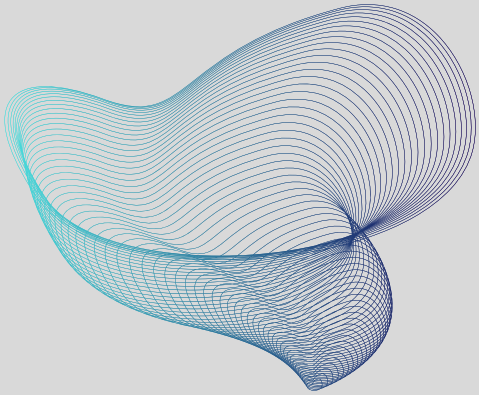
Kungliga Tekniska Högskolan

Transport and Geoinformation Technology

GIS Architecture and Algorithms

Implement a GIS software application



Group 5

Carl Vilhelm Boström
Jasmin Zdovc
Weihua Wang
Agni Kontorini
Ming Yang

Stockholm, 2022

Welcome to the world that digital mapping is fun and the only limit is lack of imagination.

Carl Vilhelm BostrÖm, Jasmin Zdovc, Weihua Wang, Agni Kontorini, Ming Yang

My Little FunGIS OPERATION MANUAL

Contents

Introduction

My Little FunGIS is a gis application made by Carl Vilhelm Boström, Jasmin Zdovc, Weihua Wang, Agni Kontorini and Ming Yang for the course AG2411: GIS Architecture and Algorithms. In this application are implemented the algorithms and the architecture that have been taught during a period of four weeks. The program contains several statistical operations using raster data.

Operations

1.Open file/ Save, and Export to JPG

In this application you can open a text file that is converted into an ASCII raster file, by clicking on the "File" button and then "Open file". Then a pop up window opens and you can select the text files and click "Open".

File ➔ *Open file*

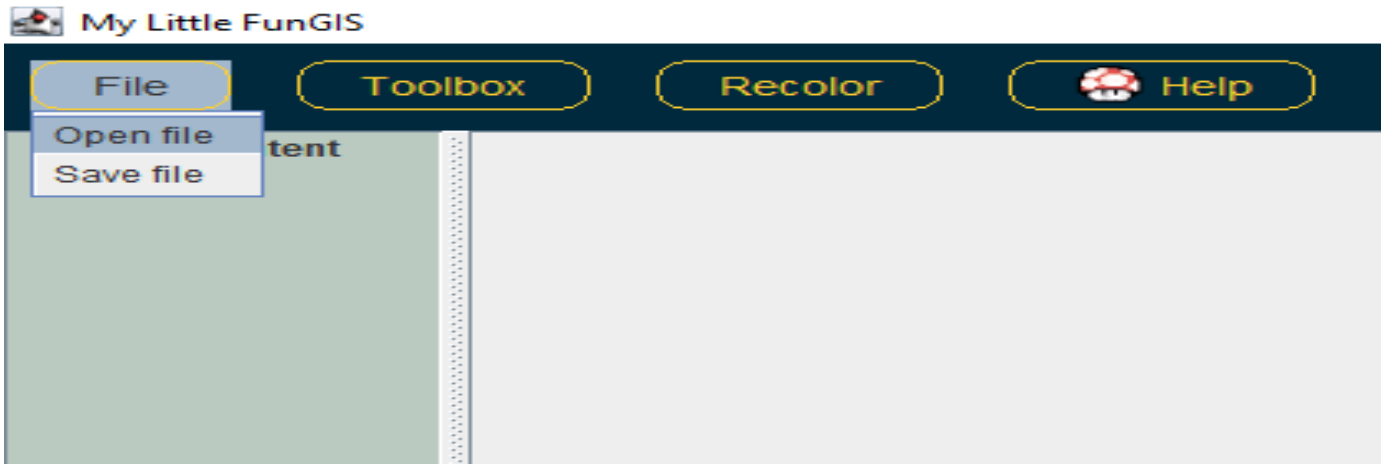


Figure 1: Selecting and opening a text file

Opening a file with a name that already exists in the program will automatically add a suffix to it, e.g. "vegetation_2" if "vegetation" and "vegetation_1" already exist. This is also the case when trying to create and save new layers using the *local*, *focal*, *zonal* or *shortest path* operations explained further down in this document.

To save the file you can click the "File" button and then "Save file".

File ➔ *Save file*

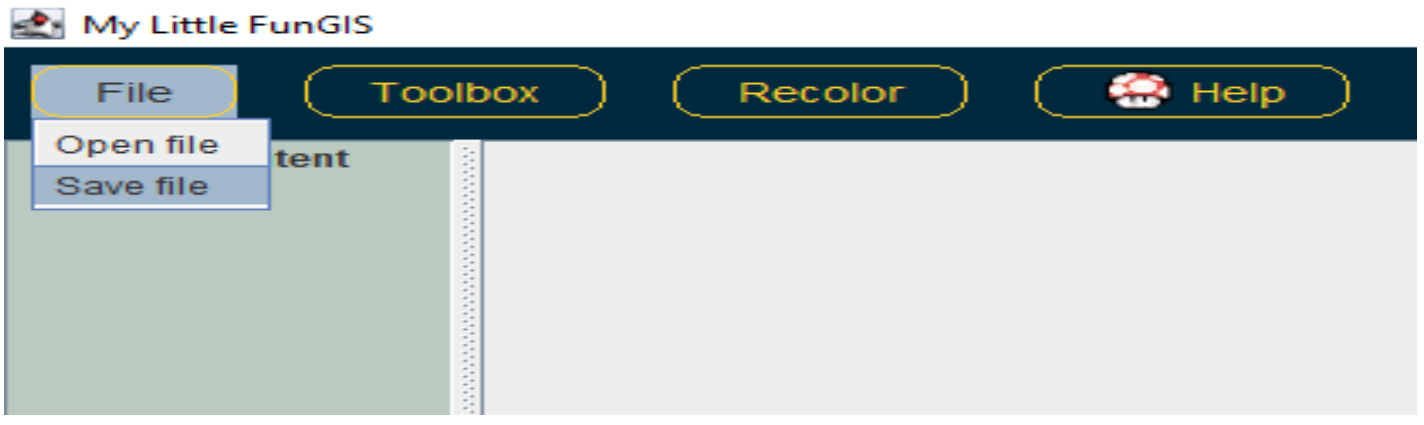


Figure 2: Save file

Any layer can also be saved directly to a JPG image from the *File* menu by clicking the "File" button and then "Export to JPG". Exporting the file to an image will keep any visualization changes the user has made, such as changing a layer's color scheme.

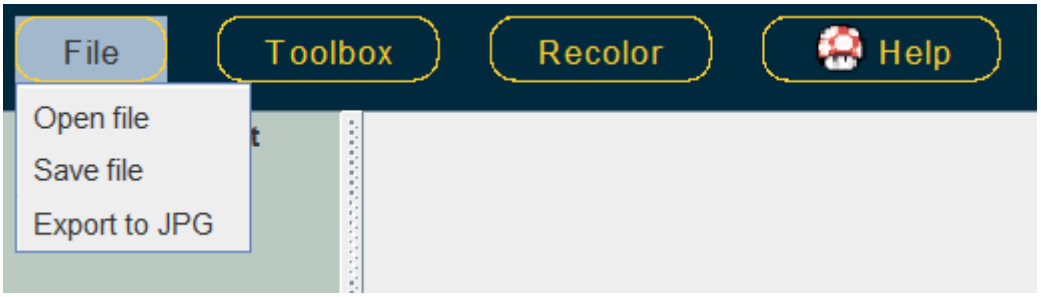


Figure 3: Export to JPG.

2. Toolbox

The Toolbox includes the list of the available operations which are Local operations, Focal operations, Zonal Operations, Distance and the Shortest path.

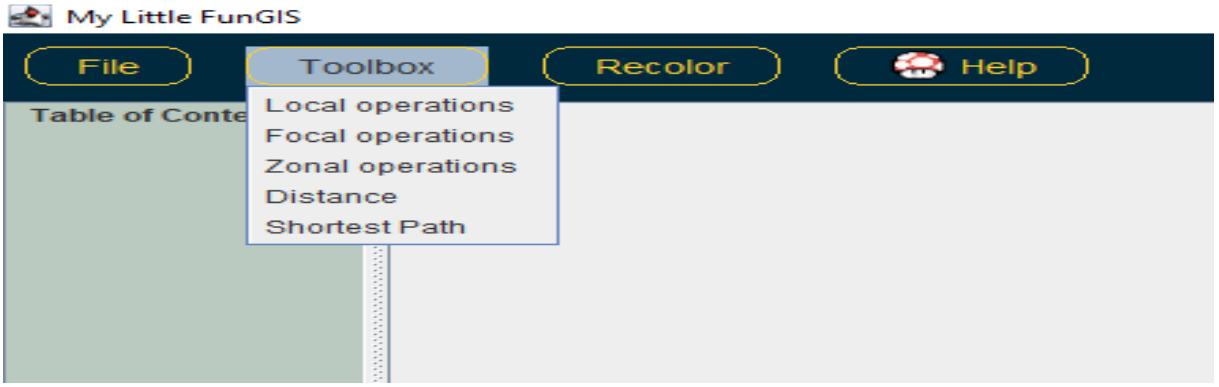


Figure 3: Operations included in the Toolbox

i. Local operations

Using two input raster layers, the local operations compute an output layer where each location's (cell's) value is a function of the two corresponding input layers. If one of the two input layers have a "no data" cell, the value of the output layer in that corresponding location will also be "no data". The local operations available are listed below.

LocalSum: The sum is determined by adding the values for the two corresponding cells of the input data.

LocalDiff: The value for each cell is determined as the difference of values between the two input layers.

ii. Focal operations

An output dataset is computed because of the focal operations, with each cell's value at each position being a function of both its input value and the values of the cells in a certain neighborhood around it. Six focal operations are available, listed below.

FocalVariety: The variety is determined by the number of unique values of the center cell within the neighborhood.

FocalSum: The sum is determined by the summation of the center cell within the rectangular neighborhood.

FocalProduct: The product is determined for the center cell as the product of all the neighbors in the neighborhood. **DOES NOT WORK.**

FocalRanking: DOES NOT WORK.

iii. Zonal Operations

The zonal operations take two input layers, one with values and one which decides which zone each of the cells are in. Zonal operations produce a raster layer where the values at each location is based on the information found in that cell's zone, as well as the relationships between those cells. There are six zonal operations available, listed below.

ZonalVariety:Records, in each output cell, the variety of values (the number of unique values) of all cells in the value raster that belong to the same zone as the output cell. Zones are identified by the values of the cells in the input.

ZonalSum:Records, in each output cell, the sum of the values of all cells in the value raster that belong to the same zone as the output cell. Zones are identified by the values of the cells in the input zone raster.

ZonalProduct: Record, in each output cell, the product of the values of all cells in the value raster that belong to the same zone as the output cell. Zones are identified as above. **DOES NOT WORK.**

ZonalMean: In each zone, the arithmetic mean of the values in the zone is assigned to all the cells in the zone.

ZonalMinimum: In each zone, the lowest value from the value layer in that zone is assigned to all the cells in the zone.

ZonalMaximum: In each zone, the highest value from the value layer in that zone is assigned to all the cells in the zone.

iv. Distance

Measures the distance from an origin point to a destination point in a layer of choice. Points are defined as *x, y* coordinate pairs and can either be entered manually or by clicking the *Pick*

from map option and selecting an origin or destination point by clicking it on the map. The output is two numbers, defined as follows:

Euclidean distance: Calculated by using Pythagoras theorem.

Manhattan distance: Calculated as the sum of the distance in *x* and *y*.

v. Shortest Path

Finds and prints the shortest path between an origin point and a destination point. Points are defined, chosen and input into the program in the same manner as for *Distance*, which can be read about above.

Currently, only Dijkstra's algorithm is supported. For more info about how Dijkstra's algorithm works, the user is encouraged to read online, e.g. from here:

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, or here:

https://www.freecodecamp.org/news/dijkstras-algorithm-explained-with-a-pseudocode-example/.

Warning: The current implementation of Dijkstra's algorithm may take a few moments to perform.

3. Recolor

Pressing the *Recolor* button in the menu will allow the user to access the *Color scheme* manager. From there, users can visualize any layer using their favorite color schemes. **Users are encouraged to change the color scheme for map's with shortest paths.**

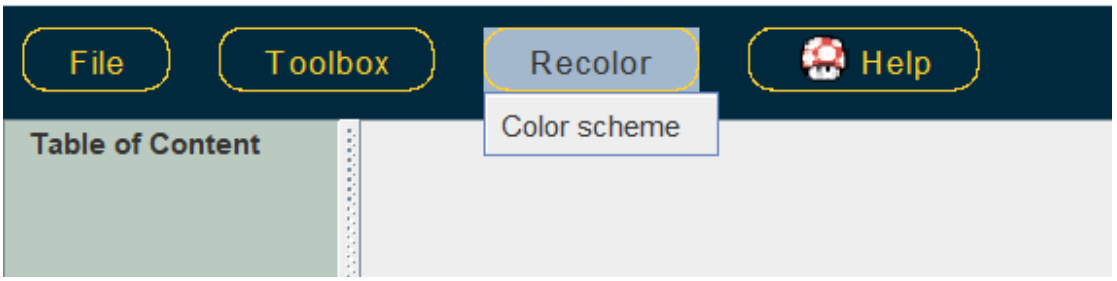


Figure 4: By clicking the Color scheme button a new window pops up, allowing the user to visualize their layers in their favorite color schemes.

4. Help

The "Help" button is an easy and fast way to access the Manual.

Help ➥ *Open App Manual*

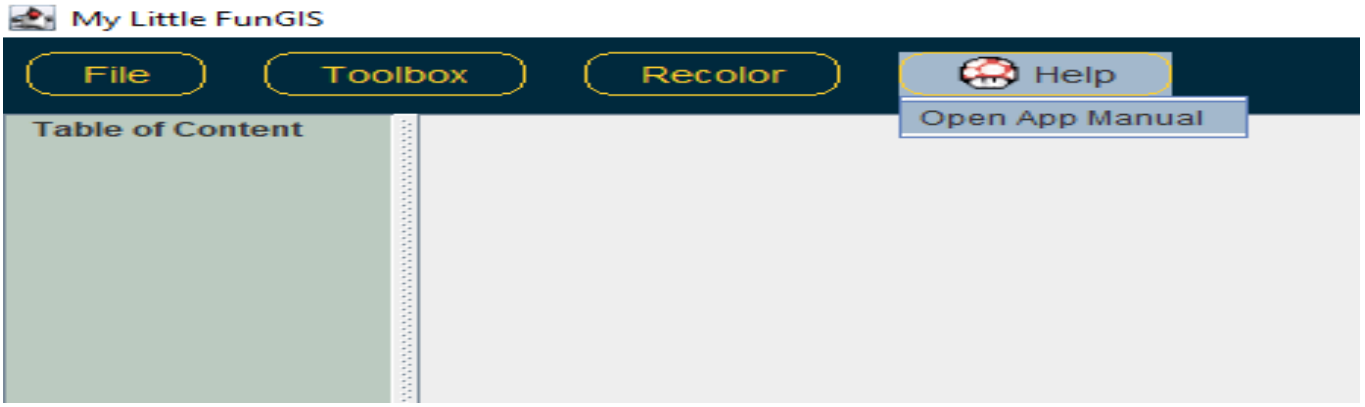


Figure 5: By clicking the Help button the Manual pops up in a new window

5. Get Learnt Button

An upcoming feature with visualizations of how local, focal and zonal algorithms work. The animations can be found in the provided code but have not been implemented into the program.

6. Table of Contents

The Table of contents (TOC) shows all stored and modified data. Users can retrieve their own loaded data and pictures from it.

The user can visualize a layer of choice by double clicking it in the TOC. Double clicking a layer will not remove previous changes to a layer's visualization, e.g. if a color scheme is changed for a layer, that color scale will remain until it is either changed again or deleted from the program.

Furthermore, it is possible to directly save the layer to a txt file, export the layer to a JPG image or delete it from the program by right-clicking the layer in the TOC.

Figure 6: The layer can be saved as a txt-file, exported as a JPG image or deleted directly from the TOC by right-clicking it.

7. Position and Zooming

Any map that is loaded into the program will have its scale dynamically resized to best fit the window size.

It is then possible to either zoom in or zoom out by pressing the *zoom in button* marked by a "+", the *zoom out button* marked by a "-", by using the mouse wheel or touchpad.

The magnifier in the program's bottom panel keeps track of how much larger the map has been made, and the scale takes the magnification and the input map's resolution and produces a scale unit.

If one loses track of where the map is, where the user is in the map, or if one simply wants to conveniently return to the scale and positioning of the map when it was loaded, the *Full extent* button can be used.

The program also allows the user to move the map around by clicking and dragging it. Any time a new layer is loaded or created, the map's positioning and scale is reset to that of when it was first loaded.

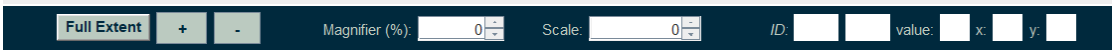


Figure 7: The bottom panel of the program with the zoom in, zoom out and full extent buttons to the left.

It is also possible to hover over and select pixels in the loaded maps. Hovering over a pixel will show the pixel ID in the leftmost *ID field* in the bottom panel, its value in the *value field*, its x and its y coordinates in the *x* and *y coordinate fields*. Selecting any pixel by clicking on it will then record its ID in the rightmost *ID field* in the bottom panel

Pseudocode for the Toolbox operations

NOTE: Some of these operations do not work properly and have thus been removed from the program.

```
1.  ////////////////////////////// Local methods ////////////////////////////
2.
3.      // LocolSum:
4.      // Create a new Layer in which the value of each pixel is
5.      // the sum of counterparts in this Layer and in an input Layers.
6.      // Input:   inLayer:        an input Layer ready for addition
7.      //          outLayerName:   the name of output Laye
8.      // Output:  outLayer:       an new Layer, value of each pixel: outLayer = (this)Layer +
    inLayer
9.      public Layer localSum(Layer inLayer, String outLayerName){
10.         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
11.         for(int i = 0; i < nRows; i++){
12.             for (int j = 0; j < nCols; j++) {
13.                 outLayer.values[i][j] = values[i][j] + inLayer.values[i][j];
14.             }
15.         }
16.         return outLayer;
17.     }
18.
19.     // LocolDifference:
20.     // Create a new Layer in which the value of each pixel is the result of
21.     // counterpart in this Layer subtracts from in an input Layers
22.     // Input:   inLayer:        an input Layer ready for subtract
23.     //          outLayerName:   the name of output Laye
24.     // Output:  outLayer:       an new Layer, value of each pixel: outLayer = (this)Layer -
    inLaye
25.     public Layer localDifference(Layer inLayer, String outLayerName){
26.         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
27.         for(int i = 0; i < nRows; i++){
28.             for (int j = 0; j < nCols; j++) {
29.                 outLayer.values[i][j] = values[i][j] - inLayer.values[i][j];
30.             }
31.         }
32.         return outLayer;
33.     }
34.
35. ////////////////////////////////////// Focal methods
    ////////////////////////////////////////////
36.
37.     // FocalSum:
38.     // Create a new Layer in which the value of each pixel is the sum of its neighborhood
    in this Layer.
39.     // The radius and shape of neighborhood should be given as parameters
40.     // Input:   radius:         the radius of neighborhood
41.     //          IsSquare:       the shape of neighborhood, true--square, false--circle
42.     //          outLayerName:   the name of output Layer
43.     // Output:  outLayer:       an new Layer, value of each pixel:
44.     //                          outLayer =
    neighborhood[0]+neighborhood[1]+...+neighborhood[n]
45.     public Layer focalSum(int radius, boolean IsSquare, String outLayerName) {
46.         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
47.         outLayer.values = new double[nRows][nCols];
48.         for (int i = 0; i < nRows; i++) {
49.             for (int j = 0; j < nCols; j++) {
50.                 //the neighborhood's shape is a square (if true) or a circle (if false)
51.                 int[][] neighborhood = getNeighborhood(i*nCols+j, radius, IsSquare);
52.                 double sumNeigh = 0;
53.                 for (int k = 0; k < neighborhood.length; k++) {
54.                     int l = neighborhood[k][0];
55.                     int m = neighborhood[k][1];
56.                     sumNeigh += this.values[l][m];
57.                 }
58.                 outLayer.values[i][j] = sumNeigh;
```

```
            }
        }
        return outLayer;
    }

    // FocalProduct:
    // Create a new Layer in which the value of each pixel is the product of its neighborhood in this Layer.
    // The radius and shape of neighborhood should be given as parameters.
    // Input:   radius:         the radius of neighborhood
    //          IsSquare:       the shape of neighborhood, true--square, false--circle
    //          outLayerName:   the name of output Laye
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = neighborhood[0]*neighborhood[1]*...*neighborhood[n]
    public Layer focalProduct(int radius, boolean IsSquare, String outLayerName) {
        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution, nullValue);
        outLayer.values = new double[nRows][nCols];
        for (int i = 0; i < nRows; i++) {
            for (int j = 0; j < nCols; j++) {
                int[][] neighborhood = getNeighborhood(i*nCols+j, radius, IsSquare);
                //the neighborhood's shape is a square (if true) or a circle (if false)
                double sumNeigh = 0;
                for (int k = 0; k < neighborhood.length; k++) {
                    int l = neighborhood[k][0];
                    int m = neighborhood[k][1];
                    sumNeigh *= this.values[l][m];
                }
                outLayer.values[i][j] = sumNeigh;
            }
        }
        return outLayer;
    }

    // FocalRanking:
    // Create a new Layer in which the value of each pixel is its rank in its neighborhood in this Layer.
    // The radius and shape of neighborhood should be given as parameters.
    // Input:   radius:         the radius of neighborhood
    //          IsSquare:       the shape of neighborhood, true--square, false--circle
    //          outLayerName:   the name of output Layer
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = its rank in neighborhood(Descending order
    public Layer focalRanking(int radius, boolean IsSquare, String outLayerName) {
        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution, nullValue);
        outLayer.values = new double[nRows][nCols];
        for (int i = 0; i < nRows; i++) {
            for (int j = 0; j < nCols; j++) {
                int[][] neighborhood = getNeighborhood(i*nCols+j, radius, IsSquare);
                //the neighborhood's shape is a square (if true) or a circle (if false)
                HashMap<Integer, Double> neighMap = new HashMap<Integer, Double>();
                double rank = -1;
                for (int k = 0; k < neighborhood.length; k++) {
                    int l = neighborhood[k][0];
                    int m = neighborhood[k][1];
                    neighMap.put(l*nCols+j, values[l][m]);
                }
                List<Map.Entry<Integer, Double>> list = new ArrayList<>(neighMap.entrySet());
                //sort in ascending order
                Collections.sort(list, new Comparator<Map.Entry<Integer, Double>>(){
                    public int compare(Map.Entry<Integer, Double> o1,Map.Entry<Integer, Double> o2) {
                        return o1.getKey().compareTo(o2.getKey());
                    }
                });
//              Arrays.sort(neighValues, Collections.reverseOrder());
                for(int ii=0;ii<list.size();ii++) {
                    if(list.get(ii).getKey()==i*nCols+j){
                        rank = list.size()-ii;//descending order
                    }
```

```
                }
                outLayer.values[i][j] = rank;
            }
        }
        return outLayer;
    }

    // FocalVariety:
    // Create a new Layer in which the value of each pixel is the number of unique values in its neighborhood.
    // The radius and shape of neighborhood should be given as parameters
    // Input:   radius:         the radius of neighborhood
    //          IsSquare:       the shape of neighborhood, true--square, false--circle
    //          outLayerName:   the name of output Laye
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = the number of unique values in neighborhood
    //                    e.g. the number of unique values in [1, 0, 2 , 1, 0, 0, 2] = 3
    public Layer focalVariety(int r, boolean isSquare, String outLayerName) {
         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution, nullValue);
            for(int i = 0; i < nRows; i++) {
                for (int j = 0; j < nCols; j++) {
                    int input = i * nCols + j;
                    int[][] neighborhood = outLayer.getNeighborhood(input, r, isSquare);

                    // Create list of unique values for specified neighborhood
                    ArrayList<Double> list = new ArrayList<Double>();
                    for (int[] neighbor: neighborhood) {
                        int row = neighbor[0];
                        int col = neighbor[1];

                        Double valueObj = values[row][col];

                        if (!list.contains(valueObj)) {
                            list.add(valueObj);
                        }
                    }
                    outLayer.values[i][j] = list.size();
                }
            }
            return outLayer;
    }
////////////////////////////// Zonal methods ////////////////////////////////////////////////////

    // ZonalSum
    // Create a new Layer in which the value of each pixel is the sum of its zone in this Layer.
    // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique numbers.
    //          outLayerName:   the name of output Laye
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = the sum of zon
    public Layer zonalSum(Layer zoneLayer, String outLayerName) {
        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution, nullValue);
        outLayer.values = new double[nRows][nCols];
        if ((nRows != zoneLayer.nRows) || (nCols != zoneLayer.nCols)) {
            System.out.println("The input layer does not match the exiting layer.");
        }
        else {//create a dictionary for No. of zone and an array of all values
            HashMap<Double,ArrayList<Double>> hm = new HashMap<Double, ArrayList<Double>>();
            for (int i = 0; i < nRows; i++) {
                for (int j = 0; j < nCols; j++) {
                    double key = zoneLayer.values[i][j];
                    if(hm.containsKey(key)) { //key exists
                        ArrayList<Double> zoneList = hm.get(key);
                        zoneList.add(this.values[i][j]);//add new element
                        hm.put(key, zoneList);
                    }
                    else {//add key
                        ArrayList<Double> zoneList = new ArrayList<Double>();
```

```
                        hm.put(key, zoneList);
                    }
                }
            }
            for (int i = 0; i < nRows; i++) {
                for (int j = 0; j < nCols; j++) {
                    double key = zoneLayer.values[i][j];
                    //https://blog.csdn.net/Android_Mrchen/article/details/107355803
                    DoubleSummaryStatistics statistics =
hm.get(key).stream().mapToDouble(Number::doubleValue).summaryStatistics();
                    outLayer.values[i][j] = statistics.getSum();
                }
            }
        }
        return outLayer;
    }

    // ZonalMean
    // Create a new Layer in which the value of each pixel is the mean of its zone in this
Layer.
    // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique
numbers.
    //          outLayerName:   the name of output Layer
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = the mean of zon
    public Layer zonalMean(Layer zoneLayer, String outLayerName) {
        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
nullValue);
        outLayer.values = new double[nRows][nCols];
        if ((nRows != zoneLayer.nRows) || (nCols != zoneLayer.nCols)) {
            System.out.println("The input layer does not match the exiting layer.");
        }
        else {//create a dictionary for No. of zone and an array of all values
            HashMap<Double,ArrayList<Double>> hm = new HashMap<Double,
ArrayList<Double>>();
            for (int i = 0; i < nRows; i++) {
                for (int j = 0; j < nCols; j++) {
                    double key = zoneLayer.values[i][j];
                    if(hm.containsKey(key)) { //key exists
                        ArrayList<Double> zoneList = hm.get(key);
                        zoneList.add(this.values[i][j]);//add new element
                        hm.put(key, zoneList);
                    }
                    else {//add key
                        ArrayList<Double> zoneList = new ArrayList<Double>();
                        hm.put(key, zoneList);
                    }
                }
            }
            for (int i = 0; i < nRows; i++) {
                for (int j = 0; j < nCols; j++) {
                    double key = zoneLayer.values[i][j];
                    DoubleSummaryStatistics statistics =
hm.get(key).stream().mapToDouble(Number::doubleValue).summaryStatistics();
                    outLayer.values[i][j] = statistics.getAverage();
                    //System.out.println(statistics.getAverage());
                }
            }
        }
        return outLayer;
    }

    // ZonalMaximum
    // Create a new Layer in which the value of each pixel is the maximum of its zone in
this Layer.
    // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique
numbers.
    //          outLayerName:   the name of output Laye
    // Output:  outLayer:       an new Layer, value of each pixel:
    //                          outLayer = the maximum of zon
    public Layer zonalMaximum(Layer zoneLayer, String outLayerName) {
        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
nullValue);
```

```
255.         outLayer.values = new double[nRows][nCols];
256.         if ((nRows != zoneLayer.nRows) || (nCols != zoneLayer.nCols)) {
257.             System.out.println("The input layer does not match the exiting layer.");
258.         }
259.         else {//create a dictionary for No. of zone and Maximum
260.             HashMap<Double, Double> hm = new HashMap<Double, Double>();
261.             for (int i = 0; i < nRows; i++) {
262.                 for (int j = 0; j < nCols; j++) {
263.                     double key = zoneLayer.values[i][j];
264.                     if(hm.containsKey(key)) { //key exists
265.                         if(this.values[i][j] > hm.get(key)) {
266.                             hm.put(key, this.values[i][j]);//renew value
267.                         }
268.                     }
269.                     else {//add key
270.                         hm.put(key, this.values[i][j]);
271.                     }
272.                 }
273.             }
274.             for (int i = 0; i < nRows; i++) {
275.                 for (int j = 0; j < nCols; j++) {
276.                     double key = zoneLayer.values[i][j];
277.                     outLayer.values[i][j] = hm.get(key);
278.                 }
279.             }
280.         }
281.         return outLayer;
282.     }
283.
284.     // ZonalMinimum
285.     // Create a new Layer in which the value of each pixel is the minimum of its zone in
    this Layer
286.     // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique
    numbers.
287.     //          outLayerName:   the name of output Laye
288.     // Output:  outLayer:       an new Layer, value of each pixel:
289.     //                          outLayer = the minimum of zon
290.     public Layer zonalMinimum(Layer zoneLayer, String outLayerName) {
291.         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
292.
293.         // Create hashmap with associated lowest values
294.         HashMap<Double, Double> hm = new HashMap<Double, Double>();
295.         for(int i = 0; i < nRows; i++){
296.             for (int j = 0; j < nCols; j++) {
297.                 Double key = zoneLayer.values[i][j];
298.                 Double value = values[i][j];
299.                 if (!hm.containsKey(key) || hm.get(key) > value) {
300.                     hm.put(key,  value);
301.                 }
302.             }
303.         }
304.         // Assign lowest values to output layer
305.         for(int i = 0; i < nRows; i++){
306.             for (int j = 0; j < nCols; j++) {
307.                 Double key = zoneLayer.values[i][j];
308.                 outLayer.values[i][j] = hm.get(key);
309.             }
310.         }
311.         return outLayer;
312.     }
313.
314.     // ZonalProduct
315.     // Create a new Layer in which the value of each pixel is the product of its zone in
    this Layer.
316.     // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique
    numbers.
317.     //          outLayerName:   the name of output Layer
318.     // Output:  outLayer:       an new Layer, value of each pixel:
319.     //                          outLayer = the product of zone
320.     public Layer zonalProduct(Layer zoneLayer, String outLayerName) {
321.         Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
```

```
322.        outLayer.values = new double[nRows][nCols];
323.        if ((nRows != zoneLayer.nRows) || (nCols != zoneLayer.nCols)) {
324.            System.out.println("The input layer does not match the exiting layer.");
325.        }
326.        else {//create a dictionary for No. of zone and an array of all values
327.            HashMap<Double,ArrayList<Double>> hm = new HashMap<Double,
    ArrayList<Double>>();
328.            for (int i = 0; i < nRows; i++) {
329.                for (int j = 0; j < nCols; j++) {
330.                    double key = zoneLayer.values[i][j];
331.                    if(hm.containsKey(key)) { //key exists
332.                        ArrayList<Double> zoneList = hm.get(key);
333.                        zoneList.add(this.values[i][j]);//add new element
334.                        hm.put(key, zoneList);
335.                    }
336.                    else {//add key
337.                        ArrayList<Double> zoneList = new ArrayList<Double>();
338.                        hm.put(key, zoneList);
339.                    }
340.                }
341.            }
342.            for (int i = 0; i < nRows; i++) {
343.                for (int j = 0; j < nCols; j++) {
344.                    double key = zoneLayer.values[i][j];
345.                    Double product = hm.get(key).stream().reduce((number, number2)->
    number*number2).get();
346.                    outLayer.values[i][j] = product.doubleValue();
347.                }
348.            }
349.        }
350.        return outLayer;
351.    }
352.
353.    // ZonalVariety
354.    // Create a new Layer in which the value of each pixel is the number of the unique
    values in its zone in this Layer.
355.    // Input:   zoneLayer:      a Layer that divide this layer into several zones by unique
    numbers.
356.    //          outLayerName:   the name of output Layer
357.    // Output:  outLayer:       an new Layer, value of each pixel:
358.    //                          outLayer = the number of unique values in its zone
359.    //                      e.g. the number of unique values in [1, 0, 2 , 1, 0, 0, 2] = 3
360.    public Layer zonalVariety(Layer zoneLayer, String outLayerName) {
361.        Layer outLayer = new Layer(outLayerName, nRows, nCols, origin, resolution,
    nullValue);
362.        outLayer.values = new double[nRows][nCols];
363.        if ((nRows != zoneLayer.nRows) || (nCols != zoneLayer.nCols)) {
364.            System.out.println("The input layer does not match the exiting layer.");
365.        }
366.        else {//create a dictionary for No. of zone and an array of all values
367.            HashMap<Double,ArrayList<Double>> hm = new HashMap<Double,
    ArrayList<Double>>();
368.            for (int i = 0; i < nRows; i++) {
369.                for (int j = 0; j < nCols; j++) {
370.                    double key = zoneLayer.values[i][j];
371.                    if(hm.containsKey(key)) { //key exists
372.                        ArrayList<Double> zoneList = hm.get(key);
373.                        zoneList.add(this.values[i][j]);//add new element
374.                        hm.put(key, zoneList);
375.                    }
376.                    else {//add key
377.                        ArrayList<Double> zoneList = new ArrayList<Double>();
378.                        hm.put(key, zoneList);
379.                    }
380.                }
381.            }
382.            for (int i = 0; i < nRows; i++) {
383.                for (int j = 0; j < nCols; j++) {
384.                    double key = zoneLayer.values[i][j];
385.                    ArrayList<Double> zoneList = hm.get(key);
386.                    double[] zonelist= new double[zoneList.size()];
387.                    int counter = 0;
388.                    for (Double zoneObj: zoneList) {
```

```
                        zonelist[counter] = zoneObj;
                        counter++;
                    }
                    outLayer.values[i][j] = getNumUnique(zonelist);
                }
            }
        }
        return outLayer;
    }
```