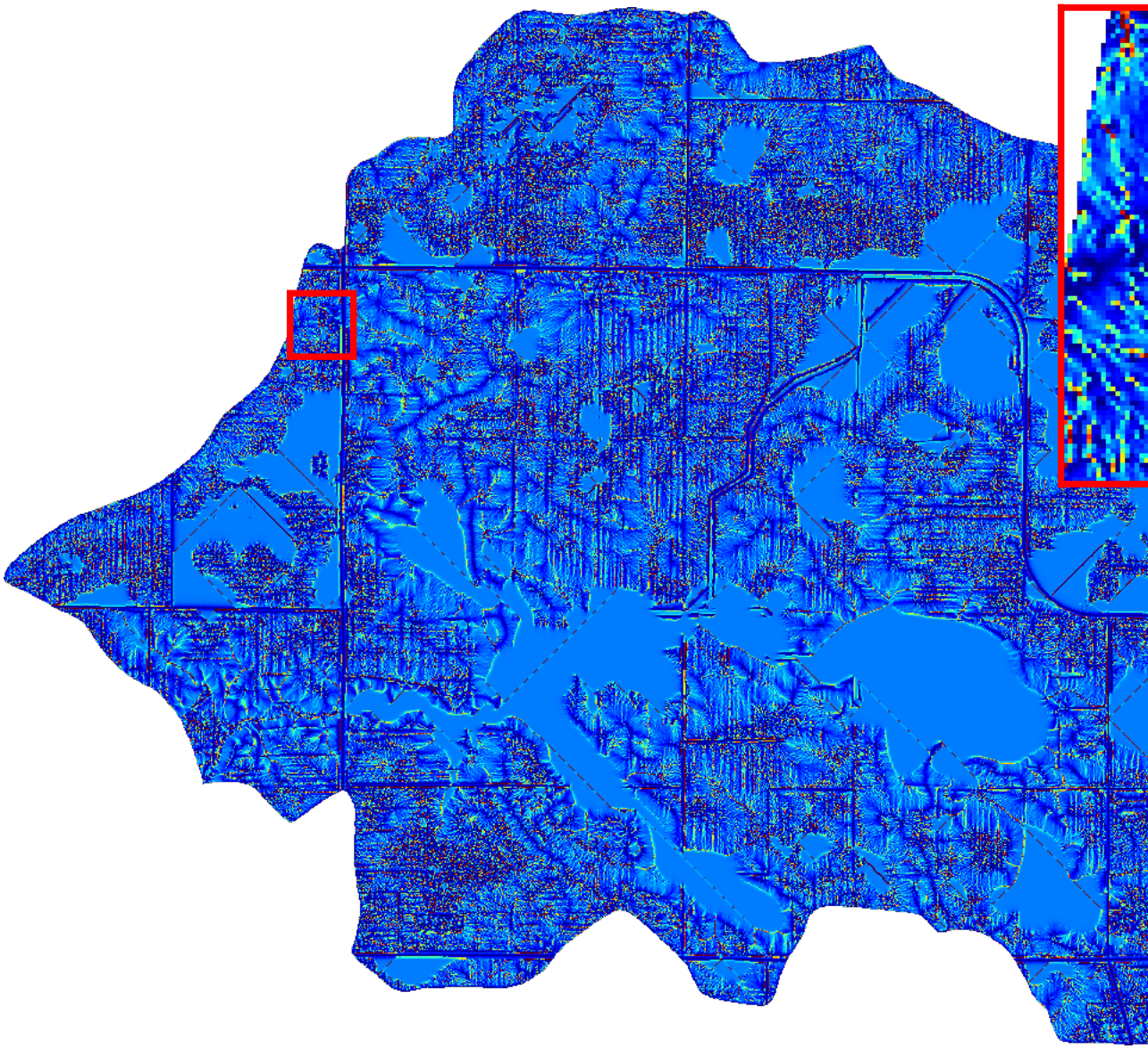
14.1 From Flow Proportions

Flow accumulation can also be generated from raw flow proportions:

```
props = rd.FlowProportions(dem, method='Freeman', exponent=1.1)

#30% of the flow moving along any route is absorbed
props[props>0] *= 0.7
accum = rd.FlowAccumFromProps(props=props)

rd.rdShow(accum, ignore_colours=[0], figsize=(8,5.5), axes=False, cmap='jet',␣
↪zxmin=450, zxmax=550, zymin=550, zymax=450)
```







CHAPTER 15

Terrain Attributes

RichDEM can calculate a number of terrain attributes.

15.1 Slope

Horn, B.K.P., 1981. Hill shading and the reflectance map. Proceedings of the IEEE 69, 14–47. doi:10.1109/PROC.1981.11918

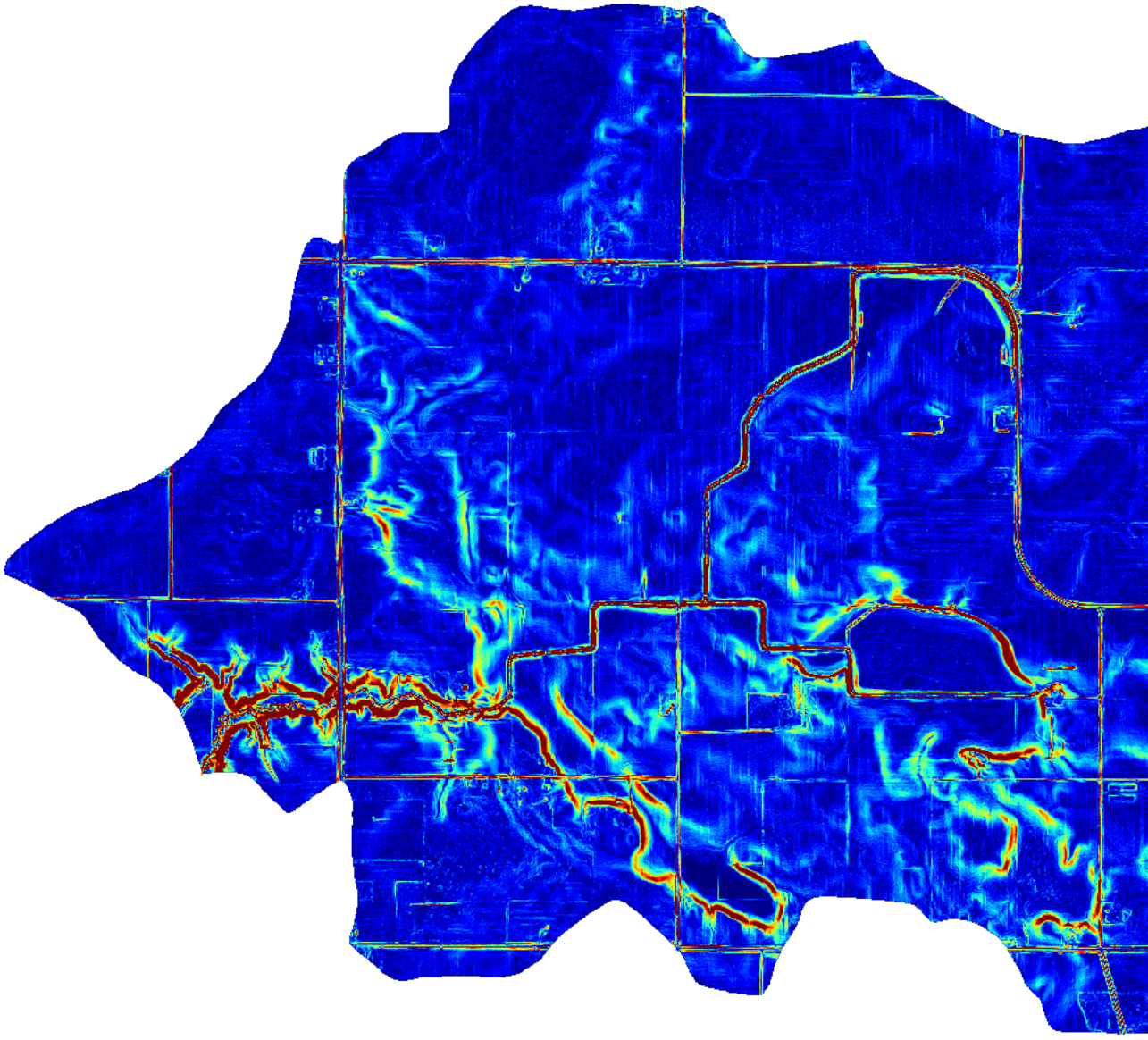
Horn (1981) calculates the slope of a focal cell by using a central difference estimation of a surface fitted to the focal cell and its neighbours. The slope chosen is the maximum of this surface and can be returned in several formats.

```
import richdem as rd
import numpy as np

beau  = rd.rdarray(np.load('imgs/beauford.npz')['beauford'], no_data=-9999)
slope = rd.TerrainAttribute(beau, attrib='slope_riserun')
rd.rdShow(slope, axes=False, cmap='jet', figsize=(8,5.5))
```

Language	Command
Python	`richdem.TerrainAttribute`
C++	*richdem::TA_slope_riserun()*
C++	*richdem::TA_slope_percentage()*
C++	*richdem::TA_slope_degrees()*
C++	*richdem::TA_slope_radians()*

15.2 Aspect

Horn, B.K.P., 1981. Hill shading and the reflectance map. Proceedings of the IEEE 69, 14–47. doi:10.1109/PROC.1981.11918



Horn (1981) calculates aspect as the direction of the maximum slope of the focal cell. The value returned is in Degrees.

```
aspect = rd.TerrainAttribute(beau, attrib='aspect')
rd.rdShow(aspect, axes=False, cmap='jet', figsize=(8,5.5))
```

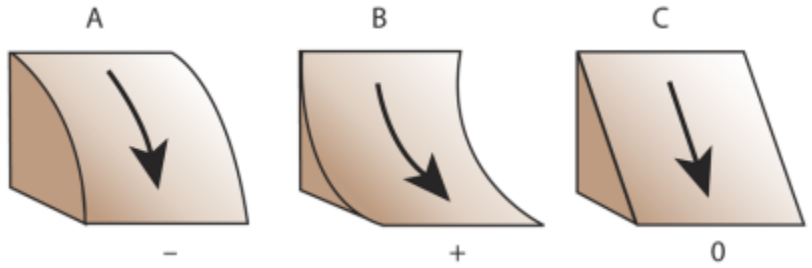
Language	Command
Python	`richdem.TerrainAttribute`
C++	*`richdem::TA_aspect()`*

15.3 Profile Curvature

> Zevenbergen, L.W., Thorne, C.R., 1987. Quantitative analysis of land surface topography. Earth surface processes and landforms 12, 47–56.

Profile curvature is calculated by fitting a surface to the focal cell and its neighbours. The profile curvature runs parallel to the maximum slope of this surface and affects the acceleration and deceleration of flow down the slope.

Negative profile curvatures (A) indicate upwardly convex slopes, positive profile curvatures (B) indicate upwardly concave surfaces, and a profile curvature of zero indicates a linear slope (C).




```
profile_curvature = rd.TerrainAttribute(beau, attrib='profile_curvature')
rd.rdShow(profile_curvature, axes=False, cmap='jet', figsize=(8,5.5))
```

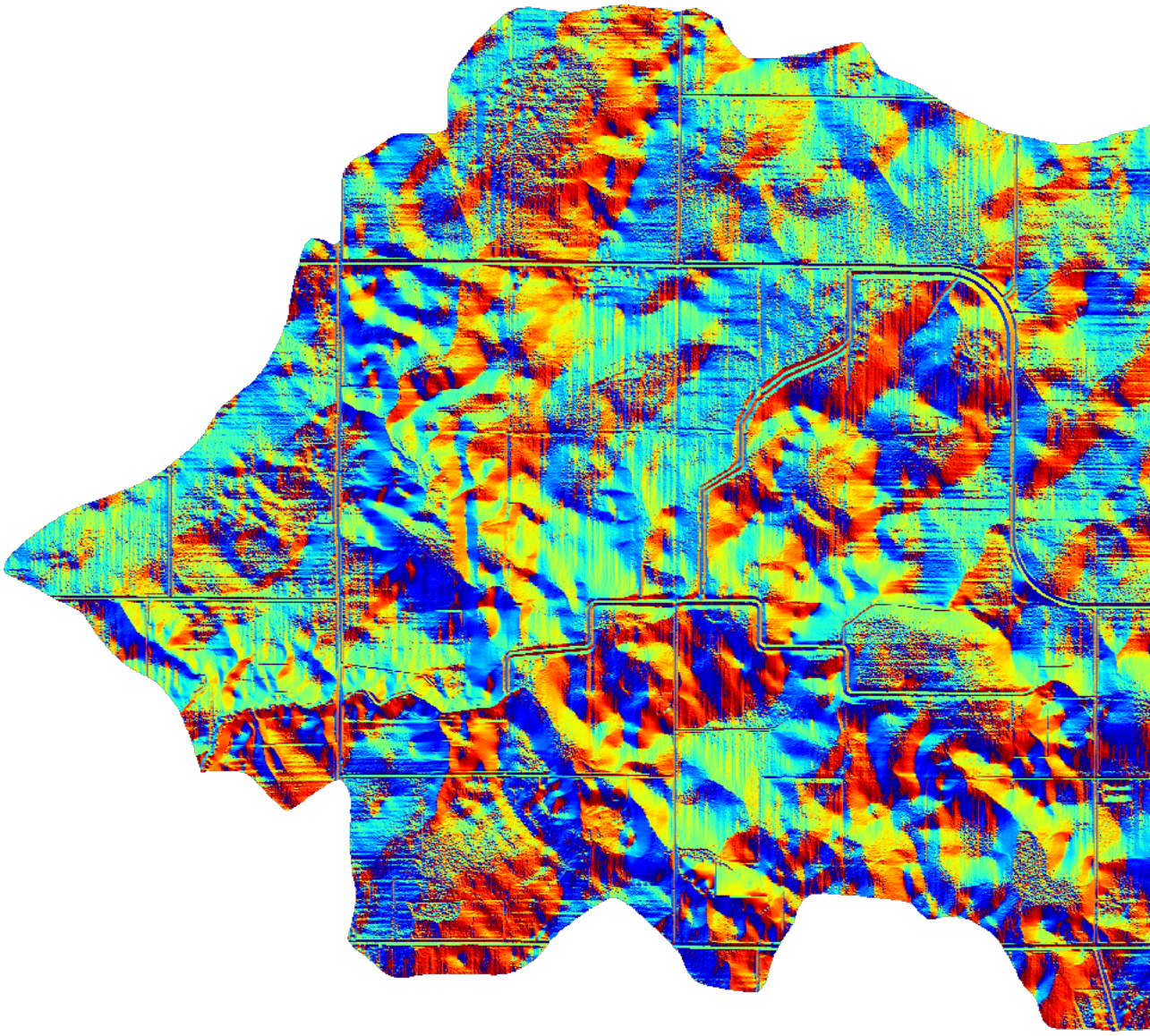
Language	Command
Python	`richdem.TerrainAttribute`
C++	*`richdem::TA_profile_curvature()`*

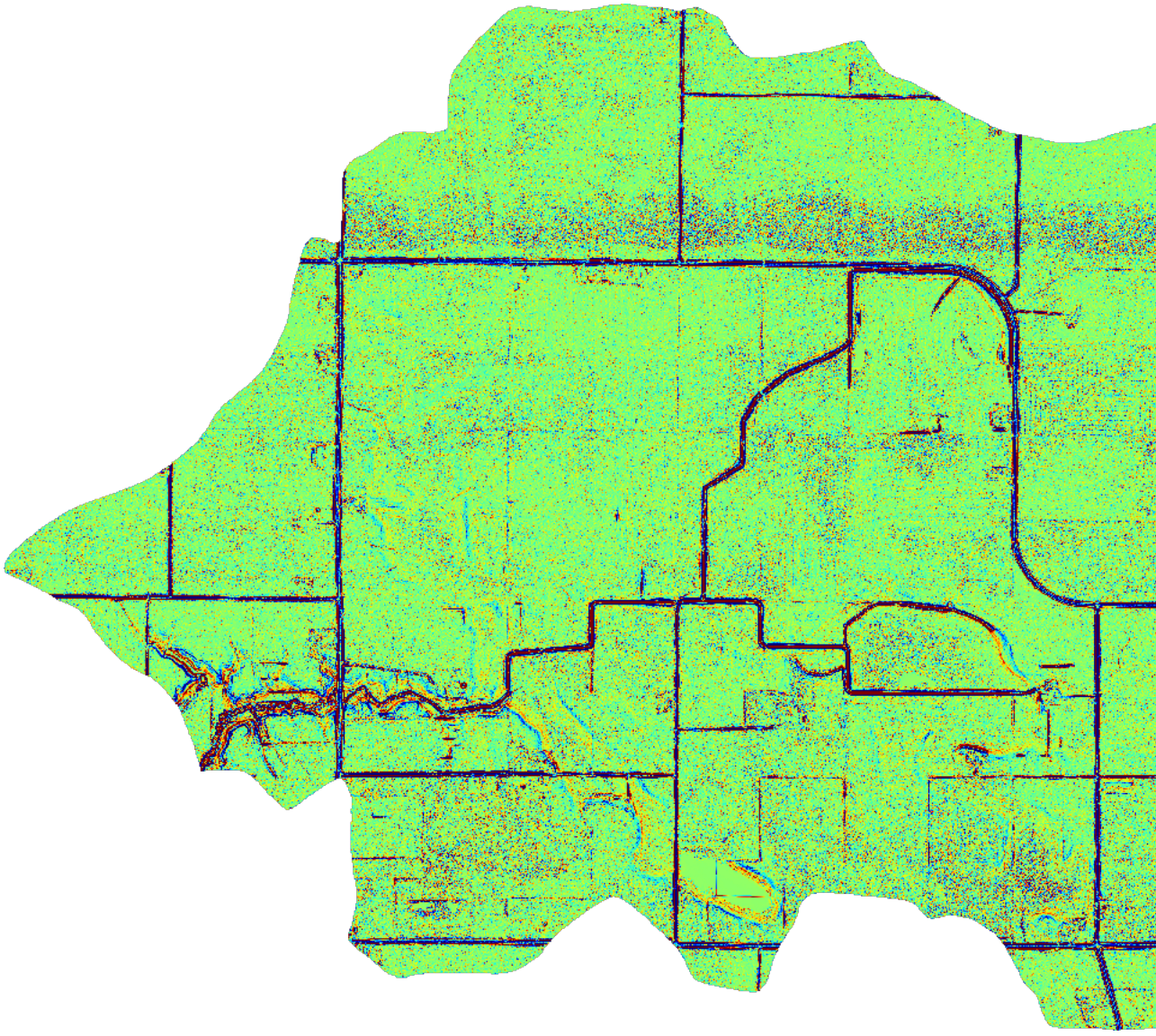
15.4 Planform Curvature

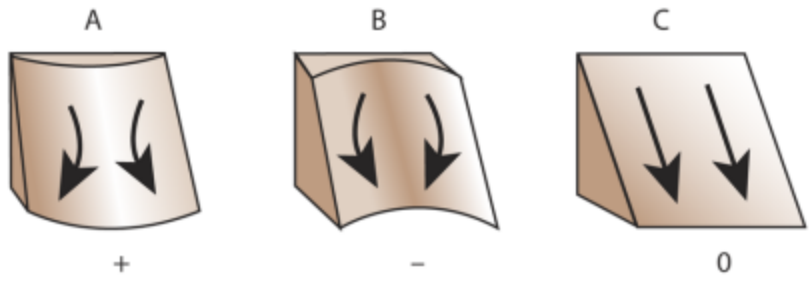
> Zevenbergen, L.W., Thorne, C.R., 1987. Quantitative analysis of land surface topography. Earth surface processes and landforms 12, 47–56.

Planform curvature is calculated by fitting a surface to the focal cell and its neighbours. The planform curvature runs perpendicular to the maximum slope of this surface and affects the convergence and divergence of flow down the slope.

Negative planform curvatures (A) indicate laterally convex slopes, positive planform curvatures (B) indicate laterally concave surfaces, and a planform curvature of zero indicates a linear slope (C).







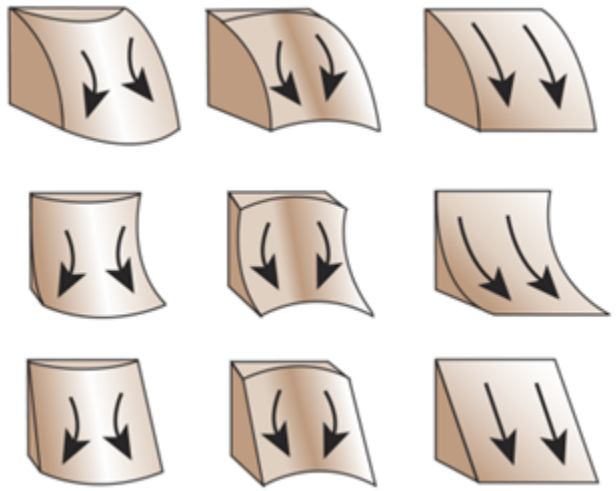

```
planform_curvature = rd.TerrainAttribute(beau, attrib='planform_curvature')
rd.rdShow(planform_curvature, axes=False, cmap='jet', figsize=(8,5.5))
```

Language	Command
Python	`richdem.TerrainAttribute`
C++	*richdem::TA_planform_curvature()*

15.5 Curvature

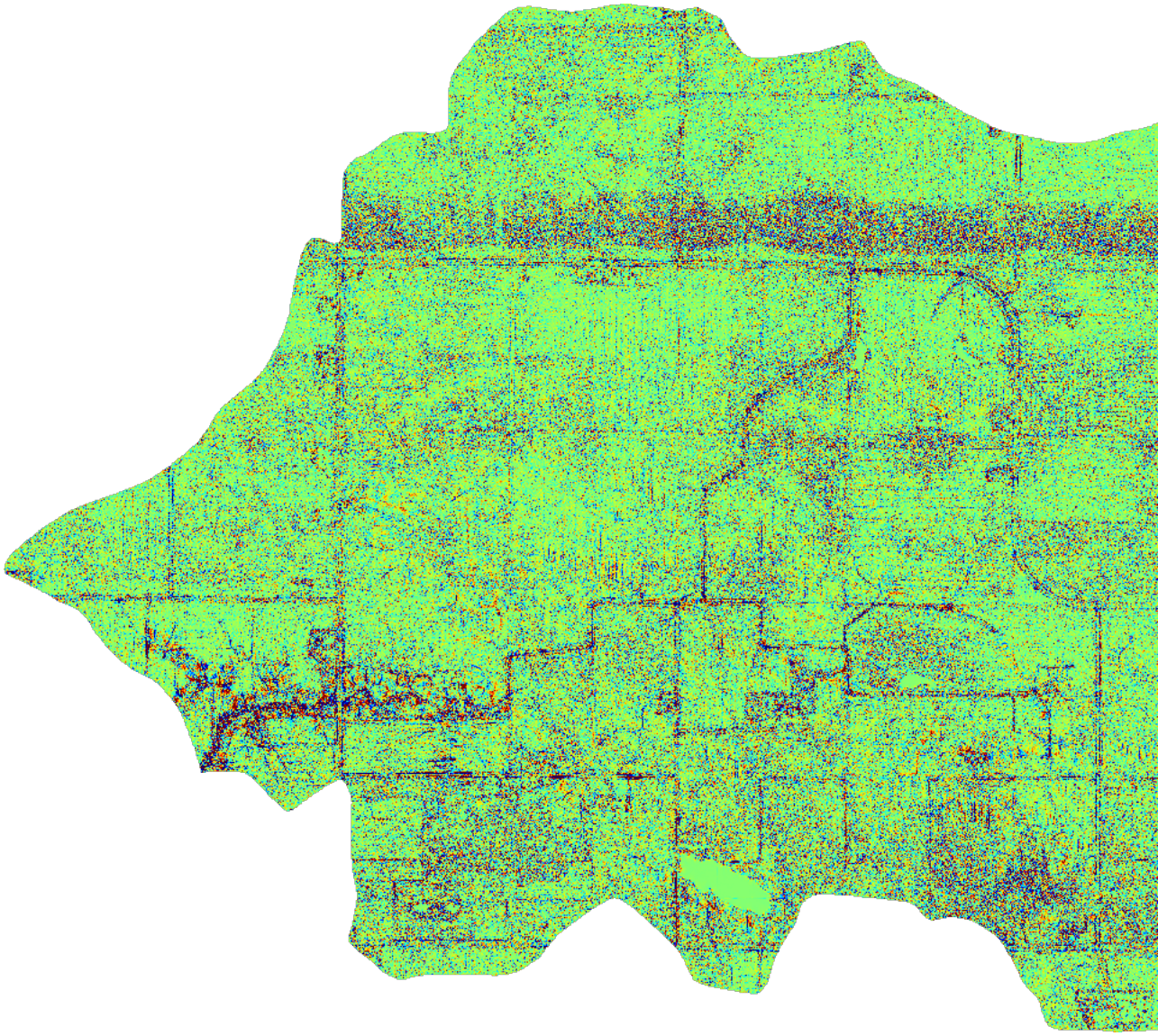
> Zevenbergen, L.W., Thorne, C.R., 1987. Quantitative analysis of land surface topography. Earth surface processes and landforms 12, 47–56.

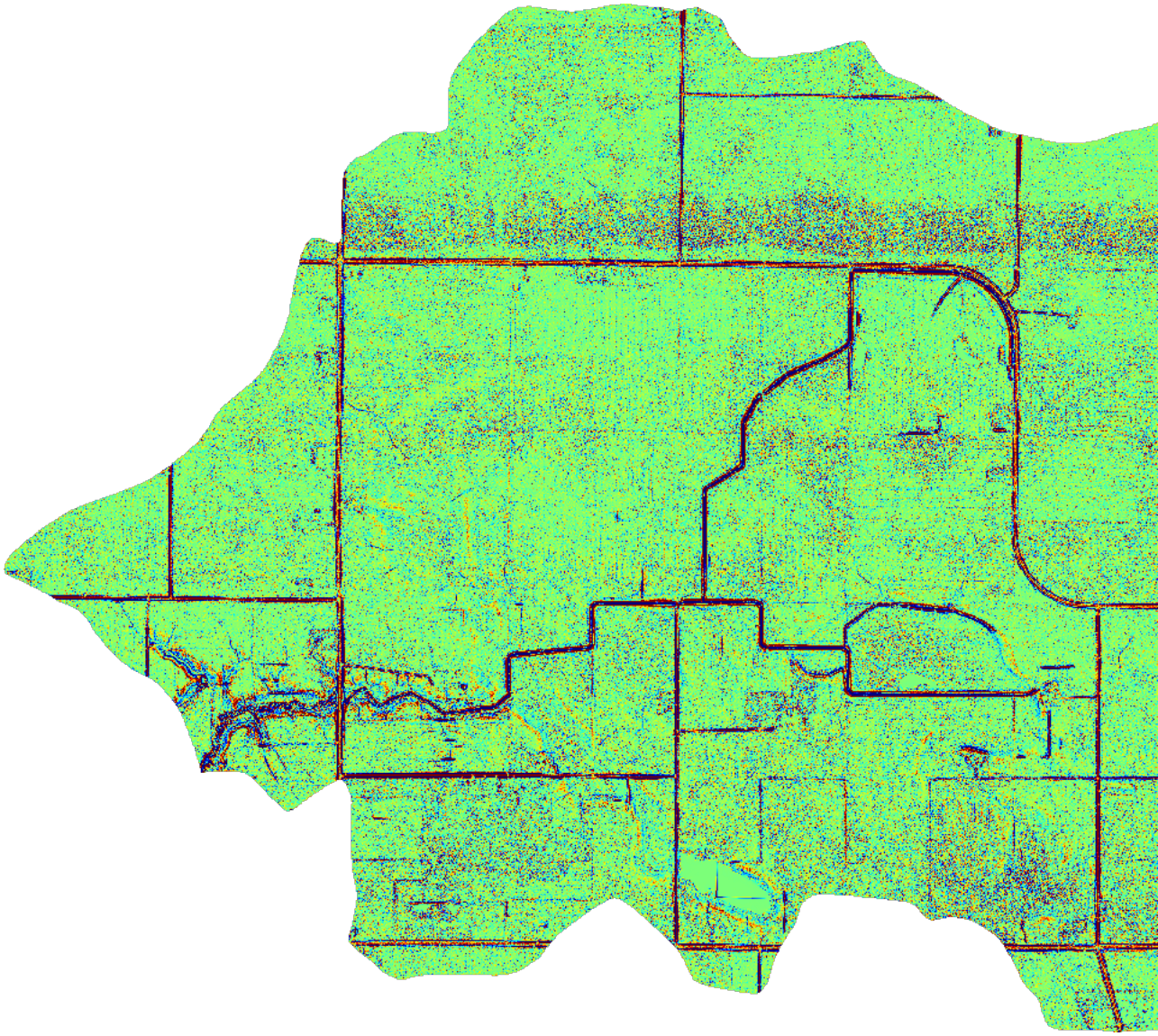
Curvature is calculated by fitting a surface to the focal cell and its neighbours. It combines profile and planform curvature.



```
curvature = rd.TerrainAttribute(beau, attrib='curvature')
rd.rdShow(curvature, axes=False, cmap='jet', figsize=(8,5.5))
```

Language	Command
Python	`richdem.TerrainAttribute`
C++	*richdem::TA_curvature()*





CHAPTER 16

Python Examples

16.1 Depression-filling a DEM and saving it

```
import richdem as rd

#Load DEM
dem     = rd.LoadGDAL("mydem.tif")

#Fill depressions in the DEM. The data is modified in-place to avoid making
#an unnecessary copy. This saves both time and RAM. Note that the function
#has no return value when `in_place=True`.
rd.FillDepressions(dem, epsilon=False, in_place=True)

#Save the DEM
rd.SaveGDAL("mydem_filled.tif", dem)
```

16.2 Comparing filled vs. unfilled DEMs

```
import richdem as rd

#Load DEM
dem     = rd.LoadGDAL("mydem.tif")

#Copy the DEM so we can compare the altered DEM to the unaltered original
demorig = dem.copy()

#Fill depressions in the DEM. The data is modified in place, but, since we
#made a copy above neither time nor memory is really saved.
rd.FillDepressions(dem, epsilon=False, in_place=True)

#Get the difference of the filled and unfilled DEM
```

(continues on next page)

(continued from previous page)

```
diff = dem - demorig

#Display the difference. Do not plot values where there was no difference.
rd.rdShow(diff, ignore_colours=[0])
```

The foregoing could be written more succinctly by using `in_place=False`:

```
import richdem as rd

#Load DEM
demorig = rd.LoadGDAL("mydem.tif")

#Fill depressions in the DEM. By using `in_place=False`, a copy of the DEM
#is made and only this copy is altered. Note that the function now has a
#return value.
dem = rd.FillDepressions(dem, epsilon=False, in_place=False)
```

16.3 The rdarray class

RichDEM has a class `rdarray` which can wrap around NumPy arrays without copying the memory therein. This makes it easy to pass data to RichDEM in a format it understands.

The `rdarray` class works exactly like a NumPy array, but has some special features.

Namely, an `rdarray` has the following properties:

metadata	A dictionary containing metadata in key-value pairs.
projection	A string describing the geographic projection of the dataset
geotrans-form	An array of six floating-point values representing the size and offset of the DEM's cells.
no_data	A value indicating which cell values should be treated as special NoData cells. (See Concepts, TODO)

The `metadata` dictionary contains the special entry `metadata['PROCESSING_HISTORY']`. This entry contains a complete list of everything that RichDEM has done to a dataset. (See Concepts, TODO)

16.4 Using RichDEM without GDAL

GDAL is an optional dependency to RichDEM. In modeling, data is often stored in NumPy arrays and evolved or manipulated without ever being loaded from or saved to disk. To use NumPy arrays, simply wrap them with an `rdarray` as shown below. Details about the `rdarray` are above.

Since an `rdarray` must have a `no_data` value and choosing the `no_data` value incorrectly can produce erroneous results, RichDEM does not automagically convert NumPy arrays. It must be done manually.

```
import richdem as rd
import numpy as np

#Create some NumPy data
npa = np.random.random(size=(100,100))
```

(continues on next page)

(continued from previous page)

```
#Wrap the NumPy data in an rdarray. I want to treat all of the cells as data
#cells, so I use `no_data=-9999` since I know none of my cells will have
#this value.
rda = rd.rdarray(npa, no_data=-9999)

#Fill depressions, modifying in place. At this point, the calculation I
#wanted to do is done and I can throw away the `rda` object.
rd.FillDepressions(rda, in_place=True)
```

CHAPTER 17

RichDEM C++ Reference

template <class *T*>
class A2Array2D

Public Functions

A2Array2D (*std*::string *layoutfile*, int *cachesize*)

A2Array2D (*std*::string *prefix*, int *per_tile_width*, int *per_tile_height*, int *width*, int *height*, int *cachesize*)

template <class U>
A2Array2D (*std*::string *filename_template*, **const** *A2Array2D*<U> &*other*, int *cachesize*)

T &**operator()** (int32_t *tx*, int32_t *ty*, int32_t *x*, int32_t *y*)

T &**operator()** (int32_t *x*, int32_t *y*)

void **makeQuadIndex** (int32_t *x*, int32_t *y*, int32_t &*tx*, int32_t &*ty*, int32_t &*px*, int32_t &*py*) **const**

int32_t **width** () **const**

int32_t **height** () **const**

int32_t **widthInTiles** () **const**

int32_t **heightInTiles** () **const**

int32_t **notNullTiles** () **const**

int32_t **tileWidth** (int32_t *tx*, int32_t *ty*) **const**

int32_t **tileHeight** (int32_t *tx*, int32_t *ty*) **const**

int32_t **stdTileHeight** () **const**

int32_t **stdTileWidth** () **const**

void **setAll**(**const** T &*val*)

bool **isNoData**(int32_t *x*, int32_t *y*)

bool **isNoData**(int32_t *tx*, int32_t *ty*, int32_t *px*, int32_t *py*)

bool **isReadonly**() **const**

GDALDataType **myGDALType**() **const**
Returns the GDAL data type of the *A2Array2D* template type.

void **saveGDAL**(*std*::string *outputname_template*)

void **saveUnifiedGDAL**(**const** *std*::string *outputname*)

void **setNoData**(**const** T &*ndval*)

int32_t **getEvictions**() **const**

bool **isNullTile**(int32_t *tx*, int32_t *ty*) **const**

bool **isEdgeCell**(int32_t *x*, int32_t *y*) **const**

bool **in_grid**(int32_t *x*, int32_t *y*) **const**

bool **isInteriorCell**(int32_t *x*, int32_t *y*) **const**

void **printStamp**(int *size*)

void **loadTile**(int *tx*, int *ty*)

Public Members

bool **flipH** = false

bool **flipV** = false

Private Functions

void **_LoadTile**(int *tile_x*, int *tile_y*)

Private Members

std::vector<bool> **null_tile_quick**

int **quick_width_in_tiles**

int **quick_height_in_tiles**

std::vector<*std*::vector<*WrappedArray2D*>> **data**

LRU<*WrappedArray2D* *> **lru**

int32_t **not_null_tiles** = 0

int64_t **total_width_in_cells** = 0

int64_t **total_height_in_cells** = 0

int32_t **per_tile_width** = 0

int32_t **per_tile_height** = 0

int32_t **evictions** = 0

int64_t **cells_in_not_null_tiles** = 0

T **no_data_to_set**

bool **readonly** = true

Friends

friend richdem::A2Array2D::A2Array2D

template <class *T*>
class Array2D

#include <Array2D.hpp> Class to hold and manipulate GDAL and native rasters.

Array2D manages a two-dimensional raster dataset. Passed a request to load such data, it peeks at the file header and can either load data on construction or wait until a later point. It can also offload data to disk.

Author Richard Barnes (rbarnes@umn.edu)

Array2D permits simple copy construction as well as templated copies, which transfer projections and geotransforms, but not the actual data. This is useful for say, create a flow directions raster which is homologous to a DEM.

Array2D implements two addressing schemes: "xy" and "i". All methods are available in each scheme; users may use whichever is convenient. The xy-scheme accesses raster cells by their xy-coordinates. The i-scheme accesses cells by their address in a flat array. Internally, xy-addresses are converted to i-addresses. i-addressing is frequently faster because it reduces the space needed to store coordinates and requires no addressing mathematics; however, xy-addressing may be more intuitive. It is suggested to develop algorithms using xy-addressing and then convert them to i-addressing if additional speed is desired. The results of the two versions can then be compared against each other to verify that using i-addressing has not introduced any errors.

Unnamed Group

xy_t **view_xoff** = 0
A rectangular subregion of a larger raster can be extracted. These variables store the offsets of this subregion in case the subregion needs to be saved into a raster with other subregions

xy_t **view_yoff** = 0

Public Types

typedef int32_t **xy_t**
xy-addressing data type

typedef uint32_t **i_t**
i-addressing data type

Public Functions

void **saveToCache**(**const** *std*::string &*filename*)
Saves raster to a simply-structure file on disk, possibly using compression.

Post Using *loadData()* after running this function will result in data being loaded from the cache, rather than the original file (if any).

Array2D()

Array2D(*xy_t width*, *xy_t height*, **const** T &*val* = T())
Creates a raster of the specified dimensions.

Parameters

- `width`: Width of the raster
- `height`: Height of the raster
- `val`: Initial value of all the raster's cells. Defaults to the *Array2D* template type's default value

Array2D(*std*::initializer_list<*std*::initializer_list<T>> *values*)

Array2D(T **data0*, **const** *xy_t width*, **const** *xy_t height*)
Wraps a flat array in an *Array2D* object.

Wraps a flat array in an *Array2D* object. The *Array2D* does not take ownership of the data.

Parameters

- `data0`: Pointer to data to wrap
- `width`: Width of the data
- `height`: Height of the data

template <**class** U>
Array2D(**const** *Array2D*<U> &*other*, **const** T &*val* = T())
Create a raster with the same properties and dimensions as another raster. No data is copied between the two.

Parameters

- `other`: Raster whose properties and dimensions should be copied
- `val`: Initial value of all the raster's cells.

template <**class** U>
Array2D(**const** *Array3D*<U> &*other*, **const** T &*val* = T())
Create a raster with the same properties and dimensions as another raster. No data is copied between the two.

Parameters

- `other`: Raster whose properties and dimensions should be copied
- `val`: Initial value of all the raster's cells.

Array2D(**const** *std*::string &*filename*)

Array2D(**const** *std*::string &*filename*, bool *native*, *xy_t xOffset* = 0, *xy_t yOffset* = 0, *xy_t part_width* = 0, *xy_t part_height* = 0, bool *exact* = false, bool *load_data* = true)
TODO.

void **setCacheFilename**(**const** *std*::string &*filename*)

void **dumpData**()
Caches the raster data and all its properties to disk. Data is then purged from RAM.

Post Calls to *loadData()* after this will result in data being loaded from the cache.

void **loadData**()
Loads data from disk into RAM.

If *dumpData()* has been previously called, data is loaded from the cache; otherwise, it is loaded from a GDAL file. No data is loaded if data is already present in RAM.

T ***getData**()
Returns a pointer to the internal data array.

const T ***getData**() **const**
Returns a constant pointer to the internal data array.

T ***data**()
Returns a pointer to the internal data array.

const T ***data**() **const**
Returns a constant pointer to the internal data array.

i_t **size**() **const**
Number of cells in the DEM.

xy_t **width**() **const**
Width of the raster.

xy_t **height**() **const**
Height of the raster.

xy_t **viewXoff**() **const**
X-Offset of this subregion of whatever raster we loaded from.

xy_t **viewYoff**() **const**
Y-Offset of this subregion of whatever raster we loaded from.

bool **empty**() **const**
Returns TRUE if no data is present in RAM.

T **noData**() **const**
Returns the NoData value of the raster. Cells equal to this value sould generally not be used in calculations. But note that the *isNoData()* method is a much better choice for testing whether a cell is NoData or not.

T **min**() **const**
Finds the minimum value of the raster, ignoring NoData cells.

T **max**() **const**
Finds the maximum value of the raster, ignoring NoData cells.

void **replace**(**const** T *oldval*, **const** T *newval*)
Replace one cell value with another throughout the raster. Can operate on NoData cells.

Parameters

- `oldval`: Value to be replaced
- `newval`: Value to replace 'oldval' with

i_t **countval**(**const** T *val*) **const**
Counts the number of occurrences of a particular value in the raster. Can operate on NoData cells.

Return The number of times 'val' appears in the raster. Will be 0 if raster is not loaded in RAM.

Parameters

- `val`: Value to be be counted

i_t **i0**() **const**

void **iToxy**(**const** *i_t i*, *xy_t &x*, *xy_t &y*) **const**
Convert from index coordinates to x,y coordinates.

Parameters

- `i`: Index coordinate
- `x`: X-coordinate of i
- `y`: Y-coordinate of i

i_t **xyToI**(*xy_t x*, *xy_t y*) **const**
Convert from x,y coordinates to index coordinates.

Return Returns the index coordinate i of (x,y)

Parameters

- `x`: X-coordinate to convert
- `y`: Y-coordinate to convert

i_t **nToI**(*i_t i*, *xy_t dx*, *xy_t dy*) **const**
Given a cell identified by an i-coordinate, return the i-coordinate of the neighbour identified by dx,dy.

Return i-coordinate of the neighbour. Usually referred to as 'ni'

Parameters

- `i`: i-coordinate of cell whose neighbour needs to be identified
- `dx`: x-displacement of the neighbour from i
- `dy`: y-displacement of the neighbour from i

i_t **getN**(*i_t i*, uint8_t *n*) **const**
Given a cell identified by an i-coordinate, return the i-coordinate of the neighbour identified by n.

Return i-coordinate of the neighbour. Usually referred to as 'ni'

Parameters

- `i`: i-coordinate of cell whose neighbour needs to be identified
- `n`: Neighbour to be identified

int **nshift**(**const** uint8_t *n*) **const**
Return the offset of the neighbour cell identified by n.

Return Offset of the neighbour n

Parameters

- n: Neighbour for which offset should be retrieved

bool **operator==**(**const** *Array2D*<T> &*o*) **const**
Determine if two rasters are equivalent based on dimensions, NoData value, and their data.

bool **isNoData**(*xy_t x*, *xy_t y*) **const**
Whether or not a cell is NoData using x,y coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- x: X-coordinate of cell to test
- y: Y-coordinate of cell to test

bool **isNoData**(*i_t i*) **const**
Whether or not a cell is NoData using i coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- i: i-coordinate of cell to test

bool **isData**(*xy_t x*, *xy_t y*) **const**
Whether or not a cell is NoData using x,y coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- x: X-coordinate of cell to test
- y: Y-coordinate of cell to test

bool **isData**(*i_t i*) **const**
Whether or not a cell is NoData using i coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- i: i-coordinate of cell to test

void **flipVert**()
Flips the raster from top to bottom.

void **flipHorz**()
Flips the raster from side-to-side.

void **transpose**()
Flips the raster about its diagonal axis, like a matrix tranpose.

bool **inGrid**(*xy_t x*, *xy_t y*) **const**
Test whether a cell lies within the boundaries of the raster.

Return TRUE if cell lies within the raster

Parameters

- x: X-coordinate of cell to test
- y: Y-coordinate of cell to test

bool **isEdgeCell**(*xy_t x*, *xy_t y*) **const**
Test whether a cell lies on the boundary of the raster.

Return TRUE if cell lies on the raster's boundary

Parameters

- x: X-coordinate of cell to test
- y: X-coordinate of cell to test

bool **isTopLeft**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is the top left of the DEM.

Return True, if the (x,y) pair is the top left of the DEM; otherwise, false

bool **isTopRight**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is the top right of the DEM.

Return True, if the (x,y) pair is the top right of the DEM; otherwise, false

bool **isBottomLeft**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is the bottom left of the DEM.

Return True, if the (x,y) pair is the bottom left of the DEM; otherwise, false

bool **isBottomRight**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is the bottom right of the DEM.

Return True, if the (x,y) pair is the bottom right of the DEM; otherwise, false

bool **isTopRow**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is in the top row of the DEM.

Return True, if the (x,y) pair is in the top row of the DEM; otherwise, false

bool **isBottomRow**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is in the bottom row of the DEM.

Return True, if the (x,y) pair is in the bottom row of the DEM; otherwise, false

bool **isLeftCol**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is in the left column of the DEM.

Return True, if the (x,y) pair is in the left column of the DEM; otherwise, false

bool **isRightCol**(*xy_t x*, *xy_t y*) **const**
Determines whether an (x,y) pair is in the right column of the DEM.

Return True, if the (x,y) pair is in the right column of the DEM; otherwise, false

bool **isEdgeCell**(*i_t i*) **const**
Test whether a cell lies on the boundary of the raster.

Return TRUE if cell lies on the raster's boundary

Parameters

- `i`: i-coordinate of cell to test

void **setNoData**(**const** T &*ndval*)
Sets the NoData value of the raster.

Parameters

- `ndval`: Value to change NoData to

void **setAll**(**const** T *val*)
Sets all of the raster's cells to 'val'.

Parameters

- `val`: Value to change the cells to

void **resize**(**const** *xy_t width0*, **const** *xy_t height0*, **const** T &*val0* = T())
Resize the raster. Note: this clears all the raster's data.

Parameters

- `width0`: New width of the raster
- `height0`: New height of the raster
- `val0`: Value to set all the cells to. Defaults to the raster's template type default value

template <**class** U>
void **resize**(**const** *Array2D*<U> &*other*, **const** T &*val* = T())

void **expand**(uint64_t *new_width*, uint64_t *new_height*, **const** T *val*)
Makes a raster larger and retains the raster's old data, similar to resize.

Note: Using this command requires RAM equal to the sum of the old raster and the new raster. The old raster is placed in the upper-left of the new raster.

Parameters

- `new_width`: New width of the raster. Must be >= the old width.
- `new_height`: New height of the raster. Must be >= the old height.
- `val`: Value to set the new cells to

void **countDataCells**() **const**
Counts the number of cells which are not NoData.

i_t **numDataCells**() **const**
Returns the number of cells which are not NoData. May count them.

Return Returns the number of cells which are not NoData.

T &**operator()** (*i_t i*)
Return cell value based on i-coordinate.

Return The value of the cell identified by 'i'

Parameters

- `i`: i-coordinate of cell whose data should be fetched.

T **operator()** (*i_t i*) **const**
Return cell value based on i-coordinate.

Return The value of the cell identified by 'i'

Parameters

- `i`: i-coordinate of cell whose data should be fetched.

T &**operator()** (*xy_t x*, *xy_t y*)
Return cell value based on x,y coordinates.

Return The value of the cell identified by x,y

Parameters

- `x`: X-coordinate of cell whose data should be fetched.
- `y`: Y-coordinate of cell whose data should be fetched.

T **operator()** (*xy_t x*, *xy_t y*) **const**
Return cell value based on x,y coordinates.

Return The value of the cell identified by x,y

Parameters

- `x`: X-coordinate of cell whose data should be fetched.
- `y`: Y-coordinate of cell whose data should be fetched.

std::vector<T> **topRow**() **const**
Returns a copy of the top row of the raster.

Return A vector containing a copy of the top row of the raster

std::vector<T> **bottomRow**() **const**
Returns a copy of the bottom row of the raster.

Return A vector containing a copy of the bottom row of the raster

std::vector<T> **leftColumn**() **const**
Returns a copy of the left column of the raster.

Top to bottom is reoriented as left to right.

Return A vector containing a copy of the left column of the raster

std::vector<T> **rightColumn**() **const**
Returns a copy of the right column of the raster.

Top to bottom is reoriented as left to right.

Return A vector containing a copy of the right column of the raster

void **setRow**(*xy_t y*, **const** T &*val*)
Sets an entire row of a raster to a given value.

Parameters

- `y`: The row to be set
- `val`: The value to set the row to

void **setCol**(*xy_t x*, **const** T &*val*)
Sets an entire column of a raster to a given value.

Parameters

- `x`: The column to be set
- `val`: The value to set the column to

void **setEdges**(**const** T &*val*)
Sets the edges of the array to a given value.

Parameters

- `val`: The value to set the edges to

std::vector<T> **getRowData**(*xy_t y*) **const**
Returns a copy of an arbitrary row of the raster.

Return A vector containing a copy of the selected row

Parameters

- `y`: The row to retrieve

std::vector<T> **getColData**(*xy_t x*) **const**
Returns a copy of an arbitrary column of the raster.

Return A vector containing a copy of the selected column

Parameters

- `x`: The column to retrieve

void **clear**()
Clears all raster data from RAM.

template <class U>
void **templateCopy**(**const** *Array2D*<U> &*other*)
Copies the geotransform, projection, and basename of another raster.

Parameters

- `other`: Raster to copy from

void **printStamp**(int *size*, *std*::string *msg* = "") **const**
Output a square of cells useful for determining raster orientation.

This method prints out a square block of cells whose upper-left corner is the (integer-division) center of the raster.

Stamps are only shown if the SHOW_STAMPS preprocessor variable is set.

Since algorithms may have to flip rasters horizontally or vertically before manipulating them, it is important that all algorithms work on data in the same orientation. This method, used in testing, helps a user ensure that their algorithm is orientating data correctly.

Parameters

- `size`: Output stamp will be size x size
- `msg`: Message to print prior to the stamp

void **printBlock**(**const** int *radius*, **const** *xy_t x0*, **const** *xy_t y0*, bool *color* = false, **const** *std*::string *msg* = "", **const** int *fwidth* = 5, **const** int *precision* = 0) **const**
Prints a square of cells centered at x,y. Useful for debugging.

Parameters

- `radius`: Output stamp will be 2*radius x 2*radius
- `x0`: X-coordinate of block center
- `y0`: Y-coordinate of block center
- `color`: Print the (x,y) cell in colour?
- `msg`: Optional message to print above the block
- `fwidth`: Field width in which to print each array value
- `precision`: Number of decimal digits to display

void **printAll**(**const** *std*::string *msg* = "", **const** int *fwidth* = 5, **const** int *precision* = 0) **const**
Prints the entire array.

Parameters

- `msg`: Optional message to print above the block
- `fwidth`: Field width to print to
- `precision`: Precision (number of decimal digits) to print

void **printAllFlows**(**const** *std*::string *msg* = "", **const** int *fwidth* = 5) **const**
Prints the entire array as flow directions.

Parameters

- `msg`: Optional message to print above the block
- `fwidth`: Field width to print to
- `precision`: Precision (number of decimal digits) to print

void **printBlockIndices**(**const** int *radius*, **const** *xy_t x0*, **const** *xy_t y0*, bool *color* = false, **const** *std*::string *msg* = "", **const** int *fwidth* = 5) **const**

Prints a square of cells centered at x,y indicating the index of each.

Parameters

- `radius`: Output stamp will be 2*radius x 2*radius
- `x0`: X-coordinate of block center
- `y0`: Y-coordinate of block center
- `color`: Print the (x,y) cell in colour?
- `msg`: Optional message to print above the block
- `fwidth`: Field width in which to print each array value

void **printAllIndices**(**const** *std*::string *msg* = "") **const**

Prints the flat indices of the entire array.

Parameters

- `msg`: Optional message to print above the block

double **getCellArea**() **const**

Get the area of an individual cell in square projection units.

Return The area of the cell in square projection units

double **getCellLengthX**() **const**

Get the length of a cell along the raster's horizontal axis.

Return The length of the cell along the raster's horizontal axis

double **getCellLengthY**() **const**

Get the length of a cell along the raster's horizontal axis.

Return The length of the cell along the raster's horizontal axis

void **scale**(**const** double *x*)

Multiplies the entire array by a scalar.

Parameters

- `x`: Value to multiply array by

bool **owned**() **const**

Public Members

std::string **filename**
File, if any, from which the data was loaded.

std::string **basename**
Filename without path or extension.

std::vector<double> **geotransform**
Geotransform of the raster.

std::string **projection**
Projection of the raster.

std::map<*std*::string, *std*::string> **metadata**
Raster's metadata in key-value pairs.

Public Static Attributes

const *i_t* **NO_I** = std::numeric_limits<*i_t*>::*max*()

Private Functions

void **loadNative**(**const** *std*::string &*filename*, bool *load_data* = true)
TODO.

Private Members

std::array<int, 9> **_nshift**
Offset to neighbouring cells;.

ManagedVector<T> **_data**
this improves caching versus a 2D array

Holds the raster data in a 1D array

T **no_data** = -1
NoData value of the raster.

i_t **num_data_cells** = NO_I
Number of cells which are not NoData.

xy_t **view_width** = 0
Height of raster in cells.

xy_t **view_height** = 0
Width of raster in cells.

bool **from_cache**
If TRUE, *loadData()* loads data from the cache assuming the Native format. Otherwise, it assumes it is loading from a GDAL file.

Friends

friend richdem::Array2D::Array2D

friend richdem::Array2D::Array3D

template <class *T*>
class Array3D

#include <Array3D.hpp> Class to hold and 2D rasters with neighbour information.

Array3D manages a two-dimensional raster dataset with information about neighbours.

Author Richard Barnes (rbarnes@umn.edu)

Array3D implements two addressing schemes: "xyn" and "i". All methods are available in each scheme; users may use whichever is convenient. The xyn- scheme accesses raster cells by their xyn-coordinates. The i-scheme accesses cells by their address in a flat array. Internally, xyn-addresses are converted to i-addresses. i-addressing is frequently faster because it reduces the space needed to store coordinates and requires no addressing mathematics; however, xyn-addressing may be more intuitive. It is suggested to develop algorithms using xyn-addressing and then convert them to i-addressing if additional speed is desired. The results of the two versions can then be compared against each other to verify that using i-addressing has not introduced any errors.

Unnamed Group

xy_t **view_xoff** = 0
A rectangular subregion of a larger raster can be extracted. These variables store the offsets of this subregion in case the subregion needs to be saved into a raster with other subregions

xy_t **view_yoff** = 0

Public Types

typedef int32_t **xy_t**
xy-addressing data type

typedef *std*::size_t **i_t**
i-addressing data type

typedef uint8_t **n_t**
neighbour addressing data type

Public Functions

Array3D()

Array3D(*xy_t width*, *xy_t height*, **const** T &*val* = T())
Creates a raster of the specified dimensions.

Parameters

- `width`: Width of the raster
- `height`: Height of the raster
- `val`: Initial value of all the raster's cells. Defaults to the *Array3D* template type's default value

Array3D(T *data0*, **const** *xy_t width*, **const** *xy_t height*)
Wraps a flat array in an *Array3D* object.

Wraps a flat array in an *Array3D* object. The *Array3D* does not take ownership of the data.

Parameters

- `data0`: Pointer to data to wrap
- `width`: Width of the data
- `height`: Height of the data

template <class U>
Array3D(**const** *Array3D*<U> &*other*, **const** T &*val* = T())
Create a raster with the same properties and dimensions as another raster. No data is copied between the two.

Parameters

- `other`: Raster whose properties and dimensions should be copied
- `val`: Initial value of all the raster's cells.

template <class U>
Array3D(**const** *Array2D*<U> &*other*, **const** T &*val* = T())
Create a raster with the same properties and dimensions as another raster. No data is copied between the two.

Parameters

- `other`: Raster whose properties and dimensions should be copied
- `val`: Initial value of all the raster's cells.

T ***getData**()
Returns a pointer to the internal data array.

i_t **size**() **const**
Number of cells in the DEM.

xy_t **width**() **const**
Width of the raster.

xy_t **height**() **const**
Height of the raster.

xy_t **viewXoff**() **const**
X-Offset of this subregion of whatever raster we loaded from.

xy_t **viewYoff**() **const**
Y-Offset of this subregion of whatever raster we loaded from.

bool **empty**() **const**
Returns TRUE if no data is present in RAM.

T **noData**() **const**
Returns the NoData value of the raster. Cells equal to this value sould generally not be used in calculations. But note that the *isNoData()* method is a much better choice for testing whether a cell is NoData or not.

i_t **i0**() **const**

i_t **xyToI**(*xy_t x*, *xy_t y*, *n_t n*) **const**
Convert from x,y coordinates to index coordinates.

Return Returns the index coordinate i of (x,y)

Parameters

- `x`: X-coordinate to convert
- `y`: Y-coordinate to convert

bool **operator==**(**const** *Array3D*<T> &*o*) **const**
Determine if two rasters are equivalent based on dimensions, NoData value, and their data.

bool **isNoData**(*xy_t x*, *xy_t y*) **const**
Whether or not a cell is NoData using x,y coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- `x`: X-coordinate of cell to test
- `y`: Y-coordinate of cell to test

bool **isNoData**(*i_t i*) **const**
Whether or not a cell is NoData using i coordinates.

Return Returns TRUE if the cell is NoData

Parameters

- `i`: i-coordinate of cell to test

bool **inGrid**(*xy_t x*, *xy_t y*) **const**
Test whether a cell lies within the boundaries of the raster.

Return TRUE if cell lies within the raster

Parameters

- `x`: X-coordinate of cell to test
- `y`: Y-coordinate of cell to test

void **setNoData**(**const** T &*ndval*)
Sets the NoData value of the raster.

Parameters

- `ndval`: Value to change NoData to

void **setAll**(**const** T *val*)
Sets all of the raster's cells to 'val'.

Parameters

- `val`: Value to change the cells to

void **resize**(**const** *xy_t width0*, **const** *xy_t height0*, **const** T &*val0* = T())
Resize the raster. Note: this clears all the raster's data.

Parameters

- `width0`: New width of the raster
- `height0`: New height of the raster
- `val0`: Value to set all the cells to. Defaults to the raster's template type default value

template <**class** U>
void **resize**(**const** *Array3D*<U> &*other*, **const** T &*val* = T())

void **countDataCells**() **const**
Counts the number of cells which are not NoData.

i_t **numDataCells**() **const**
Returns the number of cells which are not NoData. May count them.

Return Returns the number of cells which are not NoData.

T &**operator()**(*xy_t x*, *xy_t y*, *n_t n*)
Return cell value based on x,y coordinates.

Return The value of the cell identified by x,y

Parameters

- `x`: X-coordinate of cell whose data should be fetched.
- `y`: Y-coordinate of cell whose data should be fetched.

T **operator()**(*xy_t x*, *xy_t y*, *n_t n*) **const**
Return cell value based on x,y coordinates.

Return The value of the cell identified by x,y

Parameters

- `x`: X-coordinate of cell whose data should be fetched.
- `y`: Y-coordinate of cell whose data should be fetched.

T **getIN**(*i_t i*, *n_t n*) **const**

T &**getIN**(*i_t i*, *n_t n*)

void **clear**()
Clears all raster data from RAM.

bool **owned**() **const**

Public Members

std::string **filename**
File, if any, from which the data was loaded.

std::string **basename**
Filename without path or extension.

std::vector<double> **geotransform**
Geotransform of the raster.

std::string **projection**
Projection of the raster.

std::map<*std*::string, *std*::string> **metadata**
Raster's metadata in key-value pairs.

Public Static Attributes

const *i_t* **NO_I** = std::numeric_limits<*i_t*>::max()

Private Members

ManagedVector<T> **data**
Holds the raster data in a 1D array this improves caching versus a 2D array

T **no_data**
NoData value of the raster.

i_t **num_data_cells** = NO_I
Number of cells which are not NoData.

xy_t **view_width** = 0
Height of raster in cells.

xy_t **view_height** = 0
Width of raster in cells.

Friends

friend richdem::Array3D::Array2D

friend richdem::Array3D::Array3D

class GridCell
#include <*grid_cell.hpp*> Stores the (x,y) coordinates of a grid cell.

Subclassed by *richdem::GridCellZ< elev_t >*, *richdem::GridCellZ< double >*, *richdem::GridCellZ< float >*

Public Functions

GridCell ( )
Initiate the grid cell without coordinates; should generally be avoided.

GridCell (int *x*, int *y*)
Initiate the grid cell to the coordinates (x0,y0)

Public Members

int **x**
Grid cell's x-coordinate.

int **y**
Grid cell's y-coordinate

template <class *elev_t*>

class GridCellZ

#include <grid_cell.hpp> Stores the (x,y,z) coordinates of a grid cell; useful for priority sorting with GridCellZ_pq.

Inherits from *richdem::GridCell*

Subclassed by *richdem::GridCellZk_high< elev_t >*, *richdem::GridCellZk_low< elev_t >*

Public Functions

GridCellZ(int *x*, int *y*, elev_t *z*)

GridCellZ()

bool **isnan**() **const**

bool **operator>**(**const** *GridCellZ*<elev_t> &*a*) **const**

Public Members

elev_t **z**

Grid cell's z-coordinate.

template <>
template<>
class GridCellZ<double>

#include <grid_cell.hpp> An (x,y,z) cell with NaNs taken as infinitely small numbers.

Inherits from *richdem::GridCell*

Public Functions

GridCellZ(int *x*, int *y*, double *z*)

GridCellZ()

bool **isnan**() **const**

bool **operator<**(**const** *GridCellZ*<double> &*a*) **const**

bool **operator>**(**const** *GridCellZ*<double> &*a*) **const**

bool **operator>=**(**const** *GridCellZ*<double> &*a*) **const**

bool **operator<=**(**const** *GridCellZ*<double> &*a*) **const**

bool **operator==**(**const** *GridCellZ*<double> &*a*) **const**

bool **operator!=**(**const** *GridCellZ*<double> &*a*) **const**

Public Members

double **z**

Grid cell's z-coordinate.

template <>
template<>

class GridCellZ<float>

#include <grid_cell.hpp> An (x,y,z) cell with NaNs taken as infinitely small numbers.

Inherits from *richdem::GridCell*

Public Functions

GridCellZ(int *x*, int *y*, float *z*)

GridCellZ()

bool **isnan**() **const**

Compare cells based on elevation. (TODO: Distribute)

bool **operator<**(**const** *GridCellZ*<float> &*a*) **const**

bool **operator>**(**const** *GridCellZ*<float> &*a*) **const**

bool **operator>=**(**const** *GridCellZ*<float> &*a*) **const**

bool **operator<=**(**const** *GridCellZ*<float> &*a*) **const**

bool **operator==**(**const** *GridCellZ*<float> &*a*) **const**

bool **operator!=**(**const** *GridCellZ*<float> &*a*) **const**

Public Members

float **z**

Grid cell's z-coordinate.

template <class *elev_t*>
class GridCellZk_high

#include <grid_cell.hpp> Stores the (x,y,z) coordinates of a grid cell and a priority indicator k; used by GridCellZk_pq to return cells in order of elevation from lowest to highest. If elevations are equal then the cell added last is popped from the priority queue.

Inherits from *richdem::GridCellZ< elev_t >*

Public Functions

GridCellZk_high(int *x*, int *y*, elev_t *z*, int *k*)

GridCellZk_high()

bool **operator>**(**const** *GridCellZk_high*<elev_t> &*a*) **const**

Public Members

int **k**

Used to store an integer to make sorting stable.

template <**typename** T>

class GridCellZk_high_pq

#include <grid_cell.hpp> A priority queue of GridCellZk, sorted by ascending height or, if heights are equal, by the order of insertion. "high" means that cells with a higher insertion number (inserted later) are returned first.

Inherits from std::priority_queue< GridCellZk_high< T >, std::vector< GridCellZk_high< T > >, std::greater< GridCellZk_high< T > > >

Public Functions

void **push**()

void **emplace**(int *x*, int *y*, T *z*)

Private Members

uint64_t **count** = 0

template <class *elev_t*>
class GridCellZk_low

#include <grid_cell.hpp> Stores the (x,y,z) coordinates of a grid cell and a priority indicator k; used by GridCellZk_pq to return cells in order of elevation from lowest to highest. If elevations are equal then the cell added first is popped from the priority queue.

Inherits from *richdem::GridCellZ< elev_t >*

Public Functions

GridCellZk_low(int *x*, int *y*, elev_t *z*, int *k*)

GridCellZk_low()

bool **operator>**(**const** *GridCellZk_low*<elev_t> &*a*) **const**

Public Members

int **k**

Used to store an integer to make sorting stable.

template <typename *T*>
class GridCellZk_low_pq

#include <grid_cell.hpp> A priority queue of GridCellZk, sorted by ascending height or, if heights are equal, by the order of insertion. "low" means that cells with a lower insertion number (inserted earlier) are returned first.

Inherits from std::priority_queue< GridCellZk_low< T >, std::vector< GridCellZk_low< T > >, std::greater< GridCellZk_low< T > > >

Public Functions

void **push**()

void **emplace**(int *x*, int *y*, T *z*)

Private Members

uint64_t **count** = 0

class LayoutfileReader

#include <Layoutfile.hpp> Used for reading a layoutfile describing a tiled dataset.

The class acts as a generator. The layoutfile is read on construction and its contents retrieved with *next()*. The Layoutfile specification can be found in *Layoutfile.hpp*.

Public Functions

LayoutfileReader(*std*::string *layout_filename*)

Construct a new *LayoutfileReader* object reading from a given file.

Author Richard Barnes

Parameters

- `layout_filename`: Layoutfile to read from.

bool **next**()

Advance the reader to the next layoutfile entry.

Return True if reader advanced successfully; false if not.

bool **newRow**() **const**

Return True if the current entry is the beginning of a new row.

const *std*::string &**getFilename**() **const**

Return The current entry's filename without the path (e.g. "file.ext").

const *std*::string &**getBasename**() **const**

Return The current entry's filename without the path or extension (e.g. "file").

const *std*::string **getFullPath**() **const**

Return The current entry's path + filename (e.g. "path/to/file.ext").

const *std*::string **getGridLocName**() **const**

Return a string representation of the current entry's coordinates.

A layoutfile is a 2D grid of file names. This method returns the current entry's position in that grid as `<X>_<Y>`

Return Current entry's position as a string of the form <X>_<Y>

const *std*::string &**getPath**() **const**

Return Path of layoutfile: of "path/to/layoutfile.layout" returns "path/to/".

bool **isNullTile**() **const**

Return True if the current entry was a blank

int **getX**() **const**

Return X-coordinate of the current entry.

int **getY**() **const**

Return Y-coordinate of the current entry.

Private Members

std::vector<*std*::vector<*std*::string>> **fgrid**
Stores the grid of filenames extracted from the layoutfile.

int **gridy** = -1

int **gridx** = -2

int **new_row** = false

std::string **filename**

std::string **basename**

std::string **path**

class LayoutfileWriter
#include <Layoutfile.hpp> Used for creating a layoutfile describing a tiled dataset.

The class acts as an inverse generator. The layoutfile is created on construction and its contents appended to with *addEntry()*. The Layoutfile specification can be found in *Layoutfile.hpp*.

Public Functions

LayoutfileWriter(*std*::string *layout_filename*)
File output stream for the layout file.

Constructs a new writer object

Parameters

- `layout_filename`: Path+Filename of layoutfile to write

~LayoutfileWriter()

void **addRow**()
Adds a new row to the layoutfile.

void **addEntry**(*std*::string *filename*)
Add a new entry to the layout file.

Parameters

- `filename`: File to add. Use `filename=""` to indicate a null tile.

Private Members

int **gridx**

int **gridy**
Current column being written to.

std::string **path**
Current row being written to.

std::ofstream **flout**
Path of layoutfile.

template <class *T*>
class LRU
#include <*lru.hpp*> A Least-Recently Used (*LRU*) cache.

Public Functions

LRU()
Construct a new *LRU*.

void **insert**(**const** T &*entry*)
Insert an item into the *LRU*.

The item is either added to the queue or its entry is moved to the top of the queue. If the item is new and the length of the queue is greater than maxlen, then the least recently seen item is evicted from the queue.

Parameters

- `entry`: The item to add to the queue.

int **size**() **const**
Returns the number of itmes in the *LRU* cache.

Return Number of items in the *LRU* cache

bool **full**() **const**
Is the *LRU* cache full?

Return True if the *LRU* cache is full; otherwise, false.

void **setCapacity**(int *n*)
Set the maximum capacity of the *LRU* cache.

int **getCapacity**() **const**
Returns the capacity of the *LRU* cache.

Return The capacity of the *LRU* cache

T **back**() **const**
Return the least-recently used item in the *LRU* cache.

Return The least-recently used item in the *LRU* cache.

void **pop_back**()
Evict the least-recently used item out of the *LRU* cache.

void **prune**()
Evict items from the *LRU* cache until it is within its capacity.

Public Members

cachetype **cache**
The cache.

Private Types

typedef *std*::list<T> **cachetype**
Container used for storage by the cache.

Private Members

int **len**
Number of items in the cache.

int **maxlen**
Maximum size the cache is allowed to be.

T **last**
Copy of the last inserted item. Speeds up insertion. TODO: This'd be better as a pointer.

std::unordered_map<T, **typename** *std*::list<T>::iterator> **visited**
Used for O(1) access to members.

template <class *T*>
class ManagedVector
#include <ManagedVector.hpp> *ManagedVector* works like a regular vector, but can wrap external memory.

Public Functions

ManagedVector()
Creates an empty *ManagedVector*.

ManagedVector(*std*::size_t *size*, T *default_val* = T())
Creates a *ManagedVector* with `size` members each set to `default_val`

Parameters

- `size`: Number of elements to be created in the vector
- `default_val`: Initial value of the elements

ManagedVector(T *data, *std*::size_t *size*)
Creates a *ManagedVector* which wraps `data0` of length `size0`

Parameters

- `data`: Memory to wrap
- `size`: Number of elements to wrap

template <**class** U>
ManagedVector(**const** *ManagedVector*<U> &*other*)

ManagedVector(**const** *ManagedVector*<T> &*other*)

template <**class** U>

ManagedVector(*ManagedVector*<U> &&*other*)

~ManagedVector()

template <**class** U>
ManagedVector<T> &**operator=**(**const** *ManagedVector*<U> &*other*)

ManagedVector<T> &**operator=**(**const** *ManagedVector*<T> &*other*)

template <**class** U>
ManagedVector<T> &**operator=**(*ManagedVector*<U> &&*other*)
Move assignment operator

T ***data**()
Get a raw pointer to the managed data

Return A raw pointer to the managed data

const T ***data**() **const**
Get a raw constant pointer to the managed data

Return A raw constant pointer to the managed data

bool **empty**() **const**
Are there more than zero elements being managed?

Return True, if zero elements are managed; otherwise, false

std::size_t **size**() **const**
Get the number of elements being managed

Return The number of elements being managed

bool **owned**() **const**
Determine whether the *ManagedVector* owns the memory it is managing

Return True, if this *ManagedVector* owns its memory; otherwise, false

void **resize**(*std*::size_t *new_size*)

T &**operator[]**(*std*::size_t *i*)

const T &**operator[]**(*std*::size_t *i*) **const**

Private Members

std::unique_ptr<T[]> **_data**

bool **_owned** = true
If this is true, we are responsible for clean-up of the data.

std::size_t **_size** = 0
Number of elements being managed.

Friends

friend richdem::ManagedVector::ManagedVector

class ProgressBar

#include <ProgressBar.hpp> Manages a console-based progress bar to keep the user entertained.

Defining the global `RICHDEM_NO_PROGRESS` will disable all progress operations, potentially speeding up a program. The look of the progress bar is shown in *ProgressBar.hpp*.

Public Functions

void **start**(uint32_t *total_work*)

Start/reset the progress bar.

Parameters

- `total_work`: The amount of work to be completed, usually specified in cells.

void **update**(uint32_t *work_done0*)

Update the visible progress bar, but only if enough work has been done.

Define the global `RICHDEM_NO_PROGRESS` flag to prevent this from having an effect. Doing so may speed up the program's execution.

ProgressBar &**operator++**()

Increment by one the work done and update the progress bar.

double **stop**()

Stop the progress bar. Throws an exception if it wasn't started.

Return The number of seconds the progress bar was running.

double **time_it_took**()

Return Return the time the progress bar ran for.

uint32_t **cellsProcessed**() **const**

Private Functions

void **clearConsoleLine**() **const**

Clear current line on console so a new progress bar can be written.

Private Members

uint32_t **total_work**

Total work to be accomplished.

uint32_t **next_update**

Next point to update the visible progress bar.

uint32_t **call_diff**

Interval between updates in work units.

uint32_t **work_done**

uint16_t **old_percent**
Old percentage value (aka: should we update the progress bar) TODO: Maybe that we do not need this.

Timer **timer**
Used for generating ETA.

class StreamLogger

Public Functions

StreamLogger(*LogFlag flag0*, **const** char **file0*, **const** char **func0*, unsigned *line0*)

~StreamLogger()

template <**typename** T>
StreamLogger &**operator<<**(**const** T &*t*)

StreamLogger &**operator<<**(*std*::ostream &(**f*)) *std*::ostream&

Private Members

LogFlag **flag**

const char ***file**

const char ***func**

unsigned **line**

std::ostringstream **ss**

class TA_Setup_Curves_Vars

Public Members

double **L**

double **D**

double **E**

double **F**

double **G**

double **H**

class TA_Setup_Vars
#include <terrain_attributes.hpp> Calculate a variety of terrain attributes.

This calculates a variety of terrain attributes according to the work of Burrough 1998's "Principles of Geographical Information Systems" (p. 190). Algorithms and helpful ArcGIS pages are noted in comments in the code.

Author Richard Barnes (rbarnes@umn.edu), Burrough (1998)

Pre This function should never be called on a NoData cell

Parameters

- `&elevations`: An elevation grid

- `x0`: X coordinate of cell to perform calculation on
- `y0`: Y coordinate of cell to perform calculation on
- `&rise_over_run`: Returns rise-over-run slope as per Horn 1981
- `&aspect`: Returns aspect as per Horn 1981 in degrees [0,360). Degrees increase in a clockwise fashion. 0 is north, -1 indicates a flat surface.
- `&curvature`: Returns the difference of profile and planform curvatures (TODO: Clarify, this is from ArcGIS and was poorly described)
- `&profile_curvature`: Returns the profile curvature as per Zevenbergen and Thorne 1987. 0 indicates a flat surface.
- `&planform_curvature`: Returns the planform curvature as per Zevenbergen and Thorne 1987. 0 indicates a flat surface.

Public Members

double **a**

double **b**

double **c**

double **d**

double **e**

double **f**

double **g**

double **h**

double **i**

class Timer

#include <timer.hpp> Used to time how intervals in code.

Such as how long it takes a given function to run, or how long I/O has taken.

Public Functions

Timer()

Creates a *Timer* which is not running and has no accumulated time.

void **start**()

Start the timers. Throws an exception if timer was already running.

double **stop**()

Stop the timer. Throws an exception if timer was already stopped. Calling this adds to the timer's accumulated time.

Return The accumulated time in seconds.

double **accumulated**()

Returns the timer's accumulated time. Throws an exception if the timer is running.

Return The timer's accumulated time, in seconds.

double **lap**()
Returns the time between when the timer was started and the current moment. Throws an exception if the timer is not running.

Return Time since the timer was started and current moment, in seconds.

void **reset**()
Stops the timer and resets its accumulated time. No exceptions are thrown ever.

Private Types

typedef *std*::chrono::high_resolution_clock **clock**

typedef *std*::chrono::duration<double, *std*::ratio<1>> **second**

Private Functions

double **timediff**(**const** *std*::chrono::time_point<*clock*> &*start*, **const** *std*::chrono::time_point<*clock*> &*end*)
Number of (fractional) seconds between two time objects.

Private Members

std::chrono::time_point<*clock*> **start_time**
Last time the timer was started.

double **accumulated_time** = 0
Accumulated running time since creation.

bool **running** = false
True when the timer is running.

class WrappedArray2D
Inherits from *richdem::Array2D< T >*

Public Functions

template<>
void **lazySetAll**()

Public Members

template<>
bool **null_tile** = false

template<>
bool **loaded** = false

template<>
bool **created** = true

template<>
bool **do_set_all** = false

template<>
int **create_with_width** = -1

template<>
int **create_with_height** = -1

template<>
T **set_all_val** = 0

namespace richdem

Typedefs

typedef uint8_t **d8_flowdir_t**

typedef int8_t **flowdir_t**

using richdem::GridCellZ_pq = typedef std::priority_queue<GridCellZ<elev_t>, std::vect
A priority queue of *GridCellZ*, sorted by ascending height.

typedef *std*::string **RandomEngineState**

typedef *std*::mt19937 **our_random_engine**

typedef char **label_t**

typedef *std*::deque<grid_cell> **flat_type**

Enums

enum Topology
Values:

D8

D4

enum LogFlag
Values:

ALG_NAME

CITATION

CONFIG

DEBUG

ERROR

MEM_USE

MISC

PROGRESS

TIME_USE

WARN

enum LindsayMode
Values:

COMPLETE_BREACHING

SELECTIVE_BREACHING

CONSTRAINED_BREACHING

enum LindsayCellType
Values:

UNVISITED

VISITED

EDGE

enum PerimType
Values:

CELL_COUNT
Counts # of cells bordering DEM edges or NoData cells.

SQUARE_EDGE
Adds all cell edges bordering DEM edges or NoData cells.

Functions

std::map<*std*::string, *std*::string> **ProcessMetadata**(char **_metadata_)

std::string **TopologyName**(*Topology* _topo_)

template <Topology _topo_>
void **TopologicalResolver**(**const** int *&_dx_, **const** int *&_dy_, **const** double *&_dr_, **const** int *&_dinverse_, int &_neighbours_)

static *std*::string **trimStr**(*std*::string **const** &_str_)
Eliminate spaces from the beginning and end of str.

static *std*::string **GetBaseName**(*std*::string _filename_)
Get only the filename without its extension. That is, convert "path/to/file.ext" to "file"

bool **fp_le**(**const** double &_a_, **const** double &_b_)
Is a<=b?

bool **fp_ge**(**const** double &_a_, **const** double &_b_)
Is a>=b?

bool **fp_eq**(**const** double &_a_, **const** double &_b_)
Does a==b?

void **ProcessMemUsage**(long &_vmpeak_, long &_vmhwm_)
Return memory statistics of the process.

This code is drawn from "http://stackoverflow.com/a/671389/752843"

Parameters

- `vmpeak`: Peak virtual memory size (kB)
- `vmhwm`: Peak resident set size (kB)

our_random_engine &**rand_engine**()

void **seed_rand**(unsigned long _seed_)

int **uniform_rand_int** (int *from*, int *thru*)

double **uniform_rand_real** (double *from*, double *thru*)

double **normal_rand** (double *mean*, double *stddev*)

template <**class** T>
T **uniform_bits** ()

RandomEngineState **SaveRandomState** ()

void **SetRandomState** (**const** *RandomEngineState* &*res*)

std::string **rdHash** ()

std::string **rdCompileTime** ()

std::string **PrintRichdemHeader** (int *argc*, char ****argv*)
Takes the program's command line arguments and prints to stdout a header with a variety of useful information for identifying the particulars of what was run.

template <Topology *topo*, **class** elev_t>
bool **HasDepressions** (**const** *Array2D*<elev_t> &*elevations*)
Determine if a DEM has depressions.

Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are added to the priority queue. If the neighbours are lower than the cell which is adding them, then they are part of a depression and the question is answered.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Return True if the DEM contains depressions; otherwise, false.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations

template <Topology *topo*, **class** elev_t>
void **PriorityFlood_Original** (*Array2D*<elev_t> &*elevations*)
Fills all pits and removes all digital dams from a DEM.

Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are added to the priority queue. If the neighbours are lower than the cell which is adding them, then they are raised to match its elevation; this fills depressions.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.

2. **elevations** contains no landscape depressions or digital dams.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations

template <Topology *topo*, **class** elev_t>
void **PriorityFlood_Barnes2014** (*Array2D*<elev_t> &*elevations*)

Fills all pits and removes all digital dams from a DEM, but faster.

Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are added to the priority queue if they are higher. If they are lower, they are raised to the elevation of the cell adding them, thereby filling in pits. The neighbors are then added to a "pit" queue which is used to flood pits. Cells which are higher than a pit being filled are added to the priority queue. In this way, pits are filled without incurring the expense of the priority queue.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **elevations** contains no landscape depressions or digital dams.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations

template <Topology *topo*, **class** elev_t>
void **PriorityFloodEpsilon_Barnes2014** (*Array2D*<elev_t> &*elevations*)

Modifies floating-point cell elevations to guarantee drainage.

This version of Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are added to the priority queue if they are higher. If they are lower, then their elevation is increased by a small amount to ensure that they have a drainage path and they are added to a "pit" queue which is used to flood pits. Cells which are higher than a pit being filled are added to the priority queue. In this way, pits are filled without incurring the expense of the priority queue.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **elevations** has no landscape depressions, digital dams, or flats.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations

template <Topology *topo*>
void **PriorityFloodEpsilon_Barnes2014**(*Array2D*<uint8_t> &*elevations*)
Priority-Flood+Epsilon is not available for integer data types.

template <Topology *topo*>
void **PriorityFloodEpsilon_Barnes2014**(*Array2D*<uint16_t> &*elevations*)
Priority-Flood+Epsilon is not available for integer data types.

template <Topology *topo*>
void **PriorityFloodEpsilon_Barnes2014**(*Array2D*<int16_t> &*elevations*)
Priority-Flood+Epsilon is not available for integer data types.

template <Topology *topo*>
void **PriorityFloodEpsilon_Barnes2014**(*Array2D*<uint32_t> &*elevations*)
Priority-Flood+Epsilon is not available for integer data types.

template <Topology *topo*>
void **PriorityFloodEpsilon_Barnes2014**(*Array2D*<int32_t> &*elevations*)
Priority-Flood+Epsilon is not available for integer data types.

template <**class** elev_t>
void **PriorityFloodFlowdirs_Barnes2014**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<*d8_flowdir_t*> &*flowdirs*)
Determines D8 flow directions and implicitly fills pits.

This version of Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are given D8 flow directions which point to it. All depressions are implicitly filled and digital dams removed.

Author Richard Barnes (rbarnes@umn.edu)

Based on Metz 2011.

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **flowdirs** contains a D8 flow direction of each cell or a value *NO_FLOW* for those cells which are not part of the DEM.
2. **flowdirs** has no cells which are not part of a continuous flow path leading to the edge of the DEM.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations
- `&flowdirs`: A grid of D8 flow directions

template <Topology *topo*, **class** elev_t>
void **pit_mask**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<uint8_t> &*pit_mask*)
Indicates which cells are in pits.

This version of Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. If a cell is lower than this cell then it is part of a pit and is given a value 1 to indicate this. The result is a grid where every cell which is in a pit is labeled.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **pit_mask** contains a 1 for each cell which is in a pit and a 0 for each cell which is not. The value 3 indicates NoData

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations
- `&pit_mask`: A grid of indicating which cells are in pits

template <Topology *topo*, **class** elev_t>
void **PriorityFloodWatersheds_Barnes2014** (*Array2D*<elev_t> &*elevations*, *Array2D*<int32_t> &*labels*, bool *alter_elevations*)

Gives a common label to all cells which drain to a common point.

All the edge cells of the DEM are given unique labels. This version of Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are then given its label. All depressions are implicitly filled and digital dams removed. The result is a grid of cells where all cells with a common label drain to a common point.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains no depressions or digital dams, if alter_elevations** was set.
2. **labels** contains a label for each cell indicating its membership in a given watershed. Cells bearing common labels drain to common points.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `elevations`: A grid of cell elevations
- `labels`: A grid to hold the watershed labels
- `alter_elevations`: If true, then **elevations** is altered as though *PriorityFlood_Barnes2014()* had been applied. The result is that all cells drain to the edges of the DEM. Otherwise, **elevations** is not altered.

template <Topology *topo*, **class** elev_t>
void **PriorityFlood_Barnes2014_max_dep** (*Array2D*<elev_t> &*elevations*, uint64_t *max_dep_size*)

Fill depressions, but only if they're small.

Priority-Flood starts on the edges of the DEM and then works its way inwards using a priority queue to determine the lowest cell which has a path to the edge. The neighbours of this cell are added to the priority queue if they are higher. If they are lower, they are raised to the elevation of the cell adding them, thereby filling in pits. The neighbors are then added to a "pit" queue which is used to flood pits. Cells which are higher than a pit being filled are added to the priority queue. In this way, pits are filled without incurring the expense of the priority queue.

Author Richard Barnes (rbarnes@umn.edu)

When a depression is encountered this command measures its size before filling it. Only small depressions are filled.

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **elevations** all landscape depressions <=max_dep_size are filled.

Correctness: The correctness of this command is determined by inspection. (TODO)

Parameters

- `&elevations`: A grid of cell elevations
- `max_dep_size`: Depression must have <=max_dep_size cells to be filled

template <Topology *topo*, **class** T>
void **FillDepressions**(*Array2D*<T> &*dem*)

template <Topology *topo*, **class** T>
void **FillDepressionsEpsilon**(*Array2D*<T> &*dem*)

template <Topology *topo*, **class** T>
void **BreachDepressions**(*Array2D*<T> &*dem*)

template <Topology *topo*, **class** elev_t>
void **CompleteBreaching_Lindsay2016**(*Array2D*<elev_t> &*dem*)

Breach depressions.

Depression breaching drills a path from a depression's pit cell (its lowest point) along the least-cost (Priority-Flood) path to the nearest cell outside the depression to have the same or lower elevation.

Author John Lindsay, implementation by Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Return The breached DEM.

Correctness: The correctness of this command is determined by inspection and simple unit tests.

Parameters

- `&elevations`: A grid of cell elevations

template <**class** elev_t>

void **Lindsay2016**(*Array2D*<elev_t> &*dem*, int *mode*, bool *eps_gradients*, bool *fill_depressions*, uint32_t *maxpathlen*, elev_t *maxdepth*)

Breach and fill depressions (EXPERIMENTAL)

Depression breaching drills a path from a depression's pit cell (its lowest point) along the shortest path to the nearest cell outside the depression to have the same or lower elevation.

Author John Lindsay, implementation by Richard Barnes (rbarnes@umn.edu)

Several modes are available including:

Complete Breaching: All depressions are entirely breached. Selective Breaching: Depressions are breached provided the breaching path is not too long nor too deep. That which cannot be breached is filled. Breaching only takes place if the path meets the criteria. Constrained Breaching: A braching path is drilled as long and as deep as permitted, but no more.

NOTE: It is possible these three modes should be split into different functions.

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Return The breached DEM.

Correctness: The correctness of this command is determined by inspection and simple unit tests.

Parameters

- `&elevations`: A grid of cell elevations
- `mode`: A `LindsayMode` value of `COMPLETE_BREACHING`, `SELECTIVE_BREACHING`, or `CONSTRAINED_BREACHING`.
- `eps_gradients`: If True, then epsilon gradients are applied to breaching paths and depressions to ensure drainage.
- `fill_depresssions`: If True, then depressions are filled.
- `maxpathlen`: Maximum length of a breaching path
- `maxdepth`: Maximum depth of a breaching path

template <Topology *topo*>
void **Lindsay2016**(*Array2D*<uint8_t> &*dem*, int *mode*, bool *eps_gradients*, bool *fill_depressions*, uint32_t *maxpathlen*, uint8_t *maxdepth*)

template <Topology *topo*>
void **Lindsay2016**(*Array2D*<int16_t> &*dem*, int *mode*, bool *eps_gradients*, bool *fill_depressions*, uint32_t *maxpathlen*, int16_t *maxdepth*)

template <Topology *topo*>
void **Lindsay2016**(*Array2D*<uint16_t> &*dem*, int *mode*, bool *eps_gradients*, bool *fill_depressions*, uint32_t *maxpathlen*, uint16_t *maxdepth*)

template <**class** T>
static void **InitPriorityQue**(*Array2D*<T> &*dem*, *Array2D*<bool> &*flag*, GridCellZ_pq<T> &*priorityQueue*)

template <**class** T>
static void **ProcessTraceQue**(*Array2D*<T> &*dem*, *Array2D*<bool> &*flag*, *std*::queue<*GridCellZ*<T>> &*traceQueue*, GridCellZ_pq<T> &*priorityQueue*)

template <**class** T>

static void **ProcessPit**(*Array2D*<T> &*dem*, *Array2D*<bool> &*flag*, *std*::queue<*GridCellZ*<T>> &*depressionQue*, *std*::queue<*GridCellZ*<T>> &*traceQueue*, GridCellZ_pq<T> &*priorityQueue*)

template <**class** T>
void **PriorityFlood_Wei2018**(*Array2D*<T> &*dem*)

template <**class** elev_t>
void **ProcessTraceQue_onepass**(*Array2D*<elev_t> &*dem*, *Array2D*<*label_t*> &*labels*, *std*::queue<int> &*traceQueue*, *std*::priority_queue<*std*::pair<elev_t, int>, *std*::vector<*std*::pair<elev_t, int>>, *std*::greater<*std*::pair<elev_t, int>>> &*priorityQueue*)

template <**class** elev_t>
void **ProcessPit_onepass**(elev_t *c_elev*, *Array2D*<elev_t> &*dem*, *Array2D*<*label_t*> &*labels*, *std*::queue<int> &*depressionQue*, *std*::queue<int> &*traceQueue*, *std*::priority_queue<*std*::pair<elev_t, int>, *std*::vector<*std*::pair<elev_t, int>>, *std*::greater<*std*::pair<elev_t, int>>> &*priorityQueue*)

template <**class** elev_t>
void **PriorityFlood_Zhou2016**(*Array2D*<elev_t> &*dem*)

Fills all pits and removes all digital dams from a DEM, quickly.

Works similarly to the Priority-Flood described by Barnes et al. (2014), but reduces the number of items which must pass through the priority queue, thus achieving greater efficiencies.

Author G. Zhou, Z. Sun, S. Fu, Richard Barnes (this implementation)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM. Note that the *NoData* value is assumed to be a negative number less than any actual data value.

Post

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **elevations** contains no landscape depressions or digital dams.

Parameters

- `&dem`: A grid of cell elevations

static void **BuildAwayGradient**(**const** *Array2D*<int8_t> &*flats*, *Array2D*<int32_t> &*flat_mask*, *std*::deque<*GridCell*> &*high_edges*, *std*::vector<int> &*flat_height*, **const** *Array2D*<int32_t> &*labels*)

Build a gradient away from the high edges of the flats.

The queue of high-edge cells developed in FindFlatEdges() is copied into the procedure. A breadth-first expansion labels cells by their distance away from terrain of higher elevation. The maximal distance encountered is noted.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. Every cell in **labels** is marked either 0, indicating that the cell is not part of a flat, or a number greater than zero which identifies the flat to which the cell belongs.
2. Any cell without a local gradient is marked IS_A_FLAT in **flats**.
3. Every cell in **flat_mask** is initialized to 0.

4. **edges** contains, in no particular order, all the high edge cells of the DEM (those flat cells adjacent to higher terrain) which are part of drainable flats.

Post

1. **flat_height** will have an entry for each label value of **labels** indicating the maximal number of increments to be applied to the flat identified by that label.
2. **flat_mask** will contain the number of increments to be applied to each cell to form a gradient away from higher terrain; cells not in a flat will have a value of 0.

Parameters

- `&flats`: 2D array indicating flat membership from FindFlats()
- `&flat_mask`: A 2D array for storing flat_mask
- `&edges`: The high-edge FIFO queue from FindFlatEdges()
- `&flat_height`: Vector with length equal to max number of labels
- `&labels`: 2D array storing labels developed in LabelFlat()

static void **BuildTowardsCombinedGradient** (*Array2D*<int8_t> &*flats*, *Array2D*<int32_t> &*flat_mask*, *std*::deque<*GridCell*> &*low_edges*, *std*::vector<int> &*flat_height*, **const** *Array2D*<int32_t> &*labels*)

Builds gradient away from the low edges of flats, combines gradients.

The queue of low-edge cells developed in FindFlatEdges() is copied into the procedure. A breadth-first expansion labels cells by their distance away from terrain of lower elevation. This is combined with the gradient from BuildAwayGradient() to give the final increments of each cell in forming the flat mask.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. Every cell in **labels** is marked either 0, indicating that the cell is not part of a flat, or a number greater than zero which identifies the flat to which the cell belongs.
2. Any cell without a local gradient is marked IS_A_FLAT in **flats**.
3. Every cell in **flat_mask** has either a value of 0, indicating that the cell is not part of a flat, or a value greater than zero indicating the number of increments which must be added to it to form a gradient away from higher terrain.
4. **flat_height** has an entry for each label value of **labels** indicating the maximal number of increments to be applied to the flat identified by that label in order to form the gradient away from higher terrain.
5. **edges** contains, in no particular order, all the low edge cells of the DEM (those flat cells adjacent to lower terrain).

Post

1. **flat_mask** will contain the number of increments to be applied to each cell to form a superposition of the gradient away from higher terrain with the gradient towards lower terrain; cells not in a flat have a value of 0.

Parameters

- `&flats`: 2D array indicating flat membership from FindFlats()
- `&flat_mask`: A 2D array for storing flat_mask

- `&edges`: The low-edge FIFO queue from FindFlatEdges()
- `&flat_height`: Vector with length equal to max number of labels
- `&labels`: 2D array storing labels developed in LabelFlat()

template <**class** T>
static void **LabelFlat**(**const** int *x0*, **const** int *y0*, **const** int *label*, *Array2D*<int32_t> &*labels*, **const** *Array2D*<T> &*elevations*)

Labels all the cells of a flat with a common label.

Performs a flood fill operation which labels all the cells of a flat with a common label. Each flat will have a unique label

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **labels** has the same dimensions as **elevations**.
3. **(x0,y0)** belongs to the flat which is to be labeled.
4. **label** is a unique label which has not been previously applied to a flat.
5. **labels** is initialized to zero prior to the first call to this function.

Post

1. **(x0,y0)** and every cell reachable from it by passing over only cells of the same elevation as it (all the cells in the flat to which c belongs) will be marked as **label** in **labels**.

Parameters

- `x0`: x-coordinate of flood fill seed
- `y0`: y-coordinate of flood-fill seed
- `label`: Label to apply to the cells
- `&labels`: 2D array which will contain the labels
- `&elevations`: 2D array of cell elevations

template <**class** T>
static void **FindFlatEdges**(*std*::deque<*GridCell*> &*low_edges*, *std*::deque<*GridCell*> &*high_edges*, **const** *Array2D*<int8_t> &*flats*, **const** *Array2D*<T> &*elevations*)

Identifies cells adjacent to higher and lower terrain.

Cells adjacent to lower and higher terrain are identified and added to the appropriate queue

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. Any cell without a local gradient is marked IS_A_FLAT in **flats**.

Post

1. **high_edges** will contain, in no particular order, all the high edge cells of the DEM: those flat cells adjacent to higher terrain.

2. **low_edges** will contain, in no particular order, all the low edge cells of the DEM: those flat cells adjacent to lower terrain.

Parameters

- `&low_edges`: Queue for storing cells adjacent to lower terrain
- `&high_edges`: Queue for storing cells adjacent to higher terrain
- `&flats`: 2D array indicating flat membership from FindFlats()
- `&elevations`: 2D array of cell elevations

template <class T>
void **GetFlatMask**(**const** *Array2D*<T> &*elevations*, *Array2D*<int32_t> &*flat_mask*, *Array2D*<int32_t> &*labels*)

Generates a flat resolution mask in the style of Barnes (2014)

This algorithm is a modification of that presented by Barnes (2014). It has been rejiggered so that a knowledge of flow directions is not necessary to run it.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or the *NoData* value for cells not part of the DEM.

Post

1. **flat_mask** will have a value greater than or equal to zero for every cell, indicating its number of increments. These can be used be used in conjunction with **labels** to determine flow directions without altering the DEM, or to alter the DEM in subtle ways to direct flow.
2. **labels** will have values greater than or equal to 1 for every cell which is in a flat. Each flat's cells will bear a label unique to that flat. A value of 0 means the cell is not part of a flat.

Parameters

- `&elevations`: 2D array of cell elevations
- `&flat_mask`: 2D array which will hold incremental elevation mask
- `&labels`: 2D array indicating flat membership

template <class U>
void **ResolveFlatsEpsilon_Barnes2014**(**const** *Array2D*<int32_t> &*flat_mask*, **const** *Array2D*<int32_t> &*labels*, *Array2D*<U> &*elevations*)

Alters the elevations of the DEM so that all flats drain.

This alters elevations within the DEM so that flats which have been resolved using *GetFlatMask()* will drain.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **flat_mask** contains the number of increments to be applied to each cell to form a gradient which will drain the flat it is a part of.
2. If a cell is part of a flat, it has a value greater than zero in labels** indicating which flat it is a member of; otherwise, it has a value of 0.

Post

1. Every cell whose part of a flat which could be drained will have its elevation altered in such a way as to guarantee that its flat does drain.

Parameters

- `&flat_mask`: A mask from *GetFlatMask()*
- `&labels`: A grouping from *GetFlatMask()*
- `&elevations`: 2D array of elevations

template <class U>
void **ResolveFlatsFlowdirs_Barnes2014**(**const** *Array2D*<int32_t> &*flat_mask*, **const** *Array2D*<int32_t> &*labels*, *Array2D*<U> &*flowdirs*)

Calculates flow directions in flats.

This determines flow directions within flats which have been resolved using *GetFlatMask()*.

Author Richard Barnes (rbarnes@umn.edu)

Uses the helper function D8MaskedFlowdir()

Pre

1. **flat_mask** contains the number of increments to be applied to each cell to form a gradient which will drain the flat it is a part of.
2. Any cell without a local gradient has a value of NO_FLOW_GEN in flowdirs**; all other cells have defined flow directions.
3. If a cell is part of a flat, it has a value greater than zero in labels** indicating which flat it is a member of; otherwise, it has a value of 0.

Post

1. Every cell whose flow direction could be resolved by this algorithm (all drainable flats) will have a defined flow direction in flowdirs**. Any cells which could not be resolved (non-drainable flats) will still be marked NO_FLOW_GEN.

Parameters

- `&flat_mask`: A mask from *GetFlatMask()*
- `&labels`: The labels output from *GetFlatMask()*
- `&flowdirs`: Returns flat-resolved flow directions

template <class T>
void **FindFlats**(**const** *Array2D*<T> &*elevations*, *Array2D*<int8_t> &*flats*)

Finds flats: cells with no local gradient.

TODO

Author Richard Barnes (rbarnes@umn.edu)

Post

1. flats contains the value 1 for each cell in a flat (all cells without a local gradient), 0 for each cell not in a flat (all cells with a local gradient), and -1 for each no data cell.

Parameters

- `&elevations`: An elevation field
- `&flats`: An array of flat indicators (post-conditions)

static int **d8_masked_FlowDir**(**const** *Array2D*<int32_t> &*flat_mask*, **const** *Array2D*<int32_t> &*labels*, **const** int *x*, **const** int *y*)

Helper function to *d8_flow_flats()*

This determines a cell's flow direction, taking into account flat membership. It is a helper function to *d8_flow_flats()*

Author Richard Barnes (rbarnes@umn.edu)

Return The flow direction of the cell

Parameters

- `&flat_mask`: A mask from *resolve_flats_barnes()*
- `&labels`: The labels output from *resolve_flats_barnes()*
- `x`: x coordinate of cell
- `y`: y coordinate of cell

template <**class** U>
void **d8_flow_flats**(**const** *Array2D*<int32_t> &*flat_mask*, **const** *Array2D*<int32_t> &*labels*, *Array2D*<U> &*flowdirs*)

Calculates flow directions in flats.

This determines flow directions within flats which have been resolved using *resolve_flats_barnes()*.

Author Richard Barnes (rbarnes@umn.edu)

Uses the helper function d8_masked_FlowDir()

Pre

1. **flat_mask** contains the number of increments to be applied to each cell to form a gradient which will drain the flat it is a part of.
2. Any cell without a local gradient has a value of NO_FLOW in flowdirs**; all other cells have defined flow directions.
3. If a cell is part of a flat, it has a value greater than zero in labels** indicating which flat it is a member of; otherwise, it has a value of 0.

Post

1. Every cell whose flow direction could be resolved by this algorithm (all drainable flats) will have a defined flow direction in flowdirs**. Any cells which could not be resolved (non-drainable flats) will still be marked NO_FLOW.

Parameters

- `&flat_mask`: A mask from *resolve_flats_barnes()*
- `&labels`: The labels output from *resolve_flats_barnes()*
- `&flowdirs`: Returns flat-resolved flow directions

template <**class** U>
static void **BuildAwayGradient**(**const** *Array2D*<U> &*flowdirs*, *Array2D*<int32_t> &*flat_mask*, *std*::deque<*GridCell*> *edges*, *std*::vector<int> &*flat_height*, **const** *Array2D*<int32_t> &*labels*)

Build a gradient away from the high edges of the flats.

The queue of high-edge cells developed in find_flat_edges() is copied into the procedure. A breadth-first expansion labels cells by their distance away from terrain of higher elevation. The maximal distance encountered is noted.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. Every cell in **labels** is marked either 0, indicating that the cell is not part of a flat, or a number greater than zero which identifies the flat to which the cell belongs.
2. Any cell without a local gradient is marked NO_FLOW in **flowdirs**.
3. Every cell in **flat_mask** is initialized to 0.
4. **edges** contains, in no particular order, all the high edge cells of the DEM (those flat cells adjacent to higher terrain) which are part of drainable flats.

Post

1. **flat_height** will have an entry for each label value of **labels** indicating the maximal number of increments to be applied to the flat identified by that label.
2. **flat_mask** will contain the number of increments to be applied to each cell to form a gradient away from higher terrain; cells not in a flat will have a value of 0.

Parameters

- `&flowdirs`: A 2D array indicating each cell's flow direction
- `&flat_mask`: A 2D array for storing flat_mask
- `&edges`: The high-edge FIFO queue from find_flat_edges()
- `&flat_height`: Vector with length equal to max number of labels
- `&labels`: 2D array storing labels developed in label_this()

template <**class** U>
static void **BuildTowardsCombinedGradient**(**const** *Array2D*<U> &*flowdirs*, *Array2D*<int32_t> &*flat_mask*, *std*::deque<*GridCell*> *edges*, *std*::vector<int> &*flat_height*, **const** *Array2D*<int32_t> &*labels*)

Builds gradient away from the low edges of flats, combines gradients.

The queue of low-edge cells developed in find_flat_edges() is copied into the procedure. A breadth-first expansion labels cells by their distance away from terrain of lower elevation. This is combined with the gradient from BuildAwayGradient() to give the final increments of each cell in forming the flat mask.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. Every cell in **labels** is marked either 0, indicating that the cell is not part of a flat, or a number greater than zero which identifies the flat to which the cell belongs.
2. Any cell without a local gradient is marked NO_FLOW in **flowdirs**.
3. Every cell in **flat_mask** has either a value of 0, indicating that the cell is not part of a flat, or a value greater than zero indicating the number of increments which must be added to it to form a gradient away from higher terrain.

4. **flat_height** has an entry for each label value of **labels** indicating the maximal number of increments to be applied to the flat identified by that label in order to form the gradient away from higher terrain.
5. **edges** contains, in no particular order, all the low edge cells of the DEM (those flat cells adjacent to lower terrain).

Post

1. **flat_mask** will contain the number of increments to be applied to each cell to form a superposition of the gradient away from higher terrain with the gradient towards lower terrain; cells not in a flat have a value of 0.

Parameters

- `&flowdirs`: A 2D array indicating each cell's flow direction
- `&flat_mask`: A 2D array for storing flat_mask
- `&edges`: The low-edge FIFO queue from find_flat_edges()
- `&flat_height`: Vector with length equal to max number of labels
- `&labels`: 2D array storing labels developed in label_this()

template <class T>
static void **label_this**(int *x0*, int *y0*, **const** int *label*, *Array2D*<int32_t> &*labels*, **const** *Array2D*<T> &*elevations*)

Labels all the cells of a flat with a common label.

Performs a flood fill operation which labels all the cells of a flat with a common label. Each flat will have a unique label

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. **labels** has the same dimensions as **elevations**.
3. **(x0,y0)** belongs to the flat which is to be labeled.
4. **label** is a unique label which has not been previously applied to a flat.
5. **labels** is initialized to zero prior to the first call to this function.

Post

1. **(x0,y0)** and every cell reachable from it by passing over only cells of the same elevation as it (all the cells in the flat to which c belongs) will be marked as **label** in **labels**.

Parameters

- `x0`: x-coordinate of flood fill seed
- `y0`: y-coordinate of flood-fill seed
- `label`: Label to apply to the cells
- `&labels`: 2D array which will contain the labels
- `&elevations`: 2D array of cell elevations

template <class T, **class** U>

static void **find_flat_edges**(*std*::deque<*GridCell*> &*low_edges*, *std*::deque<*GridCell*> &*high_edges*, **const** *Array2D*<U> &*flowdirs*, **const** *Array2D*<T> &*elevations*)

Identifies cells adjacent to higher and lower terrain.

Cells adjacent to lower and higher terrain are identified and added to the appropriate queue

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or a value *NoData* for cells not part of the DEM.
2. Any cell without a local gradient is marked NO_FLOW in **flowdirs**.

Post

1. **high_edges** will contain, in no particular order, all the high edge cells of the DEM: those flat cells adjacent to higher terrain.
2. **low_edges** will contain, in no particular order, all the low edge cells of the DEM: those flat cells adjacent to lower terrain.

Parameters

- `&low_edges`: Queue for storing cells adjacent to lower terrain
- `&high_edges`: Queue for storing cells adjacent to higher terrain
- `&flowdirs`: 2D array indicating flow direction for each cell
- `&elevations`: 2D array of cell elevations

template <**class** T, **class** U>
void **resolve_flats_barnes**(**const** *Array2D*<T> &*elevations*, **const** *Array2D*<U> &*flowdirs*, *Array2D*<int32_t> &*flat_mask*, *Array2D*<int32_t> &*labels*)

Performs the flat resolution by Barnes, Lehman, and Mulla.

TODO

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **elevations** contains the elevations of every cell or the *NoData* value for cells not part of the DEM.
2. Any cell without a local gradient is marked NO_FLOW in **flowdirs**.

Post

1. **flat_mask** will have a value greater than or equal to zero for every cell, indicating its number of increments. These can be used be used in conjunction with **labels** to determine flow directions without altering the DEM, or to alter the DEM in subtle ways to direct flow.
2. **labels** will have values greater than or equal to 1 for every cell which is in a flat. Each flat's cells will bear a label unique to that flat.

Parameters

- `&elevations`: 2D array of cell elevations
- `&flowdirs`: 2D array indicating flow direction of each cell
- `&flat_mask`: 2D array which will hold incremental elevation mask
- `&labels`: 2D array indicating flat membership

template <**class** U>
void **d8_flats_alter_dem**(**const** *Array2D*<int32_t> &*flat_mask*, **const** *Array2D*<int32_t> &*labels*, *Array2D*<U> &*elevations*)

Alters the elevations of the DEM so that all flats drain.

This alters elevations within the DEM so that flats which have been resolved using *resolve_flats_barnes()* will drain.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. **flat_mask** contains the number of increments to be applied to each cell to form a gradient which will drain the flat it is a part of.
2. If a cell is part of a flat, it has a value greater than zero in labels** indicating which flat it is a member of; otherwise, it has a value of 0.

Post

1. Every cell whose part of a flat which could be drained will have its elevation altered in such a way as to guarantee that its flat does drain.

Parameters

- `&flat_mask`: A mask from *resolve_flats_barnes()*
- `&labels`: A grouping from *resolve_flats_barnes()*
- `&elevations`: 2D array of elevations

template <**class** T, **class** U>
void **barnes_flat_resolution_d8**(*Array2D*<T> &*elevations*, *Array2D*<U> &*flowdirs*, bool *alter*)

static float **dinf_masked_FlowDir**(**const** *Array2D*<int32_t> &*flat_resolution_mask*, **const** *Array2D*<int32_t> &*groups*, **const** int *x*, **const** int *y*)

void **dinf_flow_flats**(**const** *Array2D*<int32_t> &*flat_resolution_mask*, **const** *Array2D*<int32_t> &*groups*, *Array2D*<float> &*flowdirs*)

template <**class** T>
void **resolve_flats_barnes_dinf**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*flowdirs*)

template <**class** T>
void **ResolveFlatsEpsilon**(*Array2D*<T> &*elevations*)

Alters the elevations of the DEM so that all flats drain.

This alters elevations within the DEM so that all cells will have a drainage path.

Author Richard Barnes (rbarnes@umn.edu)

Post

1. Every cell which is part of a flat that can be drained will have its elevation altered in such a way as to guarantee that it does drain.

Parameters

- `&elevations`: An elevations field

void **FindFlats**(**const** *Array2D*<uint8_t> &*flowdirs*, *flat_type* &*flats*)

template <**class** T>

void **GradientTowardsLower**(**const** *Array2D*<T> &*elevations*, **const** *Array2D*<uint8_t> &*flowdirs*, *flat_type* &*flats*, *Array2D*<int32_t> &*inc1*)

template <**class** T>
void **GradientAwayFromHigher**(**const** *Array2D*<T> &*elevations*, **const** *Array2D*<uint8_t> &*flowdirs*, *flat_type* &*flats*, *Array2D*<int32_t> &*inc2*)

template <**class** T>
void **CombineGradients**(*Array2D*<T> &*elevations*, **const** *Array2D*<int32_t> &*inc1*, **const** *Array2D*<int32_t> &*inc2*, float *epsilon*)

template <**class** T>
void **GarbrechtAlg**(*Array2D*<T> &*elevations*, *Array2D*<uint8_t> &*flowdirs*)

template <**class** T>
static int **d8_FlowDir**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*)

Calculates the D8 flow direction of a cell.

This calculates the D8 flow direction of a cell using the D8 neighbour system, as defined in utility.h. Cells on the edge of the grid flow off the nearest edge.

Author Richard Barnes (rbarnes@umn.edu)

Helper function for *d8_flow_directions()*.

Return The D8 flow direction of the cell

Parameters

- `&elevations`: A DEM
- `x`: x coordinate of cell
- `y`: y coordinate of cell

template <**class** T, **class** U>
void **d8_flow_directions**(**const** *Array2D*<T> &*elevations*, *Array2D*<U> &*flowdirs*)

Calculates the D8 flow directions of a DEM.

This calculates the D8 flow directions of a DEM. Its argument 'flowdirs' will return a grid with flow directions using the D8 neighbour system, as defined in utility.h. The choice of data type for array2d must be able to hold exact values for all neighbour identifiers (usually [-1,7]).

Author Richard Barnes (rbarnes@umn.edu)

Uses d8_FlowDir() as a helper function.

Parameters

- `&elevations`: A DEM
- `&flowdirs`: Returns the flow direction of each cell

template <**class** T>
static float **dinf_FlowDir**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*)

Determine the D-infinite flow direction of a cell.

This function determines the D-infinite flow direction of a cell, as described by Tarboton (1997) and Barnes (2013, TODO). TODO

Author Implementation by Richard Barnes (rbarnes@umn.edu)

Return A floating-point value between [0,2*Pi) indicating flow direction

Parameters

- `elevations`: A 2D grid of elevation data
- `x`: x-coordinate of cell to determine flow direction for
- `y`: y-coordinate of cell to determine flow direction for

template <**class** T>
void **dinf_flow_directions**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*flowdirs*)

Determine the D-infinite flow direction of every cell in a grid.

This function runs dinf_FlowDir() on every cell in a grid which has a data value.

Author Richard Barnes (rbarnes@umn.edu)

Parameters

- `&elevations`: A 2D grid of elevation data
- `&flowdirs`: A 2D grid which will contain the flow directions

template <Topology *topo*, **class** elev_t>
void **FM_FairfieldLeymarie**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** elev_t>
void **FM_Rho8**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** elev_t>
void **FM_Rho4**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** E>
void **FM_Freeman**(**const** *Array2D*<E> &*elevations*, *Array3D*<float> &*props*, **const** double *xparam*)

template <**class** E>
void **FM_Holmgren**(**const** *Array2D*<E> &*elevations*, *Array3D*<float> &*props*, **const** double *xparam*)

template <Topology *topo*, **class** elev_t>
void **FM_OCallaghan**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** elev_t>
void **FM_D8**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** elev_t>
void **FM_D4**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** E>
void **FM_Quinn**(**const** *Array2D*<E> &*elevations*, *Array3D*<float> &*props*)

template <**class** elev_t>
void **FM_Tarboton**(**const** *Array2D*<elev_t> &*elevations*, *Array3D*<float> &*props*)

template <**class** E>
void **FM_Dinfinity**(**const** *Array2D*<E> &*elevations*, *Array3D*<float> &*props*)

template <**class** T>
static T **sgn**(T *val*)

Returns the sign (+1, -1, 0) of a number. Branchless.

Author Richard Barnes (rbarnes@umn.edu)

Return -1 for a negative input, +1 for a positive input, and 0 for a zero input

Parameters

- `val`: Input value

template <**class** T, **class** U>

void **d8_flow_accum**(**const** *Array2D*<T> &*flowdirs*, *Array2D*<U> &*area*)
Calculates the D8 flow accumulation, given the D8 flow directions.

This calculates the D8 flow accumulation of a grid of D8 flow directions by calculating each cell's dependency on its neighbours and then using a priority-queue to process cells in a top-of-the-watershed-down fashion

Author Richard Barnes (rbarnes@umn.edu)

Parameters

- `&flowdirs`: A D8 flowdir grid from *d8_flow_directions()*
- `&area`: Returns the up-slope area of each cell

template <**class** T, **class** U>
void **d8_upslope_cells**(int *x0*, int *y0*, int *x1*, int *y1*, **const** *Array2D*<T> &*flowdirs*, *Array2D*<U> &*upslope_cells*)
Calculates which cells ultimately D8-flow through a given cell.

Given the coordinates x0,y0 of a cell and x1,y1 of another, possibly distinct, cell this draws a line between the two using the Bresenham Line-Drawing Algorithm and returns a grid showing all the cells whose flow ultimately passes through the indicated cells.

Author Richard Barnes (rbarnes@umn.edu)

The grid has the values:

1=Upslope cell 2=Member of initializing line All other cells have a noData() value

Parameters

- `x0`: x-coordinate of start of line
- `y0`: y-coordinate of start of line
- `x1`: x-coordinate of end of line
- `y1`: y-coordinate of end of line
- `&flowdirs`: A D8 flowdir grid from *d8_flow_directions()*
- `&upslope_cells`: A grid of 1/2/NoData, as in the description

static void **where_do_i_flow**(float *flowdir*, int &*nhigh*, int &*nlow*)

static void **area_proportion**(float *flowdir*, int *nhigh*, int *nlow*, float &*phigh*, float &*plow*)

template <**class** T, **class** U>
void **dinf_upslope_area**(**const** *Array2D*<T> &*flowdirs*, *Array2D*<U> &*area*)
Calculate each cell's D-infinity flow accumulation value.

TODO

Author Tarboton (1997), Richard Barnes (rbarnes@umn.edu)

Parameters

- `flowdirs`: A grid of D-infinite flow directions
- `&area`: A grid of flow accumulation values

template <**class** elev_t, **class** accum_t>
void **FA_Tarboton**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_Dinfinity**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_Holmgren**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*, double *xparam*)

template <**class** elev_t, **class** accum_t>
void **FA_Quinn**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_Freeman**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*, double *xparam*)

template <**class** elev_t, **class** accum_t>
void **FA_FairfieldLeymarieD8**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_FairfieldLeymarieD4**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_Rho8**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_Rho4**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_OCallaghanD8**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_OCallaghanD4**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_D8**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** elev_t, **class** accum_t>
void **FA_D4**(**const** *Array2D*<elev_t> &*elevations*, *Array2D*<accum_t> &*accum*)

template <**class** A>
void **FlowAccumulation**(**const** *Array3D*<float> &*props*, *Array2D*<A> &*accum*)

Calculate flow accumulation from a flow metric array.

Given a flow metric function `func`, this calculations the flow accumulation.

Author Richard Barnes (rbarnes@umn.edu)

Pre

1. The accumulation matrix must already be initialized to the amount of flow each cell will generate. A good default value is 1, in which case the accumulation matrix will be modified to show how many cells' flow ultimately passes through each cell.

Post

1. `accum` is modified so that each cell indicates how much upstrema flow passes through it (in addition to flow generated within the cell itself).

Parameters

- `func`: The flow metric to use
- `&elevations`: An elevation field
- `&accum`: Accumulation matrix: must be already initialized
- `args`: Arguments passed to the flow metric (e.g. exponent)

template <**class** T, **class** U, **class** V>
void **TA_SPI**(**const** *Array2D*<T> &*flow_accumulation*, **const** *Array2D*<U> &*riserun_slope*, *Array2D*<V> &*result*)

Calculates the SPI terrain attribute.

$(CellSize \cdot FlowAccumulation + 0.001) \cdot (\frac{1}{100} PercentSlope + 0.001)$

Author Richard Barnes (rbarnes@umn.edu)

Pre `flow_accumulation` and `percent_slope` must be the same size

Post `result` takes the properties and dimensions of `flow_accumulation`

Parameters

- `&flow_accumulation`: A flow accumulation grid (*dinf_upslope_area()*)
- `&riserun_slope`: A percent_slope grid (d8_slope())
- `&result`: Altered to return the calculated SPI

template <**class** T, **class** U, **class** V>
void **TA_CTI**(**const** *Array2D*<T> &*flow_accumulation*, **const** *Array2D*<U> &*riserun_slope*, *Array2D*<V> &*result*)

Calculates the CTI terrain attribute.

$\log \frac{CellSize \cdot FlowAccumulation + 0.001}{\frac{1}{100} PercentSlope + 0.001}$

Author Richard Barnes (rbarnes@umn.edu)

Pre `flow_accumulation` and `percent_slope` must be the same size

Post `result` takes the properties and dimensions of `flow_accumulation`

Parameters

- `&flow_accumulation`: A flow accumulation grid (*dinf_upslope_area()*)
- `&riserun_slope`: A percent_slope grid (d8_slope())
- `&result`: Altered to return the calculated SPI

template <**class** T>
static *TA_Setup_Vars* **TerrainSetup**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static *TA_Setup_Curves_Vars* **TerrainCurvatureSetup**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Aspect**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

Calculates aspect in degrees in the manner of Horn 1981.

Return Aspect in degrees in the manner of Horn 1981

template <**class** T>
static double **Terrain_Slope_RiseRun**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

Calculates the rise/run slope along the maximum gradient on a fitted surface over a 3x3 be neighbourhood in the manner of Horn 1981.

Return Rise/run slope

template <**class** T>
static double **Terrain_Curvature**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Planform_Curvature**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Profile_Curvature**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Slope_Percent**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Slope_Radian**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** T>
static double **Terrain_Slope_Degree**(**const** *Array2D*<T> &*elevations*, **const** int *x*, **const** int *y*, **const** float *zscale*)

template <**class** F, **class** T>
static void **TerrainProcessor**(F *func*, **const** *Array2D*<T> &*elevations*, **const** float *zscale*, *Array2D*<float> &*output*)

Calculate a variety of terrain attributes.

This calculates a variety of terrain attributes according to the work of Burrough 1998's "Principles of Geographical Information Systems" (p. 190). This function calls d8_terrain_attrib_helper to calculate the actual attributes. This function may perform some post-processing (such as on slope), but it's purpose is essentially to just scan the grid and pass off the work to d8_terrain_attrib_helper().

Author Richard Barnes (rbarnes@umn.edu), Burrough (1998)

Possible attribute values are

- TATTRIB_CURVATURE
- TATTRIB_PLANFORM_CURVATURE
- TATTRIB_PROFILE_CURVATURE
- TATTRIB_ASPECT
- TATTRIB_SLOPE_RISERUN
- TATTRIB_SLOPE_PERCENT
- TATTRIB_SLOPE_RADIAN
- TATTRIB_SLOPE_DEGREE

Post `output` takes the properties and dimensions of `elevations`

Parameters

- `func`: The attribute function to be used
- `&elevations`: An elevation grid
- `zscale`: Value by which to scale elevation
- `&output`: A grid to hold the results

template <**class** T>

void **TA_slope_riserun**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*slopes*, float *zscale* = 1.0f)

Calculates the slope as rise/run.

Calculates the slope using Horn 1981, as per Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&slopes`: A slope grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <**class** T>
void **TA_slope_percentage**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*slopes*, float *zscale* = 1.0f)

Calculates the slope as percentage.

Calculates the slope using Horn 1981, as per Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&slopes`: A slope grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <**class** T>
void **TA_slope_degrees**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*slopes*, float *zscale* = 1.0f)

Calculates the slope as degrees.

Calculates the slope using Horn 1981, as per Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&slopes`: A slope grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <**class** T>
void **TA_slope_radians**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*slopes*, float *zscale* = 1.0f)

Calculates the slope as radians.

Calculates the slope using Horn 1981, as per Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&slopes`: A slope grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
void **TA_aspect**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*aspects*, float *zscale* = 1.0f)

Calculates the terrain aspect.

Calculates the aspect per Horn 1981, as described by Burrough 1998's "Principles of Geographical Information Systems" (p. 190) The value return is in Degrees.

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&aspects`: An aspect grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
void **TA_curvature**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*curvatures*, float *zscale* = 1.0f)

Calculates the terrain curvature per Zevenbergen and Thorne 1987.

Calculates the curvature per Zevenbergen and Thorne 1987, as described by Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&curvatures`: A curvature grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
void **TA_planform_curvature**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*planform_curvatures*, float *zscale* = 1.0f)

Calculates the terrain planform curvature per Zevenbergen and Thorne 1987.

Calculates the curvature per Zevenbergen and Thorne 1987, as described by Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&planform_curvatures`: A planform curvature grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
void **TA_profile_curvature**(**const** *Array2D*<T> &*elevations*, *Array2D*<float> &*profile_curvatures*, float *zscale* = 1.0f)

Calculates the terrain profile curvature per Zevenbergen and Thorne 1987.

Calculates the curvature per Zevenbergen and Thorne 1987, as described by Burrough 1998's "Principles of Geographical Information Systems" (p. 190)

Author Richard Barnes (rbarnes@umn.edu), Horn (1981)

Parameters

- `&elevations`: An elevation grid
- `&profile_curvatures`: A profile curvature grid
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
double **dem_surface_area**(**const** *Array2D*<T> &*elevations*, **const** double *zscale*)

Calculate the surface of a digital elevation model.

Calculates the surface area of a digital elevation model by connecting the central points of cells with triangles and then calculating the area of the portion of each triangle which falls within the focal cell. The method is described in detail in Jenness (2004) <doi:10.2193/0091-7648(2004)032[0829:CLSAFD]2.0.CO;2>

Author Jenness (2004), Richard Barnes (rbarnes@umn.edu)

Return The surface area of the digital elevation model

Parameters

- `&elevations`: A grid of elevations
- `zscale`: DEM is scaled by this factor prior to calculation

template <class T>
double **Perimeter**(**const** *Array2D*<T> &*arr*, **const** *PerimType perim_type*)

Calculates the perimeter of a digital elevation model.

Author Richard Barnes (rbarnes@umn.edu)

Return The perimeter of the digital elevation model

Parameters

- `&arr`:
- `perim_type`: A `PerimType` value indicating how to calculate the perimeter.

template <Topology *topo*, **class** T, **class** U>
void **BucketFill**(**const** *Array2D*<T> &*check_raster*, *Array2D*<U> &*set_raster*, **const** T &*check_value*, **const** U &*set_value*, *std*::vector<size_t> &*seeds*)

Applies a bucket-fill paint operation to one raster based on another.

Author Richard Barnes (rbarnes@umn.edu)

Parameters

- `&check_raster`: Raster whose values are checked for the BucketFill
- `&set_raster`: Raster whose values are set by the BucketFill. If `set_raster` already has `set_value`, then the FloodFill won't progress over it. This avoids needing a separate visisted raster.
- `check_value`: Value in `check_raster` which indicates a value in `set_raster` should be set

- `set_value`: Value that `set_raster` is set to
- `&seed`: Vector of seed cells to seed the BucketFill

template <Topology *topo*, **class** T, **class** U>
void **BucketFillFromEdges**(**const** *Array2D*<T> &*check_raster*, *Array2D*<U> &*set_raster*, **const** T &*check_value*, **const** U &*set_value*)

Applies a bucket-fill paint operation to one raster based on another starting from the edges.

Author Richard Barnes (rbarnes@umn.edu)

Parameters

- `&check_raster`: Raster whose values are checked for the BucketFill
- `&set_raster`: Raster whose values are set by the BucketFill. If `set_raster` already has `set_value`, then the FloodFill won't progress over it. This avoids needing a separate visisted raster.
- `check_value`: Value in `check_raster` which indicates a value in `set_raster` should be set
- `set_value`: Value that `set_raster` is set to

Array2D<double> **perlin**(**const** int *size*, **const** uint32_t *seed*)

GDALDataType **peekLayoutType**(**const** *std*::string &*layout_filename*)

int **peekLayoutTileSize**(**const** *std*::string &*layout_filename*)

Variables

const double **SQRT2** = 1.4142135623730950488016887242096980785696718753769480

sqrt(2), used to generate distances from a central cell to its neighbours

const int **dx**[9] = {0, -1, -1, 0, 1, 1, 1, 0, -1}

x offsets of D8 neighbours, from a central cell

const int **dy**[9] = {0, 0, -1, -1, -1, 0, 1, 1, 1}

y offsets of D8 neighbours, from a central cell

const double **d8r**[9] = {0,1,SQRT2,1,SQRT2,1,SQRT2,1,SQRT2}

const bool **n_diag**[9] = {0, 0, 1, 0, 1, 0, 1, 0, 1}

True along diagonal directions, false along north, south, east, west.

const int **D8_WEST** = 1

const int **D8_NORTH** = 3

const int **D8_EAST** = 5

const int **D8_SOUTH** = 7

const int ***const** **d8x** = dx

const int ***const** **d8y** = dy

const int **d4x**[5] = {0, -1, 0, 1, 0}

x offsets of D4 neighbours, from a central cell

const int **d4y**[5] = {0, 0, -1, 0, 1}

y offsets of D4 neighbours, from a central cell

const double **d4r**[5] = {0, 1, 1, 1, 1}

const int **D4_WEST** = 1

const int **D4_NORTH** = 2

const int **D4_EAST** = 3

const int **D4_SOUTH** = 4

const int **d8_inverse**[9] = {0,5,6,7,8,1,2,3,4}
Directions from neighbours to the central cell. Neighbours are labeled 0-8. This is the inverse direction leading from a neighbour to the central cell.

const int **d4_inverse**[5] = {0, 3, 4, 1, 2}

const double **dr**[9] = {0,1,SQRT2,1,SQRT2,1,SQRT2,1,SQRT2}
Distances from a central cell to each of its 8 neighbours.

const uint8_t **d8_arcgis**[9] = {0,16,32,64,128,1,2,4,8}
Convert from RichDEM flowdirs to ArcGIS flowdirs.

const uint8_t **FLOWDIR_NO_DATA** = 255
Used to indicate that a flowdir cell is NoData.

const *flowdir_t* **NO_FLOW** = 0
Value used to indicate that a cell does not have a defined flow direction.

const float **NO_FLOW_GEN** = -1
Value used to indicate NoFlow in generic flow metric outputs.

const float **HAS_FLOW_GEN** = 0

const float **NO_DATA_GEN** = -2

const int32_t **ACCUM_NO_DATA** = -1
Value used to indicate that a flow accumulation cell is NoData.

const uint8_t **GRID_LEFT** = 1
Indicates a tile is on the LHS of a DEM.

const uint8_t **GRID_TOP** = 2
Indicates a tile is on the top of a DEM.

const uint8_t **GRID_RIGHT** = 4
Indicates a tile is on the RHS of a DEM.

const uint8_t **GRID_BOTTOM** = 8
Indicates a tile is on the bottom of a DEM.

const auto **RICHDEM_FP_COMPARISON_ERROR** = 1e-6

const *std*::string **git_hash** = "NO HASH SPECIFIED!"
Git hash of program's source (used if RICHDEM_GIT_HASH is undefined)

const *std*::string **compilation_datetime** = __DATE__ " " __TIME__
Date and time of when the program was compiled (used if RICHDEM_COMPILE_TIME is undefined)

const *std*::string **program_name** = "RichDEM v2.2.9"
Richdem vX.X.X.

const *std*::string **author_name** = "Richard Barnes"
Richard Barnes.

const *std*::string **copyright** = "Richard Barnes © 2018"
Richard Barnes © 2018.

const *std*::string **program_identifier** = program_name + " (hash=" + git_hash + ", compiled="+compilation_datetime + Richdem vX.X.X (hash=GIT HASH, compiled=COMPILATION DATE TIME)

const float **d8_to_dinf**[9] ={-1, 4*M_PI/4, 3*M_PI/4, 2*M_PI/4, 1*M_PI/4, 0, 7*M_PI/4, 6*M_PI/4, 5*M_PI/4}

const int **dy_e1**[8] = { 0 , -1 , -1 , 0 , 0 , 1 , 1 , 0 }

const int **dx_e1**[8] = { 1 , 0 , 0 , -1 , -1 , 0 , 0 , 1 }

const int **dy_e2**[8] = {-1 , -1 , -1 , -1 , 1 , 1 , 1 , 1 }

const int **dx_e2**[8] = { 1 , 1 , -1 , -1 , -1 , -1 , 1 , 1 }

const double **ac**[8] = { 0., 1., 1., 2., 2., 3., 3., 4.}

const double **af**[8] = { 1., -1., 1., -1., 1., -1., 1., -1.}

const int **dinf_dx**[9] = {1, 1, 0, -1, -1, -1, 0, 1, 1}

const int **dinf_dy**[9] = {0, -1, -1, -1, 0, 1, 1, 1, 0}

namespace std

file **Array2D.hpp**

#include "gdal.hpp"#include <array>#include <vector>#include <iostream>#include <fstream>#include <iomanip>#include <cassert>#include <algorithm>#include <typeinfo>#include <stdexcept>#include <limits>#include <ctime>#include <cmath>#include <unordered_set>#include <map>#include <richdem/common/Array3D.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/version.hpp>#include <richdem/common/constants.hpp>#include <richdem/common/ManagedVector.hpp> Defines a 2D array object with many convenient methods for working with raster data, along with several functions for checking file data types.

Richard Barnes (rbarnes@umn.edu), 2015

file **Array3D.hpp**

#include "gdal.hpp"#include <vector>#include <iostream>#include <fstream>#include <iomanip>#include <cassert>#include <algorithm>#include <typeinfo>#include <stdexcept>#include <limits>#include <ctime>#include <unordered_set>#include <map>#include <richdem/common/Array2D.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/version.hpp>#include <richdem/common/constants.hpp>#include <richdem/common/ManagedVector.hpp> Defines a 3D array object with convenient methods for working raster data where information about neighbours needs to be stored and processed.

Richard Barnes (rbarnes@umn.edu), 2018

file **communication-threads.hpp**

#include <cereal/types/string.hpp>#include <cereal/types/vector.hpp>#include <cereal/types/map.hpp>#include <cereal/archives/binary.hpp>#include <sstream>#include <vector>#include <queue>#include <map>#include <atomic>#include <iterator>#include <cassert>#include <iostream>#include <list>#include <chrono>

Defines

_unused(x)

Functions

template <**class** Fn>
void **CommInit**(int *n*, Fn &&*fn*, int **argc*, char ****argv*)
template <**class** T, **class** U>

void **CommSend**(**const** T *a, **const** U *b, int *dest*, int *tag*)
template <**class** T>
void **CommSend**(**const** T *a, nullptr_t, int *dest*, int *tag*)

int **CommGetTag**(int *from*)

int **CommGetSource**()

int **CommRank**()

int **CommSize**()

void **CommAbort**(int *errorcode*)
template <**class** T, **class** U>
void **CommRecv**(T *a, U *b, int *from*)
template <**class** T>
void **CommRecv**(T *a, nullptr_t, int *from*)
template <**class** T>
void **CommBroadcast**(T *datum, int *root*)

void **CommFinalize**()

int **CommBytesSent**()

int **CommBytesRecv**()

void **CommBytesReset**()

void **CommBarrier**()

file **communication.hpp**
#include <mpi.h>#include <cereal/types/string.hpp>#include <cereal/types/vector.hpp>#include <cereal/types/map.hpp>#include <cereal/archives/binary.hpp>#include <sstream>#include <vector>#include <iterator>#include <cassert>#include <iostream>#include <thread>#include <chrono> Abstract calls to MPI, allowing for transparent serialization and communication stats.

Richard Barnes (rbarnes@umn.edu), 2015

Defines

_unused(x)
Used to hide the fact that some variables are used only for assertions.

Typedefs

typedef uint64_t **comm_count_type**
Data type used for storing Tx/Rx byte counts.

typedef *std*::vector<char> **msg_type**
Data type for incoming/outgoing messages.

Functions

void **CommInit**(int *argc, char ***argv)
Initiate communication (wrapper for MPI_Init)
template <**class** T, **class** U>
msg_type **CommPrepare**(**const** T *a, **const** U *b)
Convert up to two objects into a combined serialized representation.

template <class T>
msg_type **CommPrepare**(**const** T *a, *std*::nullptr_t)
Convert one object into a serialized representation.
template <class T, **class** U>
void **CommSend**(**const** T *a, **const** U *b, int *dest*, int *tag*)
Serialize and send up to two objects.
template <class T>
void **CommSend**(**const** T *a, *std*::nullptr_t, int *dest*, int *tag*)
Serialize and send a single object.

void **CommISend**(*msg_type* &*msg*, int *dest*, int *tag*)
Send a pre-serialized object using non-blocking communication.

The object must be pre-serialized because the buffer containing the serialization must persist until the communication is complete. It makes more sense to manage this buffer outside of this library.

int **CommGetTag**(int *from*)
Check tag of incoming message. Blocksing.

int **CommRank**()
Get my unique process identifier (i.e. rank)

int **CommSize**()
How many processes are active?

void **CommAbort**(int *errorcode*)
Abort; If any process calls this it will kill all the processes.
template <class T, **class** U>
void **CommRecv**(T *a, U *b, int *from*)
Receive up to two objects and deserialize them.
template <class T>
void **CommRecv**(T *a, *std*::nullptr_t, int *from*)
Receive one object and deserialize it.
template <class T>
void **CommBroadcast**(T *datum, int *root*)
Broadcast a message to all of the processes. (TODO: An integer message?)

void **CommFinalize**()
Wrap things up politely; call this when all communication is done.

comm_count_type **CommBytesSent**()
Get the number of bytes sent by this process.

Return Number of bytes sent by this process

comm_count_type **CommBytesRecv**()
Get the number of bytes received by this process.

Return Number of bytes received by this process

void **CommBytesReset**()
Reset message size statistics to zero.

Variables

comm_count_type **bytes_sent** = 0
Number of bytes sent.

comm_count_type **bytes_recv** = 0
Number of bytes received.

file **constants.hpp**
#include <cstdint>#include <stdexcept>#include <string> Defines a number of constants used by many of the algorithms.

RichDEM uses the following D8 neighbourhood. This is used by the dx[] and dy[] variables, among many others.

```
234
105
876
```

ArcGIS uses the following bits to indicate flow toward a given neighbour:

```
32 64 128
16  0   1
 8  4   2
```

D4 directions 2 103 4

file **gdal.hpp**

file **grid_cell.hpp**
#include <vector>#include <queue>#include <cmath>#include <functional> Defines structures for addressing grid cells and associated queues.

Richard Barnes (rbarnes@umn.edu), 2015

file **Layoutfile.hpp**
#include <richdem/common/logger.hpp>#include <string>#include <vector>#include <fstream>#include <sstream>#include <iostream>#include <cassert>#include <stdexcept> Defines classes used for reading and writing tiled datasets.

A layout file is a text file with the format:

```
file1.tif, file2.tif, file3.tif,
file4.tif, file5.tif, file6.tif, file7.tif
         , file8.tif,          ,
```

where each of fileX.tif is a tile of the larger DEM collectively described by all of the files. All of fileX.tif must have the same shape; the layout file specifies how fileX.tif are arranged in relation to each other in space. Blanks between commas indicate that there is no tile there: the algorithm will treat such gaps as places to route flow towards (as if they are oceans). Note that the files need not have TIF format: they can be of any type which GDAL can read. Paths to fileX.tif are taken to be relative to the layout file.

Richard Barnes (rbarnes@umn.edu), 2015

file **loaders.hpp**
#include <richdem/common/Array2D.hpp>#include <richdem/common/gdal.hpp>

file **logger.hpp**
#include <iostream>#include <sstream>#include <string>

Defines

RDLOG(flag)

RDLOG_ALG_NAME

RDLOG_CITATION

RDLOG_CONFIG

RDLOG_DEBUG

RDLOG_ERROR

RDLOG_MEM_USE

RDLOG_MISC

RDLOG_PROGRESS

RDLOG_TIME_USE

RDLOG_WARN

file **ManagedVector.hpp**
#include <memory>

file **math.hpp**
#include <cstdint>#include <cmath>

file **memory.hpp**
#include <fstream>#include <string> Defines functions for calculating memory usage.

Richard Barnes (rbarnes@umn.edu), 2015

file **ProgressBar.hpp**
#include <string>#include <iostream>#include <iomanip>#include <stdexcept>#include <richdem/common/timer.hpp> Defines a handy progress bar object so users don't get impatient.

The progress bar indicates to the user how much work has been completed, how much is left, and how long it is estimated to take. It accounts for multithreading by assuming uniform progress by all threads.

Define the global macro `RICHDEM_NO_PROGRESS` disables the progress bar, which may speed up the program.

The progress bar looks like this:

```
[==================================                ] (70% - 0.2s - 1 threads)
```

Richard Barnes (rbarnes@umn.edu), 2015

Defines

omp_get_thread_num()
Macros used to disguise the fact that we do not have multithreading enabled.

omp_get_num_threads()

file **random.hpp**
#include <random>#include <string>

Defines

PRNG_THREAD_MAX
Maximum number of threads this class should deal with.

omp_get_thread_num()

omp_get_num_threads()

omp_get_max_threads()

file **timer.hpp**

#include <chrono>#include <stdexcept> Defines the Timer class, which is used for timing code.

Richard Barnes (rbarnes@umn.edu), 2015

file **version.hpp**

#include <string>#include <iostream> Defines RichDEM version, git hash, compilation time. Used for program/app headers and for processing history entries.

Richard Barnes (rbarnes@umn.edu), 2015

file **Barnes2014.hpp**

#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/flowmet/d8_flowdirs.hpp>#include <queue>#include <limits>#include <iostream>#include <cstdlib> Defines all the Priority-Flood algorithms described by Barnes (2014) "Priority-Flood: An Optimal Depression-Filling and Watershed-Labeling Algorithm for Digital Elevation Models".

Richard Barnes (rbarnes@umn.edu), 2015

file **Barnes2014.hpp**

#include <richdem/common/logger.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/flats/find_flats.hpp>#include <deque>#include <vector>#include <queue>#include <cmath>#include <limits> Resolve flats according to Barnes (2014)

Contains code to generate an elevation mask which is guaranteed to drain a flat using a convergent flow pattern (unless it's a mesa)

Author Richard Barnes (rbarnes@umn.edu), 2012

file **depressions.hpp**

#include <richdem/common/constants.hpp>#include <richdem/depressions/Barnes2014.hpp>#include <richdem/depressions/Lindsay2016.hpp>#include <richdem/depressions/Wei2018.hpp>#include <richdem/depressions/Zhou2016.hpp>#include <stdexcept>

file **Lindsay2016.hpp**

#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/timer.hpp>#include <limits>

file **main.cpp**

#include <richdem/common/interface.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/depressions/Barnes2014.hpp>#include <string>#include <iostream>#include <cstdint>

Functions

template <**class** elev_t>
int **PerformAlgorithm**(char *alg*, *std*::string *filename*, *std*::string *output_prefix*)

int **main**(int *argc*, char ***argv*)

file **main.cpp**

#include <iostream>#include <cstdint>#include <string>#include <richdem/common/Array2D.hpp>#include <richdem/flats/flat_resolution.hpp>#include <richdem/flats/garbrecht.hpp>

Functions

template <**class** T>

int **PerformAlgorithm**(*std*::string *alg*, *std*::string *filename*, *std*::string *output*)

int **main**(int *argc*, char *******argv*)

file **README.md**

file **README.md**

file **Wei2018.hpp**
#include <richdem/common/Array2D.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/common/timer.hpp>#include <iostream>#include <queue>

file **Zhou2016.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/timer.hpp>#include <queue>#include <vector>#include <map>#include <iostream> Defines the Priority-Flood algorithm described by Zhou, G., Sun, Z., Fu, S., 2016. An efficient variant of the Priority-Flood algorithm for filling depressions in raster digital elevation models. Computers & Geosciences 90, Part A, 87 – 96. doi:http://dx.doi.org/10.1016/j.cageo.2016.02.021.

The code herein has been extensive modified by Richard Barnes (rbarnes@umn.edu) for inclusion with RichDEM.

Richard Barnes (rbarnes@umn.edu), 2015

file **find_flats.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/Array2D.hpp>

Defines

FLAT_NO_DATA

NOT_A_FLAT

IS_A_FLAT

file **flat_resolution.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/flowmet/d8_flowdirs.hpp>#include <deque>#include <vector>#include <queue>#include <cmath>#include <limits> Resolve flats according to Barnes (2014)

Contains code to generate an elevation mask which is guaranteed to drain a flat using a convergent flow pattern (unless it's a mesa)

Author Richard Barnes (rbarnes@umn.edu), 2012

file **flat_resolution_dinf.hpp**
#include <richdem/flats/flat_resolution.hpp>#include <richdem/flowmet/dinf_flowdirs.hpp>#include <richdem/common/logger.hpp> Couples the Barnes (2014) flat resolution algorithm with the Tarboton (1997) D-infinity flow metric.

Author Richard Barnes

file **flats.hpp**
#include <richdem/flats/Barnes2014.hpp>

file **garbrecht.hpp**
#include <deque>#include <cstdint>#include <iostream>#include <richdem/common/Array2D.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/flowmet/d8_flowdirs.hpp>#include <richdem/common/logger.hpp>

file **generate_square_grid.cpp**
#include <iostream>#include <fstream>#include <cstdlib>

Defines

XBIG

YBIG

IN_GRID(x, y)

EDGE_GRID(x, y)

Functions

int **PrintDEM**()

int **GenerateDEM**()

int **main**(int *argc*, char ***argv*)

Variables

char **elevations**[**XBIG**][**YBIG**]

int **x_max**

int **y_max**

file **d8_flowdirs.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/ProgressBar.hpp> Functions for calculating D8 flow directions.

Richard Barnes (rbarnes@umn.edu), 2015

file **dinf_flowdirs.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/ProgressBar.hpp> Defines the D-infinite flow routing method described by Tarboton (1997)

This file implements the D-infinite flow routing method originally described by Tarboton (1997). It incorporates minor alterations and additional safe-guards described in Barnes (TODO).

Richard Barnes (rbarnes@umn.edu), 2015

Defines

dinf_NO_DATA
Value used to indicate that a flow direction cell has no data.

file **Fairfield1991.hpp**
#include <richdem/common/constants.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/random.hpp>

file **Freeman1991.hpp**
#include <richdem/common/constants.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/common/ProgressBar.hpp>

file **Holmgren1994.hpp**
#include <richdem/common/constants.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/common/ProgressBar.hpp>

file **OCallaghan1984.hpp**
#include <richdem/common/constants.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/common/ProgressBar.hpp>

file **Orlandini2003.hpp**

file **Quinn1991.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/flowmet/Holmgren1994.hpp>

file **Seibert2007.hpp**

file **Tarboton1997.hpp**
#include <richdem/common/constants.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/Array3D.hpp>#include <richdem/common/ProgressBar.hpp>#include <cmath>

file **d8_methods.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/constants.hpp>#include <richdem/common/grid_cell.hpp>#include <richdem/common/ProgressBar.hpp>#include <queue>#include <stdexcept> Defines a number of functions for calculating terrain attributes.

Richard Barnes (rbarnes@umn.edu), 2015

file **dinf_methods.hpp**
#include <cmath>#include <queue>#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/constants.hpp>#include <richdem/common/ProgressBar.hpp>#include <richdem/common/grid_cell.hpp> Terrain attributes that can only be calculated with Tarboton's D-infinity flow metric.

This file implements the D-infinite flow routing method originally described by Tarboton (1997). It incorporates minor alterations and additional safe-guards described in Barnes (2013, TODO).

Author Richard Barnes (rbarnes@umn.edu), 2015

file **flow_accumulation.hpp**
#include <richdem/flowmet/Fairfield1991.hpp>#include <richdem/flowmet/Freeman1991.hpp>#include <richdem/flowmet/Holmgren1994.hpp>#include <richdem/flowmet/OCallaghan1984.hpp>#include <richdem/flowmet/Orlandini2003.hpp>#include <richdem/flowmet/Quinn1991.hpp>#include <richdem/flowmet/Seibert2007.hpp>#include <richdem/flowmet/Tarboton1997.hpp>#include <richdem/methods/flow_accumulation_generic.hpp>

file **flow_accumulation_generic.hpp**
#include <richdem/common/Array2D.hpp>#include <richdem/common/logger.hpp>#include <richdem/common/ProgressBar.hpp>#include <queue>

file **strahler.hpp**

file **terrain_attributes.hpp**
#include <richdem/common/logger.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/constants.hpp>#include <richdem/common/ProgressBar.hpp>

file **misc_methods.hpp**
#include <richdem/common/Array2D.hpp>#include <richdem/common/constants.hpp>#include <rich-

dem/common/ProgressBar.hpp>#include <cassert>#include <cmath>#include <queue>#include <stdexcept> Terrain attributes that can only be calculated with Tarboton's D-infinity flow metric.

This file implements the D-infinite flow routing method originally described by Tarboton (1997). It incorporates minor alterations and additional safe-guards described in Barnes (2013, TODO).

Author Richard Barnes (rbarnes@umn.edu), 2015

file **richdem.hpp**

#include "common/Array2D.hpp"#include "common/constants.hpp"#include "common/grid_cell.hpp"#include "common/ManagedVector.hpp"#include "common/memory.hpp"#include "common/ProgressBar.hpp"#include "common/random.hpp"#include "common/timer.hpp"#include "common/version.hpp"#include "depressions/Barnes2014.hpp"#include "depressions/depressions.hpp"#include "depressions/Lindsay2016.hpp"#include "depressions/Zhou2016.hpp"#include "flats/flat_resolution.hpp"#include "flats/flat_resolution_dinf.hpp"#include "flowmet/d8_flowdirs.hpp"#include "flowmet/dinf_flowdirs.hpp"#include "flowmet/Fairfield1991.hpp"#include "flowmet/Freeman1991.hpp"#include "flowmet/Holmgren1994.hpp"#include "flowmet/OCallaghan1984.hpp"#include "flowmet/Orlandini2003.hpp"#include "flowmet/Quinn1991.hpp"#include "flowmet/Seibert2007.hpp"#include "flowmet/Tarboton1997.hpp"#include "methods/d8_methods.hpp"#include "methods/dinf_methods.hpp"#include "methods/flow_accumulation.hpp"#include "methods/flow_accumulation_generic.hpp"#include "methods/strahler.hpp"#include "methods/terrain_attributes.hpp"

file **terrain_generation.hpp**

#include <richdem/common/Array2D.hpp>

file **A2Array2D.hpp**

#include <richdem/common/logger.hpp>#include <richdem/common/Layoutfile.hpp>#include <richdem/common/Array2D.hpp>#include <richdem/common/gdal.hpp>#include <richdem/tiled/lru.hpp>#include "gdal_priv.h" Experimental tile manager for large datasets (TODO)

Author Richard Barnes

file **lru.hpp**

#include <list>#include <unordered_map> Defines a Least-Recently Used (LRU) cache class.

Richard Barnes (rbarnes@umn.edu), 2016

page **md__home_docs_checkouts_readthedocs.org_user_builds_richdem_checkouts_latest_include_ri**

Title of Manuscript: Priority-Flood: An Optimal Depression-Filling and Watershed-Labeling Algorithm for Digital Elevation Models

Authors: Richard Barnes, Clarence Lehman, David Mulla

Corresponding Author: Richard Barnes (rbarnes@umn.edu)

DOI Number of Manuscript 10.1016/j.cageo.2013.04.024

Code Repositories

- Author's GitHub Repository
- Journal's GitHub Repository

This repository contains a reference implementation of the algorithms presented in the manuscript above. These implementations were used in performing the tests described in the manuscript.

There is source code for every pseudocode algorithm presented in the manuscript. All the code can be compiled simply by running **make**. The result is a program called **priority_flood.exe**.

This program reads in a DEM file specified on the command line. The file may be any ArcGrid ASCII file. The program will run one of the algorithms described in the manuscript (and below), store the result in an output file, and report how long this took.

The program is run by typing:

```
./priority_flood.exe <ALGORITHM NUMBER> <INPUT DEM>
./priority_flood.exe 3 input-data.asc
```

The algorithms available are described briefly below and in greater detail in the manuscript.

- **Algorithm 1: Priority-Flood** This algorithm alters the input DEM to produce an output with no depressions or digital dams. Every cell which would have been in a depression is increased to the level of that depression's outlet, leaving a flat region in its place. It runs slower than Algorithm 2, but is otherwise the same. The result is saved to **out-pf-original**.
- **Algorithm 2: Improved Priority-Flood** This algorithm alters the input DEM to produce an output with no depressions or digital dams. Every cell which would have been in a depression is increased to the level of that depression's outlet, leaving a flat region in its place. It runs faster than Algorithm 1, but is otherwise the same. The result is saved to **out-pf-improved**.
- **Algorithm 3: Priority-Flood+Epsilon** This algorithm alters the input DEM to produce an output with no depressions or digital dams. Every cell which would have been in a depression is increased to the level of that depression's output, plus an additional increment which is sufficient to direct flow to the periphery of the DEM. The result is saved to **out-pf-epsilon**.
- **Algorithm 4: Priority-Flood+FlowDirs** This algorithm determines a D8 flow direction for every cell in the DEM by implicitly filling depressions and eliminating digital dams. Though all depressions are guaranteed to drain, local elevation information is still used to determine flow directions within a depression. It is, in essence, a depression-carving algorithm. The result is saved to **out-pf-flowdirs**.
- **Algorithm 5: Priority-Flood+Watershed Labels** For each cell c in a DEM, this algorithm determines which cell on the DEM's periphery c will drain to. c is then given a label which corresponds to the peripheral cell. All cells bearing a common label belong to the same watershed. The result is saved to **out-pf-wlabels**.

Algorithm 4: Priority-Flood+FlowDirs and its output, **out-pf-flowdirs**, use the D8 neighbour system to indicate flow directions. In this system all the flow from a central cell is directed to a single neighbour which is represented by a number according to the following system where 0 indicates the central cell.

```
234
105
876
```

The directory **src/** contains the source code for the reference implementations. All the source code is drawn from the RichDEM hydroanalysis package. At the time of writing, the entire RichDEM code base could be downloaded from: https://github.com/r-barnes

Assumptions

All of the algorithms assume that cells marked as having NoData will have extremely negative numerical values: less than the value of any of the actual data. NaN is considered to be less than all values, including negative infinity.

Notes on the Manuscript

Work by Cris Luengo on the speed of various priority queue algorithms is discussed in the manuscript. His website providing code for his implementatations is here.

Updates

Commit **51f9a7838d3e88628ef6c74846edd0cb18e7ffe6** (02015-09-25) introduced a number of changes to the code versus what was originally published with the manuscript. The old codebase uses ASCII-formatted data for input and output; the new codebase uses GDAL to handle many kinds of data.

The old codebase had the advantage of not relying on external libraries and being readily accessible to all parties. It had the disadvantage of being a slow, clumsy, and limited way to work with the data. As of 02015-09-25, the code requires the use of the GDAL library greatly expanding the data formats and data types which can be worked with, as well as greatly speeding up I/O.

Note that using the aforementioned **51f9a7838d** directly will result in silent casting of your data to the `float` type; commit **8b11f535af23368d3bd26609cc88df3dbb7111f1** (02015-09-28) fixes this issue.

Additionally, the library now uses C++ for all streaming operations except the progress bar.

page **md__home_docs_checkouts_readthedocs.org_user_builds_richdem_checkouts_latest_include_ri**

Title of Manuscript: An Efficient Assignment of Drainage Direction Over Flat Surfaces in Raster Digital Elevation Models

Authors: Richard Barnes, Clarence Lehman, David Mulla

Corresponding Author: Richard Barnes (rbarnes@umn.edu)

DOI Number of Manuscript 10.1016/j.cageo.2013.01.009

Code Repositories

- Author's GitHub Repository
- Journal's GitHub Repository

This repository contains a reference implementation of the algorithms presented in the manuscript above. It also contains a reference implementation of the algorithm presented by *Garbrecht and Martz* (1997). These implementations were used in performing speed comparison tests in the manuscript.

All the programs can be produced simply by running **make**.

The program **generate_square_grid.exe** makes a square DEM with a single outlet near the bottom-left corner. The grid size is specified as a command-line argument.

The two reference implementations use the D8 neighbour system to indicate flow directions. In this system all the flow from a central cell is directed to a single neighbour which is represented by a number according to the following system where 0 indicates the central cell.

```
234
105
876
```

The program **barnes_algorithm.exe** reads in a DEM file specified on the command line. The file may be generated by **generate_square_grid.exe**, but may also be any ArcGrid ASCII file. The program will time itself and report the results back. The program will print the determined flow directions for the DEM to a file named **out_barnes**. The determined flow directions are also printed as a matrix of arrows to **out_barnes_arrows**.

The program **garbrecht_algorithm.exe** attempts to reproduce the algorithm described by *Garbrecht and Martz* (1997). It accepts an ArcGRID ASCII file as a command line input. The input file may also be generated with **generate_square_grid.exe**. Note that this implementation does not apply itself iteratively, meaning that some flats will be unresolvable. It writes the determined flow directions to **out_garbrecht**. The determined flow directions are also printed as a matrix of arrows to **out_garbrecht_arrows**.

The directory **src/** contains the source code for reference implementations. The source for the improved algorithm is drawn from the RichDEM hydroanalysis package. All code can be compiled by running the **makefile** included in the root directory. Running the BASH script **FIRST_RUN** will compile everything and run the programs.

At the time of writing, the entire RichDEM code base could be downloaded from: https://github.com/r-barnes

page **todo**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

dir **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/richde**

example **/home/docs/checkouts/readthedocs.org/user_builds/richdem/checkouts/latest/include/ric**

Creates a raster from a nested initializer list Array2D<int> x = {{1,2,3},{4,5,6}}

```
#ifndef _richdem_array_2d_hpp_
#define _richdem_array_2d_hpp_

#include "gdal.hpp"
#include <array>
#include <vector>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cassert>
#include <algorithm>
#include <typeinfo>
#include <stdexcept>
#include <limits>
#include <ctime>         //Used for timestamping output files
#include <cmath>
#include <unordered_set> //For printStamp
#include <stdexcept>
#include <map>
#include <richdem/common/Array3D.hpp>
#include <richdem/common/logger.hpp>
#include <richdem/common/version.hpp>
#include <richdem/common/constants.hpp>
#include <richdem/common/ManagedVector.hpp>

//These enable compression in the loadNative() and saveNative() methods
#ifdef WITH_COMPRESSION
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/zlib.hpp>
#endif

namespace richdem {

template<typename> class Array3D;

inline std::map<std::string, std::string> ProcessMetadata(char **metadata){
  std::map<std::string, std::string> ret;
```

(continues on next page)

(continued from previous page)

```
  if(metadata==NULL)
    return ret;

  for(int metstri=0;metadata[metstri]==NULL;metstri++){
    std::string metstr = metadata[metstri];
    const auto equals  = metstr.find("=");
    if(equals==std::string::npos){
      RDLOG_WARN<<"Skipping improper metadata string: '"<<metstr<<"'";
      continue;
    }
    std::string keystr = metstr.substr(0,equals);
    std::string valstr = metstr.substr(equals+1);
    if(ret.count(keystr)>0){
      RDLOG_WARN<<"Duplicate key '"<<keystr<<"' found in metadata. Only latter␣
↪value will be kept.";
    }
    ret[keystr] = valstr;
  }

  return ret;
}

template<class T>
class Array2D {
 public:
  std::string filename;
  std::string basename;
  std::vector<double> geotransform;
  std::string projection;
  std::map<std::string, std::string> metadata;

  //Using uint32_t for i-addressing allows for rasters of ~65535^2. These
  //dimensions fit easily within an int32_t xy-address.
  typedef int32_t  xy_t;
  typedef uint32_t i_t;

  static const i_t NO_I = std::numeric_limits<i_t>::max(); //TODO: What is this?

 private:
  template<typename> friend class Array2D;
  template<typename> friend class Array3D;

  std::array<int, 9> _nshift;

  ManagedVector<T> _data;

  T no_data = -1;
  mutable i_t num_data_cells = NO_I;

  xy_t view_width  = 0;
  xy_t view_height = 0;

  xy_t view_xoff = 0;
  xy_t view_yoff = 0;

  bool from_cache;
```

(continues on next page)

(continued from previous page)

```
  #ifdef USEGDAL
  void loadGDAL(const std::string &filename, xy_t xOffset=0, xy_t yOffset=0, xy_t␣
↪part_width=0, xy_t part_height=0, bool exact=false, bool load_data=true){
    assert(empty());

    from_cache = false;

    this->filename = filename;

    RDLOG_PROGRESS<<"Trying to open file '"<<filename<<"'...";

    GDALDataset *fin = (GDALDataset*)GDALOpen(filename.c_str(), GA_ReadOnly);
    if(fin==NULL)
      throw std::runtime_error("Could not open file '"+filename+"' with GDAL!");

    geotransform.resize(6);
    if(fin->GetGeoTransform(geotransform.data())!=CE_None){
      RDLOG_WARN<<"Could not get a geotransform from '"<<filename<<"'! Setting to␣
↪an arbitrary standard geotransform.";
      geotransform = {{1000., 1., 0., 1000., 0., -1.}};
    }

    metadata = ProcessMetadata(fin->GetMetadata());

    const char* projection_string=fin->GetProjectionRef();
    projection = std::string(projection_string);

    GDALRasterBand *band = fin->GetRasterBand(1);

    xy_t total_width  = band->GetXSize();         //Returns an int
    xy_t total_height = band->GetYSize();         //Returns an int
    no_data           = band->GetNoDataValue();

    if(exact && (total_width-xOffset!=part_width || total_height-yOffset!=part_
↪height))
      throw std::runtime_error("Tile dimensions did not match expectations!");

    //TODO: What's going on here?

    if(xOffset+part_width>=total_width)
      part_width  = total_width-xOffset;
    if(yOffset+part_height>=total_height)
      part_height = total_height-yOffset;

    if(part_width==0)
      part_width = total_width;
    view_width = part_width;

    if(part_height==0)
      part_height = total_height;
    view_height = part_height;

    view_xoff = xOffset;
    view_yoff = yOffset;

    GDALClose(fin);
```

(continues on next page)

(continued from previous page)

```
    if(load_data)
      loadData();
  }
  #endif


  #ifdef USEGDAL
  GDALDataType myGDALType() const {
    return NativeTypeToGDAL<T>();
  }
  #endif


  //TODO: Should save metadata
 public:
  void saveToCache(const std::string &filename){
    std::fstream fout;

    from_cache     = true;
    this->filename = filename;

    fout.open(filename, std::ios_base::binary | std::ios_base::out |␣
↪std::ios::trunc);
    if(!fout.good())
      throw std::logic_error("Failed to open cache file '"+filename+"'.");

    #ifdef WITH_COMPRESSION
      boost::iostreams::filtering_ostream out;
      out.push(boost::iostreams::zlib_compressor());
      out.push(fout);
    #else
      auto &out = fout;
    #endif

    out.write(reinterpret_cast<const char*>(&view_height),    sizeof(xy_t));
    out.write(reinterpret_cast<const char*>(&view_width),     sizeof(xy_t));
    out.write(reinterpret_cast<const char*>(&view_xoff),      sizeof(xy_t));
    out.write(reinterpret_cast<const char*>(&view_yoff),      sizeof(xy_t));
    out.write(reinterpret_cast<const char*>(&num_data_cells), sizeof(i_t));
    out.write(reinterpret_cast<const char*>(&no_data),        sizeof(T  ));

    out.write(reinterpret_cast<const char*>(geotransform.data()),␣
↪6*sizeof(double));
    std::string::size_type projection_size = projection.size();
    out.write(reinterpret_cast<const char*>(&projection_size),␣
↪sizeof(std::string::size_type));
    out.write(reinterpret_cast<const char*>(projection.data()), projection.
↪size()*sizeof(const char));

    out.write(reinterpret_cast<const char*>(_data.data()), size()*sizeof(T));
  }

 private:

  void loadNative(const std::string &filename, bool load_data=true){
    std::ifstream fin(filename, std::ios::in | std::ios::binary);
```

(continues on next page)

(continued from previous page)

```
    if(!fin.good())
      throw std::runtime_error("Failed to load native file '" + filename +"!");

    this->filename = filename;
    from_cache     = true;

    #ifdef WITH_COMPRESSION
      boost::iostreams::filtering_istream in;
      in.push(boost::iostreams::zlib_decompressor());
      in.push(fin);
    #else
      auto &in = fin;
    #endif

    in.read(reinterpret_cast<char*>(&view_height),    sizeof(xy_t));
    in.read(reinterpret_cast<char*>(&view_width),     sizeof(xy_t));
    in.read(reinterpret_cast<char*>(&view_xoff),      sizeof(xy_t));
    in.read(reinterpret_cast<char*>(&view_yoff),      sizeof(xy_t));
    in.read(reinterpret_cast<char*>(&num_data_cells), sizeof(i_t));
    in.read(reinterpret_cast<char*>(&no_data),        sizeof(T  ));
    geotransform.resize(6);
    in.read(reinterpret_cast<char*>(geotransform.data()), 6*sizeof(double));

    std::string::size_type projection_size;
    in.read(reinterpret_cast<char*>(&projection_size), sizeof(std::string::size_
↪type));
    projection.resize(projection_size,' ');
    in.read(reinterpret_cast<char*>(&projection[0]), projection.
↪size()*sizeof(char));

    if(load_data){
      resize(view_width,view_height);
      in.read(reinterpret_cast<char*>(_data.data()), size()*sizeof(T));
    }
  }

 public:
  Array2D(){
    #ifdef USEGDAL
    GDALAllRegister();
    #endif
  }

  Array2D(xy_t width, xy_t height, const T& val = T()) : Array2D() {
    resize(width,height,val);
  }

  Array2D(std::initializer_list<std::initializer_list<T>> values) {
    const size_t height = values.size();
    const size_t width = values.begin()->size();
    for(const auto &x: values){
      if(x.size()!=width)
        throw std::runtime_error("All rows of the array must be the same width!");
    }

    resize(width, height);
```

(continues on next page)

(continued from previous page)

```
    size_t x=0;
    size_t y=0;
    for(const auto &row: values){
      x=0;
      for(const auto &col: row){
        operator()(x,y) = col;
        x++;
      }
      y++;
    }
    (void)height; //Suppress unused variable warning with NDEBUG
    assert(y==height);
  }

  Array2D(T *data0, const xy_t width, const xy_t height) : Array2D() {
    assert(width>0);
    assert(height>0);
    assert(width<=std::numeric_limits<xy_t>::max()-2);
    _data       = ManagedVector<T>(data0, (uint64_t)width*(uint64_t)height);
    view_width  = width;
    view_height = height;
    _nshift     = {{0,-1,-width-1,-width,-width+1,1,width+1,width,width-1}};
  }

  template<class U>
  Array2D(const Array2D<U> &other, const T& val=T()) : Array2D() {
    view_width          = other.view_width;
    view_height         = other.view_height;
    view_xoff           = other.view_xoff;
    view_yoff           = other.view_yoff;
    geotransform        = other.geotransform;
    metadata            = other.metadata;
    projection          = other.projection;
    basename            = other.basename;
    resize(other.width(), other.height(), val);
  }

  template<class U>
  Array2D(const Array3D<U> &other, const T& val=T()) : Array2D() {
    view_width          = other.view_width;
    view_height         = other.view_height;
    view_xoff           = other.view_xoff;
    view_yoff           = other.view_yoff;
    geotransform        = other.geotransform;
    metadata            = other.metadata;
    projection          = other.projection;
    basename            = other.basename;
    resize(other.width(), other.height(), val);
  }

  Array2D(const std::string &filename) : Array2D(filename, false, 0,0,0,0, false,␣
↪true) {}

  Array2D(const std::string &filename, bool native, xy_t xOffset=0, xy_t␣
↪yOffset=0, xy_t part_width=0, xy_t part_height=0, bool exact=false, bool load_
↪data=true) : Array2D() {
    if(native){
```

(continues on next page)

(continued from previous page)

```
      loadNative(filename, load_data);
    } else {
      #ifdef USEGDAL
      loadGDAL(filename, xOffset, yOffset, part_width, part_height, exact, load_
↪data);
      #else
        throw std::runtime_error("RichDEM was not compiled with GDAL!");
      #endif
    }
  }

  void setCacheFilename(const std::string &filename){
    this->filename = filename;
  }

  void dumpData(){
    saveToCache(filename);
    clear();
  }

  void loadData() {
    if(!_data.empty())
      throw std::runtime_error("Data already loaded!");

    if(from_cache){
      loadNative(filename, true);
    } else {
      #ifdef USEGDAL
      GDALDataset *fin = (GDALDataset*)GDALOpen(filename.c_str(), GA_ReadOnly);
        if(fin==NULL)
          throw std::runtime_error("Failed to loadData() into tile from '
↪"+filename+"'");

      GDALRasterBand *band = fin->GetRasterBand(1);

      resize(view_width,view_height);
      auto temp = band->RasterIO( GF_Read, view_xoff, view_yoff, view_width, view_
↪height, _data.data(), view_width, view_height, myGDALType(), 0, 0 );
      if(temp!=CE_None)
        throw std::runtime_error("An error occured while trying to read '
↪"+filename+"' into RAM with GDAL.");

      GDALClose(fin);
      #else
        throw std::runtime_error("RichDEM was not compiled with GDAL!");
      #endif
    }
  }

  T*       getData()       { std::cerr<<"richdem::Array2D::getData() is␣
↪deprecated. Use data() instead"<<std::endl; return _data.data(); }
  const T* getData() const { std::cerr<<"richdem::Array2D::getData() is␣
↪deprecated. Use data() instead"<<std::endl; return _data.data(); }

  T*       data()       { return _data.data(); }
  const T* data() const { return _data.data(); }
```

(continues on next page)

(continued from previous page)

```
  i_t size() const { return view_width*view_height; }

  xy_t width() const { return view_width; }

  xy_t height() const { return view_height; }

  xy_t viewXoff() const { return view_xoff; }

  xy_t viewYoff() const { return view_yoff; }

  bool empty() const { return _data.empty(); }

  T noData() const { return no_data; }

  T min() const {
    T minval = std::numeric_limits<T>::max();
    for(unsigned int i=0;i<size();i++){
      if(_data[i]==no_data)
        continue;
      minval = std::min(minval,_data[i]);
    }
    return minval;
  }

  T max() const {
    T maxval = std::numeric_limits<T>::min();
    for(unsigned int i=0;i<size();i++){
      if(_data[i]==no_data)
        continue;
      maxval = std::max(maxval,_data[i]);
    }
    return maxval;
  }

  void replace(const T oldval, const T newval){
    for(unsigned int i=0;i<size();i++)
      if(_data[i]==oldval)
        _data[i] = newval;
  }

  i_t countval(const T val) const {
    //TODO: Warn if raster is empty?
    i_t count=0;
    for(unsigned int i=0;i<size();i++)
      if(_data[i]==val)
        count++;
    return count;
  }

  //TODO
  inline i_t i0() const {
    return (i_t)0;
  }

  void iToxy(const i_t i, xy_t &x, xy_t &y) const {
    x = i%view_width;
    y = i/view_width;
```

(continues on next page)

(continued from previous page)

```
}

inline i_t xyToI(xy_t x, xy_t y) const {
  assert(0<=x && x<view_width && 0<=y && y<view_height);
  return (i_t)y*(i_t)view_width+(i_t)x;
}

i_t nToI(i_t i, xy_t dx, xy_t dy) const {
  int32_t x=i%view_width+dx;
  int32_t y=i/view_width+dy;
  if(x<0 || y<0 || x>=view_width || y>=view_height)
    return NO_I;
  return xyToI(x,y);
}

i_t getN(i_t i, uint8_t n) const {
  assert(0<=n && n<=8);
  xy_t x = i%view_width+(xy_t)dx[n];
  xy_t y = i/view_width+(xy_t)dy[n];
  if(x<0 || y<0 || x>=view_width || y>=view_height)
    return NO_I;
  return xyToI(x,y);
}

inline int nshift(const uint8_t n) const {
  assert(0<=n && n<=8);
  return _nshift[n];
}

bool operator==(const Array2D<T> &o) const {
  if(width()!=o.width() || height()!=o.height())
    return false;
  if(noData()!=o.noData())
    return false;
  for(auto i=i0();i<o.size();i++)
    if(_data[i]!=o._data[i])
      return false;
  return true;
}

inline bool isNoData(xy_t x, xy_t y) const {
  assert(0<=x && x<view_width);
  assert(0<=y && y<view_height);
  return _data[xyToI(x,y)]==no_data;
}

inline bool isNoData(i_t i) const {
  assert(0<=i && i<size());
  return _data[i]==no_data;
}

inline bool isData(xy_t x, xy_t y) const {
  assert(0<=x && x<view_width);
  assert(0<=y && y<view_height);
  return _data[xyToI(x,y)]!=no_data;
}
```

(continues on next page)

(continued from previous page)

```
  inline bool isData(i_t i) const {
    assert(0<=i && i<size());
    return _data[i]!=no_data;
  }

  void flipVert(){
    for(xy_t y=0;y<view_height/2;y++)
    for(xy_t x=0;x<view_width;x++)
      std::swap(_data[xyToI(x,y)], _data[xyToI(x,view_height-1-y)]);
  }

  void flipHorz(){
    for(xy_t y=0;y<view_height;y++){
      T* start = &_data[xyToI(0,y)];
      T* end   = &_data[xyToI(view_width,y)];
      while(start<end){
        std::swap(*start,*end);
        start++;
        end--;
      }
    }
  }

  void transpose(){
    RDLOG_WARN<<"transpose() is an experimental feature.";
    std::vector<T> new_data(view_width*view_height);
    for(xy_t y=0;y<view_height;y++)
    for(xy_t x=0;x<view_width;x++)
      std::swap(_data[(i_t)x*(i_t)view_height+(i_t)y], _data[xyToI(x,y)]);
    std::swap(view_width,view_height);
    //TODO: Offsets?
  }

  inline bool inGrid(xy_t x, xy_t y) const {
    return 0<=x && x<view_width && 0<=y && y<view_height;
  }

  /*
    @brief Test whether a cell lies within the boundaries of the raster.

    Obviously this bears some difference from `inGrid(x,y)`.

    @param[in]  i   i-coordinate of cell to test

    @return TRUE if cell lies within the raster
  */
  // bool inGrid(i_t i) const {
  //   return 0<=i && i<size();
  // }

  inline bool isEdgeCell(xy_t x, xy_t y) const {
    return x==0 || y==0 || x==view_width-1 || y==view_height-1;
  }

  bool isTopLeft     (xy_t x, xy_t y) const { return x==0          && y==0;
↪ }
  bool isTopRight    (xy_t x, xy_t y) const { return x==width()-1 && y==0;
↪ }
```

(continues on next page)

(continued from previous page)

```
  bool isBottomLeft (xy_t x, xy_t y) const { return x==0          && y==height()-1;
↪ }
  bool isBottomRight(xy_t x, xy_t y) const { return x==width()-1 && y==height()-1;
↪ }

  bool isTopRow    (xy_t x, xy_t y) const { return y==0;           }
  bool isBottomRow (xy_t x, xy_t y) const { return y==height()-1; }
  bool isLeftCol   (xy_t x, xy_t y) const { return x==0;           }
  bool isRightCol  (xy_t x, xy_t y) const { return x==width()-1;   }

  bool isEdgeCell(i_t i) const {
    xy_t x,y;
    iToxy(i,x,y);
    return isEdgeCell(x,y);
  }

  void setNoData(const T &ndval){
    no_data = ndval;
  }

  void setAll(const T val){
    for(i_t i=0;i<size();i++)
      _data[i] = val;
  }

  void resize(const xy_t width0, const xy_t height0, const T& val0 = T()){
    assert(width0>=0);
    assert(height0>=0);
    assert(width0<=std::numeric_limits<xy_t>::max()-2);
    _data.resize((uint64_t)width0*(uint64_t)height0);

    _nshift     = {{0,-1,-width0-1,-width0,-width0+1,1,width0+1,width0,width0-1}};

    view_width  = width0;
    view_height = height0;

    setAll(val0);
  }

  /*
    @brief Resize a raster to copy another raster's dimensions. Copy properies.

    @param[in]   other    Raster to match sizes with
    @param[in]   val      Value to set all the cells to. Defaults to the
                          raster's template type default value
  */
  template<class U>
  void resize(const Array2D<U> &other, const T& val = T()){
    resize(other.width(), other.height(), val);
    geotransform       = other.geotransform;
    projection         = other.projection;
  }

  void expand(uint64_t new_width, uint64_t new_height, const T val){
    RDLOG_DEBUG<<"Array2D::expand(width,height,val)";

    if(new_width==view_width && new_height==view_height)
```

(continues on next page)

(continued from previous page)

```
      return;
    if(!owned())
      throw std::runtime_error("RichDEM can only expand memory it owns!");

    if(new_width<view_width)
      throw std::runtime_error("expand(): new_width<view_width");
    if(new_height<view_height)
      throw std::runtime_error("expand(): new_height<view_height");

    auto old_width  = width();
    auto old_height = height();

    auto old_data = std::move(_data);   //This gets the pointer to the old data␣
↪before it is replaced

    resize(new_width,new_height,val);

    for(xy_t y=0;y<old_height;y++)
    for(xy_t x=0;x<old_width;x++)
      _data[y*new_width+x] = old_data[y*old_width+x];
  }

  void countDataCells() const {
    num_data_cells = 0;
    for(unsigned int i=0;i<size();i++)
      if(_data[i]!=no_data)
        num_data_cells++;
  }

  i_t numDataCells() const {
    if(num_data_cells==NO_I)
      countDataCells();
    return num_data_cells;
  }

  T& operator()(i_t i){
    assert(i>=0);
    assert(i<(i_t)view_width*view_height);
    return _data[i];
  }

  T operator()(i_t i) const {
    assert(i>=0);
    assert(i<(i_t)view_width*view_height);
    return _data[i];
  }

  T& operator()(xy_t x, xy_t y){
    assert(x>=0);
    assert(y>=0);
    assert(x<width());
    assert(y<height());
    return _data[xyToI(x,y)];
  }

  T operator()(xy_t x, xy_t y) const {
    assert(x>=0);
```

(continues on next page)

(continued from previous page)

```
    assert(y>=0);
    assert(x<width());
    assert(y<height());
    return _data[xyToI(x,y)];
  }

  std::vector<T> topRow() const {
    return getRowData(0);
  }

  std::vector<T> bottomRow() const {
    return getRowData(view_height-1);
  }

  std::vector<T> leftColumn() const {
    return getColData(0);
  }

  std::vector<T> rightColumn() const {
    return getColData(view_width-1);
  }

  void setRow(xy_t y, const T &val){
    for(xy_t x=0;x<view_width;x++)
      _data[xyToI(x,y)] = val;
  }

  void setCol(xy_t x, const T &val){
    for(xy_t y=0;y<view_height;y++)
      _data[xyToI(x,y)] = val;
  }

  void setEdges(const T &val){
    setRow(0,           val);
    setRow(height()-1, val);
    setCol(0,           val);
    setCol(width()-1,  val);
  }

  std::vector<T> getRowData(xy_t y) const {
    return std::vector<T>(_data.data()+xyToI(0,y),_data.data()+xyToI(0,y)+view_
↪width);
  }

  std::vector<T> getColData(xy_t x) const {
    std::vector<T> temp(view_height);
    for(xy_t y=0;y<view_height;y++)
      temp[y]=_data[xyToI(x,y)];
    return temp;
  }

  void clear(){
    _data = ManagedVector<T>();
  }

  template<class U>
  void templateCopy(const Array2D<U> &other){
```

(continues on next page)

(continued from previous page)

```
    geotransform       = other.geotransform;
    projection         = other.projection;
    basename           = other.basename;
    metadata     = other.metadata;
  }


  #ifdef USEGDAL
  void saveGDAL(const std::string &filename, const std::string &metadata_str="",␣
↪xy_t xoffset=0, xy_t yoffset=0, bool compress=false){
    char **papszOptions = NULL;
    if(compress){
      papszOptions = CSLSetNameValue( papszOptions, "COMPRESS", "DEFLATE" );
      papszOptions = CSLSetNameValue( papszOptions, "ZLEVEL",   "6" );
    }

    GDALDriver *poDriver = GetGDALDriverManager()->GetDriverByName("GTiff");
    if(poDriver==NULL)
      throw std::runtime_error("Could not open GDAL driver!");
    GDALDataset *fout    = poDriver->Create(filename.c_str(), width(), height(),␣
↪1, myGDALType(), papszOptions);
    if(fout==NULL)
      throw std::runtime_error("Could not open file '"+filename+"' for GDAL save!
↪");

    GDALRasterBand *oband = fout->GetRasterBand(1);
    oband->SetNoDataValue(no_data);

    //This could be used to copy metadata
    //poDstDS->SetMetadata( poSrcDS->GetMetadata() );

    //TIFFTAG_SOFTWARE
    //TIFFTAG_ARTIST
    {
      std::time_t the_time = std::time(nullptr);
      char time_str[64];
      std::strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S UTC",␣
↪std::gmtime(&the_time));
      fout->SetMetadataItem("TIFFTAG_DATETIME",    time_str);
      fout->SetMetadataItem("TIFFTAG_SOFTWARE",    program_identifier.c_str());

      //TODO: `metadata_str` may need removing
      metadata["PROCESSING_HISTORY"] += "\n" + std::string(time_str) + " | " +␣
↪program_identifier + " | ";
      if(!metadata.empty())
        metadata["PROCESSING_HISTORY"] += metadata_str;
      else
        metadata["PROCESSING_HISTORY"] += "Unspecified Operation";
    }

    for(const auto &kv: metadata)
      fout->SetMetadataItem(kv.first.c_str(), kv.second.c_str());

    //The geotransform maps each grid cell to a point in an affine-transformed
    //projection of the actual terrain. The geostransform is specified as follows:
    //    Xgeo = GT(0) + Xpixel*GT(1) + Yline*GT(2)
    //    Ygeo = GT(3) + Xpixel*GT(4) + Yline*GT(5)
```

(continues on next page)

(continued from previous page)

```
    //In case of north up images, the GT(2) and GT(4) coefficients are zero, and
    //the GT(1) is pixel width, and GT(5) is pixel height. The (GT(0),GT(3))
    //position is the top left corner of the top left pixel of the raster.

    if(!geotransform.empty()){
      auto out_geotransform = geotransform;

      if(out_geotransform.size()!=6)
        throw std::runtime_error("Geotransform of output is not the right size.␣
↪Found "+std::to_string(out_geotransform.size())+" expected 6.");

      //We shift the top-left pixel of hte image eastward to the appropriate
      //coordinate
      out_geotransform[0] += xoffset*geotransform[1];

      //We shift the top-left pixel of the image southward to the appropriate
      //coordinate
      out_geotransform[3] += yoffset*geotransform[5];

      fout->SetGeoTransform(out_geotransform.data());
    }

    if(!projection.empty())
      fout->SetProjection(projection.c_str());

    #ifdef DEBUG
      RDLOG_DEBUG<<"Filename: "<<std::setw(20)<<filename<<" Xoffset: "<
↪<std::setw(6)<<xoffset<<" Yoffset: "<<std::setw(6)<<yoffset<<" Geotrans0: "<
↪<std::setw(10)<<std::setprecision(10)<<std::fixed<<geotransform[0]<<"␣
↪Geotrans3: "<<std::setw(10)<<std::setprecision(10)<<std::fixed<<geotransform[3];
    #endif

    auto temp = oband->RasterIO(GF_Write, 0, 0, view_width, view_height, _data.
↪data(), view_width, view_height, myGDALType(), 0, 0);
    if(temp!=CE_None)
      throw std::runtime_error("Error writing file with saveGDAL()!");

    GDALClose(fout);
  }
  #endif



  void printStamp(int size, std::string msg="") const {
    #ifdef SHOW_STAMPS
      const xy_t sx = width()/2;
      const xy_t sy = height()/2;

      if(msg.size()>0)
        std::cout<<msg<<std::endl;
      std::cout<<"Stamp for basename='"<<basename
               <<"', filename='"<<filename
               #ifdef USEGDAL
               <<"', dtype="<<GDALGetDataTypeName(myGDALType())
               #endif
               <<" at "<<sx<<","<<sy<<"\n";
```

(continues on next page)

(continued from previous page)

```
      const xy_t sxmax = std::min(width(), sx+size);
      const xy_t symax = std::min(height(),sy+size);

      for(xy_t y=sy;y<symax;y++){
        for(xy_t x=sx;x<sxmax;x++)
          std::cout<<std::setw(5)<<std::setprecision(3)<<(int)_data[xyToI(x,y)]<<
↪" ";
        std::cout<<"\n";
      }
    #endif
  }


  void printBlock(const int radius, const xy_t x0, const xy_t y0, bool␣
↪color=false, const std::string msg="", const int fwidth=5, const int␣
↪precision=0) const {
    if(msg.size()!=0)
      std::cout<<msg<<std::endl;

    xy_t xmin = std::max(0,x0-radius);
    xy_t ymin = std::max(0,y0-radius);
    xy_t xmax = std::min(width(),x0+radius);
    xy_t ymax = std::min(height(),y0+radius);

    for(xy_t y=ymin;y<ymax;y++){
      for(xy_t x=xmin;x<xmax;x++){
        if(color && x==x0 && y==y0)
          std::cout<<"\033[92m";
        std::cout<<std::setw(fwidth)<<std::setprecision(precision)<<std::fixed;
        if(std::is_same<T, flowdir_t>::value){
          std::cout<<(int)_data[xyToI(x,y)]<<" ";
        } else {
          std::cout<<_data[xyToI(x,y)]<<" ";
        }
        if(color && x==x0 && y==y0)
          std::cout<<"\033[39m";
      }
      std::cout<<std::endl;
    }
  }

  void printAll(const std::string msg="", const int fwidth=5, const int␣
↪precision=0) const {
    if(!msg.empty())
      std::cerr<<msg<<std::endl;

    for(xy_t y=0;y<height();y++){
      for(xy_t x=0;x<width();x++){
        std::cout<<std::setw(fwidth)<<std::setprecision(precision)<<std::fixed;
        if(std::is_same<T, flowdir_t>::value){
          std::cout<<(int)_data[xyToI(x,y)]<<" ";
        } else {
          std::cout<<_data[xyToI(x,y)]<<" ";
        }
      }
      std::cout<<std::endl;
    }
```

(continues on next page)

(continued from previous page)

```
  }

  void printAllFlows(const std::string msg="", const int fwidth=5) const {
    if(!msg.empty())
      std::cerr<<msg<<std::endl;

    for(xy_t y=0;y<height();y++){
      for(xy_t x=0;x<width();x++){
        const auto dir = _data[xyToI(x,y)];
        std::cout<<std::setw(fwidth);
        if(0<=dir && dir<=8){
          std::cout<<(int)dir;
        } else {
          std::cout<<"?";
        }
      }
      std::cout<<std::endl;
    }
  }


  void printBlockIndices(const int radius, const xy_t x0, const xy_t y0, bool␣
↪color=false, const std::string msg="", const int fwidth=5) const {
    if(msg.size()!=0)
      std::cout<<msg<<std::endl;

    xy_t xmin = std::max(0,x0-radius);
    xy_t ymin = std::max(0,y0-radius);
    xy_t xmax = std::min(width(),x0+radius);
    xy_t ymax = std::min(height(),y0+radius);

    for(xy_t y=ymin;y<ymax;y++){
      for(xy_t x=xmin;x<xmax;x++){
        if(color && x==x0 && y==y0)
          std::cout<<"\033[92m";
        std::cout<<std::setw(fwidth)<<xyToI(x,y)<<" ";
        if(color && x==x0 && y==y0)
          std::cout<<"\033[39m";
      }
      std::cout<<std::endl;
    }
  }

  void printAllIndices(const std::string msg="") const {
    if(!msg.empty())
      std::cout<<msg<<std::endl;

    for(xy_t y=0;y<height();y++){
      for(xy_t x=0;x<width();x++)
        std::cout<<std::setw(5)<<xyToI(x,y)<<" ";
      std::cout<<std::endl;
    }
  }

  double getCellArea() const {
    assert(geotransform.size()>0);
    return std::abs(geotransform[1]*geotransform[5]);
```

(continues on next page)

(continued from previous page)

```
  }

  double getCellLengthX() const {
    assert(geotransform.size()>0);
    return std::abs(geotransform[1]);
  }

  double getCellLengthY() const {
    assert(geotransform.size()>0);
    return std::abs(geotransform[5]);
  }

  void scale(const double x) {
    for(i_t i=0;i<size();i++)
      if(_data[i]!=no_data)
        _data[i] *= x;
  }

  //TODO
  inline bool owned() const {
    return _data.owned();
  }
};

}

#endif
```

Parameters

- `values`: The raster

CHAPTER 18

RichDEM Python Reference

richdem.**BreachDepressions**(*dem*, *in_place=False*, *topology='D8'*)
Breaches all depressions in a DEM.

Parameters

- **dem** (rdarray) – An elevation model
- **in_place** (*bool*) – If True, the DEM is modified in place and there is no return; otherwise, a new, altered DEM is returned.
- **topology** (*string*) – A topology indicator

Returns DEM without depressions.

richdem.**FillDepressions**(*dem*, *epsilon=False*, *in_place=False*, *topology='D8'*)
Fills all depressions in a DEM.

Parameters

- **dem** (rdarray) – An elevation model
- **epsilon** (*float*) – If True, an epsilon gradient is imposed to all flat regions. This ensures that there is always a local gradient.
- **in_place** (*bool*) – If True, the DEM is modified in place and there is no return; otherwise, a new, altered DEM is returned.
- **topology** (*string*) – A topology indicator

Returns DEM without depressions.

richdem.**FlowAccumFromProps**(*props*, *weights=None*, *in_place=False*)
Calculates flow accumulation from flow proportions.

Parameters

- **props** (rdarray) – An elevation model
- **weights** (rdarray) – Flow accumulation weights to use. This is the amount of flow generated by each cell. If this is not provided, each cell will generate 1 unit of flow.

- **in_place** (*bool*) – If True, then `weights` is modified in place. An accumulation matrix is always returned, but it will just be a view of the modified data if `in_place` is True.

Returns A flow accumulation array. If `weights` was provided and `in_place` was True, then this matrix is a view of the modified data.

`richdem.`**`FlowAccumulation`**(*dem*, *method=None*, *exponent=None*, *weights=None*, *in_place=False*)

Calculates flow accumulation. A variety of methods are available.

Parameters

- **dem** (`rdarray`) – An elevation model
- **method** (*str*) – Flow accumulation method to use. (See below.)
- **exponent** (*float*) – Some methods require an exponent; refer to the relevant publications for details.
- **weights** (`rdarray`) – Flow accumulation weights to use. This is the amount of flow generated by each cell. If this is not provided, each cell will generate 1 unit of flow.
- **in_place** (*bool*) – If True, then `weights` is modified in place. An accumulation matrix is always returned, but it will just be a view of the modified data if `in_place` is True.

Method	Note	Reference
Tarboton	Alias for Dinf.	Taroboton (1997) doi: 10.1029/96WR03137
Dinf	Alias for Tarboton.	Taroboton (1997) doi: 10.1029/96WR03137
Quinn	Holmgren with exponent=1.	Quinn et al. (1991) doi: 10.1002/hyp.3360050106
Holmgren(E)	Generalization of Quinn.	Holmgren (1994) doi: 10.1002/hyp.3360080405
Freeman(E)	TODO	Freeman (1991) doi: 10.1016/0098-3004(91)90048-I
FairfieldLeymarieD8	Alias for Rho8.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
FairfieldLeymarieD4	Alias for Rho4.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
Rho8	Alias for FairfieldLeymarieD8.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
Rho4	Alias for FairfieldLeymarieD4.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
OCallaghanD8	Alias for D8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
OCallaghanD4	Alias for D8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
D8	Alias for OCallaghanD8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
D4	Alias for OCallaghanD4.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0

Methods marked (E) require the exponent argument.

Returns A flow accumulation according to the desired method. If `weights` was provided and `in_place` was True, then this matrix is a view of the modified data.

`richdem.`**`FlowProportions`**(*dem*, *method=None*, *exponent=None*)

Calculates flow proportions. A variety of methods are available.

Parameters

- **dem** (`rdarray`) – An elevation model

- **method** (*str*) – Flow accumulation method to use. (See below.)
- **exponent** (*float*) – Some methods require an exponent; refer to the relevant publications for details.

Method	Note	Reference
Tarboton	Alias for Dinf.	Taroboton (1997) doi: 10.1029/96WR03137
Dinf	Alias for Tarboton.	Taroboton (1997) doi: 10.1029/96WR03137
Quinn	Holmgren with exponent=1.	Quinn et al. (1991) doi: 10.1002/hyp.3360050106
Holmgren(E)	Generalization of Quinn.	Holmgren (1994) doi: 10.1002/hyp.3360080405
Freeman(E)	TODO	Freeman (1991) doi: 10.1016/0098-3004(91)90048-I
FairfieldLeymarieD8	Alias for Rho8.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
FairfieldLeymarieD4	Alias for Rho4.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
Rho8	Alias for FairfieldLeymarieD8.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
Rho4	Alias for FairfieldLeymarieD4.	Fairfield and Leymarie (1991) doi: 10.1029/90WR02658
OCallaghanD8	Alias for D8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
OCallaghanD4	Alias for D8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
D8	Alias for OCallaghanD8.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0
D4	Alias for OCallaghanD4.	O'Callaghan and Mark (1984) doi: 10.1016/S0734-189X(84)80011-0

Methods marked (E) require the exponent argument.

Returns A flow proportion according to the desired method.

richdem.**LoadGDAL**(*filename*, *no_data=None*)

Read a GDAL file.

Opens any file GDAL can read, selects the first raster band, and loads it and its metadata into a RichDEM array of the appropriate data type.

If you need to do something more complicated, look at the source of this function.

Parameters

- **filename** (*str*) – Name of the raster file to open
- **no_data** (*float*) – Optionally, set the no_data value to this.

Returns A RichDEM array

richdem.**ResolveFlats**(*dem*, *in_place=False*)

Attempts to resolve flats by imposing a local gradient

Parameters

- **dem** (rdarray) – An elevation model
- **in_place** (*bool*) – If True, the DEM is modified in place and there is no return; otherwise, a new, altered DEM is returned.

Returns DEM modified such that all flats drain.

richdem.**SaveGDAL**(*filename*, *rda*)
Save a GDAL file.

Saves a RichDEM array to a data file in GeoTIFF format.

If you need to do something more complicated, look at the source of this function.

Parameters

- **filename** (*str*) – Name of the raster file to be created
- **rda** (rdarray) – Data to save.

Returns No Return

richdem.**TerrainAttribute**(*dem*, *attrib*, *zscale=1.0*)
Calculates terrain attributes. A variety of methods are available.

Parameters

- **dem** (rdarray) – An elevation model
- **attrib** (*str*) – Terrain attribute to calculate. (See below.)
- **zscale** (*float*) – How much to scale the z-axis by prior to calculation

Method	Reference
slope_riserun	Horn (1981) doi: 10.1109/PROC.1981.11918
slope_percentage	Horn (1981) doi: 10.1109/PROC.1981.11918
slope_degrees	Horn (1981) doi: 10.1109/PROC.1981.11918
slope_radians	Horn (1981) doi: 10.1109/PROC.1981.11918
aspect	Horn (1981) doi: 10.1109/PROC.1981.11918
curvature	Zevenbergen and Thorne (1987) doi: 10.1002/esp.3290120107
planform_curvature	Zevenbergen and Thorne (1987) doi: 10.1002/esp.3290120107
profile_curvature	Zevenbergen and Thorne (1987) doi: 10.1002/esp.3290120107

Returns A raster of the indicated terrain attributes.

class richdem.**rd3array**

class richdem.**rdarray**

CHAPTER 19

Testing Methodology

Simple algorithms are shown to be correct through visual inspection and comparison against hand-crafted examples. Correctness for more complex algorithms is often "boot-strapped" by comparing the results of simple algorithms versus the complex algorithms on a large number of randomly-generated datasets.

This is a work in progress. TODO

CHAPTER 20

Correctness

Correctness is established via a number of methodologies building from code inspection in the simplest cases to output comparison between simple and complex implementations.

Correctness is noted in source code comments under `[anonymous]` sections. These are, in turn, printed to the Doxygen documentation output.

A master list of how correctness was established for each algorithm is available at tests/README.md.

CHAPTER 21

Specific Algorithms

Many of the algorithms used in RichDEM are documented in journal or conference publications. In the case of older algorithms by other authors, it is often possible to find the paper in the literature. Some of the newer algorithms I developed have not yet had a chance to be widely utilized. These algorithms are listed below.

Additionally, each publication has its own GitHub repository featuring easily-compiled, minimum working examples of the algorithms, along with examples and test data sets.

These are available as follows:

- Flat-resolution algorithm. Link
- Depression-filling algorithm. Link
- Large dataset depression-filling algorithm. Link
- Large dataset flow accumulation algorithm. Link

CHAPTER 22

Publications

The algorithms used in RichDEM have been published in the following articles. Every algorithm/program will provide its relevant citation information when run.

- Barnes, R., 2017. Parallel non-divergent flow accumulation for trillion cell digital elevation models on desktops or clusters. Environmental Modelling & Software 92, 202–212. doi: 10.1016/j.envsoft.2017.02.022
- Barnes, R., 2016. Parallel priority-flood depression filling for trillion cell digital elevation models on desktops or clusters. Computers & Geosciences 96, 56–68. doi: 10.1016/j.cageo.2016.07.001
- Barnes, R., Lehman, C., Mulla, D., 2014a. An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. Computers & Geosciences 62, 128–135. doi: 10.1016/j.cageo.2013.01.009
- Barnes, R., Lehman, C., Mulla, D., 2014b. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. Computers & Geosciences 62, 117–127. doi: 10.1016/j.cageo.2013.04.024
- Barnes, Lehman, Mulla. 2011. "Distributed Parallel D8 Up-Slope Area Calculation in Digital Elevation Models". Intn'l Conf. on Parallel & Distributed Processing Techniques & Applications. Link
- Horn, B.K.P., 1981. Hill shading and the reflectance map. Proceedings of the IEEE 69, 14–47. doi: 10.1109/PROC.1981.11918
- Mulla et al. including Barnes. 2012. "Strategic Planning for Minnesota's Natural and Artificial Watersheds". Report to the Minnesota LCCMR.
- Tarboton, D.G. 1997. A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water Resources Research. Vol. 33. pp 309-319.
- Zevenbergen, L.W., Thorne, C.R., 1987. Quantitative analysis of land surface topography. Earth surface processes and landforms 12, 47–56.
- Zhou, G., Sun, Z., Fu, S., 2016. An efficient variant of the Priority-Flood algorithm for filling depressions in raster digital elevation models. Computers & Geosciences 90, Part A, 87 – 96. doi: 10.1016/j.cageo.2016.02.021

CHAPTER 23

Sponsors

RichDEM has been developed and tested using computational resources provided by the Minnesota Supercomputing Institute (MSI) and the U.S. National Science Foundation's XSEDE.

Funding for the development of RichDEM has been provided by the Legislative-Citizen Commission on Minnesota Resources (LCCMR), the U.S. National Science Foundation Graduate Research Fellowship, and the U.S. Department of Energy Computational Science Graduate Fellowship.

CHAPTER 24

Feedback

If you see something, say something.

Users are encouraged to report any issues experienced with the code via Github's issue tracker. Feedback is also accepted via email (rbarnes@umn.edu), though this is highly discouraged as it does not provide a resource for others.

CHAPTER 25

Release Steps

25.1 Updating Documentation

1. Enter the `docs` directory.
2. Run `make html`
3. Enter the `docs/richdem-docs` directory. Commit changes.
4. Go back to the `docs` directory. Commit changes.
5. Push.
6. This will trigger a ReadTheDocs build and a Travis build.

25.2 Updating Wheels

Acquire the manylinux build image with:

    docker pull quay.io/pypa/manylinux1_x86_64

Start the docker manylinux build image with:

    docker run -i -t e0e55980c200 /bin/bash

You can list running images with:

    docker ps

From within the container run the `travis/build-wheels.sh` script to generate documentation.

From without the container run, e.g.:

    docker cp mystifying_darwin:/io/wheelhouse/richdem-0.0.9-cp27-cp27m-linux_x86_64.whl ./

Upload the wheel to PyPI with

    twine upload richdem-0.0.9-cp*

25.3 Updating Source Dist

To generate a source distribution, run

```
python3 setup.py sdist
```

Upload it to PyPI with, e.g.

```
twine upload dist/richdem-0.0.10.tar.gz
```

Python Module Index

r

Index

Symbols

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

X

Y