Exam Number: B119374

Python Hydrological Model

OBJECT ORIENTATED SOFTWARE ENGINEERING

SPATIAL ALGORITHMS

INDEX

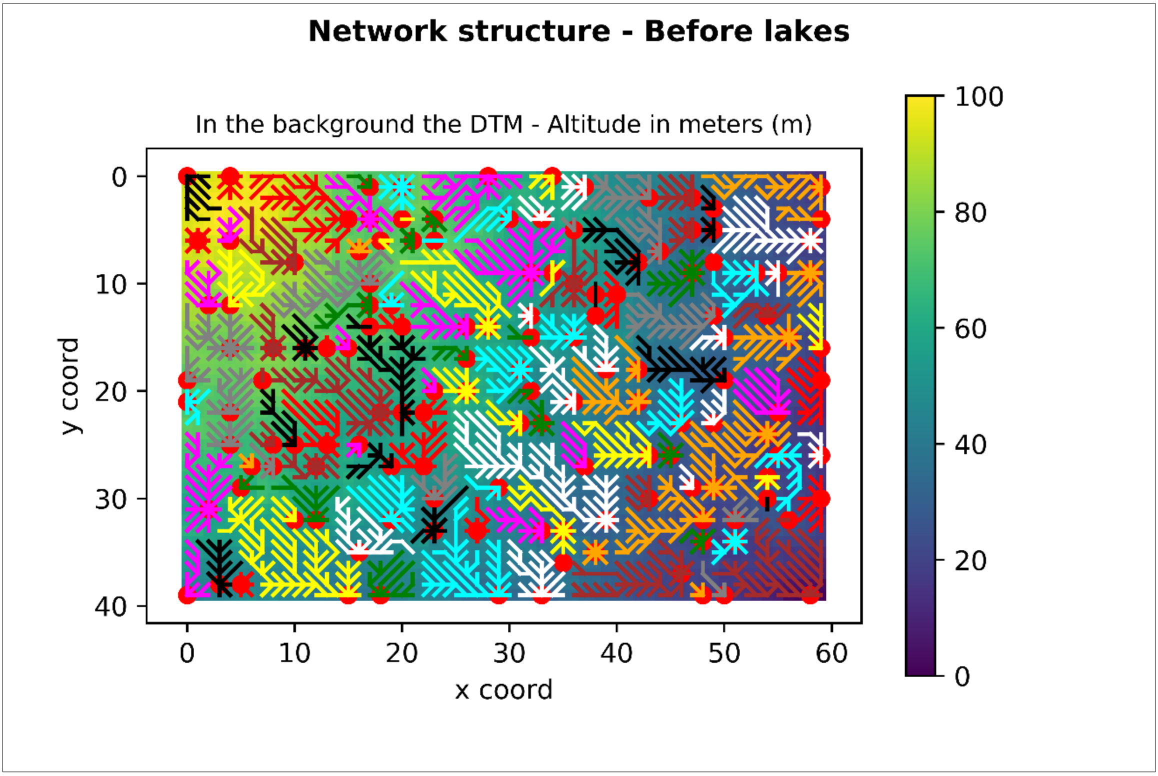
INTRODUCTION

The present report explains the functioning of an algorithm that calculates a hydrological model, starting from a Digital Elevation model and an annual Rainfall data raster.

TASK 1

The algorithm is showing a fragmented flow network. At this stage, the catchment cells called “pits” are displayed in red. Per each one of these cells is displayed the drainage basin. Different colours distinguish different basins; colours are assign with the method **plotstreams** called by **plotFlowNetwork**.
To get this the algorithm starts from the Digital Elevation Model, applying the **FlowRaster** method from the **Flow.py** module. The **FlowRaster** method generates an array with the same size of the original DTM in which each cell is represented by a Point2D element (FlowNode).
When creating the flownodes grid (from now on named **fr**) the method **setDownCells** is called. Per each flownode this method set the down cell, which uses in turn **lowestNeighbour** and **getNeighbours**. **SetDownCells** is also calling the method **setDownnode,** which both check the correctness of the relationship upnode-downnode but also compile the list of upnodes per each cell. The result is displayed in Figure 1.

Figure 1 – output for Task 1.



Code additions – Task 1

Few additions were made to the code to complete Task 1.

1) In **Flow.py** was imported the **Raster** method from the **Raster.py** module with the following code:

```
### TASK 1 ###
# Import class Raster from Module Raster.py
from Raster import Raster
```

2) It was added a method **setDownCells** in **Flow.py**:

```
def setDownCells(self):
    ''' Look for the lowest neighbour around the cell and set the
    downnodes '''
    # Loop through the raster of Point2D
    for r in range(self.getRows()):
        for c in range(self.getCols()):
            # Get the lowest neighbour
            lowestN = self.lowestNeighbour(r,c)
            # if its elevation is LOWER than the one of the center cell
            if (lowestN.getElevation() < self._data[r,c].getElevation()):
                # Set the DownNode of that Point2D as the lowestN
                self._data[r,c].setDownnode(lowestN)
            else:
                # if the center cell is a Pitflag than DownNode is NONE
                self._data[r,c].setDownnode(None)
                ### Added trying to do task 4 ###
                self._data[r,c].setPitFlag(True)
```

3) Uncommented the following lines in **CourseWork1.py**:

```
################ Step 1 find and plot the initial network #######
# Create the FlowRaster
fr=flow.FlowRaster(resampledElevations)
# Plot the network between points #
plotFlowNetwork(elevation, fr, "Network structure - before lakes", plotLakes=False)
```

TASK 2

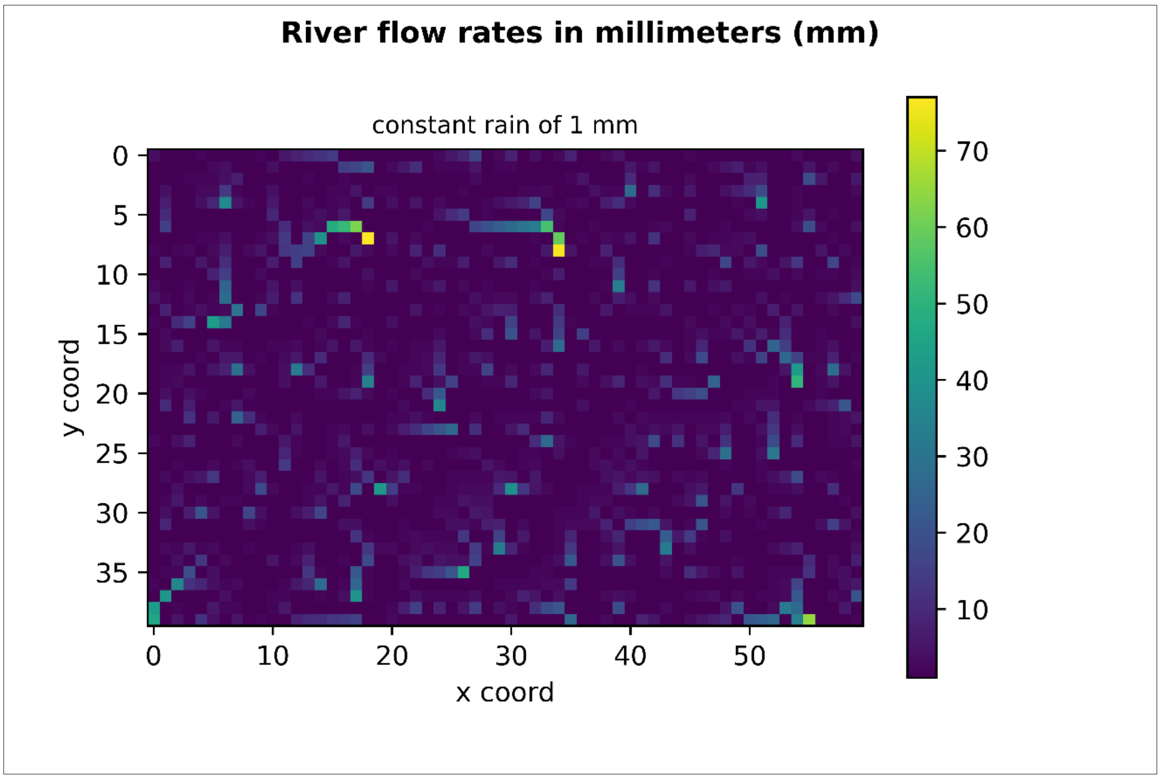
In the module **CourseWork1.py** is called the method **plotExtractedData**, which accepts as arguments (1) a raster and (2) an extractor function.

In this case, the raster is **fr,** the grid of flownodes previously created, and the function is the class **Flow.FlowExtractor**, which contains the method **getValue**. The method **getValue** returns the flow of the node under examination, calling the method **getFlow** located in the class **FlowNodes**. This last method uses the recursion to calculate the total flow of each cell summing the rain in the cell and the flow coming from the upnodes (recursion).

The recursive method **getFlow** was built immediately to be adaptable for task 3. To test it with one mm of rain the attribute **self._rain** was set to 1.

In addition, it was added a method **printFlow** in the class **FlowRaster** to create an array of flow values and crosscheck the correctness of the algorithm.

Figure 2 – Output Task 2.



Code additions – Task 2

1) Uncommented the following lines in **CourseWork1.py**:

```
################ Step 2 ########################################
# Plot the extrated data - the second arg is the extractor function
plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - constant rain")
# Check # Get an array of flow values
print(fr.printFlow())
```

2) Added the following attributes to the class **FlowNodes**:

```
self._rain=1
self._flow=0
```

3) Add the method **GetValues** to the class **FlowExtractor**:

```
class FlowExtractor():

  def getValue(self, node):
    ''' Method that extract the flow value from a specific node
    Args: Array of flownodes, the node of interest'''
    return node.getFlow()
```

4) Create the method **GetFlow** in the **FlowNode** class:

```
def getFlow(self):
    ''' Method that adds the rain to the flownodes using a recursive process
    Arg: the array of flownodes '''

    upnodes = self.getUpnodes()
    # stores the initial flow
    self._flow = self.getRain()

    # if there are no upnodes return the flow
    if upnodes == []:
        return self._flow
    # if there are more upnodes calculate the flow for these (recursion)
    else:
        for node in upnodes:
            self._flow = self._flow + node.getFlow()
    # return the flow
    return self._flow
```

5) Create the method **printFlow** in the **FlowRaster** Class:

```
def printFlow (self):
    ''' prints the flow value for each cell '''

    flowdata = []
    for r in range(self.getRows()):
        for c in range(self.getCols()):

            flowdata.append(self._data[r,c].getFlow2())
    valuesarray=np.array(flowdata)
    valuesarray.shape=self._data.shape
    return valuesarray
```

6) Create auxiliary methods in **FlowNode** Class:

```
def getFlow2(self):
    '''Method that returns the value of the flow'''
    return self._flow

def getRain(self):
    return self._rain
```

TASK 3

The method **addRainfall** is called on **fr** in the module **CourseWork1.py**.

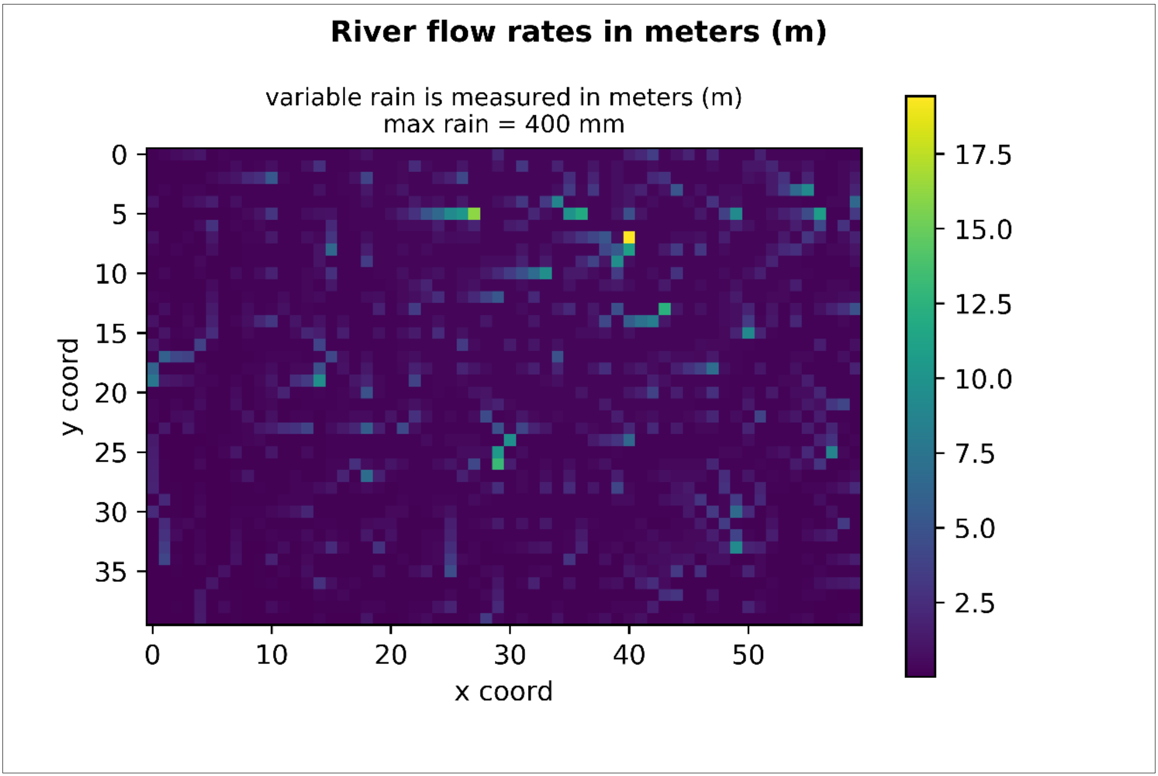
The method **addRainfall** was built in the class **FlowRaster** and accepts as arguments **fr** itself and the method **getData** applied to the **rainrasterA**, introduced as the argument "rain" in the function **calculateFlowsAndPlot.**

```
### RAIN RANDOM DATA ###
fr.addRainfall(rain.getData())
```

The **rainrasterA** is a randomly generated raster that represent random precipitations. Different parameters can be changed. In this case, the maximum rain has been set at 400 mm (0.4 m). At first the method **addRainfall** checks that the array of rain values and **fr** are of the same size using the "assert" statement. Following this the method goes over all the cells of **fr** one by one, retrieving the coordinates [r,c]. While vising a specific cell in **fr** the method visits a cell in the same position in the rain raster (same coordinates [r,c]), retrieving the rain value and inserting this in the flownode attribute **self._rain**. The initial value of zero in **self._rain** is overwritten with the new one through the method **setRain**.

In addition, it was added a method **printRain** in the class **FlowRaster** to create an array of rain values and crosscheck the correctness of the algorithm.

Figure 3 - Output Task 3.



Code additions – Task 3

1) Uncommented the following lines in CourseWork1.py:

```
################# Step 3 ########################################
### RAIN RANDOM DATA ###
fr.addRainfall(rain.getData())
plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall")

# Check # Get an array of flow values
print(fr.printRain()) # returns the values of the rain loaded in the flowraster
```

2) Added the method AddRain:

```
### TASK 3 ###
def addRainfall (self,rainArray):
   ''' method that traverses the rainfall raster and load the values in self._rain in meters
   Args: flownode raster, array with rain values'''
   # Ensure rain and data have the same size
   assert self._data.shape == rainArray.shape
  # print (self._data.shape)
  # print (rainArray.shape)
   rainValue = 0
   for r in range(self.getRows()):
      for c in range(self.getCols()):
         rainValue = rainArray[r,c]
         self._data[r,c].setRain(rainValue/1000)
```

3) Added the auxiliary method setRain:

```
   def setRain(self, value):
      self._rain = value
```

TASK 4

The method **calculateLakes** processes the initial fragmented flow network to obtain a final hydrological model. To achieve this the method starts calculating all the natural 'depression' that are lakes identifying (1) the flownodes part of each lake (2) the flownode where the lake discharges the outflow (3) the highest wall before the sea and according to this (4) the depth of the lake.

The algorithm first orders the pits from the highest to the lowest, and in this order start identifying the lakes. All the pits included in one lake will not be considered while creating the next one. Once all the lakes are completed each cell of the lake is linked (setDownNode) with the first lower cell after the highest wall (the highest wall is the last cell of the lake). The depth of the lake is calculated against the highest wall of the lake.

To achieve this the method uses a new class Lake, plus other new methods and attributes in the class FlowNode. All the sessions added for Task 4 are clearly tagged in the code in the appendix. The code is also properly commented.

Figure 4 - Output 1 Task 4 Network structure – With lakes

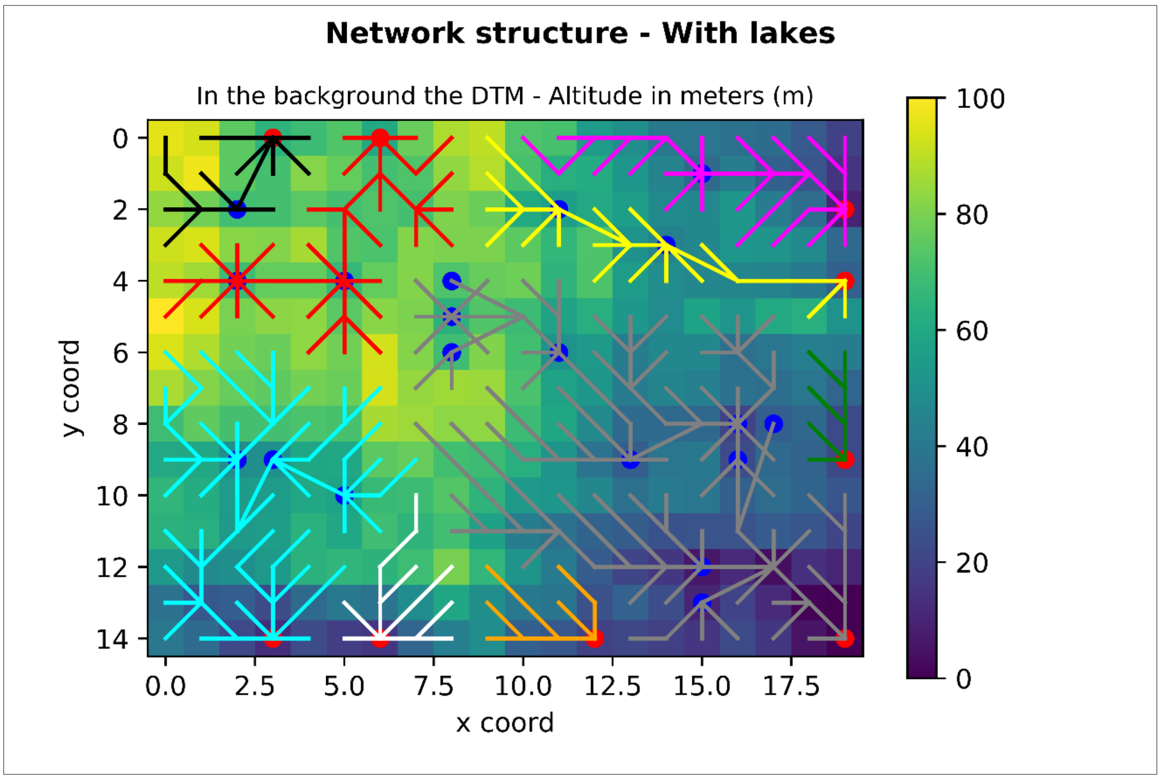


Figure 5 - Output 2 Task 4 Flow rates

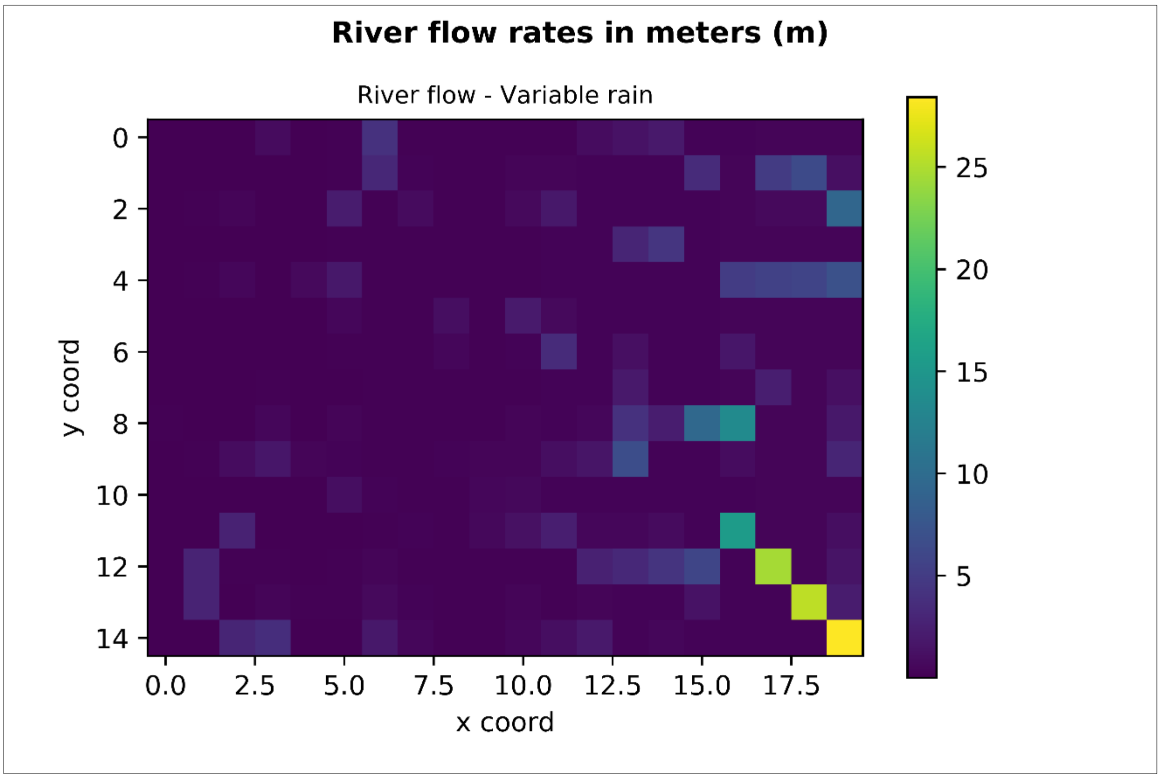
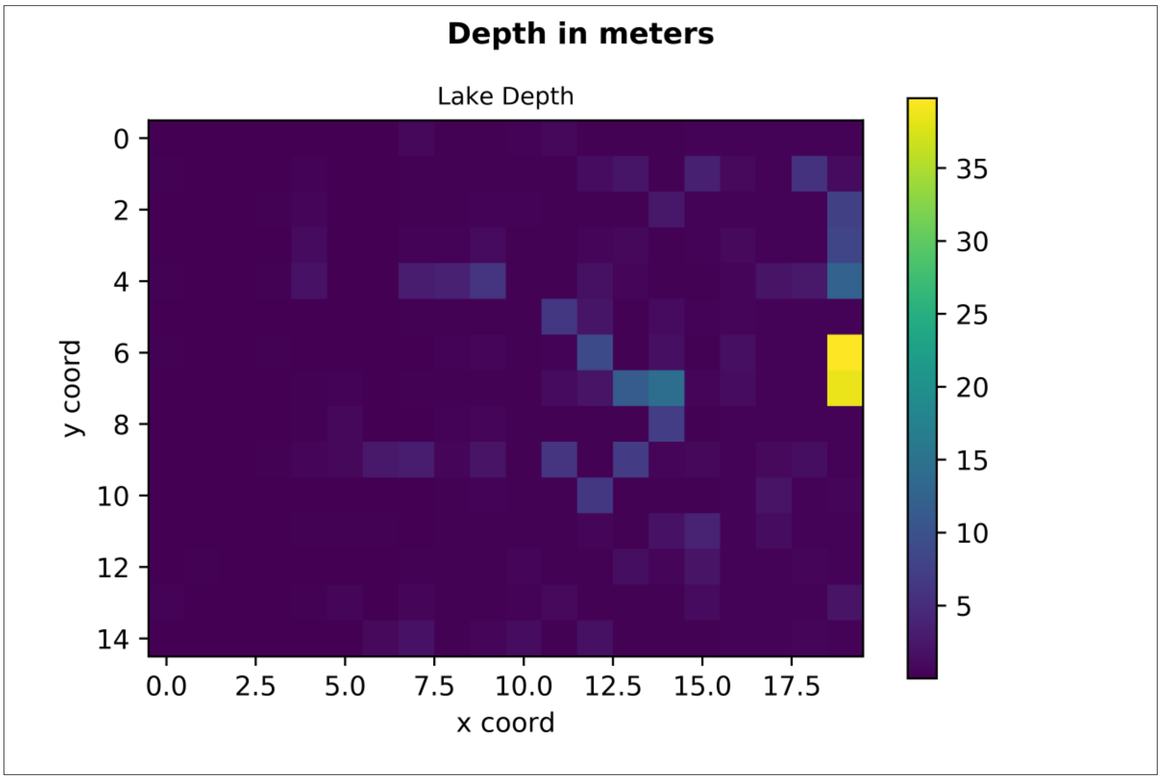


Figure 6 - Output 3 Task 4 Lake Depth
NB: it does not match with the previous figures because it was exported in a separate session



Pseudo code – Method calculate Lakes

No pseudo code for additional methods – please refer to code at the end of the docuemnt.

```
Create method calculateLakes:
  - Create a list of Pits ordered highest to lowest
  - For each pit in the list
    - if the Pit has not been marked as lake
      - Set the attribute Lake as True
      - making the pit as part of a lake

      - create instance of the class Lake
      - add lake to a list of lakes (attrib of class FlowRaster)
      - add the Pit to a list of lake nodes (attrib of class Lake)
      - get coordinates of the cell
      - while both the cell is not at the margin at is a pit keep looping (while loop)
        - set lownode as none
        - go over all the flownodes part of the lake
          - get the coordinates of the flownode in the lake
          - find the lowest neighbour to the cell - ! lowestNeighbour method has been modified !

          - memorize the lowest neighbour
        - Set the Flag True for the lowest
        - Set the depth to one
        - Append the cell to the list
      - Get the high wall after the lake
      - create two lists to split the lake according to the high wall
      - select bad half of flownodes after high wall

      - select good half of flownodes before high wall
      - High wall included in the good half
      - Turns the attribute Lake to false for the badhalf
      - keep only the nodes before the high wall included (this is the lake)
      - Set the attributes down node and high wall
      - append the lake to a list of lakes
  - go over each element in the list of lakes
    - Go over the cells in each lake
      - Set Lake depth
      - Set the new downNode
```

TASK 5

Since the original raster had a resolution that was to fine (530 * 830 = 439900 cells) it was necessary to resample it. This is done modifying the method **createWithIncreasedCellsize** in **Raster.py.**

Figure 7 – Originael Raster Task 5

```
Original elevation (m), shape is  (530, 830)
```

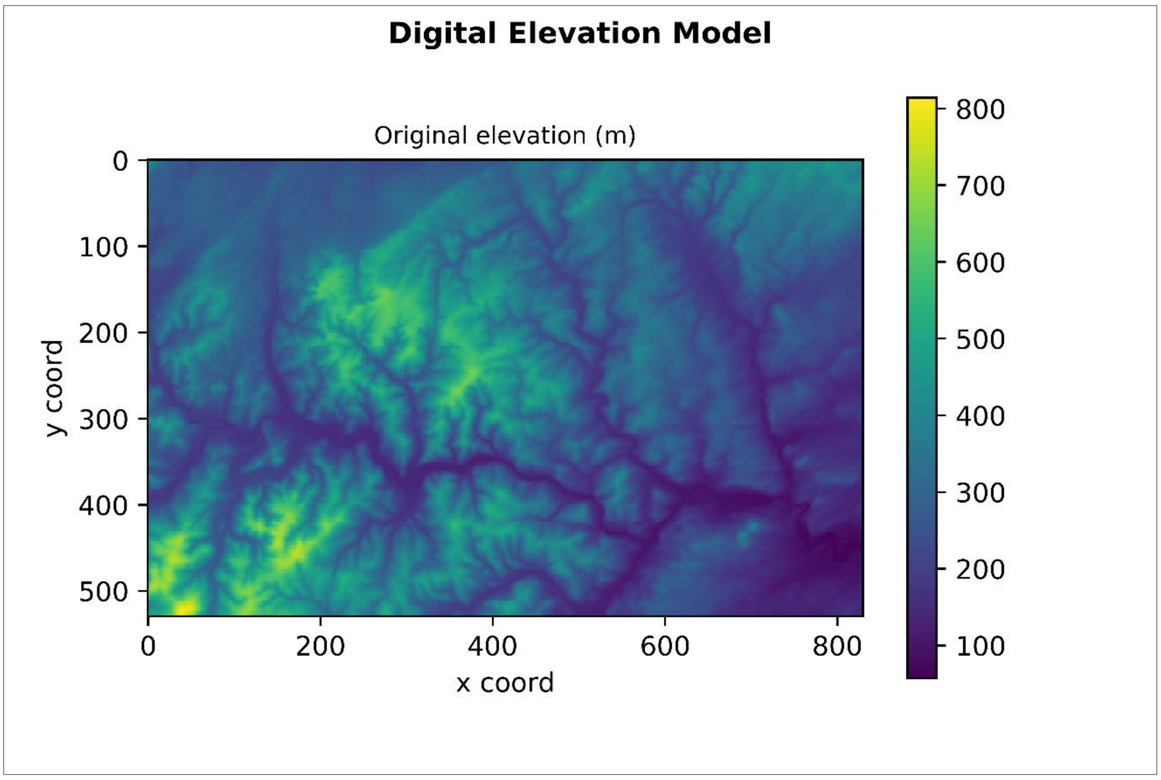
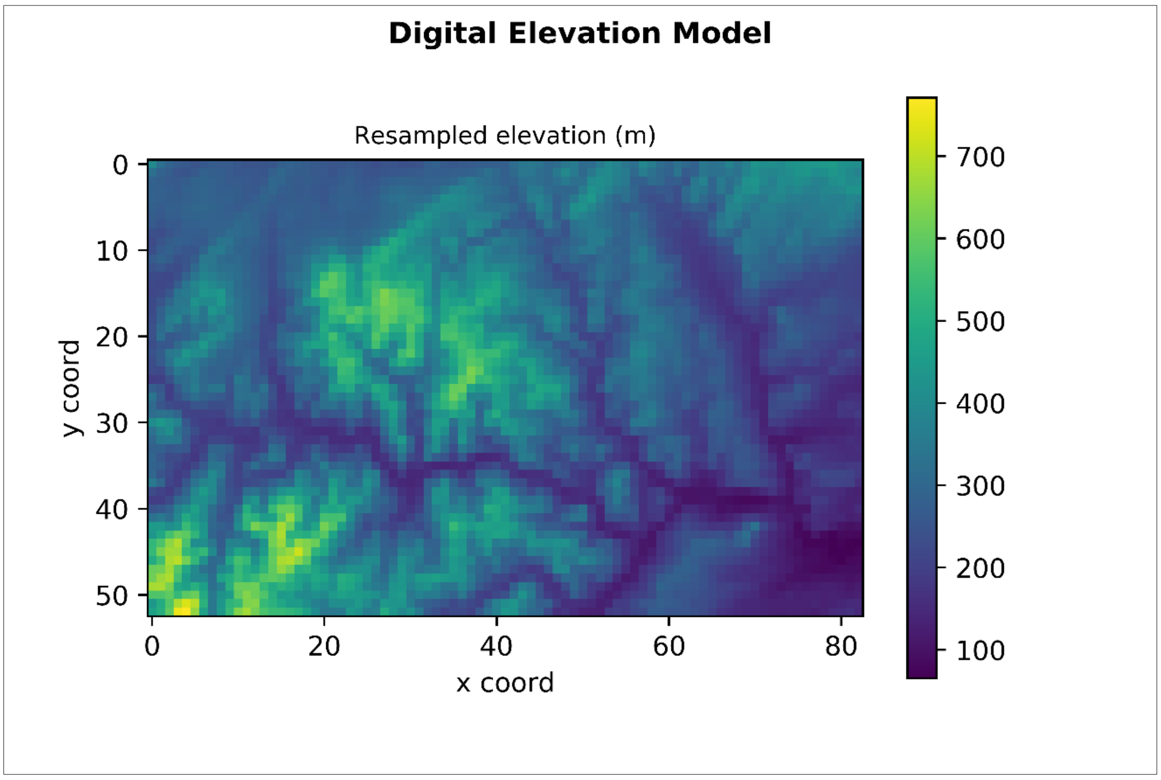


Figure 8 - Resampled Raster Task 5

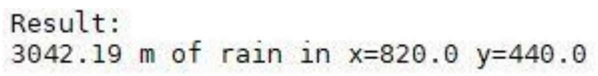
```
Resampled elevation (m), shape is  (53, 83)
```



The maximum flow rate in the DEM is of 3042.19 m of rain/ year.

The location of this flow is at the geographical location x = 820 and y = 440, which corresponds to the cell x = 82 and y = 44 of the resampled raster.

Figure 9 - Final Output Task 5



```
Result:
3042.19 m of rain in x=820.0 y=440.0
```

Figure 10 - Output 1 Task 5 Network structure – With lakes

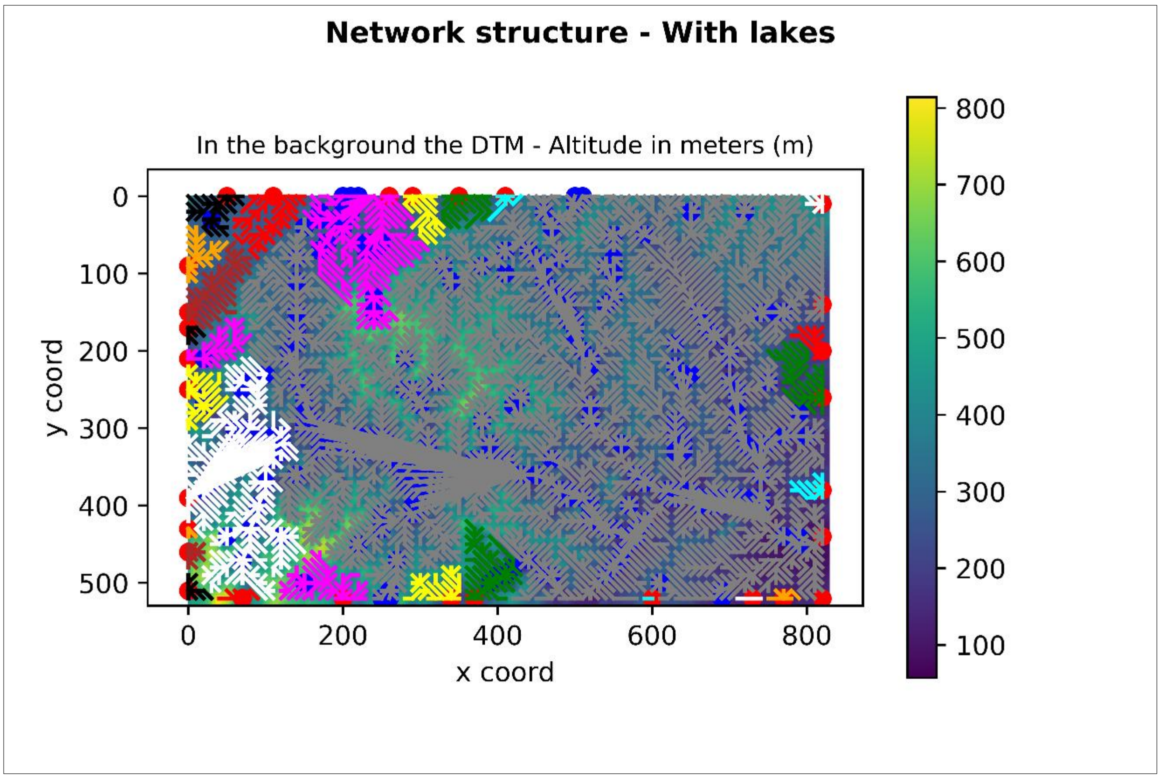


Figure 11 - Output 2 Task 5 Flow rates

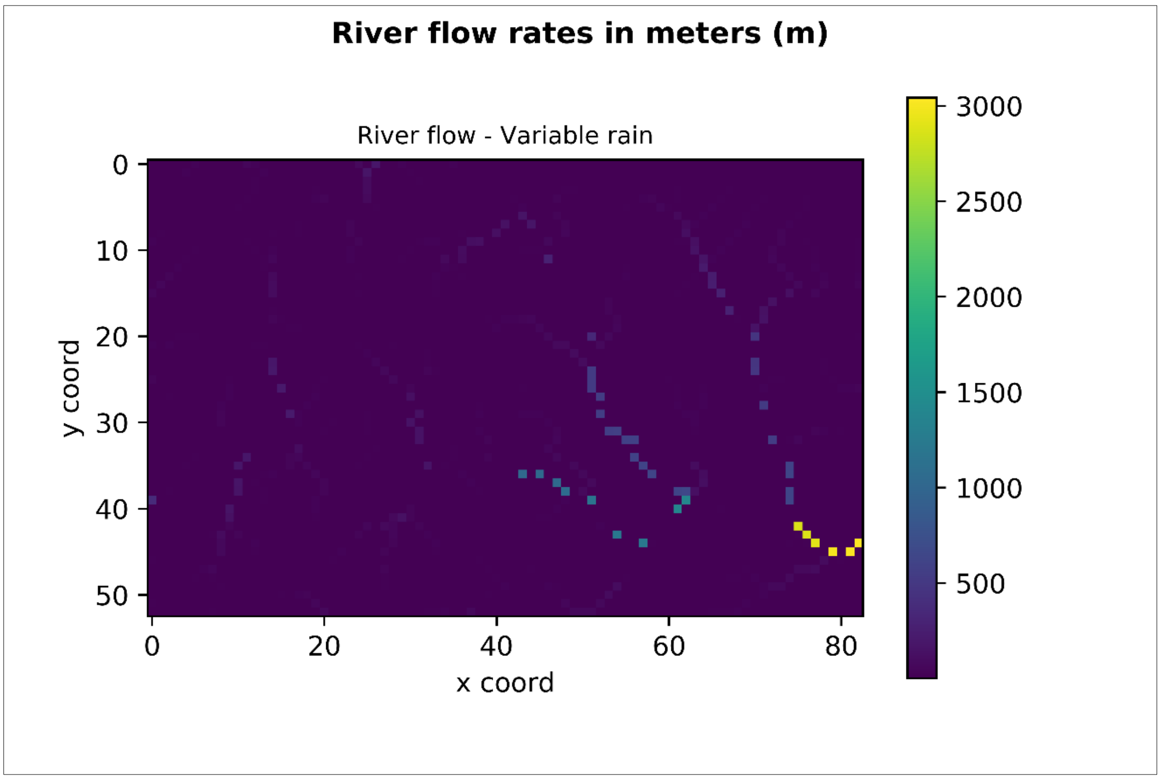
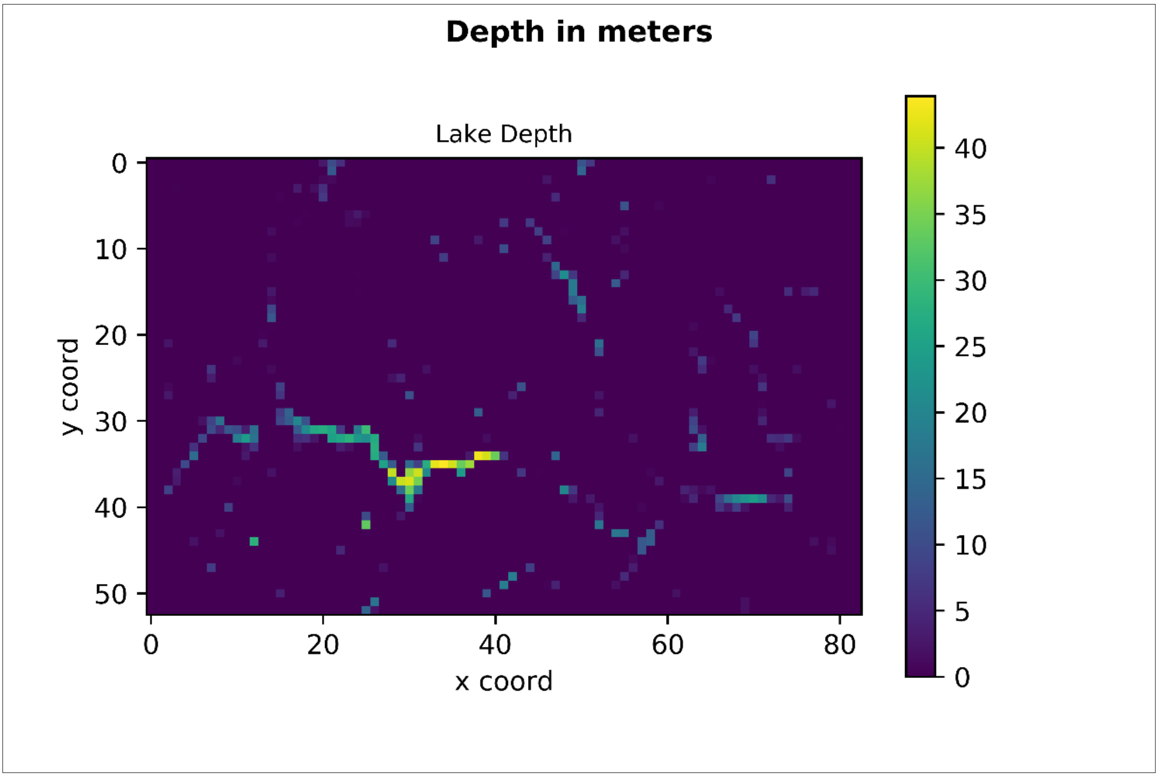


Figure 12 - Output 3 Task 5 Lake Depth



Code additions – Task 5

1) Uncommented/Commented the following lines in CourseWork1.py:

```
############# step 4 and step 5 #########################################
## handle lakes # - orig
fr.calculateLakes() # - orig
#print(fr.printFlow())
plotFlowNetwork(elevation, fr, "Network structure (i.e. watersheds) - with lakes") # - orig
plotExtractedData(fr, flow.LakeDepthExtractor(), "Lake depth") # - orig

plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall") # - orig
#print(fr.printFlow())

### Task 5 ###
nomi = fr.finalCell() #finds the cell with the highest value and it's coordinate
print (type(nomi))
print ("Result: \n{} m of rain in x={} y={}".format(nomi[0],nomi[1],nomi[2]))
```

2) And:

```
# Swap the Rasters
#calculateFlowsAndPlot(elevationRasterA, rainrasterA, resampleFactorA)

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

3) Resample method in Raster.py:

```
def createWithIncreasedCellsize(self, factor):
    ''' Methid that resamples the raster according to a resample factor
    Args: rater(self) and factor'''
    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

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

4) Methods to find the final cell and the maximum flow

```
### TASK 5 ###

def finalCell (self):
    ''' Method that returns the Max flow and the position of the cell
    Arg: flownode raster'''

    listPits = self.getPits2() #list of flownodes PITS ordered from the highest to the lowest

    lastPit = round(listPits[-1].getFlow2(),2) # get the flow
    cellCoordX = listPits[-1].get_x() # get the coordinates
    cellCoordY = listPits[-1].get_y() # get the coordinates

    return lastPit,cellCoordX,cellCoordY
    #print ("\n{} m of rain in x={} y={}".format(lastPit,cellCoordX,cellCoordY))

def getPits2 (self):
    ''' Method that identifies the pits on the boundary and return a list
    Arg: flownode raster'''
    listPits3 = []
    listPits4 = []

    for r in range(self.getRows()):
        for c in range(self.getCols()):

            pitFlag = self._data[r,c].getPitFlag() # checking if is a pitflag

            if pitFlag==True and len(self.getNeighbours(r,c)) < 8 : # ok - selecting only the pits that are on the border

                listPits3.append(self._data[r,c]) # creates a list with all the outer nodes

    # sort the list with an inbuilt function
    listPits4 = sorted(listPits3, key=lambda FlowNode: FlowNode.getFlow2(), reverse=False)

    return listPits4
```

OVERALL DISCUSSION

The developed code overall fulfils its task of calculating the final volume of water that every year passes through the point of maximum outflow. However, have been identified some limitations that can be turned in further developments.

Lakes

The algorithm correctly identifies the water bodies according to the highest wall that can stop the water before the sea. However, at this stage each cell of the lake directly connects with the outflow cell of the lake, generating direct links. This is technically not correct, since the water should move from one cell to the other. This particular also visually affects the final flow output, in which the water appears to jump over the raster. Possible solutions would be to: (1) starting from the outflow cell proceed upstream linking all the cells' upnodes; (2) less elegantly playing with the attributes and assign a fixed flow value to all the cells of the lake. This flow value should be equal to the lake outflow, so that the cells will appear as a smooth surface in the plot. However, this solution implies the modifications of other methods to avoid errors when calling **flow.FlowExtractor()**.

Outflow Cells selection

At the boundaries of the raster the algorithm looks for a Pit, considering it the point of outflow. Of course here there is an obvious assumption that the water does not go straight "in the sea".

Other outflows

At this stage the algorithm correctly identifies the final outflow cells with the highest annual flow. However, it does not detect the other outflow cells. This means that other parts of the hydrological network remain unknown. A possible solution would be to change the **finalCell** method so that it returns all the outflow points, with their annual flow, plus a calculation of the total outflow, which should match with the total annual rain. This will give a more complete picture of the hydrological network of the DTM.

Additional Datasets

A more complete and realistic model should also consider other factors that might influence the final outflow of the water.

Seasons and snow

In case the calculation is not done on a year basis but with a higher frequency it is necessary to take into account the seasonality. If the DTM reaches altitudes where there is snow accumulation, or it is located at high latitudes where there is a considerable accumulation of snow, this needs to be taken into account. The accumulation of the snow (which counts as precipitation) delays the outflow which will be higher in the spring. This dataset can be represented by a raster with values that represent the rate of snow accumulation (es: at sea level accumulation = 0, at 8000m accumulation = 1). Appropriate methods and attributes will after handle and combine this raster with the rain raster, taking into account seasonality, snow accumulation and discharge delay.

Land cover

The land cover is an important factor to take into account. Aside from the run off speed, which in a seasonal or yearly model does not count much, the land cover can considerably influence the amount of water that reaches the water bodies. This is because different types of vegetation have different water needs.

Soil

Strictly linked with the land cover and vegetation information the type of soil is also important, because it can influence the infiltration of the water (es: clay infiltration = 0, sand infiltration = 1). However, the introduction of this variable adds another dimension to the model, because it starts looking at the underground circulation. It is not said that the water that that percolates is “lost”. Normally water bodies such as lakes are recharged (and discharged) also through deep circulation. However high levels of infiltrations might alters the immediate/quick flow, since the movement of the underground water is slower that surface water.

Final considerations

In general the model does not take into account groundwater circulation, however, starting from the assumption that the lakes are full at the beginning of the iteration we can also consider the groundwater system as in equilibrium (saturated). In addition we did not consider any human interference with the model (e.s. agriculture activities pumping water from lakes or human infrastructures such as dams).

So concluding a model mimicking more the reality would be extremely more complex, however, the scope of the models are to simplify the reality, trying to represent it in a simple manner. The important thing is to be aware of the assumptions made and communicate these to the client/users.

Code

CourseWork1

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


def plotstreams(flownode,colour):
    ''' Retrieve the upnodes of a point and plot the network '''
    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):
    ''' Plot the flow network. Args: backgrond raster, flow raster, title '''
    print ("\n\n{}".format(title))
    mp.imshow(originalRaster._data)
    mp.colorbar()
    colouri=-1
    colours=["black","red","magenta","yellow","green","cyan","white","orange","grey","brown"]

    for i in range(flowRaster.getRows()):
        for j in range(flowRaster.getCols()):
            node = flowRaster._data[i,j]
            # if the point is a pitflag it calls the plotstream

            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")

    # Code to output the figure with good resolution and captions
#   mp.title('In the background the DTM - Altitude in meters (m)', fontsize=9)
#   mp.suptitle('Network structure - With lakes', fontsize=11, fontweight='bold')
#   mp.xlabel('x coord')
#   mp.ylabel('y coord')
#   mp.savefig('./Network_structure3.png', dpi=900)

    mp.show()

def plotExtractedData(flowRaster, extractor, title=""):
    ''' Plots the flow rate
    args: flow Raster, Extraction function '''
    print ("\n\n{}".format(title))
    mp.imshow(flowRaster.extractValues(extractor))
    mp.colorbar()
    # Code to output the figure with good resolution and captions
#   mp.title('Lake Depth', fontsize=9)
    mp.title('River flow - Variable rain', fontsize=9)
    mp.suptitle('River flow rates in meters (m)', fontsize=11, fontweight='bold')
#   mp.suptitle('Depth in meters', fontsize=11, fontweight='bold')
    mp.xlabel('x coord')
    mp.ylabel('y coord')
#   mp.savefig('./FlowRates_constrain1.png', dpi=900)
#   mp.savefig('./FlowRates_variable1.png', dpi=900)
    mp.savefig('./RiverFlow_task52.png', dpi=900)

#
    mp.show()

def plotRaster(araster, title=""):
    ''' Plot a Raster. Args: raster, title '''
    print ("\n\n{}, shape is  {}".format(title, araster.shape))
    mp.imshow(araster)
    mp.colorbar()
#   mp.title('Original elevation (m)', fontsize=9)
#   mp.title('Resampled elevation (m)', fontsize=9)
#   mp.suptitle('Digital Elevation Model', fontsize=11, fontweight='bold')
#   mp.xlabel('x coord')
#   mp.ylabel('y coord')
#   mp.savefig('./OriginalDTM_task5.png', dpi=900)
#   mp.savefig('./ResampleDTM_task5.png', dpi=900)
#   mp.show()


def calculateFlowsAndPlot(elevation, rain, resampleF):
    ###  PLOT INPUT RASTER ----
#   plotRaster(elevation.getData(), "Original elevation (m)")
#   plotRaster(rain.getData(), "Rainfall")
    resampledElevations = elevation.createWithIncreasedCellsize(resampleF)
#   plotRaster(resampledElevations.getData(), "Resampled elevation (m)")
    ################# Step 1 find and plot the intial network #######
    # Create the FlowRaster
    fr=flow.FlowRaster(resampledElevations)
    # Plot the network between points #
#   plotFlowNetwork(elevation, fr, "Network structure - before lakes", plotLakes=False)

    ################ Step 2 #####################################
    # Plot the extrated data - the second arg is the extractor function
#   plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - constant rain")

    # Check # Get an array of flow values
#   print(fr.printFlow()) # returns the values of the rain loaded in the flowraster

    ################# Step 3 ########################################
    ### RAIN RANDOM DATA ###
    fr.addRainfall(rain.getData())
#   plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall")

    # Check # Get an array of flow values
#   print(fr.printRain()) # returns the values of the rain loaded in the flowraster
```

```
  ############# step 4 and step 5 ########################################
   ## handle lakes # - orig
   fr.calculateLakes() # - orig
   #print(fr.printFlow())
   plotFlowNetwork(elevation, fr, "Network structure (i.e. watersheds) - with lakes") # - orig
   plotExtractedData(fr, flow.LakeDepthExtractor(), "Lake depth") # - orig

   plotExtractedData(fr, flow.FlowExtractor(), "River flow rates - variable rainfall") # - orig
   #print(fr.printFlow())

   ### Task 5 ###
   nomi = fr.finalCell() #finds the cell with the highest value and it's coordinate
   print ("Result: \n{} m of rain in x={} y={}".format(nomi[0],nomi[1],nomi[2]))

############# step 1 to 4 ########################################
# Create Random Raster
rows=15
cols=20
xorg=0.
yorg=0.
xp=1 #100
yp=1 #100
nodata=-999.999
cellsize=1.
levels=4
datahi=100.
datalow=0
randpercent=0.4

resampleFactorA = 1
# Create random Elevation Raster
elevationRasterA=createRanRasterSlope(rows,cols,cellsize,xorg,yorg,nodata,levels,datahi,datalow,xp,yp,randpercent)
# Create random Rain Raster
rainrasterA=createRanRasterSlope(rows//resampleFactorA,cols//resampleFactorA,cellsize*resampleFactorA,xorg,yorg,nodata,levels,
400,1,36,4,.1)
# Swap the Rasters
#calculateFlowsAndPlot(elevationRasterA, rainrasterA, resampleFactorA)

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

Flow

```
import numpy as np

from Points import Point2D
### TASK 1 ###
# Import class Raster from Module Raster.py
from Raster import Raster

class FlowNode(Point2D):
    ''' FlowNode is an instance of the Point2D Class '''
    def __init__(self,x,y, value):
        Point2D.__init__(self,x,y)
        self._downnode=None
        self._upnodes=[]
        self._pitflag=False
        self._value=value
        self._rain=0
        self._flow=0
        self._lakeDepth=0
        self._lakeFlag=False

    def __repr__(self):
        return 'x{} y{} z{}'.format(self.get_x(),self.get_y(),self.getElevation())


    def setDownnode(self, newDownNode):
        ''' ensures that there are no provblems when setting a down node'''

        self._pitflag=(newDownNode==None)

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

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

        self._downnode=newDownNode

    # return the downnode
    def getDownnode(self):
        return self._downnode
    # return the upnode
    def getUpnodes(self):
        return self._upnodes
    # remove one upnode - I need ot specify which node
    def _removedUpnode(self, nodeToRemove):
        self._upnodes.remove(nodeToRemove)
    # add an upnode - append to list
    def _addUpnode(self, nodeToAdd):
        self._upnodes.append(nodeToAdd)
    # number of upnodes
    def numUpnodes(self):
        return len(self._upnodes)
    # Returns True or False according of the pitflag is true or false
    def getPitFlag(self):
```

```
    return self._pitflag

# Get the other atribute, which in this case is elevation
def getElevation(self):
    return self._value
# print the coordinate of itself
def __str__(self):
    return self.__repr__()

### TASK 2 ###

def getFlow(self):
    ''' Method that adds the rain to the flownodes using a recursive process
    Arg: the array of flownodes '''

    upnodes = self.getUpnodes()
    self._flow = self.getRain()

    if upnodes == []:
        return self._flow
    else:
        for node in upnodes:
            self._flow = self._flow + node.getFlow()

    return self._flow

def getFlow2(self):
    '''Method that returns the value of the flow'''
    return self._flow


def getRain(self):
    return self._rain

### TASK 3 ###

def setRain(self, value):
    self._rain = value

### TASK 4 ###
def setLakeDepth(self, value):
    self._lakeDepth = value

def getLakeDepth(self):
    return self._lakeDepth

def setLakeFlag(self, value):
    self._lakeFlag = value

def getLakeFlag(self):
    return self._lakeFlag

def setPitFlag(self, value):
    self._pitflag = value

def setLakeOverflow(self, value):
        self._lakeOverflow = value

    def getLakeOverflow(self):
        return self._lakeOverflow

    def setOverFlag(self, value):
        self._overflag = value

    def getOverFlag(self):
        return self._overflag

    def setNewFlow(self, value):
        self._newflow = value

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


class FlowRaster(Raster):
    ''' It creates a raster grid of flow nodes
        Raster stores the values as ARRAY of values
        FlowRastes stores the values as ARRAY of Point2D with elevation data '''
    def __init__(self,araster):
        # inherit the carachteristics of the parent class Raster
        super().__init__(None,araster.getOrgs()[0],araster.getOrgs()[1],araster.getCellsize())
        # extract the data from a Raster as an ARRAY (see Raster.py)
        data = araster.getData()
        # set node as empty list
        nodes=[]
        # loop trough raster and create a grid of flownodes
        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]))

        # array of points2D with elevation
        nodearray=np.array(nodes)
        # gives the nodearray the same shape as the origin raster and load self._data with nodearray
        nodearray.shape=data.shape
        self._data = nodearray
        self._listino=[]
        #initialize with a random lake
        # set the coordinates of the points in the sorounding box
        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):
        ''' Method that get the eigh lowest neighbours '''
        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

    ### modified task 4 ##
    def lowestNeighbour(self,r,c):
        ''' Method that finds the lowest neighbour '''
        # initially set the lownode to none
        lownode=None
        # loops through a list of sorrounding point2D given by getNeighbours
        for neighbour in self.getNeighbours(r,c):
            # if the 1st iteration or the Point2D assessed is lower than the actual one it changes it
            if (lownode==None or neighbour.getElevation() < lownode.getElevation()) :
                if len(self._listino) == 0 or not self._listino[-1].isLake(neighbour) :

                    lownode=neighbour

        return lownode

    def setDownCells(self):
        ''' Look for the lowest neighbour around the cell and set the
        downnodes '''
        # Loop through the raster of Point2D
        for r in range(self.getRows()):
            for c in range(self.getCols()):
                # Get the lowest neighbour
                lowestN = self.lowestNeighbour(r,c)
                # if its elevation is LOWER than the one of the center cell
                if (lowestN.getElevation() < self._data[r,c].getElevation()):
                    # Set the DownNode of that Point2D as the lowestN
                    self._data[r,c].setDownnode(lowestN)
                else:
                    # if the center cell is a Pitflag than DownNode is NONE
                    self._data[r,c].setDownnode(None)
                    ### Added trying to do task 4 ###
                    self._data[r,c].setPitFlag(True)

    def extractValues(self, extractor):
        ''' extract values from the raster according to the extractor function'''
        values=[]
        # goes through the raster
        for i in range(self._data.shape[0]): # columns
            for j in range(self._data.shape[1]): # rows
                # extracts the values of each point
                values.append(extractor.getValue(self._data[i,j]))
        # transform the values in an array and gives to it the same shape as the data
        valuesarray=np.array(values)
        valuesarray.shape=self._data.shape
        return valuesarray

    ### TASK 2 ###
    def printFlow (self):
        ''' prints the flow value for each cell '''
        flowdata = []
        for r in range(self.getRows()):
            for c in range(self.getCols()):

                flowdata.append(self._data[r,c].getFlow2())
        valuesarray=np.array(flowdata)
        valuesarray.shape=self._data.shape
        return valuesarray

    ### TASK 3 ###
    def addRainfall (self,rainArray):
        ''' Method that traverses the rainfall raster and load the values in self._rain in meters
        Args: flownode raster, aray with rain values'''
        # Ensure rain and data have the same size
        assert self._data.shape == rainArray.shape
       # print (self._data.shape)
       # print (rainArray.shape)
        rainValue = 0
        for r in range(self.getRows()):
            for c in range(self.getCols()):
                rainValue = rainArray[r,c]
                self._data[r,c].setRain(rainValue/1000)

    def printRain (self):
        ''' Method that prints the flow value for each cell
        Arg: flownode raster'''
        # TO ADD # Way to check the two aster are the same size
        raindata = []
        for r in range(self.getRows()):
            for c in range(self.getCols()):
                #print(self._data[r,c].getFlow2())
                raindata.append(self._data[r,c].getRain())
        valuesarray=np.array(raindata)
        valuesarray.shape=self._data.shape
        return valuesarray

    ### TASK 4 ###
    def calculateLakes (self):
        ''' Method that calculates the lakes on the DTM linking different pits in bigger water basins
        according to the topography. Arg: flownode raster
        '''
        # Create a list of pits ordered highest to lowest
        ListPits = self.getPits() #list of flownodes PITS ordered from the highest to the lowest
        # For each pit in the list
        for i in ListPits:
            # if the Pit has not been marked as lake
            if i.getLakeFlag() == False:
                # Set the attribute Lake as True
                i.setLakeFlag(True) # making the pit as part of a lake

                mylake = lake() # create instance of the class Lake
                self._listino.append(mylake) # add lake to a list of lakes (attrib of class FlowRaster)
                mylake._lakeNodes.append(i) # add the Pit to a list of lake nodes (attrib of class Lake)

                # get coordinates of the cell
                c = int(mylake._lakeNodes[-1].get_x()/self.getCellsize())
```

```
        r = int(mylake._lakeNodes[-1].get_y()/self.getCellsize())

        # while both the cell is not at the margin at is a pit keep looping (while loop)
        while not ( ( len(self.getNeighbours(int(mylake._lakeNodes[-1].get_y()/self.getCellsize()),int(mylake._lakeNodes[-
1].get_x()/self.getCellsize()))) < 8) and (mylake._lakeNodes[-1].getPitFlag() == True)):

            lownode2 = None # set lownode as none
            for mm in mylake._lakeNodes: # go over all the flownodes part of the lake

                # get the coordinates of the flownode in the lake
                c = int(mm.get_x()/self.getCellsize())
                r = int(mm.get_y()/self.getCellsize())

                # fin the lowest neighbour to the cell
                # ! lowestNeighbour method has been modified
                lastCellNeigh = self.lowestNeighbour(r,c)

                # memorize the lowest neighbour
                if (lownode2==None or (lastCellNeigh != None and lastCellNeigh.getElevation() < lownode2.getElevation())):
                    lownode2=lastCellNeigh


            lownode2.setLakeFlag(True) # Set the Flag True for the lowest
            mylake._lakeNodes[-1].setLakeDepth(1) # Set the depth to one
            mylake._lakeNodes.append(lownode2) # Apend the cell to the list

        # Get the high wall after the lake
        highWall = mylake.explicit()

        # create two lists to split the lake according to the high wall
        badHalf=[]
        goodHalf=[]

        badHalf = mylake._lakeNodes[highWall[-1]+1:] # select bad half after high wall

        goodHalf = mylake._lakeNodes[:highWall[-1]+1] # select good half before high wall

        # High wall included in the good half
        # un-lake the badhalf setting attribute Lake to false
        for m in badHalf:

            m.setLakeFlag(False)
            m.setLakeDepth(0)

        # keep only the nodes before the high wall included
        mylake._lakeNodes = goodHalf

        # Set the attributes down node and high wall
        if len(badHalf) > 0:
            mylake.setLakeDownnode(badHalf[0])
            mylake.setHighWall(goodHalf[-1])

        # append the lake to a list
        self._listino.append(mylake) #this is a list of lakes, not a list of nodes

    # do over each element in the list of lakes
    for l in self._listino:
        # Go over the cells in each lake
        for k in l._lakeNodes:

            # Set Lake depth
            if l.getHighWall() != None:
                depth = l.getHighWall().getElevation() - k.getElevation()
                k.setLakeDepth(depth)

            # Set the new downNode
            k.setDownnode(l.getLakeDownnode())



def getPits (self):
    ''' Method to order the pits from the highest one (in altitude) to the lowest
    Arg: flownodes raster'''
    listPits = []
    listPits2 = []
    for r in range(self.getRows()):
        for c in range(self.getCols()):

            pitFlag = self._data[r,c].getPitFlag() # checking if is a pitflag

            if pitFlag==True and len(self.getNeighbours(r,c)) == 8 : # ok - selecting only the pits that are within the raster

                listPits.append(self._data[r,c]) # checked - it creates a list with all the inner nodes

    listPits2 = sorted(listPits, key=lambda FlowNode: FlowNode.getElevation(), reverse=True)
    return listPits2

### TASK 5 ###

def finalCell (self):
    ''' Method that returns the Max flow and the position of the cell
    Arg: flownode raster'''

    listPits = self.getPits2() #list of flownodes PITS ordered from the highest to the lowest

    lastPit = round(listPits[-1].getFlow2(),2) # get the flow
    cellCoordX = listPits[-1].get_x() # get the coordinates
    cellCoordY = listPits[-1].get_y() # get the coordinates

    return lastPit,cellCoordX,cellCoordY
    #print ("\n{} m of rain in x={} y={}".format(lastPit,cellCoordX,cellCoordY))

def getPits2 (self):
    ''' Method that identifies the pits on the boundary and return a list
    Arg: flownode raster'''
    listPits3 = []
    listPits4 = []

    for r in range(self.getRows()):
        for c in range(self.getCols()):

            pitFlag = self._data[r,c].getPitFlag() # checking if is a pitflag
            if pitFlag==True and len(self.getNeighbours(r,c)) < 8 : # ok - selecting only the pits that are on the border

                listPits3.append(self._data[r,c]) # creates a list with all the outer nodes

        # sort the list with an inbuilt function
        listPits4 = sorted(listPits3, key=lambda FlowNode: FlowNode.getFlow2(), reverse=False)

        return listPits4

    ### END TASK 5 ###

    ### TASK 4 ###

class lake(list):

    def __init__(self):
        self._lakeNodes=[]
        self._highWall=None # last cell belonging to the lake
        self._lakeUpnodes=[]
        self._lakeDownnode=None

    def explicit(self):
        ''' Method that return list of position(s) of largest element '''
        seq = []
        listin=[]
        for n in self._lakeNodes:
            seq.append(n.getElevation())
        max_val = max(seq)
        max_idx = seq.index(max_val)
        listin.append(max_idx)
        return listin

    def maxelements(self):
        ''' Method that return list of position(s) of largest element '''
        max_indices = []
        seq = []
        for n in self._lakeNodes:
            seq.append(n.getElevation())
        if seq:
            max_val = seq[0]
            for i,val in ((i,val) for i,val in enumerate(seq) if val >= max_val):
                if val == max_val:
                    max_indices.append(i)
                else:
                    max_val = val
                    max_indices = [i]

        return max_indices

    def splitLakeNodes(self,index):
        begin=[]
        end=[]
        begin=self._allPoints[:index]
        end=self._allPoints[index:]
        # v contains two polylines
        #v=[Polyline(list1a),Polyline(list1b)]
        return begin,end

    def getHighWall(self):
        return self._highWall

    def setHighWall(self, value):
        self._highWall = value

    def getLakeDownnode(self):
        return self._lakeDownnode

    def setLakeDownnode(self, value):
        self._lakeDownnode = value

    def isLake(self, node):
        return node in self._lakeNodes


class LakeDepthExtractor():
    def getValue(self, node):
        return node.getLakeDepth()

### END TASK 4 ###

class FlowExtractor():

    def getValue(self, node):
        ''' Method that extract the flow value from a specific node
        Args: Array of flownodes, the node of interest'''
        return node.getFlow()
```

Raster

```
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
    # returns the data in the raster - it is an array
    def getData(self):
        return self._data

    def getData2(self):
        values=[]
        for r in range(self.getRows()):
            for c in range(self.getCols()):
                values.append(round(self._data[r,c],2))
        return values
```

```
#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):
    ''' Methid that resamples the raster according to a resample factor
    Args: rater(self) and factor'''
    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

    return Raster(newdata, self._orgs[0], self._orgs[1], self._cellsize*factor)
   #if factor== 1:
   #    return self
   #else:
     #raise ValueError("createWithIncreasedCellsize: not fully implemented so only works for scaling by factor 1!")
```

Points – no modifications

This file has not been modified – therefore not attached

RasterHandler – no modifications

This file has not been modified – therefore not attached