Flood Modeller

Aims

This software is designed to take in DEM elevation data and model the build-up of water due to rainfall over the whole area, with a view to identify areas that are at risk from flooding.

The basic concept for how to achieve this is that if you can compare one point on the DEM to the neighbouring 8 cells (above, below, left, right and the 4 diagonals), you can work out the slope, and therefore the direction(s) that rain would flow, in effect creating the catchments. If the rain could be then passed to the downslope neighbour, you could see pools of high accumulations of rain in downslope areas.

Plan

The basic order of code building process was-

* Reading in a DEM
* Calculating slopes
* Creating grid for rainfall
* Passing rainfall based on slope
* Displaying data
* Writing out data
* Repeating for multiple iterations

Development

Firstly the code to read in data was written, using the same code as that used in the agent based model, and OS tile NN50 was download from Edina Digimap to be used with this model as a test.

Creating the code to calculate slopes to neighbouring cells was fairly straightforward, a moving window was set up that would assign the difference in elevation to each neighbour for each cell and list them. Due to this moving window, the outer perimeter of cells do not get processed, as they don’t have the 8 neighbours, only 5 on the edges and 3 in the corners, and you would not be able to tell if there were slopes outside of the area.

A ‘flow scalar’ was written in to allow for a proportion of the rain to be passed to multiple neighbours, for example. If one neighbour was twice as steeply sloping as another, it would receive twice the proportion of the rain.

The next addition was the ‘flat land rule’, there was an issue where if a cell was the same elevation as a neighbour, it wouldn’t pass any rain to it, so you could get instances where walls of water would form as it wasn’t spread over the level area, just gathered at the edge. a rule was written to equally split the sum of neighbouring flat lying cells rain values between themselves.

The major issue that was faced was that it was currently processing this code for each cell in the list in list order (left to right, one row at a time). This was causing a wave affect where all the rain was pushed to the right hand edge. this was because if rain was passed left or up, that cell had already been process so it would be left there, whereas if it was going right, that cell would be processed next, so that could roll on and accumulate all of the rain.

To make the modeller work correctly, the cell with the highest value (ie. The highest altitude) would need to be processed first, as that would have to be passing down to cells yet to be processed, then the next highest cells could be processed, leaving no change for accumulations to be left behind.

The first attempts to address this problem where to –

* make a duplicate array of elevation data
* find the highest value in that array
* index its position within a new list (call elevation index or similar)
* set high value to 0
* run the search again, theoretically finding the next highest value, as the previous one was now 0.

At the end of this process you should have a list of cell positions you could use as a run order.

This method was plagued with issues, different searching methods were used, until eventually it would return the list of values. However, having to search 40 000 values, 40 000 times equates to checking 1.6 billion entries. This was not efficient, and the code was still needing a lot of debugging to work fully, therefore an alternative solution was needed.

The solution found was quite simple, create a variable that was equal to the maximum elevation, then inside the for loop running through all of the variables, check to see if the cell is equal to this variable and only process it if it is.

Once all the cells have been checked the variable number can be reduced to find the next highest cells to process.

An animation was added to act as the framework for the loop, to allow for various ‘days of rain’ to be inputted, and every frame would represent a day.

Outputting the final rain accumulation to a text file and adding in the GUI were some of the final tasks, before parts of the code were condensed into functions and the function definitions moved to a separate file.

There were issues with adding some of the code to functions, if the code referred to a location in the array, and then and neighbouring cell in the array, the function could not be written in the same way, for example this line;

rain\_area[i][j] + environment[i][j] > rain\_area[i+x][j+y] + environment[i+x][j+y]

Could not be in the function as;

rain\_area + environment > rain\_area[+x][+y] + environment[+x][+y]

When rain\_area and environment were passed in.

To attempt to combat this variables equal to *variable[i+x][j+y]* were created so that the [+x][+y] would not need to be present In the code, however this did not solve the problem. As the needed to able to refer to themselves relative to their position to other cells, the next attempt at a solution was to put them in a class where it could refer to itself and its x and y coordinated, so (environment.x + *variable name for neighbour*) could be used. However this allow gave runtime errors. The code was left as is as no solution could be found.

Tests

Throughout the process, test DEMs I’d created that would be a smooth slope or a bowl were used, in order to better understand the behaviour of the animations, it was through this method that the wave affect caused by the code running from the top left cell in the DEM was discovered.

A second real DEM, OS tile NN60 was also downloaded to test adaptability.