**COMP 116**

**Assignment #4**

**Watershed Computations**

Due Date: March 19, 2013, **3:00 p.m.** (EDT)

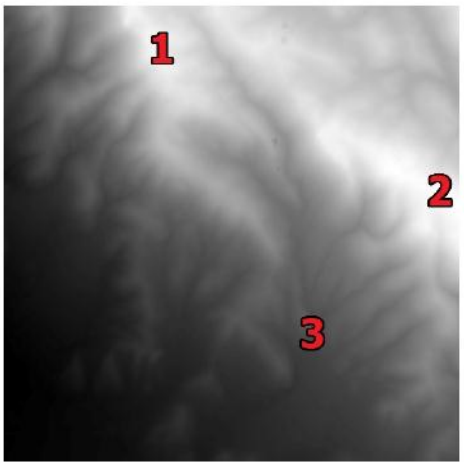
*In this assignment you are going to write functions to visualize the surface runoff across the mountains in the area of Heber City, Utah. Your solution to the problems in this assignment will be in two files. A4Func.py will contain the definitions for all of the functions described below. A4.py will* ***call*** *those functions to produce specific results. The result of this assignment will be a number of images and print statements.*

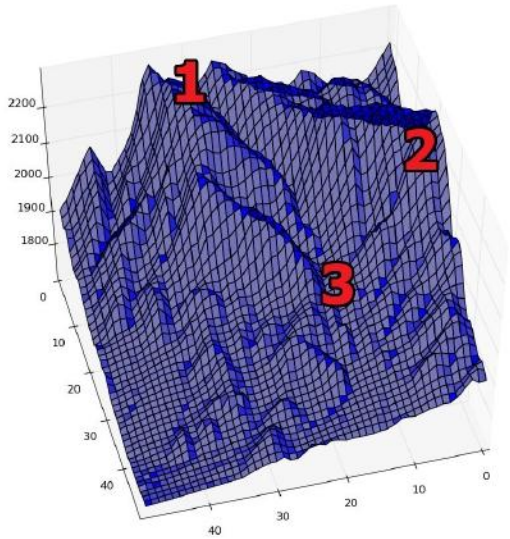
*This assignment is to help you practice with two different principles:* ***loops (for and while)*** *and* ***lists****. You're going to be using loops, lists and tuples to analyze an array of data and create interesting visualizations.*

**Watershed (a.k.a. Drainage Basin)**

The watershed (or drainage basin) is an area of land where water from rain or melting snow drains *downhill* into a body of water. The watershed includes both the terrain over which the water flows as well as the actual rivers and streams which transport the water downwards. The watershed acts like a funnel; water seeks the fastest route downhill. We can model this phenomenon using a **terrain elevation map**.

A terrain elevation map (sometimes called a *height map*) is a representation of actual terrain. It is a 2D array where every element is an elevation measurement from actual terrain. The samples are uniformly distributed along latitudinal and longitudinal lines. We can visualize a height map in several ways. We can visualize it as a 2D image, where the highest point is colored white and the lowest is colored black. Alternatively, we can make a 3D plot where the height of each value is shown in a 3D image. The two images below illustrate this with corresponding points labeled. In this assignment, we'll be using the 2D, grey-scale visualization approach.



**Required Functions**

In A4Func.py you are going to have to provide function definitions for the following functions. A4Func.py already includes place holders for these functions. Currently, each function returns None. It is your task to write the expressions which accomplish the indicated functionality.

**Watershed functions**

1. **`findLowNhbr( terrain )`**

   Inputs: **`terrain`** - a 2D-array representing a terrain elevation map.
   Outputs: **Two** 2D-arrays. One represents the offset in rows toward the direction of greatest downward change. The other represents the offset in columns toward the direction of greatest downward change.
   Functionality: Water flows in the direction where the terrain is the steepest. In our terrain model, we need to determine, at each point, which direction the terrain drops the most.

   Because our terrain model is a uniformly sampled 2D array of elevation measurements, every point, P, in our data has eight points surrounding it (except for those on the edge. We'll treat them specially.) So, our directions are limited to nine directions: the direction to each of the neighbors **as well as the point itself** (it's possible for a terrain value to be in the bottom of a bowl and lower than all of its neighbors.) We can describe that direction as the offset in the row and the column to the cell that has the lowest elevation. So, if the lowest point in the neighborhood of P is down and to the right of P, the offset would be +1 row and +1 column. Similarly, if the lowest point were to the left, the offset would be +0 row and -1 column. And if P is the lowest point in the neighborhood, then the offsets would be 0 and 0 for both row and column. We want the output arrays to contain these offsets.

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

   Imagine that we call the output arrays **`rowOffset`** and **`colOffset`**, representing the row offset and column offset values, respectively. Then for a point in the terrain at index:

   **`[ i, j ]`**,

   the index in the terrain of the neighboring cell lowest in its neighborhood is:

   **`[ i + rowOffset[ i, j ], j + colOffset[ i, j ] ]`**.

   We will be able to use these arrays to find which way the water flows.

   Because the boundary points don't have the full set of neighbors, we will simplify the problem by saying that on the boundaries, the row and column offsets should be zero. In other words, we're assuming that the boundary points are all their own lowest neighbors. So, the results of this function are two 2D-arrays the same shape as the terrain array. The values on the edges of the arrays are zero, and every other entry should be a -1, 0 or 1.

   Doing this function is going to require nested loops. One loop to iterate through each row and then, for each row, another loop to iterate through the columns. This is the point whose neighborhood you're testing. You might need *another* loop (or two) to actually test the neighborhood.

2. **findPits( terrain )**

Inputs: **terrain** - a 2D-array representing a terrain elevation map.
Outputs: An NX2 array of integer pairs representing the indices (row, col) of all of the "pits" in the terrain.
Functionality: A "pit" in the terrain is, mathematically speaking, a local minimum. It is the point in terrain that is lower than all of its neighbors. This function find all of the points in the terrain map (ignoring the boundary points) and returns the row, column index values of each pit.

We can use the two arrays produced by **findLowNhbr** to compute pits. **A pit is everywhere in the terrain except the boundaries where the row and column offsets to the lowest neighbor is 0** (this means, of course, that this point IS the lowest value in its neighborhood.)

To implement this function, you might find the function **np.where** useful. This function returns an array of indices which correspond to the non-zero values in an array. However, it returns a TUPLE of arrays, one per dimension. For example:

In one dimension case,

```
offsetX = np.array( [ 0, -1, 1, 0, 0, 1 ] )
np.where(offsetX == 0 )
   (array([ 0, 3, 4 ] ), )      <-- A tuple with one array in it
```

In two dimensions, it returns a tuple with *two* arrays. The ith element in each array contains the row and column indices for an entry in the array which satisfied the "where" condition.

```
offsetX = np.array([[ 0, -1, 1, 0, 0, 1 ],
              [ 1, -1, 0, 1, 0, 1 ],
              [ 0, 0, 1, 0, -1, -1 ]])
indices=np.where(offset==0)
>>indices
   (array([0, 0, 0, 1, 1, 2, 2, 2]), array([0, 3, 4, 2, 4, 0, 1, 3]))
=> A tuple with two arrays
```

Finally, you can combine those separate arrays into the NX2 array by doing something like this:

```
Indice_offsetX_eq_Zero =
np.column_stack( ( indices[0],indices[1] ) )
```

**!!Note: You should find out indices where both row and column offsets are zero**

Here are the position of my pits. You should make sure you have the same pits.
[ 5, 207], [ 11, 213], [ 13, 185], [ 13, 226], [ 14, 217], [ 23, 183], [ 32, 288], [ 88, 176], [ 88, 178], [101, 180], [107, 180], [127, 2], [134, 1], [138, 2], [153, 4], [167, 16], [174, 23], [182, 21], [198, 32], [225, 42], [242, 298], [245, 6], [246, 1], [250, 29], [251, 55], [252, 10], [258, 173], [259, 37], [261, 166], [262, 92], [263, 6], [264, 156], [266, 11], [266, 14], [267, 7], [268, 84], [268, 87], [269, 10], [269, 13], [271, 2], [271, 5], [273, 150], [274, 1], [274, 11], [275, 73], [276, 130], [276, 214], [277, 40], [281, 65], [282, 2], [283, 40], [285, 55], [286, 46], [286, 50], [286, 68], [289, 54], [297, 84]

3. **`traceDrop( terrain, row, column )`**

Inputs: **`terrain`** - a 2D-array representing a terrain elevation map.

**`row`** - the row index of the starting point of the drop.

**`column`** - the column index of the starting point of the drop.

Outputs: An NX2 array of integer pairs representing the path a raindrop would follow down the terrain from the initial point **`(row, column)`**. The columns are the row and column indices of all the points in the terrain that make up the path.

Functionality: This function traces the path of a drop of water down the terrain. As with findPits, we're going to use the function findLowNhbr.

We can construct a path by using a **`list`**. The first point in our path is the position given as the start position, **`(row, column)`**. We use the row and column offset arrays from **`findLowNhbr`** to figure out which neighboring cell we should move to from this cell. Every time we use the offsets to compute the next element index, we add it to the path. We continue doing this until we end up in a pit (where the row and column offsets are zero.)

This function uses a **`while`** loop. The nature of this loop is indefinite. You do not know how long the path is; it depends on where you start. However, you know the loop condition--keep following the terrain downwards until you reach a point that's lower than all of its neighbors.

To construct the path, you'll use the **`list`** datatype. The **`list`** datatype allows you to add new values to the list so you can increase the contents dynamically (something you *can't* do with numpy arrays.) You can create an empty list with this syntax:

```
myList = []
```

Then you can added pairs of row, column values by using the append method this way (assuming that the row and column values are stored in the variables r and c, respectively):

```
myList.append( (r, c) )
```

Finally, you can convert the list to a numpy array with the following syntax:

```
myArray = np.array( myList )
```

**Exercising functions**

In the file A4.py we're going to exercise the functions you've created by calling them on an input and saving the results. A4.py already imports numpy, pylab and A4Func. In addition, it reads in the terrain file "elevation.npy" and saves the 2D array in a variable called **`terrain`**. This is a large array and you might be easier to test your code on a smaller array. I've also added a smaller 10X10 simulated terrain called "test.npy". Changing the line that creates the terrain variable to load the other file will give you access to the simpler file -- but all of your code will use the same variable. So, changing it back to the full terrain is as simple as changing which file is loaded. In A4 do the following things:

1. Draw the terrain into a figure with the following command:

   ```
   pylab.imshow( terrain, cmap=pylab.cm.gray )
   ```

   This draws the terrain array as an image. Unlike the other times we've drawn images, **`terrain`** isn't a real image. The cmap argument tells python to use a gray color map to color the data. The smallest values will be black and the largest values will be white.

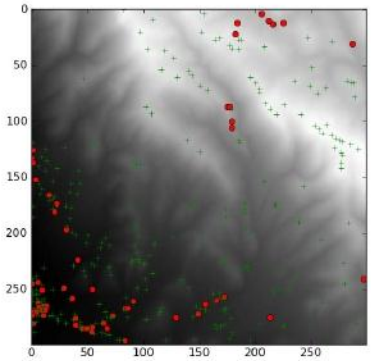
2. Compute the pits of the terrain (by calling **`findPits`** on **`terrain`**) and draw locations of the pits on top of the image as red dots. Remember that **`findPits`** returns an Nx2 array of row and column indices. But when you plot these, the rows are the y-value and the columns are the x-value. Make sure you use the right x and y values when you plot the red dots.

3. Now compute the peaks of the terrain. You can use your **`findPits`** function to find the pits by calling the function on the negation of **`terrain`**. Take these locations and plot them on top of the pits as green crosses (the pylab format string 'g+' will make green crosses.)

4. Print the number of peaks and the number of pits as follows Replace the 0 with the actual number. In your code, if you simply type in a literal number, even if the number is correct, you will lose points. Your program should compute the values and have them stored in a variable that you output (I've listed here my actual answers so you can see if you're getting the right values):

   ```
   Number of peaks: 273
   Number of pits: 57
   ```

5. Save this figure as '**peaksNPits.png**' and then call **`pylab.close()`** (so that this figure doesn't interfere with the next stage. The result should look something like the image on the right.



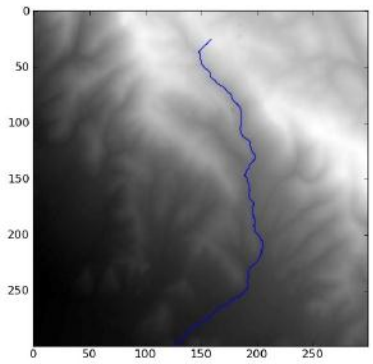

6. Use **`pylab.ginput(1)`** to let the user click on a single point in the terrain. We will use this point to test **`traceDrop`**. The point the user clicks is the starting point you'll pass to the **`traceDrop`** function.

**pylab.ginput(1)** will give you a TUPLE consisting of one tuple with two values (the x and y values.) These x and y values won't necessarily be integers, but you need integers to index into the array. The following code will turn **pylab.ginput**'s result into a row and column index:

```
pylab.imshow( terrain, cmap=pylab.cm.gray )
point = np.array( pylab.ginput(1)[0] )
r = int( np.round( point[1] ) )     # an integer row index
c = int( np.round( point[0] ) ) # an integer column index
```

7. Draw the terrain into a figure (like in step 1.)

8. Call **traceDrop** with the row and column index from step 5. Plot the path **traceDrop** returns as a blue line on top of the image (watch out for the row-column/x-y issue discussed in step 2.)




9. Save this figure as 'raindrop.png' and then call **pylab.close()**. The image should be similar to the image on the left. It won't be exact, because it depends on where you click. In this case, I clicked on the white pixel at the top end of the path.

**Grading notes**

If your script doesn't run, you automatically lose 50% of the points. Make sure that your script runs without any errors. Errors can come from two sources: 1) your code isn't even proper python. 2) Your code is syntactically correct, but you're trying to do something semantically illogical (e.g. x = np.arange( 10 ) and then trying to do this: x[20]. The index isn't valid.)

3. **THIS FUNCTION IS PART OF THE EXTRA CREDIT!!**

**`findFlow( terrain )`**

Inputs: **`terrain`** - a 2D-array representing a terrain elevation map.

Outputs: A 2D-array consisting of the total flow through each point in the terrain (except the boundaries.)

Functionality: We are going to implement an algorithm invented in the 70s (and is still commonly used) to compute water flow through the terrain. When we're done we'll have a numerical value at each point in the terrain which represents the amount of water that flows over that point. The higher the number, the more water flows. Ultimately, we can use this information to find rivers and streams.

The algorithm is as follows:

a. Create a new array of *ones* the same size as the terrain.
b. Sort the heights of the terrain map from highest to lowest.
c. For each point (i,j), except the pits, from the highest point down to the lowest:
   i. Determine the neighboring cell to (i,j) into which the water flows.
   ii. Add (i,j)'s flow into the neighboring cell's flow.

To implement this algorithm, you'll want to use the **`findLowNhbr`** function again. This is how you will do step c.i in the algorithm.

The trickiest part of this function is performing the core part of the flow algorithm from the highest point down to the lowest. You can't just work along the terrain from left to right to get the right answer. Numpy has a useful function: **`numpy.argsort`**. What **`numpy.argsort`** does is return an array of INDEX values. The first index value is the index of the smallest value in the input array. The second index value is the index of the second-smallest value in the input array, etc. You can use **`numpy.argsort`** to get the ordered indices of a sorted array into terrain. By default it sorts the data from smallest value to greatest value. So, if you called **`numpy.argsort( terrain )`**, it would give you the index of the lowest point first. You can reverse this by simply negating terrain (making the big values small and the small values big.)

Because we have to sort the whole array, we can't use a specific axis. Because of that the index values you are given are like the "size" property of the array. The index is the index you'd get if you reshaped the 2D array into a 1D array. An MXN array would have index values in the range [0, MN - 1]. To use it in the terrain, you'd have to convert this "flattened" index into a 2D index. Here's an example:

```
y = np.array([ [ 10, 120, 90 ], [ 70, 20, 110 ], [ 40, 30, 60 ], [100, 50,
80 ]])
indices = np.argsort( y, axis=None )
rows = indices / 3            # the number of columns
cols = indices % 3            # the number of columns
rows
   array([0, 1, 2, 2, 3, 2, 1, 3, 0, 3, 1, 0])
cols
   array([0, 1, 1, 0, 1, 2, 0, 2, 2, 0, 2, 1])
```

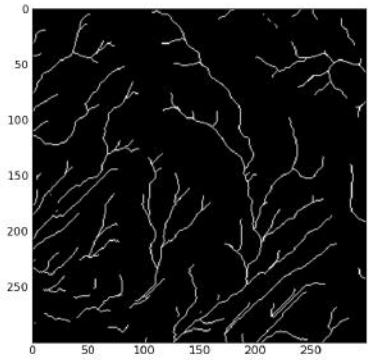
So, the smallest value is at [0, 0] and the largest is at [0, 1].

10. **EVERYTHING BELOW THIS LINE IS EXTRA CREDIT**

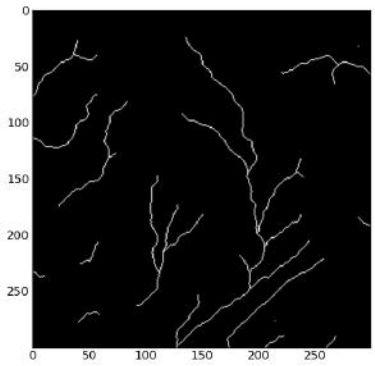
11. Compute the flow for the terrain by calling **`findFlow`** on **`terrain`**. We will use the flow to find the river and streams in the watershed.

12. To be a stream, a point in the terrain must have a flow of more than 200 (Note to the scientists out there: this number is arbitrary and the units are undefined.) Create a figure showing all of the terrain that has flow of at least 200. The code below shows you how:

```
# the variable terrainFlow holds the result of
# the call to findFlow
rivers = terrainFlow > 200
pylab.imshow( rivers , cmap=pylab.cm.gray )
```




13. Save the result as '**streams.png**' and then call **`pylab.close()`**. The image should look like the image on the right.




14. To be a river, it must have a flow of at least 1000. Create another figure consisting of the regions of terrain with a flow of at least 1000 (similar to step 10.)

15. Save the result as '**rivers.png**' and then call **`pylab.close()`**. The image should look like the image to the left.