Per the client’s request, the following report documents some of the major milestones in the development of an algorithm to estimate rainfall-adjusted waterflow rates across a given topography. The analysis begins simply, creating a flow network based on randomly generated data, building to higher complexity through the sections of the document. Rainfall rates are made variable, and lake and watershed formation modeled. In the final section real data resampled and analyzed. Along the way measures of flow and a series of plots are generated.

The algorithm has been developed in Python 3.6.7. Comments and docstrings have been added throughout, defining classes, functions, and how they fit together, for both newly added code and for that which was included when the project was handed over. Where possible, code added or modified has been brought into line with the standards of PEP-8. As requested, pseudocode and code have been provided for some tasks in appendices, helping illustrate the algorithmic logic implemented. The entire body of code has also been made available in a github repository, and has been uploaded to Learn, along with a README file which gives a roadmap for the code and how it can be tested.

Command line functionality has been added to facilitate interaction with the code. This can be accessed with the xxx.py module. Please see README.md for further details.

**Task 1.**

After adjusting the import calls where needed to ensure all required modules are available, the Task 1 code produces the plot shown in Figure 1, entitled “Network structure – before lakes.” The base of this plot is a randomly generated topographic raster, produced using *createRanRasterSlope()* from RasterHandler.py. For each ‘node’ (cell) we calculate the ‘downnode’ into which it flows, and correspondingly any ‘upnodes’ which flow into it. The streams formed are plotted by starting at each ‘pit’ (a node lower than all its neighbors and thus without a downnode) and recursing upstream until reaching a node without any upnodes.

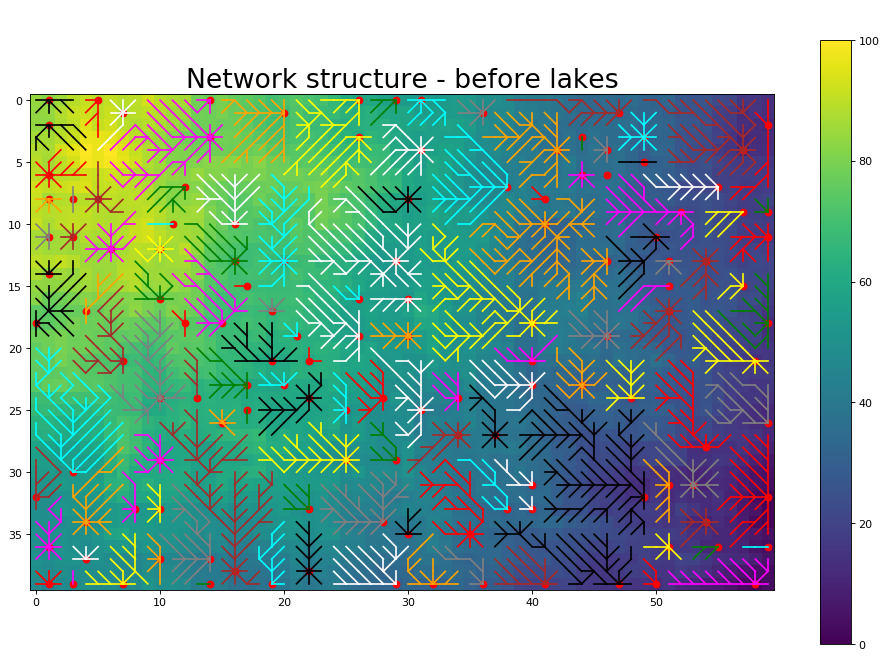


Figure 1: (Task 1.) A plot of flow direction, with networks differentiated by color, overlaying a randomly generated topographic raster. Red points indicate ‘pits.’

**Task 2.**

To calculate the flow rate with constant rainfall, we add an attribute *\_rainfall* to the FlowNodeclass, which is initialized at 1 millimeter for all nodes. The method *getFlow()* is also added to FlowNode, in alignment with the the class’s other functions and the requirement of FlowExtractor. This function initially sets a node’s flow at it its own level of rainfall, and then adds the flow of that node’s upnodes, recursing upstream until reaching a node with no upnodes. Accordingly, under constant rainfall a node with no upnodes will have a flow of 1, while a node downstream will have larger flow.

Figure 2. is produced using the random topography generated during Task 1. *getFlow()* is called on each node using FlowRaster’s *extractValues()* method (which in turn makes use of the *FlowExtractor* class). The flow values produced for each cell are stored in a 2D array (not technically a raster) of the same dimensions as the topographic raster. Yellow represents higher flow, while deep blue indicates lower flow. The most recently generated topography rendered a highest node with flow of 144 millimeters annually.

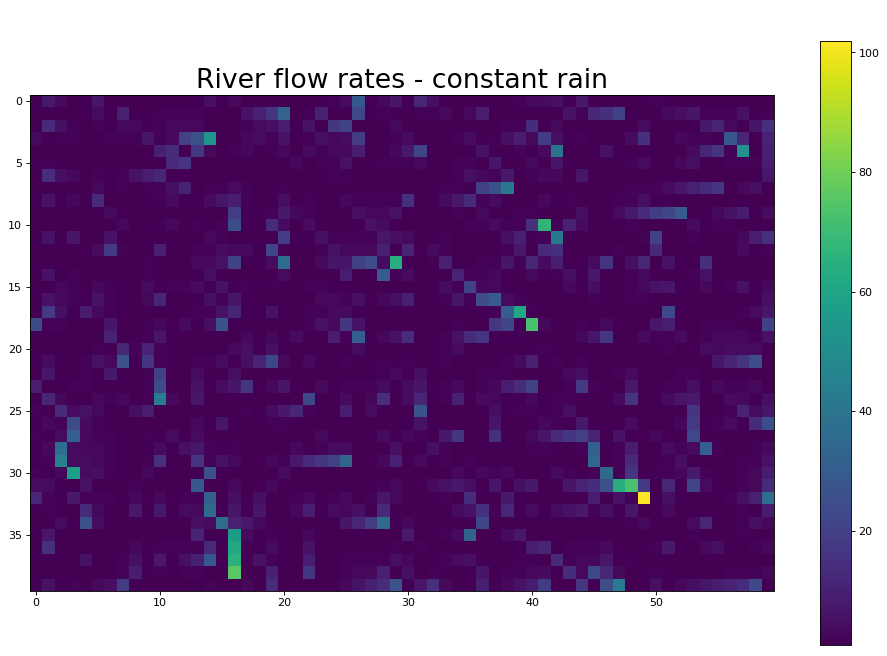


Figure 2. (Task 2.) River flow rates with constant rainfall of 1 millimeter.

**Task 3.**

The requirements of Task 2. could have been accomplished without the addition of *\_rainfall* (by directly initializing the flow for each node at 1 millimeter) but it was added with an eye to the requirements of Task 3. This attribute gives us a place to store the variable rainfall associated with each node.

The bulk of the work of accounting for variable rainfall is done with the newly added *addRainfall()* method*.* This method must be passed a rainfall raster of the same dimensions as its FlowRaster. It then iterates through the input rain dataset and overwrites rainfall variable for each associated cell. As a result, when *getFlow()* is called, the upstream recursion sums variable rainfall figures of each upnode, rather than the flat rate of 1 used in Task 2.

Figure 3. shows the flow volume under (spatially) variable rainfall. The pattern that emerges is similar to that of Figure 2., as the same topography is used, though flows have clearly shifted toward areas with heavier rain. In the case shown, each node receives between 1 and 4 meters, simulated using *ranRasterSlope()*, the same function used to generate the topography, albeit with different parameter values.

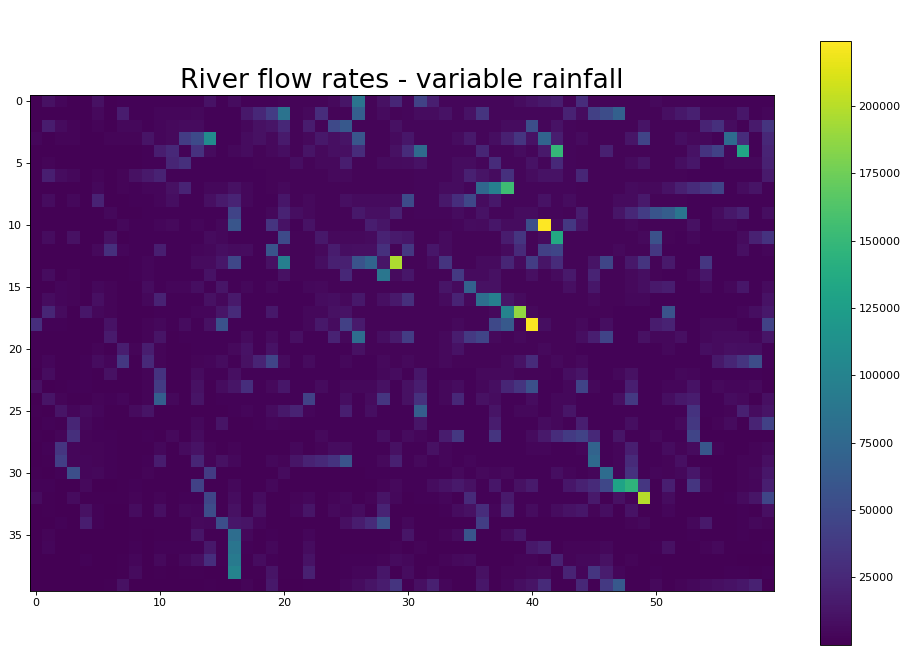


Figure 3. (Task 3.) Flow rates in millimeters, based using spatially variable rainfall.

**Task 4.**

As pointed out by the client, the algorithms through Task 3 are quite unrealistic, in that waterflow simply disappears when it reaches a ‘pit.’ To improve the algorithm, we add code that allows pits to ‘fill up,’ forming lakes of varying depths. Lakes are taken to be any pit that is not on the edge of the DEM (and thus does not flow into the ocean).

Under our improved model, the water at each pit rises and overflows into nearby nodes. When water begins to overflow into a node of another catchment, we consider the catchments joined. The bulk of this takes place in the *joinCatchments()* method added to FlowRaster. We first search a pit’s immediate neighbors. If the lowest is in another catchment, that is the overflow point; if the lowest is in the same catchment, we add the neighbors of the first low point and search again to find the new lowest point (excluding the first). The water continues rising, and we continue searching recursively, until overflow occurs into a different catchment. This process is repeated for all pits by the *calculateLakes()* FlowRaster method.

Lake depth is determined by the difference in elevation between the original pit under and the elevation of the spillover node. We also consider to be lakes same-catchment nodes into which a pit has overflowed. Depths are stored in the *\_lakedepth* attribute, and retrieved by the helper method *getLakeDepth()*, which is in turn called in the example by the helper class LakeDepthExtractor*.*

Pseudocode for the logic of how this is calculated, and the Python code that embodies it, are included in Appendix 1 of this report.

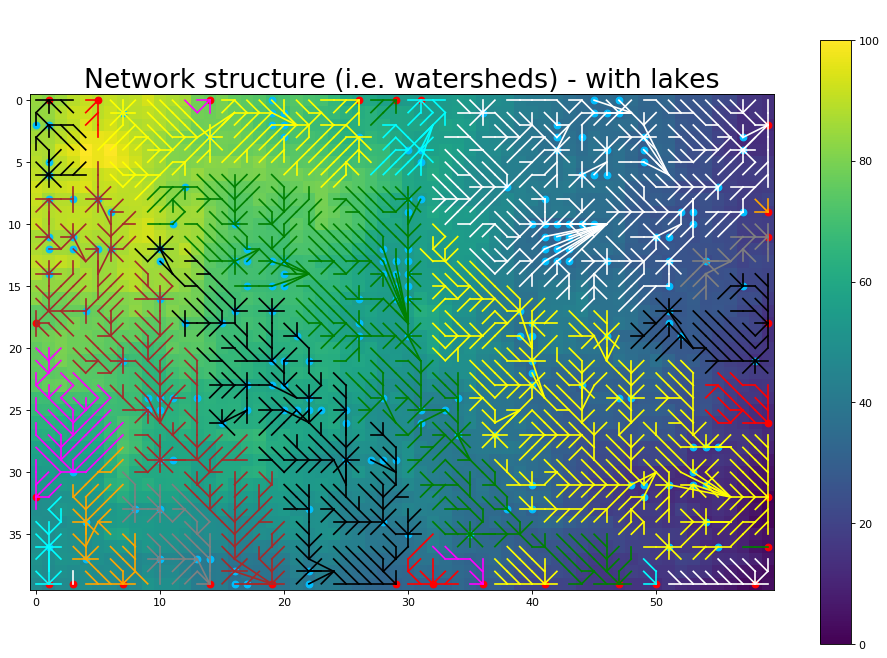


Figure 4. (Task 4.) joined network structure, including all lake nodes formed.

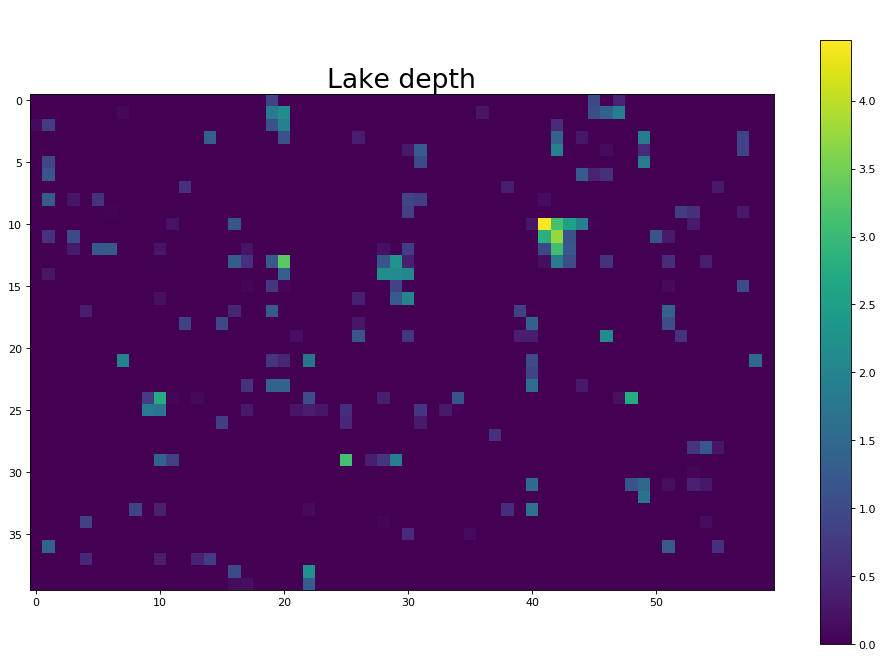


Figure 5. (Task 4.) Lake depths (in meters) resulting from rising waters.

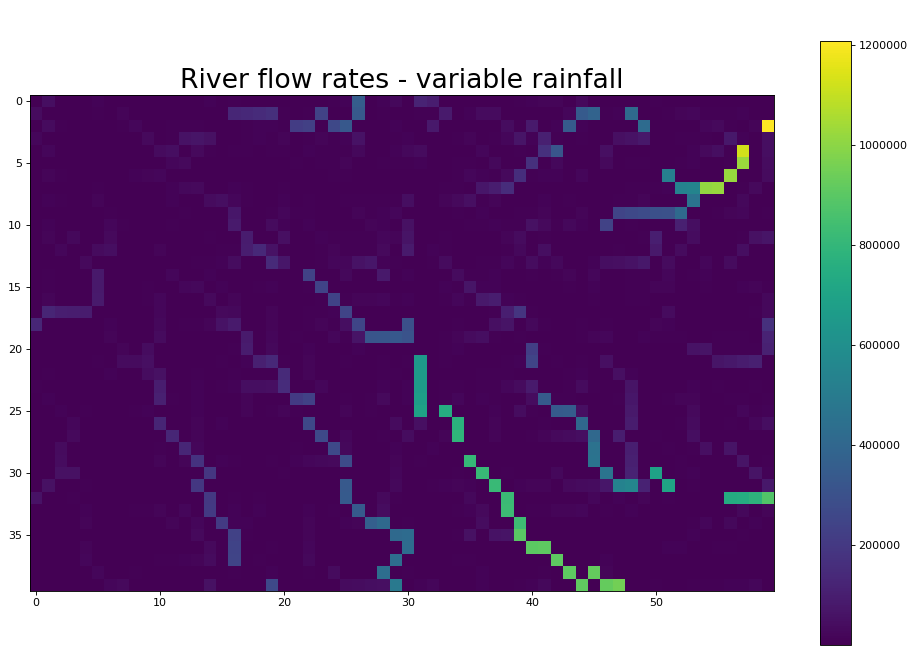


Figure 6. (Task 4.) River flow rates (in millimeters).

**Task 5.**

As given, DEM.txt has a resolution too high to be used at sufficient speed for the client’s needs. As a result, we add the capability to resample down to a lower resolution. This is implemented through completion of the *createWithIncreasedCellSize()* method of the Raster class, which one calls specifying a factor by which to resample. As initially packaged, this method gives an error if called with any value other than 1. For this task we want to resample by a factor of 10 so the DEM data matches that of Rainfall.txt.

To flesh out the method, we create a blank array of the dimensions desired for the resampled dataset. We then take the average of the corresponding cells in the original raster. These new values are returned in a new raster, which can be analyzed using all the other tools developed in this project. We take care to account for the x and y coordinates which are now different than in the underlying data initially given. Commented code of the updated function is included in Appendix 2.

Using the resampled data we calculate that the highest flow is found in cell [44, 82] through which 3,042 meters flow based on the rate given in Rainfall.txt. The code used to produce these numbers and the resampled elevation have been added to CourseWork1\_argparse.py.

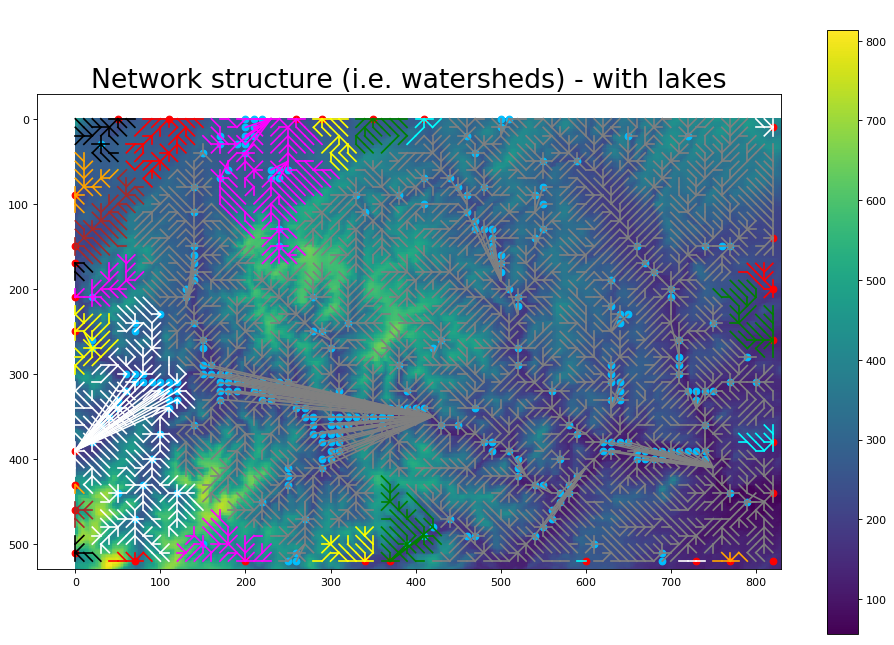
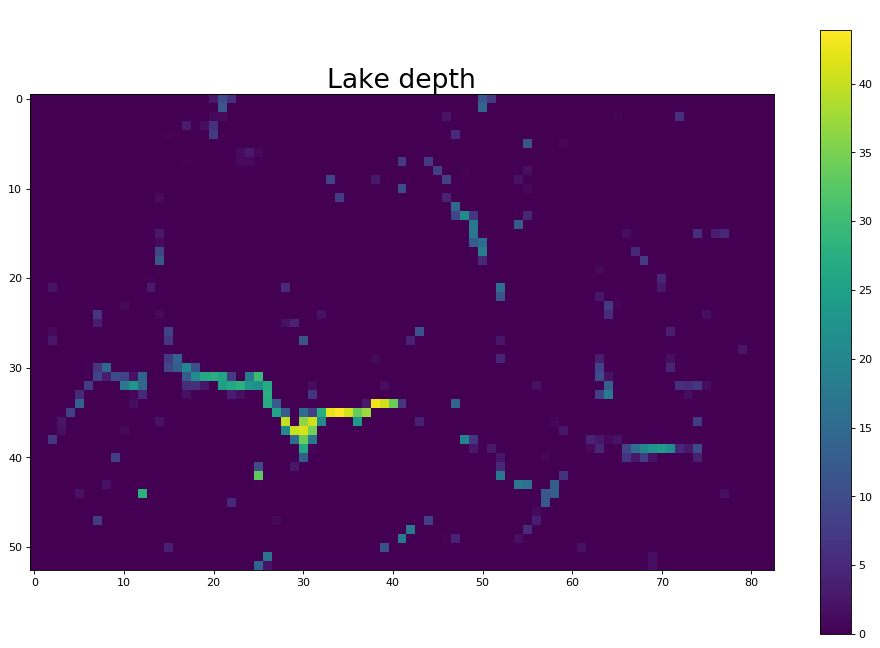


Figure 7. (Task 5.) Catchment structure calculated using a resampled DEM.txt.

Figure 8. (Task 5.) Lake depths (in meters) resulting from a resampled DEM.txt.

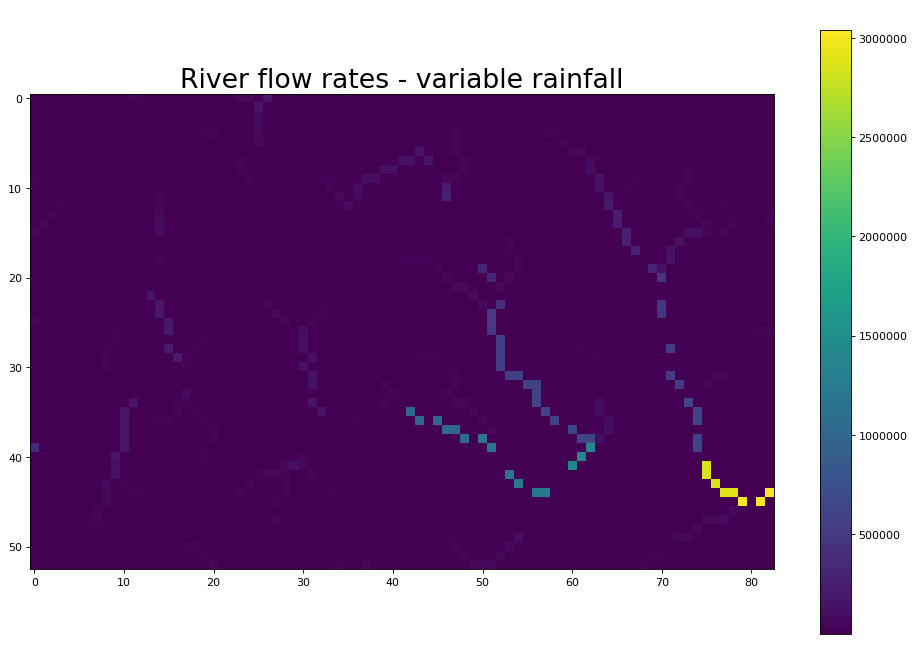


Figure 9. (Task 5.) River flow rates (in millimeters) based on resampled DEM.txt.

**General discussion**

To provide the client with the most realistic understanding of how rainfall flow rates will behave under different climate change scenarios, there are several ways our algorithms can be improved. Some of these are processing oriented, and some involve inclusion of additional data.

As implemented, as catchments are joined due to rising water, the setting of upnodes and downnodes to trace flow back to the original pit is highly simplified: each point that has become part of a lake due to rising waters (prior to finding a spillover point) is now shown draining directly into the spillover point. This could be made a more realistic by instituting a recursion within *joinCatchments()* to trace the path back from the spillover point to the pit. In pseudocode:

1. Draw a bounding box around all nodes excluded from consideration after ‘spillovernode’ has been found.
2. Find tracenode1, the lowest neighbor of spillovernode that is in the bounding box from 1.
3. Set spillovernode as the downnode of tracenode1.
4. If tracenode1 is the original pit:
   1. Traceback complete.
5. Else, recurse starting at 2:
   1. Set spillovernode = tracenode1.
   2. Find tracenode2.
   3. Set spillovernode as the downnode of tracenode2.
   4. If tracenode2 is the original pit:
      1. Traceback complete.
   5. Else…

Another improvement that could be made would more realistically account for the borders of the elevation considered. There are not a lot of perfectly rectangular islands, so to be used on a differently shaped area—for example, one that actually borders the sea—the tools would need to be extended. We currently consider all pits not on the border to be lakes, and all those on the rectangular border to be exit points to the sea. This does not realistically take into account that in addition to flowing out through the borders, water could also flow onto it. This extension could be achieved by assigning each FlowNode an *\_inflow* attribute which would also need to be accounted for in the *getFlow()* recursion.

These boundary issues could in part be addressed by adding the ability to stitch DEMs together to model flow for variable scales of the landscape. This may require the use of reprojection capabilities, and would likely lead to additional resampling to keep computation tractable.

Even with extensions of the algorithm, the model will remain severely limited by the small number of data layers incorporated. Several additional data layers would be essential to proper modeling of hydrographic behavior in a given area. Which datasets precisely are needed will depend on the client’s intended use of the algorithm, but some we would do well to consider include:

* Temporal variability. The algorithm currently incorporates spatial variability, but not temporally variability. To continue building the code to account for climate change scenarios as required by the client, time will need to be added to the rain data. While finer grained the temporal data would be required to make some of this possible, resampling (along time) may again be required to process frequent readings.
* Soil types. Our model currently assumes no absorption or drainage take place, but that the ground is fully impervious. To improve this we could add functionality to each FlowNode giving it some capacity for absorption which would be filled up after a certain amount of rainfall, and would slowly drain off. This behavior would be most interesting if implemented alongside temporal variability.
* Erosion. Though some areas of the planet truly do have hundreds of tiny lakes dotting the countryside, often rainfall interacts with soil and rock through time to erode and form new channels. This could be incorporated as a function of soil type and amount of flow passing through a node. This would significantly increase the model’s complexity, as the elevations themselves would begin to change.
* Plant life. Alongside inflow rate and soil type, the species and health of the plants covering an area will determine the amount of erosion that takes place.

Incorporation of each of these additional data layers would come with its own challenges. The addition of each adds another dimension to the analysis. If not approached strategically, this could lead to rapid increase in the complexity of the program. The fact that our initial modules have been developed in an object-oriented way sets us up to make such extensions far easier. The complexity added by layering data will demonstrate the ways in which object-oriented development can increase efficiency, reducing errors and conflicts, and minimizing redundant code, making our algorithm more accurate and easily maintainable.

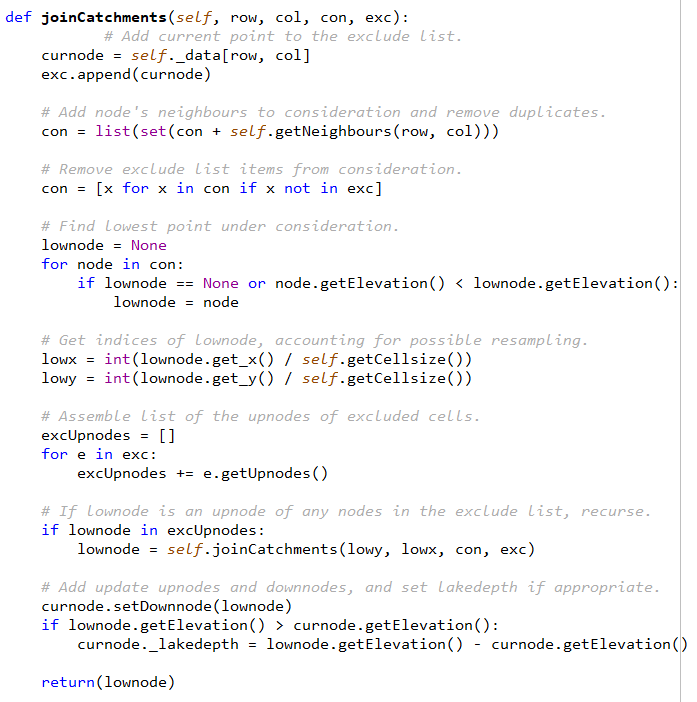
**Appendix 1. Task 4 Pseudocode and code.**

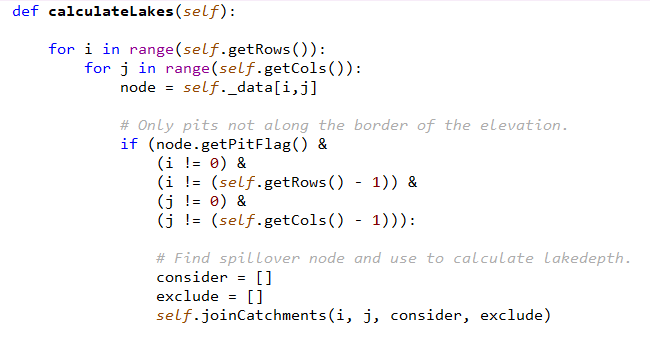
*Pseudocode*

For each pit not flowing into the sea (not on border of elevation):

1. Initiate an empty list for nodes currently under consideration as possible spillover points.
2. Initiate an empty list for nodes already considered, and thus to be excluded from future consideration.
3. Add the ‘currentnode’ (the pit on first iteration) to the exclude list, as it is part of its own catchment.
4. Add all of currentnode’s neighbors to the consideration list.
5. Remove from the consideration list those from the exclude list (if any overlap).
6. Get a list of upnodes of all nodes on the exclude list (known to be part of the same catchment).
7. Find ‘lownode1,’ the lowest of the nodes in the consideration list.
8. If lownode1 is not one of the in the exclude list’s upnodes:
   1. Spillover point found.
   2. Lake depth equals the lownode1’s elevation minus the pit’s elevation.
   3. Set lownode1 as the pit’s downnode.
9. Else, continue searching recursively, starting with step 3.
   1. currentnode = lownode1.
   2. Add currentnode to exclude list.
   3. Add currentnode’s neighbors not already on the consideration list to the list.
   4. Remove exclude list items from consideration list.
   5. Get upnodes of exclude list members.
   6. Find lownode2.
   7. If lownode2 not among exclude list’s upnodes:
      1. Spillover point found.
      2. Lake depth equals the lownode2’s elevation minus the pit’s elevation.
      3. Set downnodes to trace flow from lownode2 back to original pit.
   8. Else, continue searching recursively, starting with step 3 and lownode2.
      1. …

*Code*





**Appendix 2. Commented code for Task 5.**

