COMP 116: Assignment #5: Watershed Computations

Due date: Fri 3/19 (March 19th)

A watershed is an area of land that is drained by a particular river, stream, or lake. Watersheds are a natural unit for managing natural resources and controlling development. Terrain elevation maps represented as matrices are often used to compute an approximation to watersheds by assuming that water flows from one entry to its lowest neighbor -- defined more precisely below.

For this assignment, you will write a series of functions and scripts to visualize the surface water runoff across the mountains area around Heber City, Utah.

Once you have completed these questions turn in a publishable script on Sakai (named *solutionWatershed.m*). As usual also submit a report of your solution.

General guideline

Write your solution (including the documentation) in one publishable m-file called 'solutionWatershed.m'. This script should assume that all provided .mat files are in the same directory as the script. It should run without any previous data loading in matlab, i.e., add

clear all; close all;

(which clears all the variables currently in the workspace and closes all open figures) to the top of the script and make sure that it executes properly. Add your name as a comment to the script, e.g.,

% Solution to assignment 5: Stan Ahalt

P1: Finding the lowest neighbors

The '.mat' file elevation.mat contains the elevation data.

load elevation.mat to get the variable **map**, which is a matrix of heights in meters for a region near Heber City, Utah. You also get a small matrix **test** that is 10x10. Display the map as a shaded image with **imagesc(map); axis equal; colormap gray;**

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	P	(0,1)
(1,-1)	(1,0)	(1,1)

Note: The statement **[r,c] = size(map);** assigns **r** and **c** to the numbers of rows and columns. If you use r and c instead of the numbers, then you can run your code on other size maps without modification.

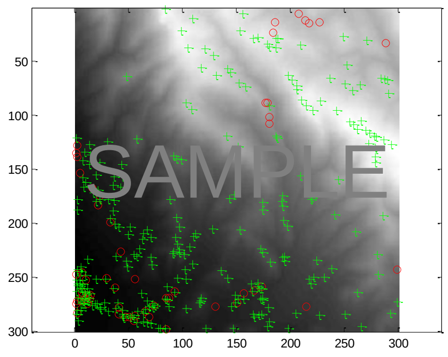
Every pixel P that is not on the boundary of the image has eight neighbors. Water follows the path of steepest descent, so we would like to know, for each non-boundary pixel, direction to the lowest height of the 9 cells around P, including P itself. One way to do that is to store both row and column offsets for each map point r,c to the neighbor: we can record these offsets in two matrices, each containing only the numbers -1, 0, and +1. Let's define boundary pixels to have both offsets zero.

Thus, write and turn in a **function [roffset, coffset] = findLowNhbr(map)** that returns two matrices of the same size as map, so that pixel map(r,c) has pixel map(r+roffset(r,c), c+coffset(r,c)) as the minimum of these 9 heights. (You need not turn in any output.)

P2: Finding pits

Any pixel where the offsets are both zero is a *pit.* Note that every boundary pixel is a pit, so let's ignore them. Write **function rc = findpits(map)** that will return a matrix containing 2 columns with row and column of all non-boundary pits in map. (You can then find the non-boundary *peaks* with `findpits(-map)`, too.) Report how many non-boundary pits and peaks are on the map, and plot them as follows:

```
pits = findpits(map);
peaks = findpits(-map);
imagesc(map); colormap(gray); axis equal
hold on
plot(pits(:,2),pits(:,1),'ro');
plot(peaks(:,2),peaks(:,1),'g+');
hold off
```



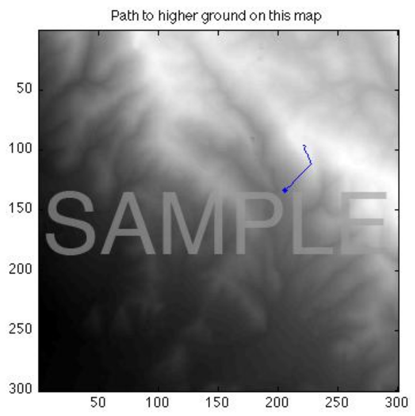
Your findpits() function will probably call findLowNhbr().

Notice that rows are y coordinates and columns are x coordinates, which explains the order in plot. We'll keep talking about rows & columns in the hope it is clearer. BTW, according to my code, **test** has pits at (3,6), (6,3), and (6,6) and peaks at (5,5) and (8,5).

P3: Path to high ground

In the event of heavy rains or flooding, it may be nice to know the path to higher ground. For any point on the map we can calculate the path to higher ground. To demonstrate this, write a function or script that draws the terrain with **imagesc(map)** then calls **[c,r] = ginput(1)**, which will let the user click on a point. (Round column and row, *c* and *r*, to integers before using them to access the map.)

Next, call your function for 1 to get the roffset and coffset matrices. Then follow the steepest ascent path represented by r and c offsets until it stops at a peak or boundary of the map, and collect the resulting r and c coordinates into a vector so that you can plot a blue line on top of the image (use **hold on** so that plotting the line does not erase the image).

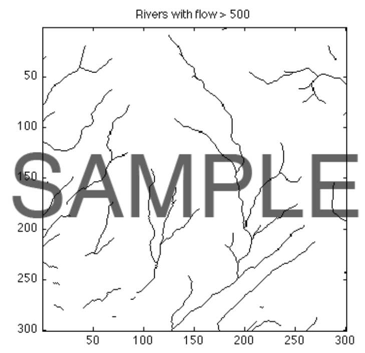


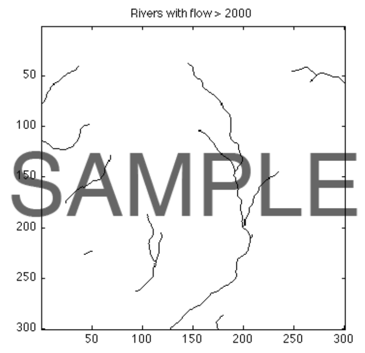
Note: To construct the vectors for r and c, we suggest using a while-loop to add successive points. The first point r,c will be the input from the user. The second point is found by adding the roffset(r,c) and coffset(r,c) to the current point. This repeats until the boundary or a peak is reached. You might also want to stop after 1000 steps, just in case two pixels point to each other…

P4: Finding rivers

This algorithm to find rivers that David Mark published in the 70s is still commonly used: Start by assigning each pixel one unit of flow, then sort all the pixels in order of decreasing height. (See doc sort.) Each pixel (except for pits or boundaries) from highest to lowest then adds its total flow amount to the flow amount at its lowest neighbor. All pixels receiving more than some amount of flow are declared to be rivers. Write and submit a **function result = flow(map)** that returns a matrix of the flow received by each pixel. Hand in images of the pixels that receive 500 and 2000 units of flow (e.g. imagesc(flow>500);).

Again, you'll probably want to use your function for 1 to get the offsets that indicate the lowest neighbors. The command [height, idx] = sort(-map(:)) lets you sort all pixels by treating the whole map as one column. The idx then gives you the list of pixels in order, you just have to extract the r and c coordinates for entry j from the value of idx(j).


Rivers with flow > 500
50
100
150
200
250
300
50
100
150
200
250
300


Rivers with flow > 2000
50
100
150
200
250
300
50
100
150
200
250
300