OOSE – Final Programming Task

Handed in with Extension on 27/02/2019

Exam Number: B130808

Task 1: Calculate flow direction with constant rainfall

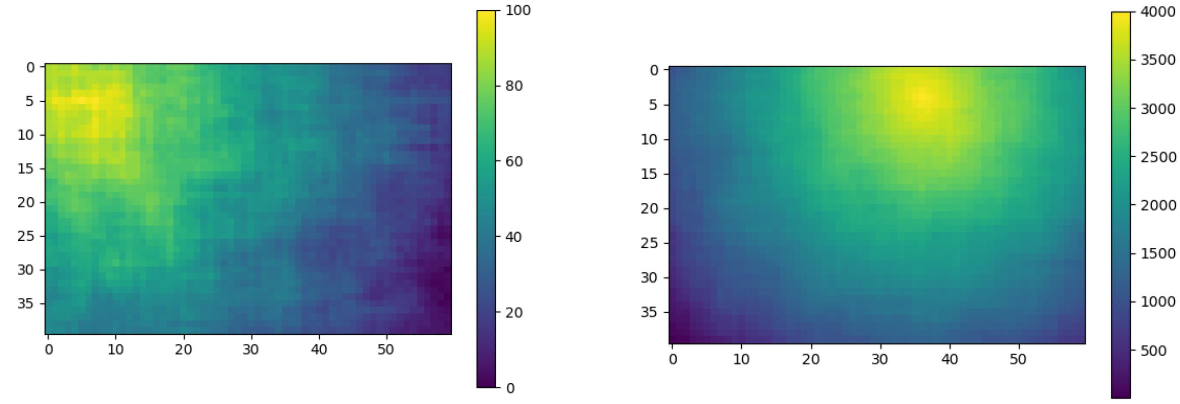

Figure 1: Original elevation (left) and rain (right), randomly generated

Figure 1 shows the elevation and rainfall of two randomly generated raster while Figure 2 shows the corresponding flow network. The workflow is straightforward, the `plotFlowNetwork` function gets here four arguments: the randomly generated elevation raster, the flow raster of the mentioned elevation raster with a resampled cell size, the title of the layer and a parameter to not plot lakes on the output. Inside of the function, first the background is displayed showing the `_data` attribute of the elevation raster without resampling. Then on top, a scatterplot and line plot of the `_data` attribute from flow

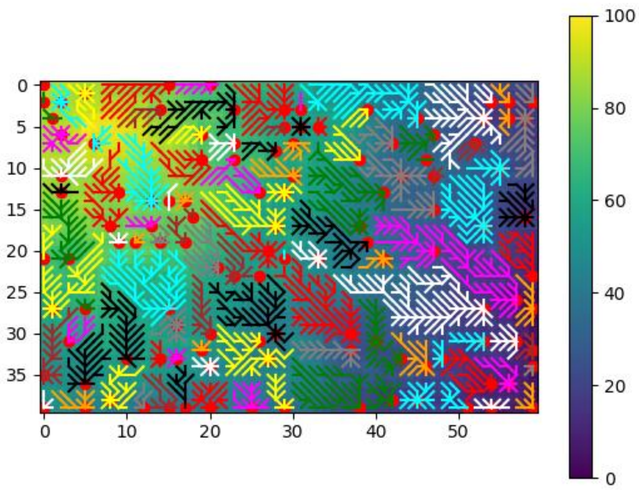

Figure 2: Flow network from randomly generated raster

raster, which is here defined as nodes, is displayed. These plots shows in total where every node and the water passages are located, so where the rain is flowing to.

To enable these steps, the flow.py module had to be enhanced with `from Raster import Raster` in the heading, because the `FlowRaster` class is inheriting functionality from `Raster`.

Task 2: Implement a method that calculates flow volume, assuming constant rainfall and plot the resultant data

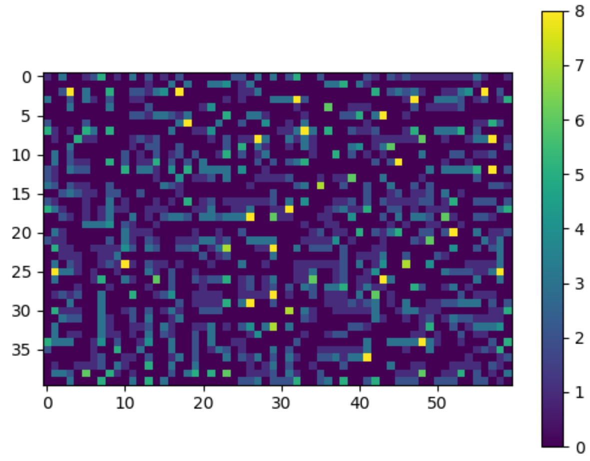

Figure 3: Simplified flow rate estimation from randomly generated raster

Figure 3 shows the volume of water which reaches each cell in the flow raster. As here constant rainfall is assumed with a value of 1 mm per cell, the unit has been skipped and the number of upper cells flowing into one down cell were calculated. For the calculation a new class and function has been created. The `FlowExtractor` contains the `getValue` function which expect a `FlowNode` object to be passed into it and which calls then the second new function: `getFlow()` in the `FlowNode` class, in which the `numUpnodes()` function is called to retrieve the number of `upnode` for a specific node in case that the `israin` parameter equals `False` and therefore no rain data was passed to the flow raster. This whole process is recursively called by the `extractValues` function from the `FlowRaster` class, which is itself called once by the `plotExtractedData` function in the driver for task 2. As a result, an array holding the number of belonging up nodes per node is plotted and shown in figure 2, which shows therefore also the flow volume in millimetre as we assume a constant rainfall of 1mm per cell. A cell of with belonging up nodes retrieves therefore also 5 millimetres of water.

Task 3: Repeat task 2 but this time using non-constant rainfall rather than assuming a constant rainfall.

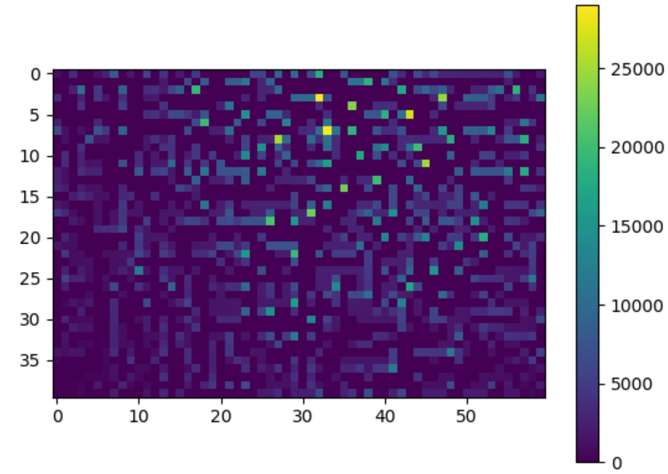

Figure 4: Sophisticated flow rate estimation from randomly generated rainfall data

For task 3, the same functions as in task 2 in `plotExtractedData` are called but now the parameter `isRainFall` is set to `True` to call a different `if` clause in the `getFlow` function. This parameter starts an iteration process, in which a variable is built which holds the summarized rain values of the belonging up nodes for the specific node called. The rain values are stored in the new `_rain` attribute, which is created before the `plotExtractedData` function in the task 3 driver is called. This creation happens by calling the new `addRainfall` function, which has been written for this specific task. The function expects to receive an array containing the values of a rainfall raster. The rainfall raster is stored inside of the `caluclateFlowsAndPlot` function as `rain` and therefore the expected array is passed into `addRainfall` by calling `rain.getData()`, a function from the `Raster` class which retrieves an array of values with the shape of the original raster. Inside `addRainfall`, a nested for loop iterates over rows and columns of the value array and stores every value of each cell in the `_rain` attribute on top of the `_data` attribute of the original flow raster. These values are then summarized in `getFlow` and plotted through `plotExtractedData` as an array of summarized variable rainfall per cell as shown here in Figure 4.

Task 4: Something seems to be missing from source so far. The client asks “where is the water going? It does not seem to flow across the landscape and exit into the sea” (we assume here the sea is the edge of the DEM). The client asks if you can improve the algorithms.

For this task, pseudo code is provided in the file Pseudocode.task4 which is close to Python syntax. In this file, the class `DealWithLakes` containing three functions, `defineLakes`, `calculateLakes` and `LakeDepthExtractor`, is provided which should enhance the current algorithm with the necessary functionality to not stop as soon as the flow reaches a lake. The coursework1.py driver shows the process, here with out commented code in section `##step 4 and 5##`. First, the `defineLakes` function is called, which iterates through all cells and defines if the current cell is part of a lake. The function assumes that a lake is a conglomerate of multiple down nodes next to each other. Therefore, it retrieves all neighbours for each cell and checks if the cell itself and one or more of its direct neighbours is marked as being a down node. If these conditions are met, then the new `_Lake` attribute on top of the belonging `_data` attribute is set to `True`. This attribute is then used in the second function, `calculateLakes`, which iterates through the raster and checks if the current cell is part of a lake (if its `_Lake` attribute is `True`) and its neighbours are also part of a lake. If so, the function checks further if one of these lake cells is marked as an up node and if so, all investigated cells are set to be an up node to allow the algorithm to run further and not to stop if a lake is reached. All these up nodes can now flow to the belonging down node. Also, the flow volume of all lake cells is extracted and summarized and then reassigned to the investigated lake cells. Accordingly, all cells which are part of the lake have the same total flow volume reaching them, because they are connected. The coursework driver also plots the depths of all lakes, which are retrieved through the `LakeDepthExtractor` function in the `DealWithLakes` class. This function iterates through the DEM and checks if the current cell is part of a lake. If so, the height value of that cell is assigned to a depth array. If the cell is not part of a lake, the value `-999` is assigned to indicate a non-relevant cell. This depth array is then returned and plotted.

Task 5: Get the code to work with real data to answer the following two questions:

1. What is the maximum flow rate in the DEM ?
2. What is the location of the maximum flow rate ?

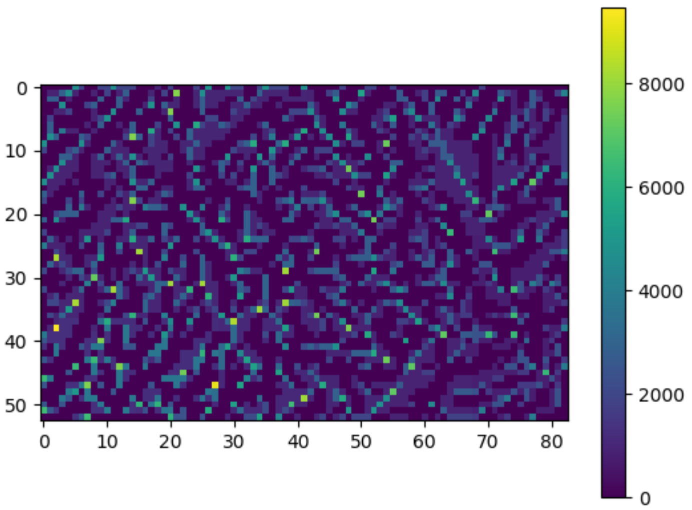

Figure 5: Flow rate estimation with real data

Up to the functionality of task 4, the code is also working with real data and some enhancements of the `extractValues` function inside of the `FlowRaster` class allowed to answer these questions. As visible in Figure 5, the real data shows less variability then the random data set and lower maximum values. The maximum flow rate in the DEM is for the real data 9474 mm at the location of x = 20 and y = 380. In both data sets the maximum number up nodes is as visible when looking at Figure 6 compared to Figure 3. To gain this functionality, inside of `extractValues` the final `valuesarray` is compared with a maximum value, which is in the beginning of the process 0 and which is then always set to current value if this value is greater then the maximum value. The location is retrieved by storing the last indices of the highest value in the variables `k` and `j`, which are then used to retrieve the cartesian

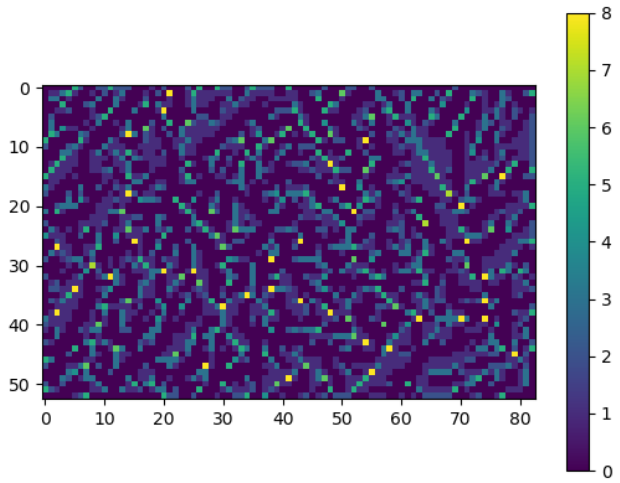

Figure 6: Simplified flow rate estimation with real data

position via the `__str__` function of the `FlowNode` class. Both, value and position get printed as soon as `extractValues` is called, therefore visible above the plots of Figure 4 and 5.

Task 6: Overall Discussion

Future improvements of the code can be achieved in multiple ways. Clearly, the shown examples are for 2-dimensional-plots, while a DEM holds 3 Dimensions. The plotting is therefore a reduction of information. Instead of looking from a bird's perspective on top of a flow diagram, the plot could extract x-,y- and -z – coordinates from every node and display them in a 3D-Model. Such a plot would also benefit from interactivity, so that the client could understand the model intuitively by looking at it without the need of many explanations. Also, the estimation of the flow rate could be improved at is it currently just adding the amount of rain from upper nodes together and assumes no loss of water. Nevertheless, in nature there is a lot of water lost on the way, for instance by plants, drainage and the sun. In the first step, the drainage could be calculated as for such a model just the knowledge of the present soil types is necessary, and a soil-type-layer could be just overlain and added to the model. On sand, more water would drain as on silt and on silt the drainage is bigger than on clay. A model which is straightforward for just a few soil types and which is quite sophisticated with a greater level of detail. In terms of plants, similar issues would arise. Here, a plant can drain very different volumes of water, depending on its height, amount of leaves and further water savoury capabilities. Also, multiple plants can be in a single raster cell and every cell can just store one final value, which requires therefore a simplification of the measurement. Therefore, a field-study could be necessary, in which the overall plant-density per cell is estimated. The pure coverage of plants per cell could be also estimated through remote sensing, but depending on the level of detail needed, a field study could be conducted. Also, the shadowing per cell by trees could be either estimated via remote sensing or via a field study depending on the level of detail needed. Adding a shadowing layer to the model, the influence of the sun through vaporization could be estimated. Still, the flow rate estimation is then based on yearly mean values of precipitation. The model could be enhanced if a time-dimension is added to it. Here, monthly precipitation rates instead of year wise once could already have a big impact on the final outcome as this enables also the other parameters to be more sophisticated. Depending on month and seasoning, the drainage of the plants can vary heavily as well as their protective shadowing effect. Also, precipitation is not the only natural source for water flowing downwards. To create a more generalized model, glacial water flowing down mountains would also add up with the rainfall, here also heavily depending on the season present. Furthermore, the model does not check fur present barrens or rivers, which may come from a natural well or out of the mountains and which would add up with the flows generated by the model. Therefore, adding a river-layer of known streams would also increase the accuracy of our flow estimations significantly.

Summarizing, the following datasets could improve our flow estimations: A river-layer, a soil-layer, a plant-coverage layer, a shadowing-layer and especially a time-scale, effecting all present layers. The precision of the flow rate estimation varies then strongly with the level of detail provided. Also, the current code is limited by just representing 2-dimensional plots while at least three dimensions are present, with a time-scale even four. It could be enhanced by creating 3-dimensional, interactive plots including a timeline to represent then also the fourth dimension.

Literature

1. Martelli, A., Ravenscroft, A., Ascher, D., & Martelli Ravenscroft, A. (2005). *Python cookbook*. (2nd edition / edited by Alex Martelli, Anna Martelli Ravenscroft, and David Ascher.. ed.). Sebastopol, CA: O'Reilly Media.
2. Martelli, A., Ravenscroft, A., & Holden, S. (2017). *Python in a nutshell* (Third ed., In a nutshell (O'Reilly & Associates)). Sebastopol: O'Reilly Media, Incorporated.
3. Lutz, M. (2013). *Learning Python* (Fifth edition / Mark Lutz.. ed.). Sebastopol, CA: O'Reilly.

Appendix

Coursework1.py

```
from RasterHandler import createRanRasterSlope
from RasterHandler import readRaster
import matplotlib.pyplot as mp
import Flow as flow

def plotstreams(flownode,colour):
    for node in flownode.getUpnodes():
        x1=flownode.get_x()
        y1=flownode.get_y()
        x2=node.get_x()
        y2=node.get_y()
        mp.plot([x1,x2],[y1,y2],color=colour)
        if (node.numUpnodes()>0):
            plotstreams(node,colour)

def plotFlowNetwork(originalRaster, flowRaster, title="", plotLakes=True):
    print ("\n\n{}".format(title))
    mp.imshow(originalRaster._data)
    mp.colorbar()
    colouri=-1

colours=["black","red","magenta","yellow","green","cyan","white","orange","grey","b
rown"]

    for i in range(flowRaster.getRows()):
        for j in range(flowRaster.getCols()):
            node = flowRaster._data[i,j]

            if (node.getPitFlag()): # dealing with a pit
                mp.scatter(node.get_x(),node.get_y(), color="red")
                colouri+=1
                plotstreams(node, colours[colouri%len(colours)])

            if (plotLakes and node.getLakeDepth() > 0):
                mp.scatter(node.get_x(),node.get_y(), color="blue")

    mp.show()
```

```
    def plotExtractedData(flowRaster, extractor, title="", isRainFall = False):
        print ("\n\n{}".format(title))
        mp.imshow(flowRaster.extractValues(extractor, isR = isRainFall))
        mp.colorbar()
        mp.show()

    def plotRaster(araster, title=""):
        print ("\n\n{}, shape is  {}".format(title, araster.shape))
        mp.imshow(araster)
        mp.colorbar()
        mp.show()


    def calculateFlowsAndPlot(elevation, rain, resampleF):
        # plot input rasters
        plotRaster(elevation.getData(), "Original elevation (m)")
        plotRaster(rain.getData(), "Rainfall")
        resampledElevations = elevation.createWithIncreasedCellsize(resampleF)

        ################# step 1 find and plot the intial network #######
        '''
        1. From Raster import Raster
        2. Execute full code
        '''

        fr=flow.FlowRaster(resampledElevations)
        plotFlowNetwork(elevation, fr, "Network structure - before lakes", plotLakes=False)

        ################Step 2 ######################################
        '''
        Calculate flow volume
        Resursively call each up-node and add one each time
        Solution:
            def getValue(self, node):
                return node.numUpnodes()
        '''
        plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - constant rain")

        ################# step 3 #######################################
        #handle variable rainfall
        '''
        addRainfall does not replace any values but adds
        the rainfall values on top of the current _data attribute of every node
        '''

        fr.addRainfall(rain.getData())
        plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall", isRainFall = True)

        ############# step 4 and step 5 #######################################
        # handle lakes
        #fr.defineLakes()
        #fr.calculateLakes()
        #plotFlowNetwork(elevation, fr, "Network structure (i.e. watersheds) - with lakes")
        #plotExtractedData(fr, flow.LakeDepthExtractor(), "Lake depth")
        #plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall")


    ############# step 1 to 4 #######################################
    # Create Random Raster
    rows=40
    cols=60
    xorg=0.
    yorg=0.
    xp=5
    yp=5
```

```
   nodata=-999.999
   cellsize=1.
   levels=4
   datahi=100.
   datalow=0
   randpercent=0.2

   resampleFactorA = 1
   elevationRasterA=createRanRasterSlope(rows,cols,cellsize,xorg,yorg,nodata,levels,
datahi,datalow,xp,yp,randpercent)
   rainrasterA=createRanRasterSlope(rows//resampleFactorA,cols//resampleFactorA,cell
size*resampleFactorA,xorg,yorg,nodata,levels,4000,1,36,4,.1)

   calculateFlowsAndPlot(elevationRasterA, rainrasterA, resampleFactorA)

   ############# step 5 #######################################
   #calculateFlowsAndPlot(readRaster('../data/dem_hack.txt'),
readRaster('../data/rain_small_hack.txt'), 10)
```

Flow.py

```
import numpy as np

from Points import Point2D
from Raster import Raster

class FlowNode(Point2D):

    def __init__(self,x,y, value):
        Point2D.__init__(self,x,y)
        self._downnode=None
        self._upnodes=[]
        self._pitflag=True
        self._value=value
        self._rain = 0

    def setDownnode(self, newDownNode):
        self._pitflag=(newDownNode==None)

        if (self._downnode!=None): # change previous
            self._downnode._removedUpnode(self)

        if (newDownNode!=None):
            newDownNode._addUpnode(self)

        self._downnode=newDownNode

    def getDownnode(self):
        return self._downnode

    def getUpnodes(self):
        return self._upnodes

    def _removedUpnode(self, nodeToRemove):
        self._upnodes.remove(nodeToRemove)

    def _addUpnode(self, nodeToAdd):
        self._upnodes.append(nodeToAdd)

    def numUpnodes(self):
        return len(self._upnodes)

    def getPitFlag(self):
        return self._pitflag

    def getElevation(self):
        return self._value

    def getFlow(self, israin = False):
```

```

        if israin == True:

            sumRain = 0

            for i in self.getUpnodes():

                sumRain = sumRain + i._rain

            return sumRain

        if israin == False:
            return self.numUpnodes()


    def __str__(self):
        return "Flownode x={}, y={}".format(self.get_x(), self.get_y())

class FlowRaster(Raster):

    def __init__(self,araster):
        super().__init__(None,
araster.getOrgs()[0],araster.getOrgs()[1],araster.getCellsize())
        data = araster.getData()
        nodes=[]
        for i in range(data.shape[0]):
            for j in range(data.shape[1]):
                y=(i)*self.getCellsize()+self.getOrgs()[0]
                x=(j)*self.getCellsize()+self.getOrgs()[1]
                nodes.append(FlowNode(x,y, data[i,j]))

        nodearray=np.array(nodes)
        nodearray.shape=data.shape

        self._data = nodearray


        self.__neighbourIterator=np.array([1,-1,1,0,1,1,0,-1,0,1,-1,-1,-1,0,-1,1] )
        self.__neighbourIterator.shape=(8,2)
        self.setDownCells()

    def getNeighbours(self, r, c):
        neighbours=[]
        for i in range(8):
            rr=r+self.__neighbourIterator[i,0]
            cc=c+self.__neighbourIterator[i,1]
            if (rr>-1 and rr<self.getRows() and cc>-1 and cc<self.getCols()):
                neighbours.append(self._data[rr,cc])

        return neighbours

    def lowestNeighbour(self,r,c):
        lownode=None

        for neighbour in self.getNeighbours(r,c):
            if lownode==None or neighbour.getElevation() < lownode.getElevation():
                lownode=neighbour

        return lownode

    def setDownCells(self):
       for r in range(self.getRows()):
           for c in range(self.getCols()):
               lowestN = self.lowestNeighbour(r,c)
               if (lowestN.getElevation() < self._data[r,c].getElevation()):
                   self._data[r,c].setDownnode(lowestN)
               else:
                   self._data[r,c].setDownnode(None)

    def addRainfall(self, rainObject):
```

```
        for i in range(rainObject.shape[0]):
            for j in range(rainObject.shape[1]):
                self._data[i,j]._rain = rainObject[i,j]



    def calculateLakes(self):
        return self

    def getPointList(self):
        return np.reshape(self._data, -1)

    def extractValues(self, extractor, isR):
        values=[]
        maxRain = 0
        for i in range(self._data.shape[0]):
            for j in range(self._data.shape[1]):
                values.append(extractor.getValue(self._data[i,j], isRainfall = isR))

        valuesarray=np.array(values)
        valuesarray.shape=self._data.shape
        for i in range(valuesarray.shape[0]):
            for j in range(valuesarray.shape[1]):
                if valuesarray[i,j] > maxRain:
                    maxRain = valuesarray[i,j]
                    k = i
                    l = j
        print('''The maximum flow rate is :'''+str(round(maxRain))+''' mm''')
        print('''Location of the maximum '''+self._data[k,l].__str__())
        return valuesarray


class FlowExtractor():

    def getValue(self, node, isRainfall):

            return node.getFlow(israin = isRainfall)
```

Pseudocode.task4

```
class DealWithLakes():

      self._Lake = False

      function defineLakes(self):

             '''Defines if a specific node is part of a lake or not'''

             for i in range(self._data.shape[0]):
                    for j in range(self._data.shape[1]):
                           neighbours = FlowRaster.getNeighbours(self._data[i,j])
                           if (self._data[i,j]._downnode == True) &&
(self._downnode %in% neighbours == True):
                                  self._data[i,j]._Lake = True


      function calculateLakes(self, Node):

             '''If a node is part of a lake, check his direct neighbours and if they
             are also part of a lake, then assign them the same FlowVolume. Check
also,
             if one of them is marked as an upnode, and if so, mark the others also
as upnodes'''

             for i in range(self._data.shape[0]):
                    for j in range(self._data.shape[1]):
```

```
                if data[i,j]._Lake == True:
                    neighbours = FlowRaster.getNeighbours(data[i,j])

                    for k in range(len(neighbours)):
                        if self._upnode %in% neighbours == True:
                            neighbours[k]._upnode = True
                            FlowVolume = FlowVolume +
FlowExtractor.getFlow(neighbours[k])

                    for l in range(len(neighbours)):
                        neighbours[l]._rain = FlowVolume


    function LakeDepthExtractor(self):

        deptharray = self._data.shape
        for i in range(self._data.shape[0]):
            for j in range(self._data.shape[1]):
                if self._data[i,j]._Lake == True:
                    deptharray[i,j] = self._data[i,j]
                else:
                    deptharray[i,j] = -999

        return deptharray
```

Points.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 05 00:46:36 2012

@author: nrjh
"""

import math

class Point2D(object):
    '''A class to represent 2-D points'''

# The initialisation methods used to instantiate an instance
    def __init__(self,x,y):
#ensure points are always reals
        self._x=x*1.
        self._y=y*1.

#return a clone of self (another identical Point object)
    def clone(self):
        return Point2D(self._x,self._y)

#return x coordinate
    def get_x(self):
        return self._x

#return y coordinate
    def get_y(self):
        return self._y

#return x coord if arg=0, else y coord
    def get_coord(self,arg):
        if arg==0:
            return self._x
        else:
            return self._y


#return x,y tupel
    def get_xys(self):
        return (self.x,self._y)
```

```
    #move points by specified x-y vector
    def move(self,x_move,y_move):
        self._x = self._x + x_move
        self._y = self._y + y_move

#calculate and return distance
    def distance(self, other_point):
 #     put in check to see if other point is a point
     xd=self._x-other_point._x
     yd=self._y-other_point._y
     return math.sqrt((xd*xd)+(yd*yd))


    def bearingTo(self, other_point):

       otherX = other_point.get_x()
       otherY = other_point.get_y()
# All geometry is in radians
# we could convert to degrees if we wanted
# math.pi is a  funtion of the math module
       distance = self.distance(other_point)
       sinTheta = (otherX - self._x) / distance
       cosTheta = (otherY - self._y) / distance

       aSinTheta = math.asin(sinTheta)

#These conditions give an angle between 0 and 2 Pi radians
#You should test them to make sure they are correct
       if (sinTheta >= 0.0 and cosTheta >= 0.0):
           theta = aSinTheta
       elif (cosTheta < 0.0):
           theta = math.pi - aSinTheta
       else:
           theta = (2.0 * math.pi + aSinTheta)
       return theta


    def samePoint(self,point):
        if point==self:
             return True

    def sameCoords(self,point,absolute=True,tol=1e-12):
        if absolute:
            return (point.get_x()==self._x and point.get_y()==self._y)
        else:
            xequiv=math.abs((self.get_x()/point.get_x())-1.)<tol
            yequiv=math.abs((self.get_y()/point.get_y())-1.)<tol
            return xequiv and yequiv


#End of calss Point 2D
#********************************************************


class PointField(object):
    '''A class to represent a field (collection) of points'''

    def __init__(self,PointsList=None):
        self._allPoints = []
        if isinstance(PointsList, list):
            self._allPoints = []
            for point in PointsList:
                if isinstance(point, Point2D):
                    self._allPoints.append(point.clone())

    def getPoints(self):
        return self._allPoints

    def size(self):
        return len(self._allPoints)
```

```

    def move(self,x_move,y_move):
        for p in self._allPoints:
            p.move(x_move,y_move)

    def append(self,p):
        self._allPoints.append(p.clone())

#method nearestPoint
    def nearestPoint(self,p,exclude=False):
        """A simple method to find the nearest Point to the passed Point2D
        object, p.  Exclude is a boolean we can use at some point to
        deal with what happens if p is in the point set of this object, i.e
        we can choose to ignore calculation of the nearest point if it is in
        the same set"""

#check we're been passed a point
        if isinstance(p,Point2D):

#set first point to be the initial nearest distance
            nearest_p=self._allPoints[0]
            nearest_d=p.distance(nearest_p)

# now itereate through all the other points in the PointField
# testing for each point, i.e start at index 1
            for testp in self._allPoints[1:]:

# calculate the distance to each point (as a test point)
                d=p.distance(testp)

# if the test point is closer than the existing closest, update
# the closest point and closest distance
                if d<nearest_d:
                    nearest_p=testp
                    nearest_d=d

# return the nearest point
            return nearest_p

#else not a Point passed, return nothing
        else:
            return None



    def sortPoints(self):
           """ A method to sort points in x using raw position sort """
           self._allPoints.sort(pointSorterOnX)



class Point3D (Point2D):

    def __init__(self, x, y, z):
        print ('I am a Point3D object')
        Point2D.__init__(self, x, y)
        self._z = z
        print ('My z coordinate is ' + str(self._z))
        print ('My x coordinate is ' + str(self._x))
        print ('My x coordinate is ' + str(self._y))

    def clone(self):
        return Point3D(self._x, self._y, self._z)

    def get_z(self):
        return self._z

    def move(self, x_move, y_move, z_move):
        Point2D.move(self,x_move, y_move)
        self._z = self._z + z_move
```

```

    def distance(self, other_point):
        zd=self._z-other_point.get_z()
#        xd=self._x-other_point.get_x()
#        yd=self._y-other_point.get_y()
        d2=Point2D.distance(self,other_point)
        d3=math.sqrt((d2*d2)+(zd*zd))
        return d3



def pointSorterOnX(p1,p2):
    x1=p1.get_x()
    x2=p2.get_x()
    if (x1<x2): return -1
    elif (x1==x2): return 0
    else: return 1

def pointSorterOnY(p1,p2):
    y1=p1.get_y()
    y2=p2.get_y()
    if (y1<y2): return -1
    elif (y1==y2): return 0
    else: return 1
```

Raster.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 31 00:44:33 2013

@author: nrjh
"""
import numpy as np

class Raster(object):

    '''A class to represent 2-D Rasters'''

# Basic constuctor method
    def __init__(self,data,xorg,yorg,cellsize,nodata=-999.999):
        self._data=np.array(data)
        self._orgs=(xorg,yorg)
        self._cellsize=cellsize
        self._nodata=nodata

    def getData(self):
        return self._data

#return the shape of the data array
    def getShape(self):
        return self._data.shape

    def getRows(self):
        return self._data.shape[0]

    def getCols(self):
        return self._data.shape[1]

    def getOrgs(self):
        return self._orgs

    def getCellsize(self):
        return self._cellsize

    def getNoData(self):
        return self._nodata

    # returns a new Raster with cell size larger by a factor (which must be an integer)
    def createWithIncreasedCellsize(self, factor):
```

```
        if not isinstance(factor, int):
            print ("Factor must be an int")
            return None

        if (self.getRows() % factor != 0):
            print ("Number of rows {} not divisible by factor {}".format(self.getRows(), factor))
            return None
        if (self.getCols() % factor != 0):
            print ("Number of cols {} not divisible by factor {}".format(self.getCols(), factor))
            return None


        newRowNum = self.getRows() // factor
        newColNum = self.getCols() // factor
        newdata = np.zeros([newRowNum, newColNum])

        for i in range(newRowNum):
            for j in range(newColNum):
                sumCellValue = 0.0

                for k in range(factor):
                    for l in range(factor):
                        sumCellValue += self._data[i*factor + k, j*factor + l]

                newdata[i,j] = sumCellValue / factor / factor + 100

        return Raster(newdata, self._orgs[0], self._orgs[1], self._cellsize*factor)

```

RasterHandler.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 31 01:00:00 2013

@author: nrjh
"""
import numpy as np
from Raster import Raster
import random
import math

def readRaster(fileName):
    """ Generates a raster object from a ARC-INFO ascii format file"""

    lines = []
    myFile=open(fileName,'r')

    end_header=False
    xll=0.
    yll=0.
    nodata=-999.999
    cellsize=1.0

    while (not end_header):
        line=myFile.readline()
        items=line.split()
        keyword=items[0].lower()
        value=items[1]
        if (keyword=='ncols'):
            ncols=int(value)
        elif (keyword=='nrows'):
            nrows=int(value)
        elif (keyword=='xllcorner'):
            xll=float(value)
        elif (keyword=='yllcorner'):
            yll=float(value)
        elif (keyword=='nodata_value'):
```

```
            nodata=float(value)
        elif (keyword=='cellsize'):
            cellsize=float(value)
        else:
            end_header=True

    if (nrows==None or ncols==None):
        print ("Row or Column size not specified for Raster file read")
        return None


    items=line.split()


    datarows=[]
    items=line.split()
    row=[]
    for item in items:
        row.append(float(item))


    datarows.append(row)

    for line in myFile.readlines():
        lines.append(line)
        items=line.split()
        row=[]
        for item in items:
            row.append(float(item))

        datarows.append(row)

    data=np.array(datarows)

    return Raster(data,xll,yll,cellsize,nodata)


#def createRanRaster(rows=25,cols-25,cellsize=1,xorg=0,yorg=0,nodata=-
999.999,levels=1,datahi=0.,datalo=0.:
def createRanRaster(rows=20,cols=30,cellsize=1,xorg=0,yorg=0,nodata=-
999.999,levels=5,datahi=100.,datalo=0.):

   #print (rows,cols,levels)

   levels=min(levels,rows)
   levels=min(levels,cols)
   data=np.zeros([levels,rows,cols])
   dataout=np.zeros([rows,cols])

   for x in np.nditer(data,op_flags=['readwrite']):
       x[...]=random.uniform(datalo,datahi)
   #print data



   for i in range(levels):
       lin=((i)*2)+1
       lin2=lin*lin
       #print (lin,lin2)
       iterator=np.zeros([lin2,2], dtype=int)
       for itx in range(lin):
           for ity in range(lin):
               iterator[itx*lin+ity,0]=(itx-i)
               iterator[itx*lin+ity,1]=(ity-i)
       #print iterator


       part=data[i]

```

```
        new=np.zeros([rows,cols])
        for j in range(rows):
            for k in range(cols):
                for it in range(lin2):
                        r=(j+iterator[it,0])%rows
                        c=(k+iterator[it,1])%cols
                        #print (i,j,k,r,c)
                        new[j,k]=new[j,k]+part[r,c]

        minval=np.min(new)
        maxval=np.max(new)
        ran=maxval-minval
        data[i]=((new-minval)/ran)*(2**i)
        #print data[i]

        dataout=dataout+data[i]

    minval=np.min(dataout)
    maxval=np.max(dataout)
    ran=maxval-minval
    datarange=datahi-datalo
    dataout=(((dataout-minval)/ran)*(datarange))+datalo
    return Raster(dataout,xorg,yorg,cellsize,nodata)

def createRanRasterSlope(rows=20,cols=30,cellsize=1,xorg=0,yorg=0,nodata=-999.999,levels=5,datahi=100.,datalo=0.,focusx=None,focusy=None,ranpart=0.5):

    if (focusx==None):
        focusx=cols/2
    if (focusy==None):
        focusy=rows/2

    rast=createRanRaster(rows,cols,cellsize,xorg,yorg,nodata,levels,1.,0.)

    slope_data=np.zeros([rows,cols])
    maxdist=math.sqrt(rows*rows+cols*cols)

    for i in range(rows):
        for j in range(cols):
            xd=focusx-j
            yd=focusy-i
            dist=maxdist-math.sqrt((xd*xd)+(yd*yd))
            slope_data[i,j]=dist/maxdist

    minval=np.min(slope_data)
    maxval=np.max(slope_data)
    ran=maxval-minval

    slope_data=((slope_data-minval)/ran)

    ran_data=rast.getData()

    data_out=slope_data*(1.-ranpart)+ran_data*(ranpart)
    minval=np.min(data_out)
    maxval=np.max(data_out)
    ran=maxval-minval
    datarange=datahi-datalo
    data_out=(((data_out-minval)/ran)*datarange)+datalo

    return Raster(data_out,xorg,yorg,cellsize)
```