UNIVERSITY OF
BIRMINGHAM

Medical Image Processing with Quadtrees

School of Computer Science
University of Birmingham

Josh Wainwright

Supervisor: Iain Styles

Date: September 2014

Abstract

This study deduces that reionization began at a redshift of $z = 17.82(+3.06, -2.4)$ and ended at a redshift of $z = 7 \pm 1.8$. This is calculated by directly applying the dynamics of star formation and the ionization rate of neutral hydrogen in the Inter-galactic Medium. A photometry strategy consisting of 3 multi-band surveys is proposed in order to observe Lyman Break Galaxies across redshifts 6–17. The surveys will locate 100.5 ± 37.0, 138.7 ± 100.6, 358.1 ± 158.6 galaxies in redshift ranges 6–8.5, 8.5–10 and 10–17 respectively. These surveys will be completed by the James Webb Space Telescope and Euclid which are planned for launch in the coming decade. A follow up spectroscopy survey will be used to confirm the redshift and properties of 24, 4 and 48 galaxies in these 3 surveys respectively. The spectroscopy will be carried out using James Webb Space Telescope and a combination of single and multi-slit spectroscopy. It is shown that the use of known gravitational lenses, located between redshift 0.5–0.7, is very beneficial for discovering high redshift candidates as it can increase the depth of surveys by up to 3 magnitudes.

Table of Contents

List of Figures

Part I

Introduction

1 Medical Imaging

In various scientific fields, viewing and imaging objects smaller than the human eye can naturally observe is an ability eagerly sought. Microscopy in medical fields has allowed us to learn about the nature of tissues, micro-organisms and cells and the way they work together and to develop preventative measures and cures for injuries and diseases.

The humble microscope, used as far back as the 1500's, is able to show us a world that is not usually visible, but in recent times, as our understanding has grown, we have desired to see beyond what ordinary microscopes are capable and have invented machines that let us catch a glimpse of some of the smallest structures, molecules. However, when the objects to be viewed get this small, of the order of a few tens of nanometers, the light that is used to view them become the limiting factor.

2 Sub-Diffraction-Limit Imaging

Imaging objects becomes more difficult as they get smaller because of the wavelength of light. Once two objects are separated by a distance of an order similar to that of the wavelength (λ) of the light used to view them, it is no longer possible to resolve these two objects apart, instead all that can be seen is a blur of the two objects together.

There have been several techniques developed for distinguishing objects apart on smaller and smaller scales. Many of these involve using different wavelengths of light. For example, instead of being limited by visible light, $\lambda \approx 5 \times 10^{-7}$m, x-ray radiation ($\lambda \approx 10^{-10}$m) or even electrons ($\lambda \approx 10^{-11}$m) can be used to resolve smaller scales in x-ray and electron microscopy respectively. These, however, have the issue that, because the smaller wavelengths imply higher energies, there is the danger of destroying the sample.

The minimum distance that two objects can be resolved at is given by Abbe's criterion,

$$d = \frac{\lambda}{2NA} \tag{1}$$

$$= \frac{\lambda}{2n \sin\theta}, \tag{2}$$

where NA is the numerical aperture of the microscope, the range of angles that the microscope's lens will let light through properly. For $\lambda \approx 500\,\text{nm}$ (in the middle of the visible range), and $NA = 1.5$, the maximum resolving distance is $d = 160\,\text{nm}$, this is the diffraction limit. This is an order of magnitude larger than the objects that need to be resolved. A few attempts to avoid this limit using exotic types of lenses have been developed [Fang et al., 2005], but these are currently far more expensive to use than traditional imaging equipment.

2.1 Image Manipulation

Instead of trying to avoid the diffraction limit using shorter wavelengths of light or other particles, other techniques employ different methods of actually capturing the image, or cleaver manipulation of the images that are produced, to get around the limitations of the diffraction problem.

For example Stochastic Optical Reconstruction Microscopy (STORM) [Rust et al., 2006] and PhotoActivation Localisation Microscopy (PALM) [Owen et al., 2010] use a technique where the objects to be imaged are molecules of a fluorescent dye. These are attached to the object of interest, a cell or sample of tissue for example. The type of dye molecule used allows the fluorescence to be switched on and off, allowing some markers to be imaged separately to others, effectively increasing the distance between points. Once an image is captured, the point spread function (PSF) of the point is used to locate the single marker, the "on" markers are changed and the image retaken. When many of these images are taken, they can be combined to provide accurate information on the original location of the markers and hence the shape and dimensions of the object.

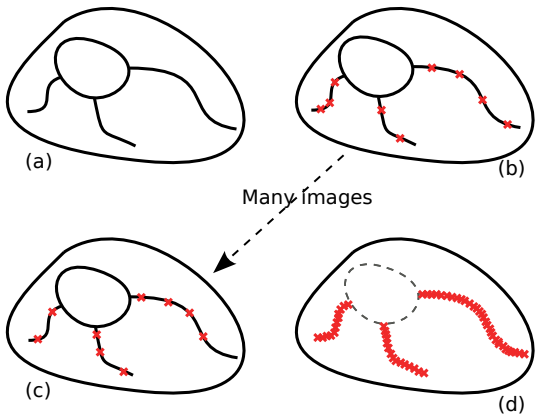


Figure 1: *STORM imaging. (a) The actual structure to be imaged is too small for regular microscopy. In stages (b) through (c), many images are taken, each with a different subset of the fluorescent markers activated. When the images are combined, (d), the points add up and reveal the nature of the object.*

3 Benchmarking

Throughout the project, a set of files will be used to test the algorithms that are developed; their correctness and effectiveness, speed and resource use. These files contain real data formated in the same way as would be expected for data given to the plugin in general use. The files that will be used are detailed in Table 1.

Note that `palm-3-small.txt` is a subset of `palm-3.txt` which is used for simply checking correctness of algorithms. A summary of the columns that are included in the files, used and unused fields, is included in Appendix A.

File Name	Size	Points
`palm-1.txt`	12 MiB	65572
`palm-2.txt`	6.4 MiB	36672
`palm-3.txt`	5.8 MiB	33342
`palm-3-small.txt`	176 KiB	1000
`uniform.txt`	22 MiB	2000000

Table 1: *These files containing sample data are used for benchmarking throughout the project.*

Part II

Data Structures

The way in which data is represented in memory has a large effect on the speed, efficiency and effectiveness of any algorithm that is performed on that data. There are almost always a number of tradeoffs that must be considered when choosing or designing a data structure: speed of access vs. speed of search or traversal, storage space used vs. time to insert a datum, etc.

Some data structures may be naïvely chosen based on one of these, at the expence of the other. For example, consider storing the pixel information for a sparse image generated by a number of single pixel points—a linear array might be selected. This would give extremely good access and modification times, both $O(1)$, but insertion and deletion are very slow. There would also be a large amount of wasted space since every pixel that is black, i.e., does not have any points in, would need to have an array index with the same entry, 0.

To decide the most approriate data structure, a number of different approaches are implemented and tested under different types of data and for a range of different operations performed on them.

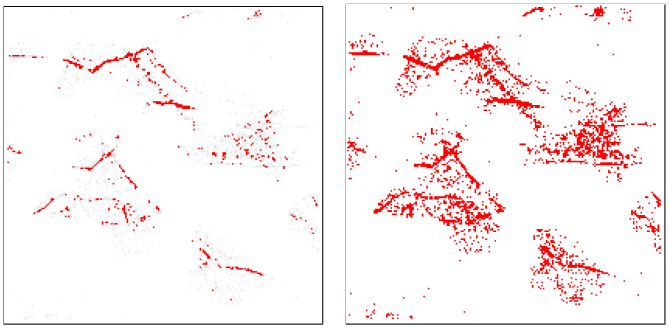
4 Simple Grid Method

The simplest method for analysing the distribution of points is to use a regular grid of cells and place the points into the cells one at a time. Once all points have been added, the number of points per cell can be treated as a grey scale brightness value. This gives a simple pixel image, with brightness as a function of density of the points, in the `pnm` image format. A thresholding filter can then be applied to remove the points that are isolated and leave the denser areas corresponding to clusters.

Though the resolution of this method can be easily changed by altering the size of the cells and the grid, it performs badly when presented with data that is even slightly noisy. If the clusters themselves have a density that is not significantly above the background noise level, the thresholding step is prone to either exclude much of the real data, or to increase the size of the clusters by including too much noise. These two effects can be seen clearly in Figure 2, where `palm-1.txt` was used with a cell size of 200. The range of the data is from 0 to 41000 for both the x and the y axes, thus the images are 205 by 205 pixels. This data took 495 ms to generate.

There are steps that can be taken to improve the approach of this simple grid when handling outlying points caused by noise.

1. First the algorithm is modified to include a thresholding step before writing the data to a file. This means that the pixels can be adjusted with greater accuracy and any arbitrary level can be chosen to threshold at.
2. Next, once an image has been generated, the number of points that contributed to each pixel is no longer of interest and so the image can be converted to a binary image. This is an image with just two possible values, the first represents white space, where there are no points, the seconds is black where there were points and so is of interest.
3. Once a binary image has been generated, erode and dilate filters can be applied to remove remaining outliers and to try to close the gaps in the structures that have been identified so that they are more solid.



(a) **(b)**

Figure 2: *Setting a low threshold, (a), means that many of the points in the clusters are lost. Setting it higher, (b), includes too many of the points deemed to be noise. Pixels are cells that ended with points, red are points that would be kept by the threshold process, black would be removed.*

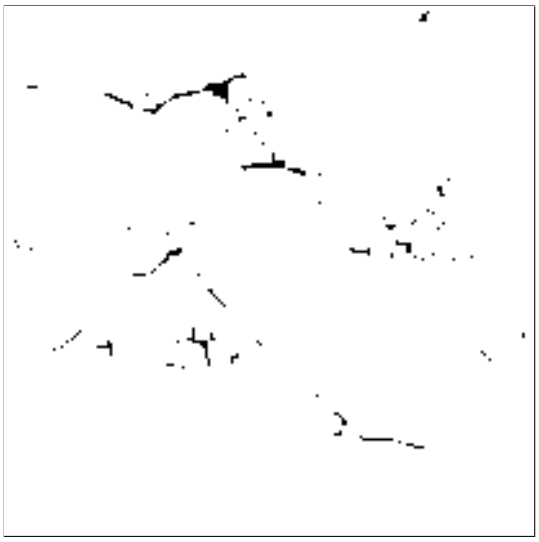


Figure 3: *Using a `close`-ing algorithm can help to emphasise the structure in the data and, at the same time, remove the isolated points representing noise.*

These steps lead to significantly better isolation of the interesting parts of the image, as can be seen in Figure 3, however, much of the detail of the structure is lost in this process.

5 Quadtrees

Since the simple grid method described in Section 4 performs slowly and does not offer good cluster analysis, a different approach is needed. The chosen method is to use a quadtree data structure.

Quadtrees are a type of recursive abstract data type in the form of a tree where every node has exactly zero or four chil-

dren. A node with zero children is a leaf and contains some information, value or quantity. A node with four children is not a leaf and cannot hold information.

Quadtrees are often used in image processing since the four children of the root node can naturally represent the four quadrants of the image; upper left, upper right, lower left and lower right. Since each of these children is also a quadtree, the image can be subdivided to any arbitrary depth. From this point, information about the image can be "seen" more easily by the computer and statistics calculated.

5.1 Quadtree Definitions

It is useful to define a few terms that shall be used in the context of quadtrees. Many of these are identical to the definitions more commonly applied to binary trees.

Node (Cell) A leaf in the tree which holds a number of points.

Root The node at the topmost position in the tree.

Child One of the four nodes that are beneath a given node for which there is a direct path of length 1 between this node and it.

Height The length of the longest path from the root node to one of the leaf nodes.

Depth The length of the path from a given node up to the root.

Completeness A quadtree is 'complete' when each of the root node's four children have the same height. In this case, the number of leaves in the tree is a maximum.

Quadtree Code (Code) The unique binary number assigned to a node by adding its position with respect to its siblings to the code of its parent.

5.2 Code Orderings

In order to identify a node uniquely in the tree, each node is given a code that is built up from it's parent code plus some value that identifies it among it's siblings. The root node is usually chosen to have an empty code so that the first four children are given the first level codes.

The choice of what order to label the children is important if the order in which the nodes are placed is important. For spatial indexing, for example, each node represents a quadrant in two dimensional space, so being able to traverse the children in a sensible and predictable way is essential.

5.2.1 Morton Order (Z-Order)

Perhaps the most natural order to give to the values in a spatial quadtree is to number them from 1 to 4, left to right, top to bottom. This can be made more appropriate for a computer to use by numbering from 0 to 3. This is called Morton Order [Morton, 1966] or Z-order because of the resulting path that would be followed by traversing the nodes in order, Figure 4a. This has several useful features.

1. First, the numbers can be converted to base 2:
   - 0 becomes 00
   - 1 becomes 01
   - 2 becomes 10 and
   - 3 becomes 11.
2. This has advantages since binary is very efficient for computers to work with and allows certain tricks to be employed (see Morton order coordinates).
3. Also, this numbering system is easily extendible to any depth of tree that can be imagined.
   (a) The root, as mentioned before, is given no value,
   (b) each of the children are numbered 00 through 11.
   (c) the children of these children are numbered 00 to 11 with the parent as a prefix. So the children of node 00 are 0000, 0001, 0010 and 0011. Likewise, the children of 11 are 1100, 1101, 1110 and 1111.
   (d) The children are always numbered in the same order. If starting at the top and going top to bottom and left to right, this is maintained for all children.

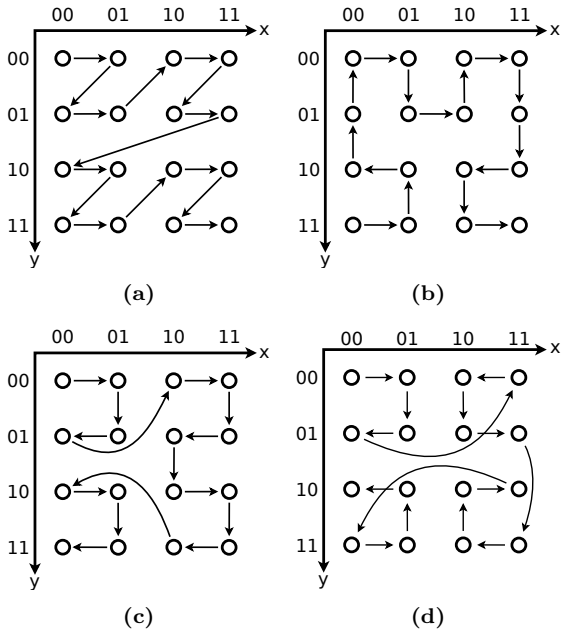


Figure 4: *A selection of possible tree traversal orderings. Each of theses is, more formally, a space filling curve which is a direct, unique mapping from a 2D grid to a 1D array. (a) shows Z-order, (b) shows Hilbert order, (c) shows Gray code order and (d) shows the modified Gray code order developed for this project.*

This method of numbering is simple and so acceptable for the standard uses of quadtrees, but it was found to be difficult to work with in a spatial context when information about neighbouring cells is needed. The steps required to calculate the neighbours of any given cell are reasonably complex and so would add computational and time complexity to calculations performed on the tree.

Morton Order Coordinates

Another useful feature of the Morton ordering is the simple conversion from quadtree code to Cartesian coordinate notation. The steps to convert to coordinate form are:

1. Ensure the code is in binary format with two bits for each level of the tree,
2. *de-interleave* the bits of the code (starting with the first being given to the y-axis, assign bits to the y and x axes building up a binary value for each),
3. convert the resulting two binary value to decimal to give a standard decimal (x, y) coordinate.

This method means that it is very easy to calculate an arbitrary number of nodes in any direction by simply converting the code for a node to coordinates, adding or subtracting the number of positions to move in the x and y directions and then converting back to quadtree code representation by following the algorithm above in the reverse order.

Of course, this method has no knowledge of the structure of the quadtree being used and so only provides the code of the node that would occupy the space at the given coordinate. That node might not exist—the tree might not extend deep enough, so the node in that position is larger than expected; or the tree might be deeper at that location meaning the node is smaller. In these cases, there are a number of options as to how to find the correct node for the code:

- The code can be shortened and/or lengthened and the resulting adapted code checked to see if it is in the tree. This trial and error method can be fastest when there is a limit on the depth range allowed when searching (see Section 7.1).
- The tree can be traversed to find the nearest node to the expected code reference.

5.2.2 Hilbert Order

One of the reasons the Z-order above becomes difficult to work with is that the resulting path from traversing the nodes in-order has to make large jumps and so cells which are numbered next to each other may, in fact, not be near each other in the image.

A number of routes exist that avoid this jumping around the image. These are based on space filling curves which have the property of being a simple recursive pattern that visits every point in a 2D space exactly once. These curves were first discovered in the early 1900's and described mathematically by D. Hilbert [Hilbert, 1970]. One of the curves that Hilbert found, the Hilbert Curve, is particularly useful since it can be represented in the simplest level in a two by two square which is then recursively repeated for each quadrant of that first square—exactly as the quadtree does.

The path that the traversal of points follows becomes fairly complicated, Figure 4b. This means, again, that the calculation of neighbours becomes difficult.

5.2.3 Gray Codes

The Gray Code [Gray, 1953], developed by Frank Gray in 1953, was originally designed to reduce the error rate produced by mechanical electronics. The code is a variation on binary where each step when counting up changes only a single bit at a time. This meant that electromechanical apparatus was less likely to make a mistake or generate errors since the actions required to count from one to two required only a single bit change, rather than two, as would be required for binary counting. When using just two bits, i.e., counting from zero to three, the steps are very similar to binary, (00, 01, 11, 10).

The path that this follows is shown in Figure 4c. This does not seem to provide any benefits since there is now more jumping around the image space than with Z-order and the neighbours are just as difficult to calculate as for Hilbert Order. However, by using a different arrangement of the sub-trees, as the Hilbert curve does, the leaf nodes group themselves in a very ordered fashion. When arranged as in Figure 4d and 5, each cell is arranged such that

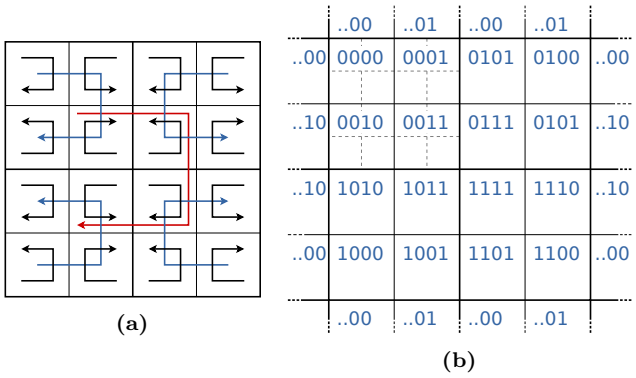


Figure 5: *The tree traversal that was developed for this project is a variation of the Gray code order discussed in Section 5.2.3. Instead of all sub-quadrants having the same orientation, each is reflected in either the x- or y-axis, (a). This has the advantage that neighbouring nodes have codes which differ by exactly one bit, (b).*

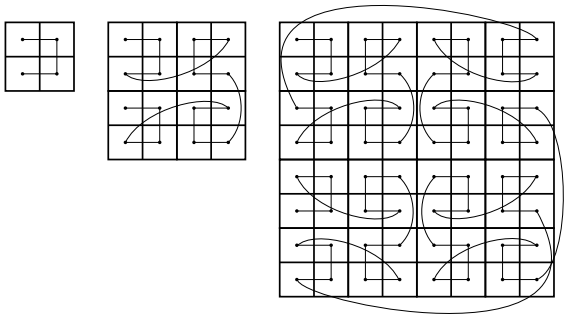


Figure 6: *modgray-Steps*

5.2.4 Other Orderings

As well as the orderings discussed above, there are also a number of other space filling curves and dis-joint orderings that were also discounted. Figure 7 shows some of these.

- Figure 7a shows *row order* traversal of the grid. This extremely simple to implement for a simple grid-like arrangement, but is very awkward and looses much of the information when using quadtrees.
- Figure 7b shows *row-prime order*, also known as snake-order [Goodchild and Grandfield, 1983], traversal. This is a slight modification of row order which is continuous, again, is not recursive so not suitable for use with quadtrees.
- Figure 7c is identical to them modified Gray order above, but with a single 90° rotation. Since the orientation of

an image has no effect on the clusters found, all rotations and reflections provide the same functionality as the original. The particular form chosen is simply due to æsthetic preference.

- Figure 7d shows an alteration to the modified Gray code order, named *pinwheel order*, where, instead of reflecting the quadrants, each is rotated about it's center 90° so that the whole image is rotational symmetric, something the modified Gray code lacks. This form turns out not to provide any additional features and makes conversion from quadtree code to coordinate representation substantially more difficult.

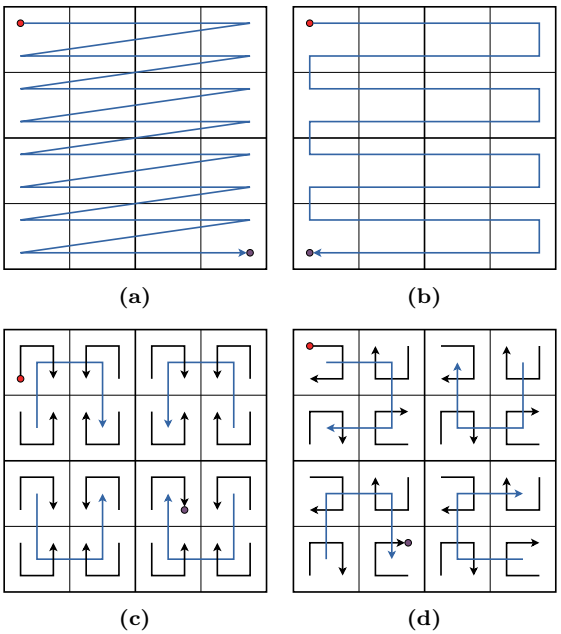


Figure 7: *There are a number of different space filling curves that can be used to map a two dimensional grid to a single dimensional array. To be considered for use in the quadtree ordering, the new ordering must provide some improvement of efficiency of some operations. Here are shown some orderings which were discounted.*

Since the quadtree is a recursive data structure, it is necessary to be able to maintain the correct orientation of child trees with respect to their parents at the construction stage. It turns out that this is easy to achieve and thus adapting it for each of the arrangements discussed above is simply a matter of adjusting the next part of the code that is added for each of the four children when creating them. Pseudo code to achieve this for the Morton order and the modified Gray code order is shown in Listings 1 and 2.

```
// Constructor
Quadtree(max_x, max_y, code)

// Child creation, Z-Order
t_l = new Quadtree(100, 100, this.code + "00");
t_r = new Quadtree(100, 100, this.code + "01");
b_l = new Quadtree(100, 100, this.code + "11");
b_r = new Quadtree(100, 100, this.code + "10");

// Child creation, Gray Code
t_l = new Quadtree(100, 100, this.code + "00");
t_r = new Quadtree(100, 100, this.code + "01");
b_l = new Quadtree(100, 100, this.code + "10");
b_r = new Quadtree(100, 100, this.code + "11");
```

Listing 1: *Code to generate children of the current quadtree while maintaining the correct ordering. Z- and Gray ordering.*

```
// Child creation, Modified Gray Code
this.position = pos;
if (pos == "tl") {
   new_code[0] = code + "00";
   new_code[1] = code + "01";
   new_code[2] = code + "10";
   new_code[3] = code + "11";

} else if (pos == "tr") {
   new_code[0] = code + "01";
   new_code[1] = code + "00";
   new_code[2] = code + "11";
   new_code[3] = code + "10";

} else if (pos == "bl") {
   new_code[0] = code + "10";
   new_code[1] = code + "11";
   new_code[2] = code + "00";
   new_code[3] = code + "01";

} else if (pos == "br") {
   new_code[0] = code + "11";
   new_code[1] = code + "10";
   new_code[2] = code + "01";
   new_code[3] = code + "00";
}

t_l = new Quadtree(100, 100, this.code + new_code[0]);
t_r = new Quadtree(100, 100, this.code + new_code[1]);
b_l = new Quadtree(100, 100, this.code + new_code[2]);
b_r = new Quadtree(100, 100, this.code + new_code[3]);
```

Listing 2: *Code to generate children of the current quadtree while maintaining the correct ordering. Modified Gray code order. Since the ordering is different for each quadrant, the ordering is changed depending on the position of the current node.*

5.3 Hash Table Implementation

In order to speed up the subsequent operations applied to the quadtree structure, it is converted to a simpler, one dimensional data structure. The method discussed above, to number the cells in a logical fashion based on the code of the parent, can be viewed as a way to map the two dimensional image to a single unique binary code, and hence to a one dimensional format. Thus, the quadtree can be easily converted to an array structure by simply using the key code as the array index. Once this is performed the access complexity is significantly reduced. This operation is only performed once after the quadtree has been generated meaning the amortized complexity is reduced.

As mentioned in Part II, using a naïve implementation has the potential to have a very poor space usage if the image is not densely populated (in which case it is likely that few clusters would be identifiable anyway). Instead, an implementation is created that makes use of the *Hash Table* data structure [Cormen et al., 2001]. This allows the data structure to increase in size dynamically as more space is needed, but also provides linear time complexity for access, modification and search. The quadtree code is used as the key for the hash table, and the data stored within the cell with that code is the hash table value.

Using this data structure means that several operations are significantly sped up. For example, for the quadtree format, the steps required to check if a given code is present involves traversing the tree as specified by the code to check and waiting to see if a node with that code is found, thus $O(\log n)$. However, for a hash table, the hash function is used to get a hash of the code to be checked and the appropriate location checked. If the codes match, then the code does appear in the tree, a total of two operations, constant $O(1)$.

Also, since the structure is defined recursively, it is not possible to directly access a given location of the data. Instead it must be arrived at by starting at the root, and, for every level, deciding which of the children the destination exists in. This would be a time consuming step for a number of operations, but the biggest effect would be when checking the neighbours of cells since for every neighbour of every node, the tree must be traversed. This is not an issue for the hash table since the single dimensional nature means that data can be accessed anywhere directly.

Little spatial information should be lost during the conversion from quadtree to hash table since this is all contained in the quadtree code assigned to the node during the quadtree generation step. However, the quadtree representation can be kept in memory for some processes. For example, to get all of the leaf nodes of a given node, it is enough to start at that node and traverse the tree in post- or pre-order and return the nodes that are arrived at, $O(\log n)$, whereas, for the hash table, each node must be checked to decide if it is a child, $O(n)$.

A hash table requires two pieces of information for each entry: a *key* and a *value*. The key is what is used to locate items, its hash is used as the index of the array used internally. The value is the information that should be accessible when using the key. For this application, since more than a single piece of information should be accessible for each quadtree code, an object is stored against each key. This has the format shown in Table 2.

Key	Value
Quadtree Code	• Set of points in this node • Cluster these points exist in • Size of this node • Number of edges this node contributes to the perimeter of the cluster

Table 2

Part III

Cluster Analysis

Cluster analysis is the grouping of a set of objects or items in a spatially or informationally logical way such that the items that are placed in the same group are more similar to each other than they are to the objects in the other groups in the set. These groups shall be called *clusters*. When dealing with images, the clustering that is of interest is based on spatial location, i.e., clusters should be composed of objects that are close together in the image and clusters should be separated by regions of emptiness or background level noise.

One of the primary reasons for chosing the quadtree method over the simple grid methods was that the simple act of placing the objects, in this case coordinates of data points, into the quadtree starts the process of analysing the data. Since the points end up in a tree structure with the number of points closely separated being on the lowest levels of the tree, the data is already clustered in a way.

There are a number of alternative methods of identifying clusters in images.

6 Rolling Ball Analysis

The accessible surface area (ASA) algorithm, also known as the "Rolling Ball Method", is a technique used in image processing for describing the outer limit of a cluster of points. It is derived from biological molecules analysis where it describes the surface area of a molecule that is accessible to a solvent.

The rolling ball method can be used to analyse a cluster of points by imagining a solid ball that sits against one of the outer-most points. From here it is "rolled" around the cluster such that it is always touching at least one point. Once the ball has reached the point it started at, the line that the ball traced is reduced in size by the radius of the ball. This line then represents the outer limit of the cluster.

The size of the ball must be chosen depending on the average separation of the points within the cluster.

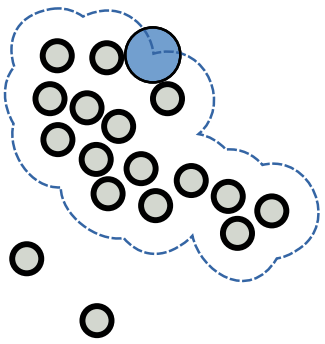


Figure 8: *The rolling ball method for cluster detection provides a way of identifying clusters, as inspired by molecular biology. A very simple implementation can be fast but is not particularly successful at finding clusters unless the data points are very dense and there is no noise.*

A simple approximation of this technique can be acheived by using the same `open`-ing and `close`-ing processes as are described in Section 4.

7 Quadtree Traversal

Whether the quadtree is stored in memory as a recursive quadtree data structure, or as hash table, Section 5.3, the most important and computationally intensive step is extracting the clusters at the correct depth and disregarding those data points that can be attributed to noise.

The quadtree numbering system chosen lends itself very well to analysis based on spatial location and the proximity of neighbours to a given node being examined.

7.1 Algorithm Description

The propagation algorithm is performed as follows:

1. Choose a starting location that has not yet been checked.
2. For this given starting location, the n neighbours are checked for validity based on the conditions of the analysis (the way these neighbours are selected is discussed in Section 7.3).
3. If a node fails the check, then it is recorded as having done so and shall not be checked again. If it passes, then it is itself propagated.
   - To be "propagated" means the perform this propagation algorithm using that node as the starting location.
4. When all nodes have been checked, return to step 1.
5. The algorithm completes either when either;
   - every one of the nodes in the image have been checked and included or ignored, or
   - the cluster has ended, so all the neighbours of all checked nodes have failed the validity tests and no further starting locations were found or specified.

This process is shown graphically in Figure 9. Here, and from now on, blue nodes are nodes that either need to be checked or are currently being checked; red are nodes that have been checked and have been included in the cluster; question marks (?) represent the neighbours of the blue nodes that will have to be checked to determine if it is included or not and crosses (x) represent neighbours that have been checked and excluded.

```
1  public propagate() {

3     node[] start_locations = get_start_locations()
      for each node in start_locations {
5        propagate(node)
      }
7  }

9  private propagate(node) {

11    if (not node.inCluster()) {
         node[] neighbours = get_node_neighbours(node)
13
         for each neighbour in neighbours {
15          propagate(neighbour)
         }
```

```
17      }
    }
```

Line 3 A set of starting locations is selected based on some heuristic that determines if a node is deep enough in the tree to be used as a starting location.

Line 11 Nodes are only propagated if they have not already been included in a cluster. If this were not the case, then clusters would overlap.

Line 14 Each of the neighbours of the current node is propagated. The method of choosing neighbours determines how far the cluster can spread, and is separated from the clustering algorithm.

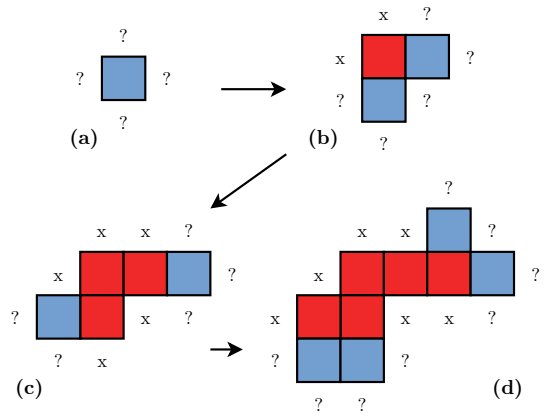


Figure 9: *For a starting location, (a), in an image, no information is known and so the neighbours are checked. Some of these are found to be part of the cluster, others are rejected. The ones that are included are then, themselves, checked and so on. As the cluster grows, (b), (c) and (d), the number of checked nodes increases.*

When discovering clusters via this propagation technique, care must be taken to avoid a run-away situation where every node in the tree gets included. This would happen when looking at the neighbours of a node and blindly including them. Since every internal node has exactly four neighbours, the propagation would terminate only when reaching the edge nodes.

Instead, the depth of the node must be considered. Again, the simplest method is not sufficient. If the propagation is limited to a given node depth, even if this is not the same as the deepest node, the size of any clusters that are identified will be limited, as shown in Figure 10a. Since the depth to consider is not able to change, when the neighbours of the blue node are checked, no correct neighbours are found and so the process terminates. When able to view the larger structure of the nodes, however, it is clear that the structure continues beyond the gap.

To avoid this, a certain amount of leniency must be given when deciding what constitutes a neighbour. Given an appropriate value, this would allow both of the larger white cells in Figure 10a to be included.

The term *depth range* shall define the levels that are to be considered when choosing neighbours with respect to a target depth. Since clusters are being considered as areas of increased

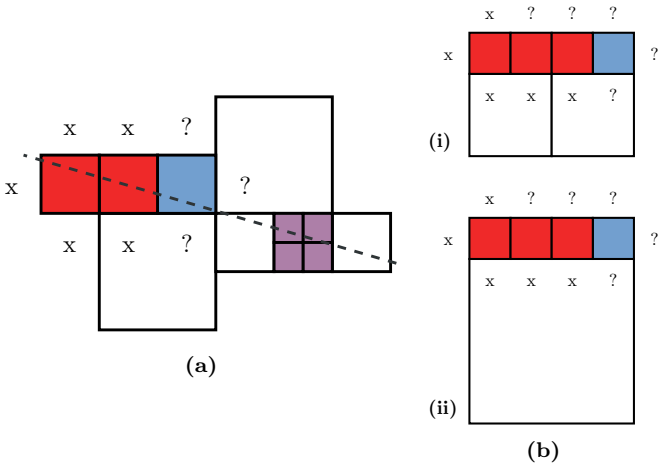


Figure 10: *Considerations regarding quadtree levels that will be accepted when propagating a node in a quadtree. (a) If the depth range that specifies how far up the tree to look for valid neighbours is too small then a cluster might be terminated too soon. (b) (i) a depth range of two and, (ii) a depth range of three.*

density of points, all cells with a depth greater than the target depth shall be allowed, so the purple cells in Figure 10a would be included when the target depth is the same as the depth of the included red cells. A depth range of zero is equivalent to the situation above where only cells of a given depth are considered. A depth range of 1 would mean that the white square in Figure 10b (i) would be included but not in (ii), whereas a depth range of three would include both and so on.

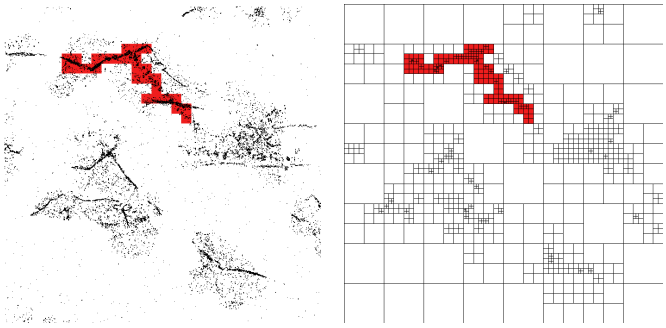
7.2 Clustering Start Locations

In order for the algorithm to proceed correctly, a good initial node, a starting location, must be chosen. Since the clusters to be found are regions of high point density, it makes sense to start the clustering algorithm at the point in the image with the highest point density. This should ensure that the most defined cluster is always found with subsequent clusters being less dense, and so less well defined.

When starting at the highest density node, i.e., the node which is at the deepest level in the tree, propagating this node and then terminating; the cluster shown in Figure 11 is found. The file `palm-1.txt` was used and generated this data in 457 ms. This shows that the algorithm works correctly. Altering the parameters that are used to generate the quadtree affects the size of the nodes that are included in the cluster and the depth that is searched.

In order to find other clusters, the algorithm must be restarted with a new starting location. This is chosen as the deepest node in the tree that is not already included in a cluster. There are a number of different way to terminate this process of finding new clusters:

- Perform a set number of iterations. This performs well if the clusters to be located can be easily counted, but in the general case, this is not possible or desired. If the number of iterations is set to a high value, of the order of 50, the algorithm will continue to locate “clusters” even if they do not exist and will eventually simply report the background noise as a cluster. To prevent this, a limit can



(a) (b)

Figure 11: *Inital versions of the clustering algorithm terminated immediately after finishing propagating the first cluster. This gives a useful test as to the correctness of the algorithm since there are no other clusters for this to collide with. It is useful to be able to check both (a) the points that were considered to be in the cluster and (b) the nodes in the quadtree that were included and where the propagating algorithm stopped.*

be set on the depth a starting location must be in order to be valid.

- Continue iterating until a depth limit is reached. This is a more general form of the previous case, but for this case, the limit on the depth of a starting location will always be reached.

- Allow the user to make a number of starting point location selections manually. Since ImageJ allows multi-point region of interest (ROI) selections, the user could be asked to place a new ROI at the places they consider a reasonable place to find a cluster. The algorithm would then convert the locations of each of these ROI points into the relevant quadtree code and begin propagating from that node and terminate when all of the ROI's had been used.

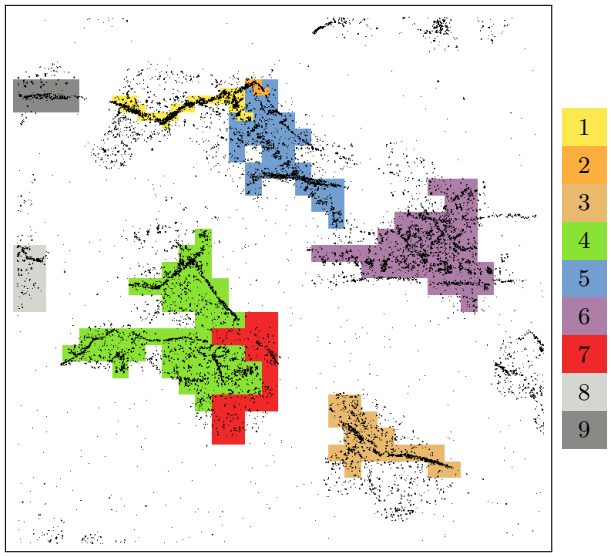


Figure 12: *Initial versions of the clustering algorithm included too many nodes, making the clusters too large. This image shows how each node cluster that is started is independent of the previous ones. The clusters were identified in the order they are listed on the right.*

7.3 Choosing Neighbours

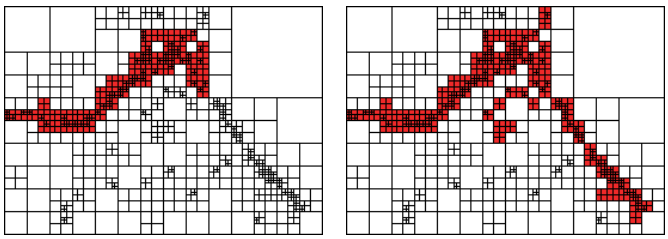
Some care must be taken when deciding what constitutes a neighbour of a node and what does not. As mentioned above, when detecting clusters using propagation, the neighbours of a node are checked and, if they are valid, are themselves propagated. For this reason, a poor choice of neighbours means that either the propagation will either:

- be cut short too early, and so not all of the clusters will be located, or

- include too many nodes, in which case the clusters will not represent the actual data.

The first neighbours that must be considered, named *rook's case* neighbours by [Abel and Mark, 1990], are the four nodes that lie to the north, east, sound and west of the current node. These are the nodes that are directly in contact with the node and so, if they are valid, represent a direct continuation of the cluster.

However, if the choice of neighbours is limited to these four, some major structures are missed. Figure 13a shows how this arrangement misses a large portion of the cluster, simply because the propagation could not consider the nodes across the boundary. If, in addition to the rook's case, the four *diagonal* neighbours are included, giving a total of eight, the results are much more complete, as shown in Figure 13b.



(a) (b)

Figure 13: *A comparison between different neighbour sets. (a) shows how some of the cluster is lost when using just the rook's case neighbours, whereas, using all eight neighbours, Figure (b) more of the cluster is included.*

The requirement to include a total of eight neighbours for each node suggests that it might be of value to be able to specify an arbitrary number of neighbours around the given cell.

In many applications in image processing, the concept of an image *kernel* is commonplace. This is a square, odd-sided matrix that is used to apply filters and other effects to an image. The kernel is placed over each of the pixels in the image in turn so that the central element in the matrix is over the current pixel and each of the other elements is over another pixel. The value of the element in the kernel is then used to manipulate the pixels in the image below it. For example, to blur an image, a 3×3 kernel composed of all $^1/_9$'s can be used. When applied, this would mean that the current pixel is given a value which is the sum of each of matrix elements multiplied by the pixel value beneath it.

A similar technique is used to choose neighbours from the quadtree. The image kernel that is used is a binary matrix, meaning that all of the elements are restricted to 0 or 1, but, in all other respects, is the same as a regular image kernel. The kernel is placed over a node of interest. Any nodes that lie

under an element with a value of 1 is included as a neighbour and 0 elements are ignored so that no neighbours are chosen. The value of the central element in the kernel does not matter, but the convention shall be to set this to 1.

Thus, the simpest kernel, though of least use, is the identity kernel which has an element with value 1 in the center and all other elements 0. This would result in no neighbours being selected, and so no clusters located. The rook's case neighbours are now represented with the first kernel in Figure 14.

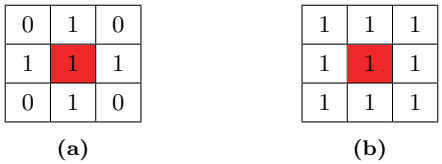


Figure 14

This functionalilty provides the ability to target certain clusters depending on the spatial orientation of the cluster, vertical or horizontal etc. Figure 15 shows a number of possibilities of kernel shapes that would allow different types of clusters to be targeted.

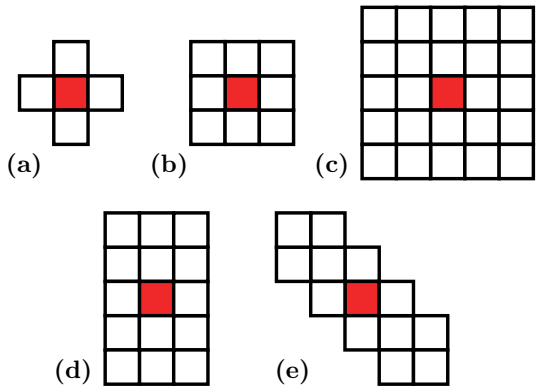


Figure 15: *Different kernel shapes can be used to find different sorts of clusters. (a) The basic rook's case neighbours and (b) the full number of closest neighbours are the simplest general cases, used for most clusters. (c) the area to search can be extended by including more neighbours—this could help find clusters that are more sparsely populated. Clusters of a specific orientation can be located, for example, predominately (d) vertical or (e) NW SE diagonal.*

Part IV

ImageJ Plugin

ImageJ is an public domain, Java based image manipulation program written and maintained by developers at the National Institute of Health. It is widly used in medical and biological research and has an open API to allow extension via macros, plugins and scripts.

Since they offer significantly betting integration, meaning speed and efficiency improvements, a plugin is chosen to inegrate this project into ImageJ over macros, which suffer performace loss when more than a few steps are involved, and scripts which do not offer such tight integration with the rest of the program.

The plugin shall allow a user to load a data set, as gathered from STORM, PALM or similar imaging techniques discussed in Section 2, choose the format of the data (the coloumns that are of interest etc.) analyse the data for clusters and receive detailed information regarding the clusters that were found.

8 ImageJ Plugin

The main deliverable for this project is a plugin for extracting clusters of points from large data sets, for the image processing program ImageJ. This plugin is written in Java and makes use of the ImageJ Java API [NIH, 2014]. Each of the following sections describes a part of the plugin. The general flow of activities from opening the plugin is:

1. click the `Data File` button,
2. choose the data file to analyse,
3. If the default separator is incorrect, input the correct one and click the `Read File` button to re-read with the new separator,
4. select the values for the x- and y-columns,
5. adjust the sliders to select the parameters that are used to build the quadtrees,
6. adjust the checkboxes to change the way the final image is displayed,
7. click the `Quadtree` button to start the process of reading and analysing the selected file.

8.1 Column Picker

To make the plugin more general purpose than being limited to only data formatted in precisely the same way as the data in the sample files used during development, the ability to specify some formatting features of the data file used are available. The user can specify the delimiter used in the file (how the columns are separated; comma, space tab etc.), and which column represents the x- and y-coordinates. These are chosen through a separate GUI.

8.2 Option Sliders

The creation of the quadtrees is subject to a number of parameters that affect the way points are added, and how it manages the propagation once all points are added. A summary of the options available and what effect they have is given below.

Density As points are added to a leaf in the quadtree, the capacity of the leaf is checked against this value. If the number of points in that leaf exceeds this value, the points are removed from the leaf, four new quadtrees are created, and each of the points added in to the appropriate new leaf, this is known as *de-leafing*.

An extremely large value for the denisty, greater than $^1/_4$ times the number of points in the data file, would result in only the first level of nodes being populated (since the root does not hold points) and the final quadtree having only 4 leaves. A value of 1 would mean that there would be the same number of leaf nodes as points in the file. This can lead to out of memory errors in some cases where the file is large and a large value for depth range is set.

A value of the order of 20 is recommended as this reduces the number of quadtrees created by 10, but still provides enough resolution for propagation. The default value is 20.

Depth Range This controls how far up the quadtree will be searched when looking for neighbours as described in Section 7.3. All neighbours that are deeper in the tree that the node being propagated are included irrespective of this setting. A large value for the depth range will mean that the propagation will go too far up the tree and, ultimately, will reach the root, thus automatically including the whole tree in the located cluster. A value of 0 means that only nodes that are in the same level as the current node, or deeper, are included.

The depth range should always be considerably less than the max depth. The default value is 3.

Max Depth As well as being controlled by the number of points in the leaf, adding points to the quadtree is also limited by the maximum depth. If the depth of a node reaches the maximum depth, then it is not de-leafed even if the number of points in the node exceeds the density. This allows a high value to be set for the denisty so that where the points are relatively clustered, the tree is created deep enough to distinguish them, but to prevent the depth getting very high for regions of localised, abnormal high clustering that might skew the results.

A small value for max depth means that the tree will never get very deep. If there is noise in the points, this can cause the depth of the noise and the depth of the clusters to be too similar and so mean clustering is not effective. Setting it to a high value can effectively remove the constraint.

Since the number of nodes in the quadtree increases exponentially with the depth, a large value for the maximum depth can lead to memory errors, though for typical data sets, it is unusual to exceed a depth of around 16. As long as this depth is localised to small areas, this should not be a problem. A depth of 16 provides a maximum of

4.29×10^9 leaf nodes, so even extremely large files can be represented to a high degree of accuracy.

The following are settings that affect the way the clusters are displayed once they are found.

Cluster Size Once the clustering algorithm has completed, the value places a limit on the minimum size of a cluster. Any clusters that are smaller than this will not be drawn in the image and will not have an entry in the results table. This can be useful for a noisy data set that contains many small clusters which make the interesting clusters less easy to see. The value is given as a fraction of the overall size of the image, so a value for the cluster size of 0.05 means that only cluster which are larger than $^1/_{20}$ of the size of the whole image are displayed.

A value of 0 means that no clusters are hidden.

Scale Internally, the points read from the file are stored exactly as they are read. The pixel grid for the image is then generated based on these values. If the maximum x- and y-coordinate are both 10 000, with a scale value of 1, the image would be 10 000 pixels by 10 000 pixels. Since each pixel requires a number of type *float* to hold the RGB colour information, this would be far too large to draw (giving an image with a size of 10 GB). Instead, the size of each coordinate is multiplied by the scale value to reduce the overall size of the image. Because this uses integer devision, there is a certain degree of error that is introduced by this process. This can be seen when viewing the quadtree structure of a deep node where the size of neighbouring nodes at the same level do not appear to be the same size. This alteration is performed only on the pixel information, the original data is maintained and used for all subsequent operations.

A default value of 5×10^{-4} is used. This gives an uncompressed image size of 11 MB for a data set with maximum x- and y-coordinates of around 50 000.

Lines

Points

Colourize

8.3 Displaying the QuadTree

The first version of the plugin simply allowed the user to visualise the data, once it had been processed and entered into a quadtree. Though this provided little benefit to the researcher producing data, it can be a useful tool to get an insight into the process that a program is using to analyse one's data so that the results can be better interpreted. For this reason, this version served as a foundation for later versions of the plugin so that, when loading data, a user gets the opportunity to view the data before proceeding with the analysis.

Again, earlier versions of the program displayed this data using the built-in GUI classes in the AWT [Zukowski, 1997] and Swing [Loy et al., 2002] libraries included in the standard Java distribution. This effectively prohibited any further actions being performed on the image once it had been generated since what was displayed was only modifiable by the JVM via compiled code. It was also very memory intensive since, in many cases, many thousands of separate objects (data points represented by zero length lines, quadtree cells by boxes, etc.) and so was slow to draw initially and redraw with any subsequent move or resize of the window.

The code used to generate this view of the data was modified to make use of the easy image generation functions present in ImageJ. Now, instead of many different objects being manipulated for each view of the data, a single array with a value for each pixel is needed. For the cases where the user wishes to view the clusters that have been found, but not have them affect the image, the image is created with a number of different *slices* in the image *stack*, Figure 16. Slices are ImageJ's representation of images with two or more layers, or alternative views, each of which resides in a stack of slices. Each slice in a stack must have the same dimensions.

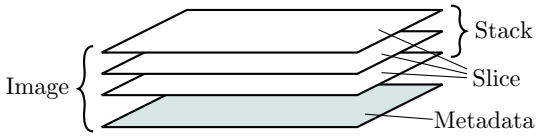


Figure 16: *An "image" in ImageJ can be composed of a number of layers, each of which is easily viewable and can be saved on its own. These layers are used to display different parts of the generated image: data points, located clusters, quadtree structure, etc.*

8.4 Results Table

In addition to the image of the data with the clusters, the plugin also calculates a number of statistics for each of the clusters that are found. These include the number of points that are included in the cluster, the area of the cluster, and it's perimeter. The area and perimeter values are given as a fraction where the whole image represents 1. Thus, to find the actual area for the real life object, the area of the cluster should be multiplied by the area of the image, and for the perimeter, the length of one side of the image should be multiplied by the perimeter.

To calculate each of these values, each node that exists in a cluster is examined and its contribution to the overall value added.

9 Cluster Analysis

Once the data has been processed and a number of clusters have been identified, they can be displayed on screen in an image to verify what was found and perform further analysis. However, since the data that was used to find the clusteres is still held in memory at this point, it is useful to take advantage of this and perform some imediate analysis and provide some statistics regarding the clusters that were found.

There are two ways of conceptually viewing the clusters which will lead to slightly different results when analysing them.

- The first is to consider the boundary of the nodes of the quadtree that were considered to be part of a cluster to be the boundary of the cluster. This will mean that the actual cluster is likely to be slightly larger than the actual data points that it is comprised of, but is computationally

simple to acheive, so fast, and can take into account any holes in the cluster.

- The second way is to, once the cluster has been located, disgard the information about the nodes themselves, and simply use them to select the appropriate points. This results in a set of points that are all considered to be spatially grouped into the same cluster. From this set, calculations can be performed on the real data. This method is garanteed to provide information that more closely represents the original data, though is more computationally intensive.

9.1 Quadtree Node Analysis

Here, the nodes of the quadtree that were considered part of the cluster are considered, rather than the points that they contain.

9.1.1 Cluster Area

To calculate the area of the clusters, each node in each cluster is examined. Since the size of the node must be known, it is calculated from the quadtree code for that node. The formula is shown in Equation 4,

$$a = \frac{1}{4^d} \tag{3}$$

$$a_i = 4^{-l_i/2}, \tag{4}$$

where a_i is the area of the node i, d is the depth in the quadtree and l_i is the length of the quadtree code of that node. For every node in the cluster, where there are n nodes, this value is summed to give the total cluster area, A:

$$A = \sum_{i=0}^{n} a_i. \tag{5}$$

9.1.2 Cluster Perimeter

Similarly to the cluster area, the perimeter is given as a fractional value of the length of one side of the whole image, as calculated for each node from the quadtree code, as shown in Equation 7,

$$p = \frac{1}{2^d} \tag{6}$$

$$p_i = 2^{-l_i/2}, \tag{7}$$

where the symbols have the same meaning as above.

However, this simply gives the length of one side of the node for any node. In order to calculate the perimeter of the cluster, it is not enough to simply sum these values, as for the cluster area, since not all nodes contribute to the perimeter. Instead, for each node, it must be decided whether it contributes to the perimeter and how much (1, 2, 3 or 4 sides), and then increase the total perimeter by this many times the length of one side. The total perimeter, P, then is given by Equation 8,

$$P = \sum_{i=0}^{n} s * p_i, \tag{8}$$

where $s \in \{0..4\}$. This is demonstrated in Figure 17 where the perimeter is simple to calculate in case (a) as the nodes

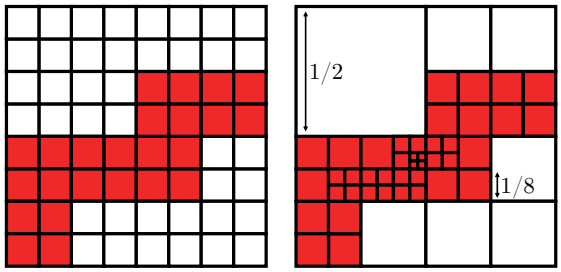


Figure 17: *When calculating the perimeter of a cluster using the nodes that contribute, the size of each node must be taken into account. In case (a), the process is simple since all nodes are the same size and a fractional value of 3.5 is calculated. For case (b), the steps are 25 lengths of size $^1/_8$ and 6 lengths of size $^1/_{16}$ which gives the same result, 3.5.*

are all the same, but the size of each node must be taken into account in case (b).

Within the cluster, *holes* occur where a node, or number of nodes, is surrounded on all sides by the same cluster. An example can be seen in Figure 13. Unfortunately, these are included in the calculation of the perimeter and so, where holes exist in a cluster, the actual perimeter is slightly smaller than that calculated.

9.1.3 Cluster Roundness

A potentially useful measure of a cluster is its *roundness*. This describes the extent to which the area and perimeter of the cluster resemble a circle. The avialible values of roundness are from 0, meaning a perfect line with no area but finite perimeter, to 1, being a perfect circle. The equation to determine roundness, Equation 9 is defined such that it is a unitless ratio of area and perimeter, such that a circle has roundness 1. The derivation of Equation 9 is found in Appendix B.

$$R = \sqrt{\frac{4\pi A}{p^2}} \tag{9}$$

The clusters that are found for data sets `palm-1.txt` and `palm-2.txt` are shown in Figure 18. The roundness values for the clusters for these data are very different, Figure 18a has an average $R = 0.350$ whereas Figure 18b has an average $R = 0.446$.

9.2 Point Analysis

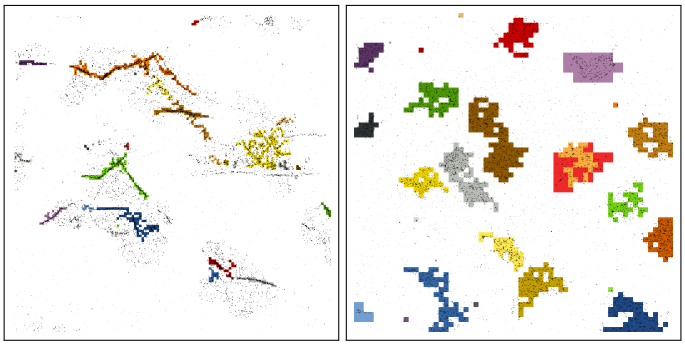
Here, the points contained in the nodes of quadtree that were considered part of the cluster are considered; the nodes are only used to select the correct points.

9.2.1 Cluster Area

9.2.2 Cluster Perimeter

[Lee and Estivill-Castro, 2002]

[Estivill-Castro and Lee, 2000]



(a) **(b)**

Figure 18: *The general shape of the clusters found can be described using the roundness measure. A value closer to 0 means longer and thinner clusters, whereas values closer to 1 mean clusters that are more circular.*

[Xia et al., 2006]

[Lee and Schachter, 1980]

Part V
Further Considerations

Some further considerations—what went well, what could have been better, improvements, next steps etc.

10 Possible Improvements

References

[Abel and Mark, 1990] Abel, D. J. and Mark, D. M. (1990). A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information System*, 4(1):21–31.

[Asano et al., 1997] Asano, T., Ranjan, D., Roos, T., Welzl, E., and Widmayer, P. (1997). Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15.

[Bailer, 2006] Bailer, W. (2006). Writing imagej plugins—a tutorial. *Upper Austria University of Applied Sciences, Austria*.

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. (2001). *Introduction to algorithms*, volume 2. MIT press Cambridge.

[Estivill-Castro and Lee, 2000] Estivill-Castro, V. and Lee, I. (2000). Autoclust: Automatic clustering via boundary extraction for mining massive point-data sets. In *In Proceedings of the 5th International Conference on Geocomputation*. Citeseer.

[Fang et al., 2005] Fang, N., Lee, H., Sun, C., and Zhang, X. (2005). Sub–diffraction-limited optical imaging with a silver superlens. *Science*, 308(5721):534–537.

[Goodchild and Grandfield, 1983] Goodchild, M. F. and Grandfield, A. W. (1983). Optimizing raster storage: an examination of four alternatives. In *Proceedings of Auto-Carto*, volume 6, pages 400–407.

[Gray, 1953] Gray, F. (1953). Pulse code communication. US Patent 2,632,058.

[Hess et al., 2006] Hess, S. T., Girirajan, T. P., and Mason, M. D. (2006). Ultra-high resolution imaging by fluorescence photoactivation localization microscopy. *Biophysical journal*, 91(11):4258–4272.

[Hilbert, 1970] Hilbert, D. (1970). Über die stetige abbildung einer linie auf ein flächenstück. In *Gesammelte Abhandlungen*, pages 1–2. Springer.

[Hjaltason and Samet, 2002] Hjaltason, G. R. and Samet, H. (2002). Speeding up construction of pmr quadtree-based spatial indexes. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(2):109–137.

[Lee and Schachter, 1980] Lee, D.-T. and Schachter, B. J. (1980). Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242.

[Lee and Estivill-Castro, 2002] Lee, I. and Estivill-Castro, V. (2002). Polygonization of point clusters through cluster boundary extraction for geographical data mining. In *Advances in Spatial Data Handling*, pages 27–40. Springer.

[Loy et al., 2002] Loy, M., Eckstein, R., Wood, D., Elliott, J., and Cole, B. (2002). *Java swing*. " O'Reilly Media, Inc.".

[Mokbel et al., 2002] Mokbel, M. F., Aref, W. G., and Kamel, I. (2002). Performance of multi-dimensional space-filling curves. In *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 149–154. ACM.

[Morton, 1966] Morton, G. (1966). A computer oriented geodetic data base and a new technique in file sequencing. *IBM, Ottawa, Canada*.

[NIH, 2014] NIH (2014). *ImageJ API Documentation*. US National Institutes of Health, 1.48t edition.

[Owen et al., 2010] Owen, D. M., Rentero, C., Rossy, J., Magenau, A., Williamson, D., Rodriguez, M., and Gaus, K. (2010). PALM imaging and cluster analysis of protein heterogeneity at the cell surface. *Journal of biophotonics*, 3(7):446–454.

[Rasband, 1997] Rasband, W. (1997). ImageJ, US National Institutes of Health. *Bethesda, Maryland, USA*, 2012.

[Rust et al., 2006] Rust, M. J., Bates, M., and Zhuang, X. (2006). Sub-diffraction-limit imaging by stochastic optical reconstruction microscopy (STORM). *Nature methods*, 3(10):793–796.

[Samet, 1990] Samet, H. (1990). Applications of spatial data structures. *Computer Graphics, Image Processing, and GIS*.

[van Oosterom, 1999] van Oosterom, P. (1999). Spatial access methods. *Geographical information systems*, 1:385–400.

[Williamson et al., 2011] Williamson, D. J., Owen, D. M., Rossy, J., Magenau, A., Wehrmann, M., Gooding, J. J., and Gaus, K. (2011). Pre-existing clusters of the adaptor lat do not participate in early t cell signaling events. *Nature immunology*, 12(7):655–662.

[Xia et al., 2006] Xia, C., Hsu, W., Lee, M. L., and Ooi, B. C. (2006). Border: efficient computation of boundary points. *Knowledge and Data Engineering, IEEE Transactions on*, 18(3):289–303.

[Zukowski, 1997] Zukowski, J. (1997). *Java AWT reference*, volume 3. O'Reilly.

Part VI
Appendix

A Data File Structure

The data files that are produced from the initial analysis of the images have a standard format.

1. Tab separated fields.
2. Single header line with names of fields.
3. One or more item of data, separated by newlines.

The columns that represent fields in the file are as follows.

Header	Meaning	Used?
`Channel Name`	Wavelength channel that was used to capture data. First value, I, is the incident wavelength of the light used to excite the dye and the second, E, is the wavelength emitted that was imaged.	no
`X`	x-coordinate of the point	no
`Y`	y-coordinate of the point	no
`Xc`	centered, normalised x-coordinate of point	yes
`Yc`	centered, normalised y-coordinate of point	yes
`Height`	the height of the fitted gaussian peak used to extract the point from the original image	not yet
`Area`	area of the point	not yet
`Width`	full width half maximum of the point	not yet
`Phi`		no
`Ax`		no
`BG`		no
`I`		no
`Frame`		no
`Length`		no
`Valid`		no
`Z`		no
`Zc`		no
`Photons`		no
`Lateral Localisation Accuracy`		no
`Xw`		no
`Yw`		no
`Xwc`		no
`Ywc`		no

B Roundness Derivation

Derivation of the equation for roundness, R, of a cluster given the area, A, and perimeter, p, of the cluster.

A circle is defined to have a roundness of 1.

$$R_{\text{circle}} = 1 \tag{10}$$

The value must be unitless, so divide area by perimeter squared,

$$A = \pi r^2 \tag{11}$$
$$p = 2\pi r \tag{12}$$
$$\frac{A}{p^2} = \frac{\pi r^2}{4\pi^2 r^2} \tag{13}$$

Remove constants to give formula,

$$R = 4\pi \times \frac{\pi r^2}{4\pi^2 r^2} \tag{14}$$
$$= \frac{4\pi A}{p} \tag{15}$$

Test for regular shapes.

Circle:

$$A = \pi r^2 \tag{16}$$
$$p = 2\pi r \tag{17}$$
$$R = 4\pi \times \frac{\pi r^2}{(2\pi r)^2} = 1 \tag{18}$$

Square:

$$A = h^2 \tag{19}$$
$$p = 4h \tag{20}$$
$$R = 4\pi \times \frac{h^2}{(4h)^2} = \frac{\pi}{4} \approx 0.785 \tag{21}$$

Equalateral Triangle:

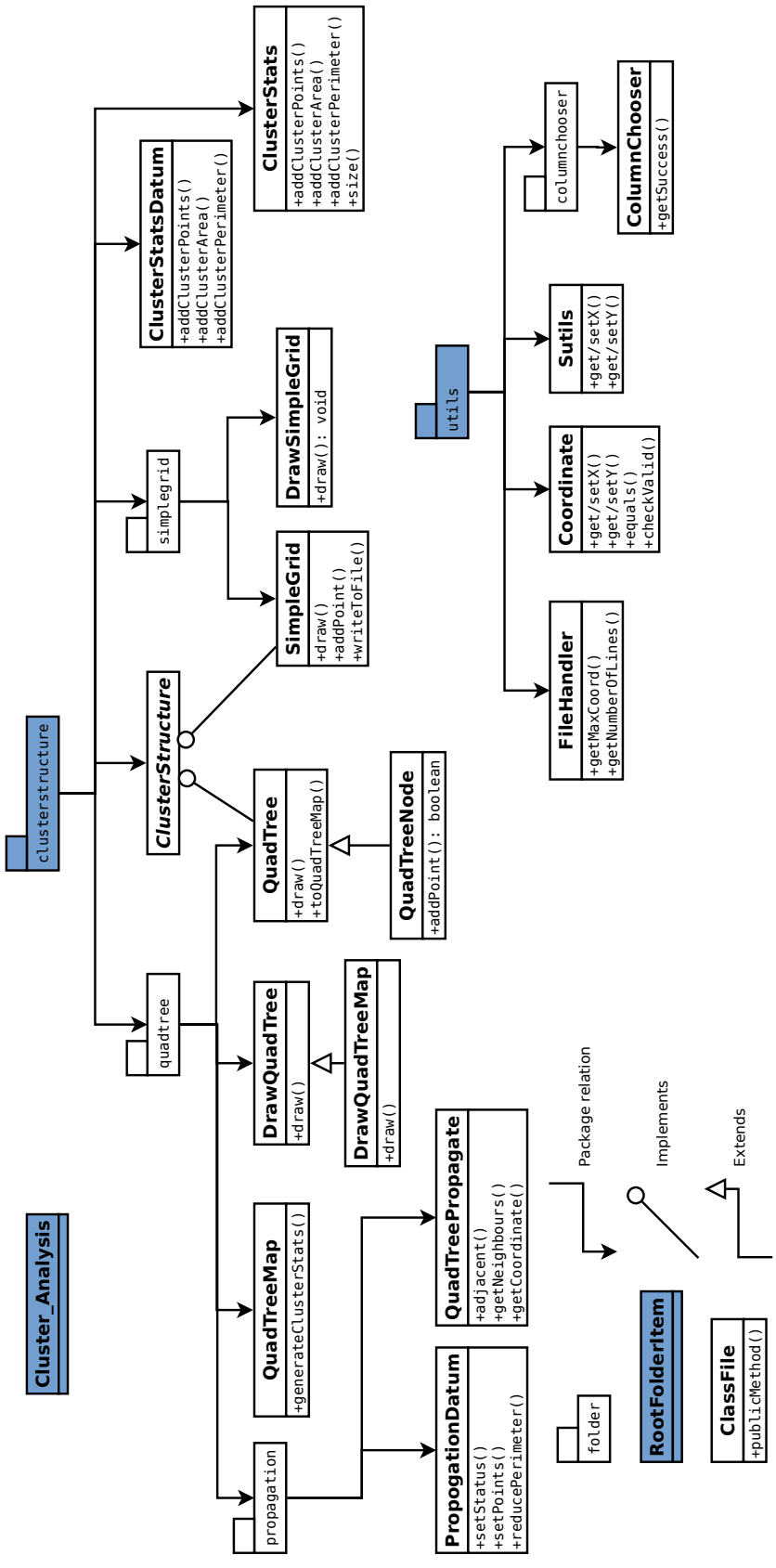
$$A = \frac{\sqrt{3}}{4} a^2 \tag{22}$$
$$p = 3h \tag{23}$$
$$R = 4\pi \times \frac{\frac{\sqrt{3}}{4} a^2}{(3a)^2} = \frac{\sqrt{3}\pi}{9} \approx 0.605 \tag{24}$$

Elipse (Eccentricity $= 0.5$): For ellipse, $\epsilon = 0.5$, let minor axis, $a = 1$ and major axis, $b = 2$.

$$A = 6.28 \tag{25}$$
$$p = 9.69 \tag{26}$$
$$R = \frac{6.28}{9.69} \approx 0.648 \tag{27}$$

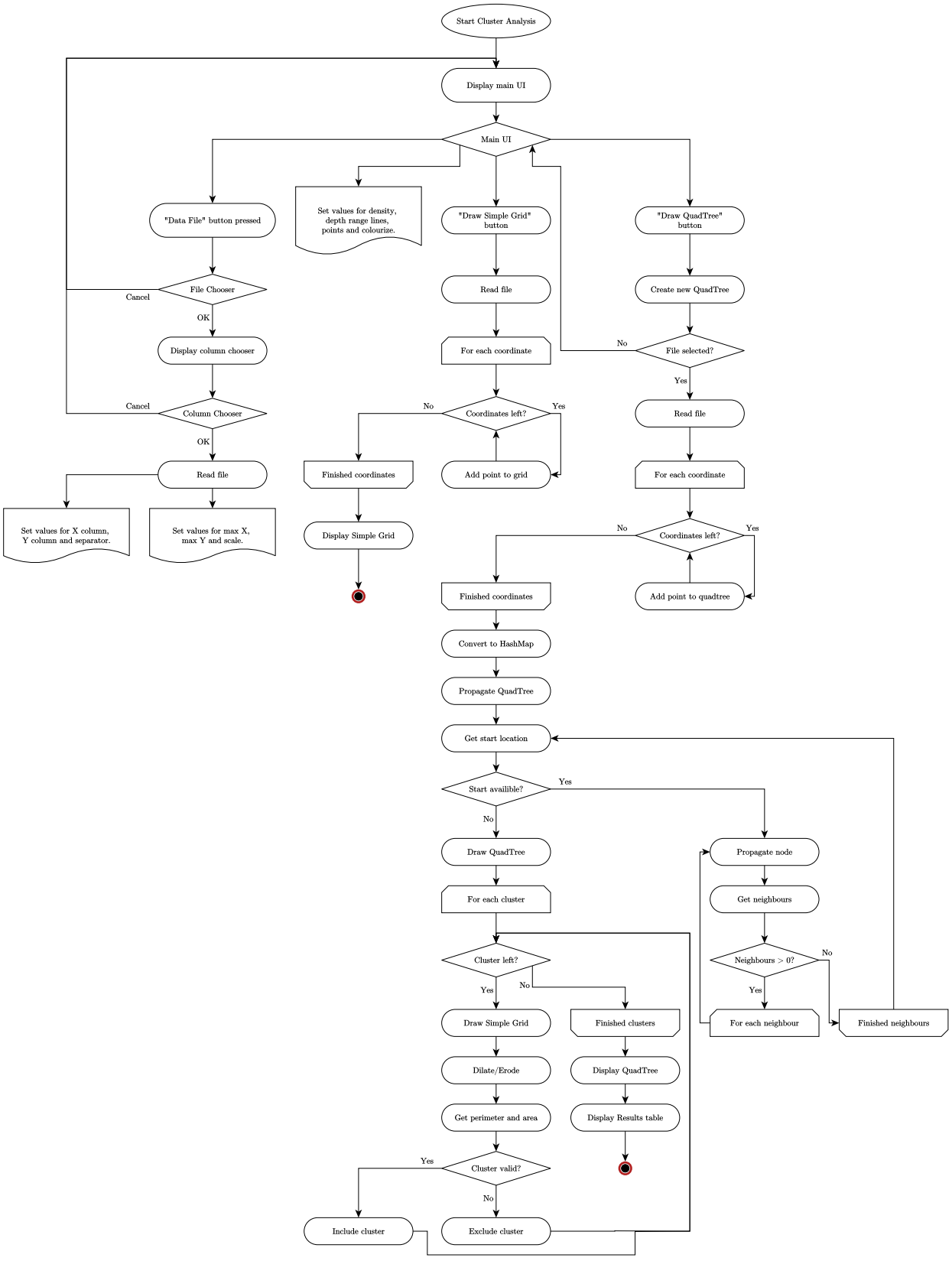
C Class Diagram

Figure C is a class diagram showing the structure of the ImageJ plugin that was created for this project.



D Flow Diagram

Figure D is a flow diagram showing the flow of events from initialisation of the plugin from ImageJ through to drawing the clusters found to an image of the data set on screen.



E Add-point Flow Diagram

Figure 19 is a flow diagram showing the flow of events from initialisation of the plugin from ImageJ through to drawing the clusters found to an image of the data set on screen.

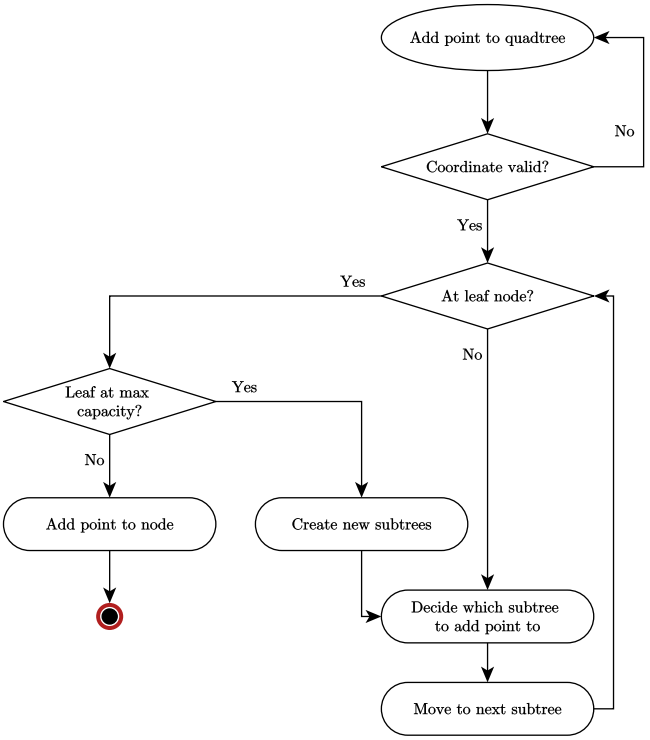


Figure 19: *Class diagram for the Cluster Analysis ImageJ plugin written for this project.*