COMP 116
Programming Assignment #5
Scientific visualization

Due Sunday, November 7st by 11:59pm

The watershed, which is the area of land that is drained by a particular river, stream, or lake, is construct often used in managing natural resources and controlling development that could effect the local environment. Terrain elevation maps represented as matrices can be used to compute approximate watersheds by assuming that water flows from one entry (patch of ground) to its lowest (in height) neighbor – this model is defined more precisely below.

In this assignment, you will write a series of functions and scripts to visualize the surface water runoff in the mountainous region around Heber City, Utah.

Assignment resources

In this assignment, you will need the file **elevation.mat**, which is on the Sakai site. This MAT-file contains the variable `map`, which is a matrix whose cells are heights (in meters) for a region near Heber City, Utah, and the variable `test`, which is a small (10x10) matrix you can use in testing. To get this into MATLAB, as in Assignment 2, make sure that **elevation.mat** is in your active directory and type `load elevation`.

Assignment plan

Once you've loaded the elevation data, you can display the map as a grayscale image by typing:

```
>> imagesc(map); axis equal; colormap gray;
```

Find the lowest neighbor

In order to generate the approximate watershed, we first need to be able to identify the lowest neighbor for each cell. (That is, for each patch of ground, which of the neighboring patches of ground is lowest in height – water generally follows the path of steepest descent, sothis is the cell to which we assume water would flow.)

Every pixel *p* that is not on the boundary of the image has eight neighbors: top-left, -right, and -center, center-left and -right, and bottom-left, -right, and -center. We can identify these neighbors by the offset from *p* needed to get to that neighbor. For example, the top-left cell is one row above and one column to the left of *p*. We can denote the top-left cell, for example, as (-1,-1). (See Fig. 1) We will assume that boundary pixels have both offsets equal to zero – that is, they are denoted (0, 0).

(-1, -1)	(-1, 0)	(-1, 1)
(0, -1)	*p* (0, 0)	(0, 1)
(1, -1)	(1, 0)	(1, 1)

Figure 1: Pixel *p* and its 8 neighbors

Write a function
 `function[rowOffset, colOffset] = findLowNeighbor(map)`
that returns two matrices of the same size as `map` (`rowOffset` and `colOffset`) such that the row and column offsets for a given cell in map are stored in that same cell in `rowOffset` and `colOffset`. That is, the lowest neighbor of pixel `map(r,c)` is pixel `map(r + rowOffset(r,c), c + colOffset(r, c)`.

Find pits
Note that it is possible to have a situation where a pixel does not have a lower neighbor (where both offsets are 0). We will call such a pixel a *pit*. (We arbitrarily defined boundary pixels as having offsets (0,0) as well. We will ignore those in this case.)

Write a function
 `function pitList = findPits(map)`
that returns one matrix which has two columns that are the row and column of all **NON-BOUNDARY** pits in `map`. That is, if `map(x,y)` is a pit, one row of pitList should be `[x y]`.

Your `findPits(map)` function should probably call `findLowNeighbor(map)`.

Not that once we have a function that finds pits, it is very easy to also find *peaks* (pixels that do not have any higher neighbors). You can simply call `findPits(-map)` to find peaks. (Note that that is a minus (-), not a not (~).)

Notice that matrix rows are *y*-coordinates and columns are *x*-coordinates, which explains the "backwards" order of things in the plot command.

N.B. According to my code, `test` has pits at (3, 6), (6, 3), and (6, 6) and peaks at (5, 5) and (8, 5).

Calculate the path of a drop of water
Now that we know the lowest neighbor of every pixel and can identify pits and peaks, it is possible to calculate the path that a drop of water will take starting from any point on the map.

Write a function (or script) that draws the terrain (by calling `imagesc(map); axis equal; colormap gray;`) and allows the user to choose a starting point by clicking on the map (by calling `[col, row] = ginput(1)`; make sure you round `col` and `row` to integers before using them to access `map`). Call `findLowNeighbor(map)` in order to get the `rowOffset` and `colOffset` matrices. Then, starting at (`row, col`), follow the steepest descent path (that is, find the next cell using the offsets in `rowOffset` and `colOffset`) until either (1) it reaches a pit, (2) it reaches the boundary of the image, or (3) you've taken 1000 steps (in case two pixels accidentally point to each other). At each step, save the `row` and `col` coordinates into a vector. Once you reach the end of the path, plot the path as a blue line on top of the image. (Make sure you use `hold on` so that plotting the line does not erase the image.)

Hints on this function
To construct the vector of `col` and `row` coordinates, we suggest using a while-loop to add successive points. The first point (`row, col`) is the user's input. The second point is found by adding `rowOffset(row,col)` and `colOffset(row,col)` to the first point. The third point is found by adding the `rowOffset` and `colOffset` values for the second point to the second point, and so on...repeat until you reach a pit, the boundary, or you've taken 1000 steps.

Your function or script should probably call `findLowNeighbor(map)` and `findPits (map)`.

<u>Finding rivers and streams</u>
In the 1970s, David Mark published an algorithm that can be used to find rivers and streams given a terrain map. This algorithm works as follows:

(1) Assign each pixel one unit of "flow."
(2) Sort the pixels in decreasing order of height. (Call `doc sort` from the command window to see how to do this.)
(3) Step through each pixel in sorted order, and add its flow to the flow of its lowest neighbor.
(4) Apply a threshold, so that each pixel whose flow is greater than some amount is declared to be part of a river.

Write a function
```
 function result = flowMap(map)
```
that returns one matrix `result` whose values are the flow amount for each pixel in `map`.

Hints on this function
Your function or script should probably call `findLowNeighbor(map)`.

The command command `[height, index] = sort(-map(:))` will sort all the pixels by treating the whole map as one big column vector. The index variable then tells you the index into that long vector; in order to use it, you need to convert this value back to a (row, column) index. You can do that with the following commands:

```
[rows, cols] = size(YOURMATRIXNAME)

% mod(X, Y) returns the remainder when X is divided by Y
% This is the row the index-th entry is in.
indexR = mod(index, rows);

% Need to account for MATLAB's 1-indexed arrays (mod(index,
%    rows) == 0 means that index is in the last (rows) row
```

```
for i = 1:size(indexR)
    if (indexR(i) == 0)
        indexR(i) = rows;
    end
end

% The column of the index-th entry, then is as follows:
indexC = ((index - indexR) / cols) + 1;
```

`IndexR`, then, is the list of row addresses, and `indexC` is the list of column addresses. As for how you would use this, say you are looking at the jth element of `index` (that is, you are looking at `index(j)`. That would mean that you were looking at pixel `map(indexR(j), indexC (j))`.

How to demonstrate your program

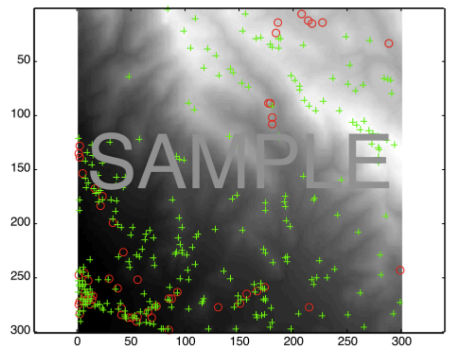
To demonstrate that your functions are working properly, please write a single script file, Assignment5.m, that performs the following tasks (note that while all this happens in a single file, Assignment4.m, this script WILL be calling functions that are defined in other M-files) :

(1) Say (in text) how many pits and peaks were in map.

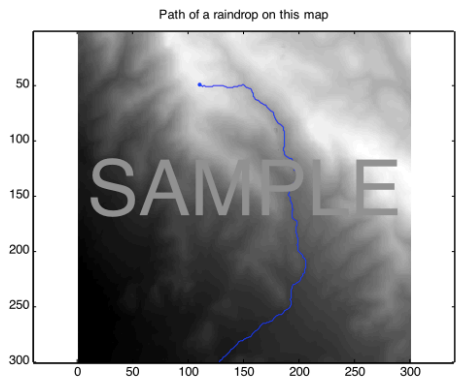
(2) Display map as a grayscale image, with pits and peaks marked on top of it. You can use the following code to do this:

```
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
```

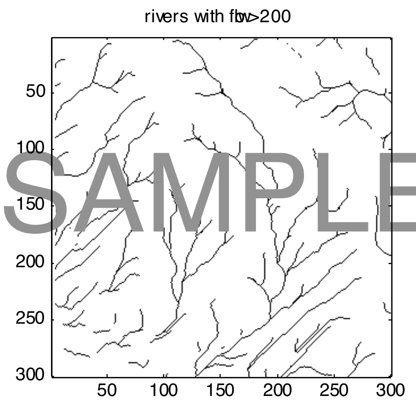
Your output should look like the image at right.



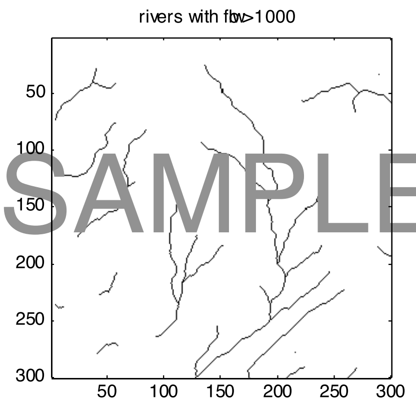
(3) Display map as a grayscale image with the path of a single raindrop drawn on it as discussed above. The resulting image should look something like the image at right (of course, it will be different depending on what pixel was clicked.



(4) Compute the flow matrix as discussed above. Plot an image of all of the pixels that have flow greater than 200. The output image should be like the image at right.



(5) Plot an image of all the pixels that have flow greater than 1000. The output image should be like the image at right.



Required Components: [100 pts]
(Note that all code for each of these components should be included in your published script, or in M-files that are included in your Dropbox.)

1. How many peaks were in map? [5 pts]
2. How many pits were in map? [5 pts]
3. Display map with pits and peaks marked. [20 pts]
4. Compute and display the path of a raindrop. [30 pts]
5. Display the rivers with flow greater than 200. [20 pts]
6. Display the rivers with flow greater than 1000. [20 pts]