UNIVERSITY OF
BIRMINGHAM

An ImageJ Plugin for Cluster Analysis of PALM and STORM Data

Josh Wainwright

Supervisor: Iain Styles

Date: September 2014

Title: An ImageJ Plugin for Cluster Analysis of PALM and STORM Data
Author: Josh Wainwright
Supervisor: Iain Styles
Date: September 2014

Declaration

The material contained within this dissertation has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this dissertation has been conducted by the author unless indicated otherwise.

Signed .....

Submitted in conformance with the requirements of the post graduate degree of MSc Computer Science at the School of Computer Science, University of Birmingham (2013/2014).

Abstract

This project introduces the concept of data clustering and its uses in medical and biological image analysis and examines some existing techniques for extracting clustering information from such data sets. A new technique based on quadtree style data structures is discussed and an implementation tested and evaluated based on this technique. This method is found to perform well even under difficult conditions, such as noisy data, where others struggle, though has limitations in data set size, resource usage and taking advantage of parallel processing.

Acknowledgements

My thanks to my supervisor, Iain Styles, for his advice, instruction and guidance through this project, to Steven Thomas for testing, evaluation and providing helpful feedback and to all those who proof read this report.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

1.1 Medical Imaging

In various scientific fields, it is an ability eagerly sought to be able to view and image objects smaller than the human eye can naturally observe. 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. In recent times, as our understanding has grown, we have desired to see beyond what ordinary microscopes are capable of and have invented machines that let us catch a glimpse of some of the smallest structures in biology—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 becomes the limiting factor.

1.2 Sub-Diffraction-Limit Imaging

Imaging objects becomes more difficult as they get smaller because of the wavelength of light (λ). Once two objects are separated by a distance of an order similar to that of the wavelength 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. The smaller wavelengths imply higher energies however, meaning that there is the danger of destroying the sample.

The minimum separation at which two objects can be resolved is given by Abbe's criterion, d,

$$d = \frac{\lambda}{2NA} \tag{1.1}$$

$$= \frac{\lambda}{2n\sin\theta}, \tag{1.2}$$

where NA is the numerical aperture of the microscope, the range of angles that the microscope's lens will let light through properly, n is the refractive index of the lens material and θ is the angle at which the light is incident on the lens. For typical values of $\lambda \approx 500\,\text{nm}$ (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.

1.2.1 Image Manipulation

Instead of trying to avoid the diffraction limit using shorter wavelengths of light or particles, other techniques employ different methods of actually capturing the image, and/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 Photo-Activation 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 image of a marker is used to locate the single point location of that marker, the "on" markers are turned off and different ones turned on 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.

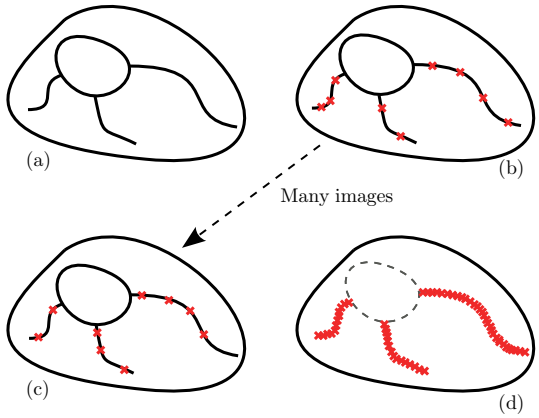


Figure 1.1: *STORM imaging. (a) The actual structure to be imaged is too small for regular microscopy. In stages (b) through (c), many images are taken, each with a different subset of the fluorescent markers activated. When the images are combined, (d), the points add up and reveal the nature of the object.*

The aim of this project is to provide a means by which the results of this method of imaging can be analysed more efficiently and with a greater degree of accuracy. This will take the form of a plugin for a piece of popular image processing software called ImageJ. The main deliverable is a plugin for extracting clusters of points from large data sets for ImageJ. This plugin is written in Java and makes use of the ImageJ Java API [NIH, 2014].

The sections included in this report are detailed below.

1 Introduction
A brief background to the project with information about the type of data to be managed.

2 Existing Cluster Analysis Algorithms
A discussion of the existing algorithms that are in use for different applications and the reasons they do not fulfill the requirements of this project.

3 Custom Algorithm
A detailed examination of the algorithms that are developed in this project and the ways in which they are superior to the existing algorithms available.

4 ImageJ Plugin
An explanation of the plugin that was developed for this project with information about how the constraints and variables of the algorithms are implemented in the plugin.

5 Evaluation
An evaluation of the success of the product at fulfilling the requirements given as well as of the processes and schedules that were followed during the project.

6 Appendix
A number of additional pieces of information and derivations to supplement the information in the body of this report.

1.3 Benchmarking

Throughout this project, a set of files is used to test the algorithms that are developed; their correctness and effectiveness, speed and resource use. Several of these files contain real data from PALM analysis of human thrombocyte (platelet) cells formatted in the same way as would be expected for data given to the plugin in general use, the others contain simulated or uniform data for testing. The files that will be used are detailed in Table 1.1.

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 1.1: *Files containing sample data are used for benchmarking throughout the project. A range of number of points per file, type of clustering expected for each file and the range of coordinates the points exist over is chosen to simulate different use cases.*

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 B and a summary of the hardware that was used when benchmarking the algorithms is shown in Appendix C.

Also note that in this report, 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.

1.4 Requirements

A number of requirements that should be achieved during this project are listed below. Each is given a value on the "MoSCoW" scale, meaning 'M' is a feature that **M**ust be included in order for the final product to fulfill the brief in Part 4, 'S' is a feature that **S**hould be included, 'C' is a feature that **C**ould be included to increase the usability and features but is not essential and 'W' is something that **W**ould be included if there is time and resources to do so.

1.4.1 Functional Requirements

The plugin will allow the user to:

1. Select data:
   - 1.1. Select a data file to process. (M)
   - 1.2. Select the appropriate column separator for the file. (C)
   - 1.3. Select which column from the data file the x- and y-coordinates appear in. (W)
   - 1.4. Adjust parameters relating to the process of analysing the data file. (S)
2. Create images:
   - 2.1. Create an image of the points from the file using native ImageJ functionality. (S)
   - 2.2. Create an image of the clusters found using native ImageJ functionality. (M)
3. Clustering algorithm:
   - 3.1. Perform a clustering algorithm on the data in the chosen file. (M)
   - 3.2. Choose from alternative clustering algorithms to perform on the data. (C)
4. Generate cluster information:
   - 4.1. Generate perimeter information for each of the clusters found. (S)
   - 4.2. Generate area information for each of the clusters found. (S)
   - 4.3. Display a results table showing a summary of information about each of the clusters found. (C)
   - 4.4. Limit which of the clusters are drawn to the image based on their size. (C)
   - 4.5. Limit which of the clusters are included in the results table based on the size of the cluster. (C)
5. Export data found by the algorithm:
   - 5.1. Export information by selecting appropriate cluster from the results table. (W)

5.2. Export data points contained in cluster by selecting appropriate cluster from the results table. (W)

1.4.2 Non-Functional Requirements

The plugin will provide functionality with the following:

1. Efficiency:
   1.1. Handle input data files of up to 100 000 data points in under 10 seconds. (S)
2. Usability:
   2.1. Be easy to use without instruction. (M)
   2.2. Have customisation ability for more advanced uses. (S)
3. Interoperability:
   3.1. Be platform independent, as long as ImageJ is present. (S)

Chapter 2

Existing Cluster Analysis Algorithms

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 often 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.

A number of clustering algorithms already exist. Many of these are designed for very specific purposes and so are not well suited for this application. Several of these are described below.

2.1 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 liquid solvent.

The rolling ball method can be used to analyse a cluster of points by imagining a solid ball or circle 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 from, 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 such that points classed as noise are not included but all interesting points are.

A simple approximation of this technique can be achieved by using the same dilating and eroding processes that are described later in Section 3.1.

2.2 Shared Nearest Neighbours

Shared nearest neighbour (SNN) clustering algorithms have been used to overcome some of the problems encountered with more traditional clustering methods when working in higher dimensions. In these cases, simple Euclidean geometry becomes unable to cope

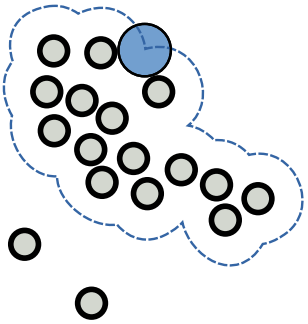


Figure 2.1: *The rolling ball method for cluster detection provides a way of identifying clusters, as inspired by molecular biology. A very simple implementation can be fast but is not particularly successful at finding clusters unless the data points are very dense and there is no noise.*

with the computation and so is replaced with other coordinate systems. For simple clustering in two dimensions, however, Euclidean geometry is sufficient and is far easier and more efficient to implement.

SNN based clustering algorithms analyse each of the points in a set, selecting those points that lie close to the current one based on certain thresholds. As the algorithm progresses, these thresholds are altered and the points considered so far are re-examined to maximise the 'similarity' of the points in any one cluster.

Implementations of SNN based algorithms, such as [Jarvis and Patrick, 1973] and [Ertoz et al., 2002] use similarity matrices, where the set of points is reduced to a graph with data points at nodes and edges between them representing the similarities between the points. This matrix is then simplified to leave a sparse graph, keeping only the edges with the highest similarity. From this matrix, points can be identified as noise and rejected, and the networks that show close similarities are designated as clusters.

This method is very good at being adaptable to complex situations such as arbitrary dimension or clusters with complicated shapes. However, the complexity in implementation and computation makes it unsuitable for small scale clustering such as the application discussed for this project. It also suffers from performance issues when the number of points to analyse becomes large as it has approximately $O(n^2)$ time complexity.

2.3 Gravity Based Clustering

A number of clustering algorithms [Zhong et al., 2010] have been developed that treat data points as physical objects under the influence of gravitational effects. These algorithms effectively model the points with these properties, allow them to move with some defined restrictions, and then examine the products that are produced.

The GRAVIclust algorithm [Indulska and Orlowska, 2002] works in two phases, first identifying n locations of interest based on the distances to neighbouring points, similar to the SNN algorithms above, and secondly applying an optimisation phase where each of these locations is further examined with the gravity effects of other points applied.

Unfortunately, these algorithms require certain information about the data set before clustering can be performed. First, a matrix of the distances between every pair of points must be calculated. This makes working with large data sets prohibitive. Secondly, the number of clusters to identify is required. This means that an algorithm to search for arbitrary numbers of clusters is not possible.

Chapter 3

Custom Algorithm

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 always a number of trade-offs that must be considered when choosing or designing a data structure for an algorithm: speed of access vs. speed of search, storage space used vs. time to insert a datum, etc.

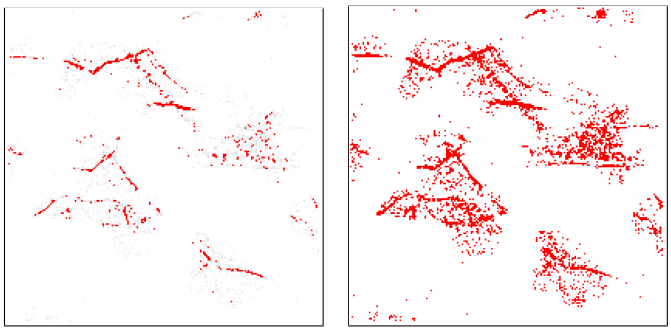
Some data structures may be naïvely chosen based on one of these, at the expense of the other. For example, consider storing the pixel information for a sparse image generated by a number of single pixel points—a linear array might be selected. This would give extremely good access and modification times, both constant, $O(1)$, but insertion and deletion are slow, $O(n)$. 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 appropriate 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.

3.1 Uniform Discrete Cell Method

The simplest method for analysing the distribution of points is to use a uniform, discrete grid of cells and place the points into these cells one at a time. Once all points have been added, the number of points that are contained in each cell can be treated as a grey scale brightness value with lighter pixels representing cells with more points and black cells having no points. This gives a simple pixel image, with brightness as a function of density of the points, in the `pnm` image format [Murray and VanRyper, 1996]. A thresholding filter can then be applied to remove the points that are isolated by removing the darker pixels 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 in 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 interesting data, or to increase the size of the clusters by including too much noise. These two effects can be seen clearly in Figure 3.1, where `palm-1.txt` is used with a cell size of 200. The range of the data is from 0 to 41 000 for both the x and the y axes, thus the images are $\left({}^{41000}/_{200} =\right)$ 205 by 205 pixels. This data took 495 ms to generate.



(a) **(b)**

Figure 3.1: *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. Red pixels are ones that would be kept by the thresholding process, black would be removed.*

There are a few 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 pixel image 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 the 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 second is black where there were some number of points and so is of interest.

3. Once a binary image has been generated, dilate and erode 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.

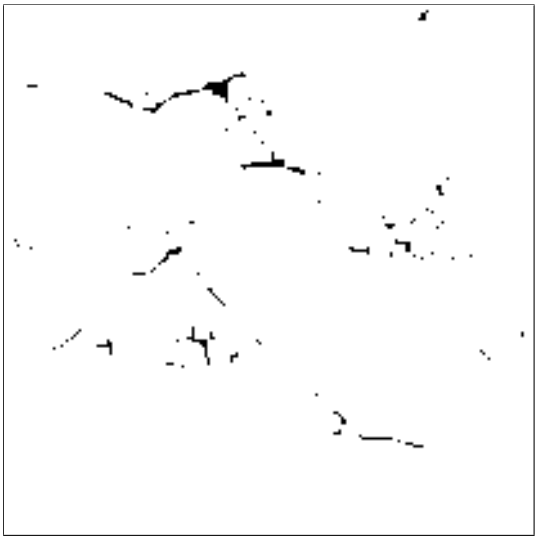


Figure 3.2: *Using dilating and eroding algorithms can help to emphasise the structure in the data and, at the same time, remove the isolated points representing noise.*

These steps lead to significantly better isolation of the interesting parts of the image, as can be seen in Figure 3.2, however, much of the detail of the structure is lost in this process.

3.2 Quadtrees

Since the simple grid method described in Section 3.1 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 often used in image processing since the four children of the root node can naturally represent the four quadrants of an 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 can be calculated.

3.2.1 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, element) A leaf in the tree which holds some information, value or quantity.

Root Node The node at the topmost position in the tree. It has no parents and is never a leaf.

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) A unique number or value that is assigned to a node by adding its position with respect to its siblings to the code of its parent.

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

3.2.2 Code Orderings

In order to identify a node uniquely in the tree, each node is given a code that is built up from its parent's code plus some value that identifies it among its 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.

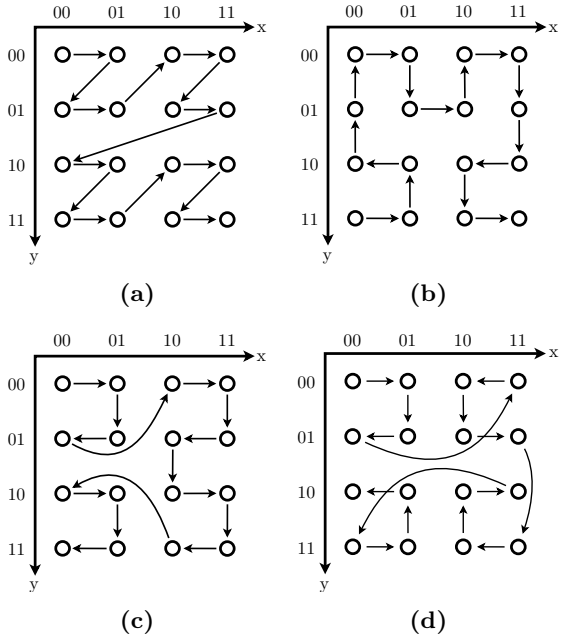


Figure 3.3: *A selection of possible tree traversal orderings. Each of these is, more formally, a space filling curve which is a direct, unique mapping from a 2D grid to a 1D array. (a) shows Z-order, (b) shows Hilbert order, (c) shows Gray code order and (d) shows the modified Gray code order developed for this project.*

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 changing to zero based indexing, 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 3.3a. This has several useful features.

- First, the numbers can be converted to base 2; 0 becomes 00, 1 becomes 01, 2 becomes 10, and 3 becomes 11.
  - This has advantages since binary is very efficient for computers to work with and allows certain tricks to be employed (see Morton Order Coordinates below).
- Also, this numbering system is easily extendible to any depth of tree that can be imagined.
  1. The root, as mentioned before, is given no value.
  2. Each of the children are numbered 00 through 11.
  3. 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.
  4. The children are always numbered in the same order for all children.

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 values to their decimal equivalent to give a standard decimal (x, y) Cartesian coordinate.

These steps result in a coordinate that represents the top left corner of the node. The coordinate is dependant on the length of the code, so to compare two coordinates using their codes, the codes must be of the same length.

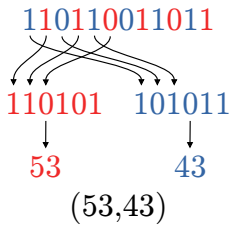


Figure 3.4: *Decomposition of a Z-order code using de-interleaving to give Cartesian coordinates. The bits of the code are alternately assigned to the y- and then x-coordinate which is then converted to decimal to give the final coordinate of the cell.*

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 3.3.1).
- The tree can be traversed, starting at the root, to find the nearest node to the expected code reference.

The first of these is used in the plugin since it allows the hash map structure to be used when searching which speeds up the process.

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. This property is known as being continuous.

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 simple recursive patterns that visit every point in a 2D space exactly once. These curves were first discovered in the early 1900's and described mathematically by David 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 is.

The path that the traversal of points follows becomes fairly complicated, Figure 3.3b, meaning that the calculation of neighbours becomes difficult as there is no simple way to traverse the path an arbitrary distance. Two of the neighbours are always simple to find by simply going a single step forward and backward, but the others are far more complex.

Gray Codes

The Gray Code, developed by Frank Gray in 1953 [Gray, 1953], was originally designed to reduce the error rate produced by mechanical electronics. The code is a variation on binary where each step, when incrementing, 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, which 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 3.3c. 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 for each of the sub-trees, as the Hilbert curve does, the leaf nodes group themselves in a very ordered fashion. When arranged as in Figures 3.3d and 3.5, each cell is arranged such that the codes for any two neighbouring nodes at the same depth differs by exactly one bit. This means that checking if any two nodes are neighbours becomes trivial.

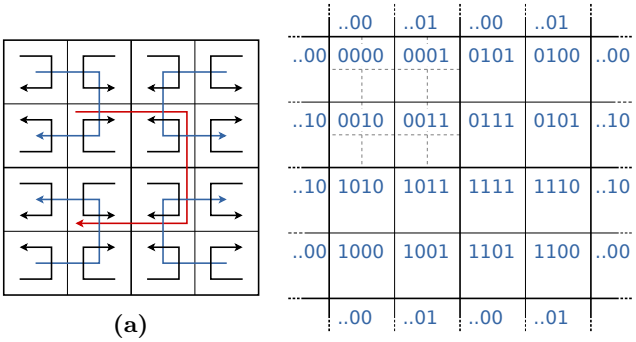


Figure 3.5: *The tree traversal that was developed for this project is a variation of the Gray code order discussed in Section 3.2.2. Instead of all sub-quadrants having the same orientation, each is reflected in either the x- or y-axis, (a). This has the advantage that neighbouring nodes have codes which differ by exactly one bit, (b).*

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 discounted. Figure 3.6 shows some of these.

- Figure 3.6a shows *row order* traversal of the grid. This is extremely simple to implement for a simple grid-like arrangement, but is very awkward and loses much of the information when using quadtrees.

- Figure 3.6b shows *row-prime order* traversal, also known as snake-order [Goodchild and Grandfield, 1983] traversal. This is a slight modification of row order which is continuous but, again, is not recursive so not suitable for use with quadtrees.

- Figure 3.6c is identical to the 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 3.6d shows an alteration to the modified Gray code order, named *pinwheel order*, where, instead of reflecting the quadrants, each is rotated about its centre 90° so that the whole image is rotationally symmetrical, 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.

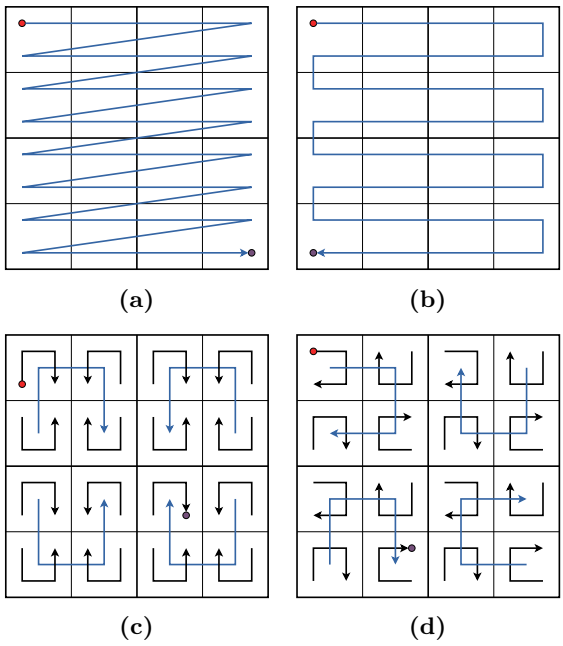


Figure 3.6: *There are a number of different space filling curves that can be used to map a two dimensional grid to a single dimensional array. To be considered for use in the quadtree ordering, the new ordering must provide some improvement of efficiency of some operations. Here are shown some orderings which were discounted.*

Node S	Neighbours	
101001	10100**0**	Guaranteed to be neighbours
	1010**1**1	
	101**1**01	Must choose two others
	10**0**001	
	1**1**1001	
	001001	

Table 3.1: *The possible neighbours for a node using modified Gray ordering. The bit that was flipped is highlighted for each neighbour.*

Chosen Ordering

Despite the promising features of the modified Gray ordering, the simple Morton ordering is used when constructing quadtrees in the ImageJ plugin. This is due to the issues encountered when developing a method of discovering neighbours of nodes in the modified Gray code system. It was found that, despite there being an assurance that two neighbours' codes differ by exactly one bit, this cannot provide a one to one mapping with the required neighbours of that node.

Consider, for example, a node, S, at a depth of three in the tree where the rook's case neighbours (see Section 3.3.3) are required. This node has a code consisting of 6 bits, two for each level. From this code, a total of six possible neighbour candidates can be identified by simply flipping each of the bits in the code. These neighbours are shown for an example node in Table 3.1 and their position relative to the original shown in Figure 3.7. The first two of these possible neighbours, the results of flipping the two least significant bits, are guaranteed always to be valid neighbours since they share the same parent as S. However, the two other neighbours which need to be selected from the remaining four possibilities, cannot be identified easily. These other possible neighbours exist at other locations in the quadtree, always either vertically above or below or horizontally to the left or right of S, but there is no simple test to determine if they are actually neighbours or not.

3.2.3 Constructing Orderings in Code

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 Morton order is shown in Listing 3.1. The constructor for a quadtree is `Quadtree(max_x, max_y, code)`, where `code` is the new portion of code to be added for that child.

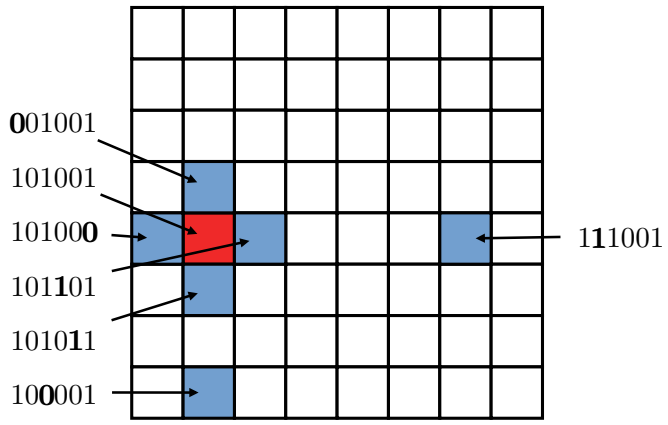


Figure 3.7: *Flipping the bits of the code to get possible neighbours gives too many possibilities to be able to select the correct neighbours. This problem is exaggerated as the node gets deeper in the tree since there are more bits in the code to change. The incorrect neighbours are always in line with the original and represent what would be neighbours if the tree was some number of levels shallower.*

```
// 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 3.1: *Code to generate children of the current quadtree while maintaining the correct ordering. Z- and Gray ordering.*

3.2.4 Hash Table Implementation

In order to speed up the subsequent operations applied to the quadtree structure, it can be 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 on page 8, 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 see if a node with that code is found, thus a time complexity of $O(\log n)$. 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, and so constant $O(1)$.

Since the quadtree 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 anywhere can be accessed 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 internal array. 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 3.2.

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

Table 3.2: *A hash map is used to associate a quadtree code with the required data for each node. The hash map value is an object containing several fields.*

3.3 Quadtree Traversal

Whether the quadtree is stored in memory as a recursive quadtree data structure, or as hash table, Section 3.2.4, 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.

3.3.1 Algorithm Description

To *propagate* is defined as the act of "spreading and promoting (an idea, theory etc.) widely" [Oxford English Dictionary, 2014]. This term shall refer to the process by which a single starting location, previously chosen, is expanded by examining neighbours to form a cluster. The algorithm is defined recursively with the main step being 'propagation' of nodes. The propagation algorithm is performed as follows:

1. Choose a starting location that has not yet been checked, l.
2. For this given starting location, the i neighbours surrounding it, n_0 to n_i, are checked for validity based on the conditions of the analysis (the way these neighbours are selected is discussed in Section 3.3.3).
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 to 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;
    - 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 3.8.

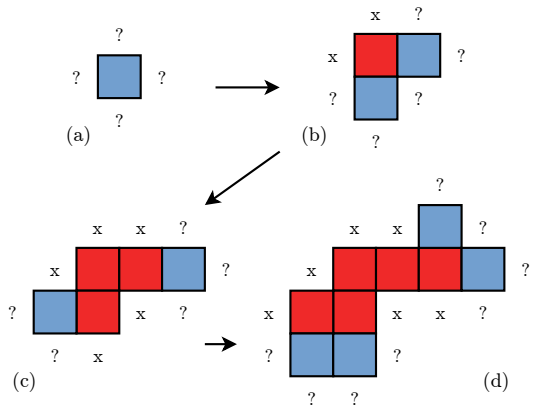


Figure 3.8: *For a starting location, (a), in an image, no information is known and so all its neighbours are checked. Some of these are found to be part of the cluster, others are rejected. The ones that are included are then, themselves, checked and so on. As the cluster grows, (b), (c) and (d), the number of checked nodes increases.*

Listing 3.2 shows pseudo code for the propagate algorithm which, starting at a given starting location, expands the cluster outwards while there still exist valid neighbours that should be included. The `getNeighbours()` method is described in Section 3.3.3. This is a recursively defined algorithm which calls itself for each of the neighbours of a node and for each of their neighbours, and so on. There is a condition that prevents the algorithm from being called again on nodes that have been checked already. This prevents a loop from forming and preventing the algorithm terminating.

```
1  function propagate() {
       node[] startLocations = getStartLocations()
3      for each node in startLocations
           propagate(node)
5  }

7  function propagate(node) {
       if (not node.inCluster())
9          node[] neighbours = getNeighbours(node)

11         for each n in neighbours
               if (n is valid) propagate(n)
13 }
```

Listing 3.2: *Code for the propagate algorithm which expands an initial starting location to a cluster.*

In reference to Listing 3.2 above:

Line 2 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 8 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 11 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.

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 boundary 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 3.9a. 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, following the dotted line.

To avoid this, a certain amount of leniency must be allowed when deciding what constitutes a neighbour. Given an appropriate value, this would allow both of the larger white cells in Figure 3.9a 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 density of points, all cells with a depth greater than the target depth shall be allowed, so the purple cells in Figure 3.9a 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

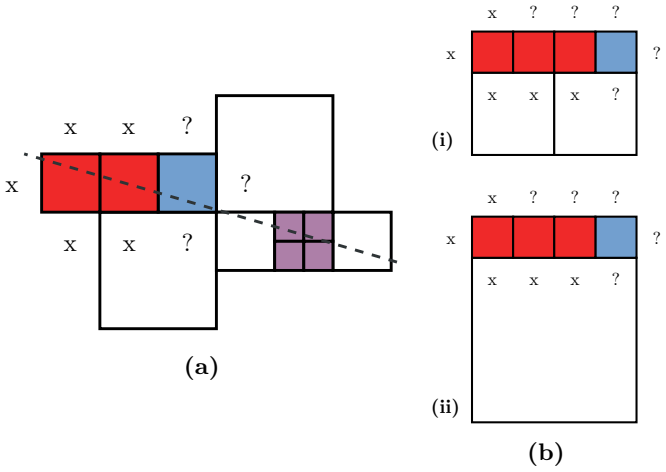


Figure 3.9: *Considerations regarding quadtree levels that will be accepted when propagating a node in a quadtree. (a) If the depth range that specifies how far up the tree to look for valid neighbours is too small then a cluster might be terminated too soon. (b)(i) a depth range of two and, (ii) a depth range of three.*

of one would mean that the white square in Figure 3.9b (i) would be included but not in (ii), whereas a depth range of two would include both and so on.

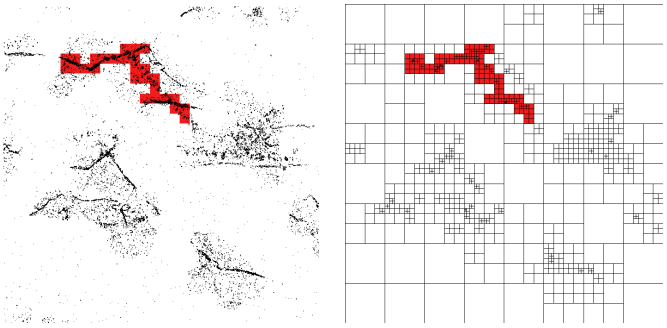
3.3.2 Clustering Start Locations

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 3.10 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 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.



(a) **(b)**

Figure 3.10: *Initial 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.*

- 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. This suffers from the same issues as method 1, since it is not helpful to require the user to make the selections manually.

The plugin developed uses a combination of the first and second approaches. Since the number of clusters cannot often be counted easily, the user cannot be expected to provide this value. However, it is also very difficult to determine when clusters that are being propagated are valid or not. The algorithm can be instructed to halt if the nodes that are being considered are too high up the tree, i.e., too many nodes have been included and the algorithm has run away, but this is generally not sufficient for all cases. Instead, a comparison with the depth range allowed (which is set by the user) with the difference between the first cluster found and the current cluster is made. If the difference is similar to the depth range, then the algorithm continues. After this process has finished, the clusters that have been found are examined and those that are too small with respect to the others are removed.

3.3.3 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 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, south 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.

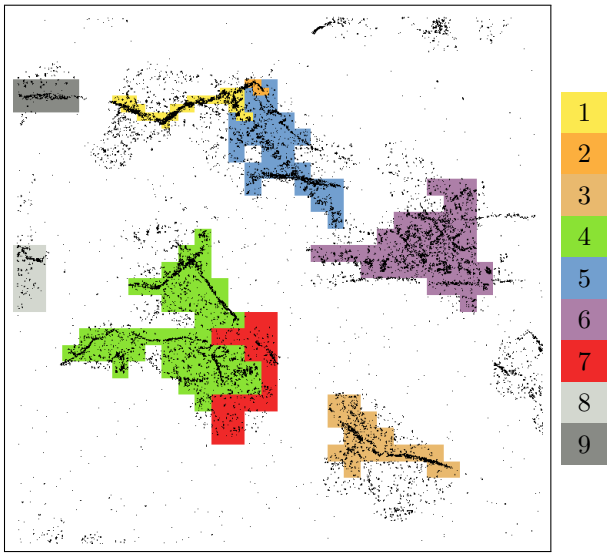
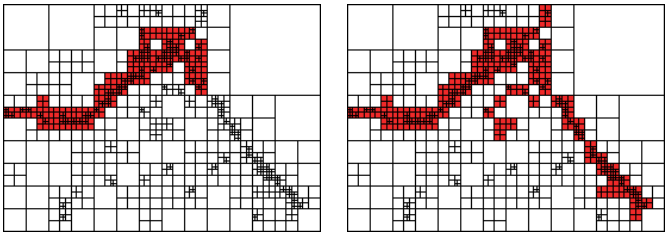


Figure 3.11: *Initial versions of the clustering algorithm included too many nodes, making the clusters too large. This image shows how each node cluster that is started is independent of the previous ones. The clusters were identified in the order they are listed on the right.*

However, if the choice of neighbours is limited to these four, some major structures are missed. Figure 3.12a 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 3.12b.



(a) **(b)**

Figure 3.12: *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.*

Image Kernel

The requirement to include eight neighbours for each node suggests that it might be of value to allow users 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. The sum of the nine elements is one, meaning the overall brightness of the image is not changed. When applied, this would mean that the current pixel, p, is given a value which is the sum of each of matrix elements, $m_i = {}^1/_9$, multiplied by the pixel value beneath it, v_i, Equation 3.1.

$$
\begin{aligned}
&s := \texttt{kernel.size} \\
&\texttt{for each pixel p in image}: \\
&\qquad p := \sum_{i=0}^{s} m_i * v_i
\end{aligned}
\tag{3.1}
$$

A similar technique is used here 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 either 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 node that lies under an element with a value of 1 is included as a neighbour and nodes under 0 elements are ignored. The value of the central element in the kernel does not matter, but the convention is to set this to 1.

Thus, the simplest kernel, though of least use, is the identity kernel which has an element with value '1' in the centre and all other elements '0', Figure 3.13a. This would result in no neighbours being selected, and so no clusters located. The rook's case neighbours are now represented with the kernel in Figure 3.13b and the case with all 8 neighbours in Figure 3.13c.

0	0	0
0	1	0
0	0	0

(a)

0	1	0
1	1	1
0	1	0

(b)

1	1	1
1	1	1
1	1	1

(c)

Figure 3.13: *Kernels for (a) the identity matrix,(b) rook's case neighbours and (b) all 8 neighbours. The central cell of the kernel can be anything as it is never included as a neighbour.*

This functionality provides the ability to target certain types of clusters depending on the spatial orientation of the cluster; vertical or horizontal etc. Figure 3.14 shows a number of possible kernel shapes that would allow different types of clusters to be targeted. Each of these show the central cell with the cells that would be represented by a 1; cells represented by 0's are not displayed.

Searching Up and Down the Tree

Once a node location has been identified as a valid neighbour, there is still no guarantee that there exists a node there. There are three possible cases concerning the location that has been identified. Each of these situations must be handled differently in order to ensure that all of the correct nodes are selected as neighbours.

1. The location represents a leaf node, in which case the process is complete and a neighbour has been identified, Figure 3.15a.

   In this case, no further action must be taken.

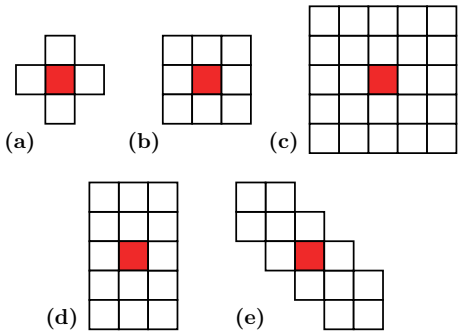


Figure 3.14: *Different kernel shapes can be used to find different sorts of clusters. (a) The basic rook's case neighbours and (b) the full number of closest neighbours are the simplest general cases, used for most clusters. (c) The area to search can be extended by including more neighbours—this could 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.*

2. The location is to be a non-existent node, i.e., there may be a leaf node at a position further up the tree than the location meaning there is nothing at this position in the tree, Figure 3.15b.

   In this case, either the code that has been obtained from the kernel analysis must be shortened until it represents a node that exists in the tree, or the tree must be traversed according to the quadtree code until a leaf node is reached. The located node must then be compared against the node to which this is a neighbour to check that the two are within the depth range specified (Section 4.1.2).

3. The location is a node in the quadtree which is not a leaf node, i.e., it has four children, each of which may be trees or leaves, Figure 3.15c.

   For this third case, the depth range is not considered since all nodes deeper than the starting node are assumed to be a part of the same cluster. Instead, all leaf nodes below the node represented by the code are included as valid neighbours, and hence are, themselves, propagated. To get these children, the sub-tree below the current node is traversed in pre- or post-order and every leaf node that is encountered is added as a neighbour.

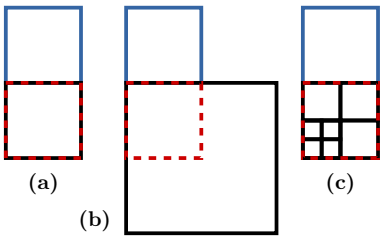


Figure 3.15: *The possible results of neighbour selection. The blue square is the node that is being propagated and the red square represents the location of the node given by the neighbour code, black squares are real nodes in the tree. (a) shows the ideal case where the neighbour is a node, (b) shows where the code could be further down in the tree than exists and (c) shows the case where a real node is selected, but is not a leaf.*

Chapter 4

ImageJ Plugin

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

Since they offer significantly better integration, meaning speed and efficiency improvements, a plugin is chosen to integrate this project into ImageJ. A plugin was chosen over macros, since these often suffer performance loss when more than a few steps are involved; and chosen over scripts since these do not offer such tight integration with the rest of the program.

The plugin shall allow a user to *a*) load a data set, as gathered from STORM, PALM or similar imaging techniques discussed in Section 1.2; *b*) choose the format of the data (the columns that are of interest etc.); *c*) analyse the data for clusters; and *d*) receive detailed information regarding the clusters that were found.

4.1 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 as follows:

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 check boxes 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. Use the additional information presented in the results table.

4.1.1 Column Picker

The first step is for the user to select a data file. This is performed with the Java built-in file chooser. The ability to specify some formatting features of the data file used are available. This makes 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 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 graphical user interface (GUI).

4.1.2 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. Each heading represents the label given to the corresponding slider in the GUI.

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 is added in to the appropriate new leaf, this is known as *re-leafing*.

An extremely large value for the **density**, 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.

`default`	20
`minimum`	1
`maximum`	100

Depth Range This controls how far up the quadtree will be searched when looking for neighbours as described in Section 3.3.3. All neighbours that are deeper in the tree than 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**.

`default`	3
`minimum`	0
`maximum`	25

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 **max depth**. If the depth of a node reaches the **max depth**, then it is not re-leafed even if the number of points in the node exceeds the **density**. This allows a high value to be set for the **density** 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, abnormally 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, as the tree is unlikely to get that deep.

Since the number of nodes in the quadtree increases exponentially with the depth, a large value for the **max 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 almost 5 billion leaf nodes, so even extremely large files can be represented to a high degree of accuracy.

`default` 10
`minimum` 5
`maximum` 25

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

Cluster Size Once the clustering algorithm has completed, this 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 clusters 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.

`default` 0
`minimum` 5×10^{-4}
`maximum` 2×10^{-2}

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 around 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 division, 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 0.03 is used. This gives an uncompressed image size of around 11 MB for a data set with maximum x- and y-coordinates of 50 000.

`default` 0.03
`minimum` 0
`maximum` 0.1

The following are purely display settings. Each is a boolean value controlled with a checkbox component. Changing any of these settings simply requires the data to be redrawn, no further computation is performed.

Lines The quadtree can be displayed on screen to check the settings that have been used, above, are most appropriate. This setting toggles the display of the quadtree lines so that the nodes can be seen.

`default` true

Points Similarly to **Lines**, **Points** simply controls whether the data points from the data file are drawn as single pixels or not at all.

If this and **Lines** are both set to false, then only the clusters themselves, as identified by the nodes that are part of the cluster, are drawn.

`default` true

Colourize By default, the clusters are drawn behind the lines of the quadtree and behind the points from the data file. The colour of each cluster is different from the previous cluster. There are 20 colours defined, meaning that when the number of clusters exceeds this number, the colours will repeat. The colourisation is simply to allow the clusters to be visually distinguished from each other. This setting stops the clusters from being colourized, instead using the same grey for all.

`default` true

4.1.3 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 plugin 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 most cases, many thousands of separate Java objects (data points represented by zero length lines, quadtree cells by boxes, etc.) were maintained in memory. This made it slow to draw with any move or resize of the window.

The code used to generate this view of the data was modified to make use of the 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*. 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.

4.1.4 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 its 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. This is discussed further in Section 4.2.1.

4.1.5 Macro Support

As mentioned on page 25, ImageJ allows extension via macros, as well as plugins and scripts. The ImageJ documentation describes macros as follows:

> A macro is a simple program that automates a series of ImageJ commands.
>
> [...]
>
> Use the macro language's built-in `run(command, options)` function to run any of the 400+ commands in the ImageJ menu bar. The first argument (`command`) is any ImageJ menu command (e.g., "Measure" and "Close All") and the second argument (`options`), which is optional, is a string containing parameter values.
>
> [NIH, 2014]

This means that the process of image processing can be simplified by writing or recording macros which perform repetitive tasks automatically.

In order to make use of these features of ImageJ, support for macros must be written into plugins. The plugin written for this project includes basic support for macros through the `run(command, options)` command described above. The command is "Cluster Analysis" and the supported options are described below.

Option	Value	Description
`sep`	**String**	Specifies the separator to use when reading the data file. If this option is not given, the separator defaults to a single tab character.
`filepath`	**String**	The data file to read from.
`quadtree`	**0 or 1**	If 1, the quadtree generated from the given file is drawn immediately. If 0, or if this option is not given, the main dialogue window is opened. If 0, or not given, but a filepath is given, the file is read and initial analysis performed once the window is open, but the image is not displayed.
`grid`	**0 or 1**	Same as for `quadtree` above but for the simple grid method.

Each of these options must be given in the form `option=value` where `value` has the type or limitations described above and each option/value pair must be separated with a comma and an optional space. All options except `filepath` are case-insensitive.

4.2 Cluster Analysis

Once the data has been processed and a number of clusters have been identified, the clusters are displayed on screen in an image to allow the user to verify them and perform further analysis. However, since the data that was used to find the clusters is still held in memory at this point, it is useful to take advantage of this and perform some immediate 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 identified as being part of a cluster. This will mean that the cluster is likely to be slightly larger than the actual data points that it is comprised of, but is computationally simple to achieve, so fast, and can take into account any holes in the cluster (see below).
- The second way is to, once the cluster has been located, discard 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 guaranteed to provide information that more closely represents the original data, though is more computationally intensive.

4.2.1 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.

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 4.2,

$$a = \frac{1}{4^d} \tag{4.1}$$

$$a_i = 4^{-l_i/2}, \tag{4.2}$$

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:

$$A = \sum_{i=0}^{n} a_i. \tag{4.3}$$

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 4.5,

$$p = \frac{1}{2^d} \tag{4.4}$$

$$p_i = 2^{-l_i/2}, \tag{4.5}$$

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 (0, 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 4.6,

$$P = \sum_{i=0}^{n} s * p_i, \tag{4.6}$$

where $s \in \{0..4\}$. This is demonstrated in Figure 4.1 where the perimeter is simple to calculate in case (a) as the nodes are all the same, but the size of each node must be taken into account in case (b).

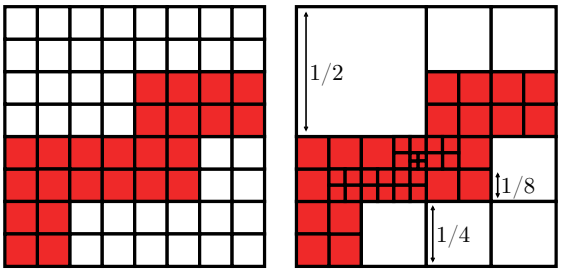


Figure 4.1: *When calculating the perimeter of a cluster using the nodes that contribute, the size of each node must be taken into account. In case (a), the process is simple since all nodes are the same size and a fractional value of 3.5 is calculated. For case (b), the steps are 25 lengths of size $^1/_8$ and 6 lengths of size $^1/_{16}$ which gives the same result, 3.5.*

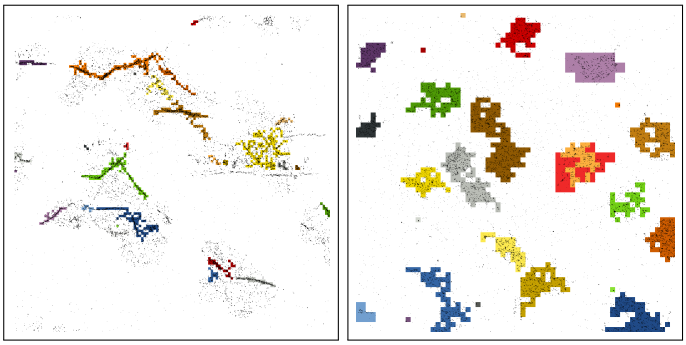
Within the cluster, *holes* occur where a node is, or number of nodes are, surrounded on all sides by the same cluster. An example can be seen in Figure 4.2. Unfortunately, these are included in the calculation of the perimeter and so, where holes exist in a cluster, the actual perimeter is slightly smaller than that calculated.

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 available 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 4.7, is defined such that it is a unit-less ratio of area and perimeter, such that a circle has roundness 1. The derivation of Equation 4.7 is found in Appendix D.

$$R = \sqrt{\frac{4\pi A}{p^2}} \tag{4.7}$$

The clusters that are found for data sets `palm-1.txt` and `palm-2.txt` are shown in Figure 4.2. The roundness values for the clusters for these data are very different, Figure 4.2a has an average $R = 0.350$ whereas Figure 4.2b has an average $R = 0.446$.



(a) (b)

Figure 4.2: *The general shape of the clusters found can be described using the roundness measure. A value closer to 0 means longer and thinner clusters, (a), whereas values closer to 1 mean clusters that are more circular, (b).*

4.2.2 Point Analysis

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

Cluster Perimeter

Convex Hull Algorithm

There are a number of algorithms that attempt to find the bounding polygon of a set of points. Ignoring any holes in the cluster, the perimeter of this bounding polygon is equivalent to the perimeter of the cluster.

The simplest method of finding this bounding polygon is using the *convex hull* method [Barber et al., 1996]. In principle, this involves starting with a circle far larger than the points it encloses, which is reduced in size, without letting any point leave the shape, until only straight lines exist between external points. The final shape can be pictured by imagining each of the points as a nail in a piece of wood. The convex hull algorithm then calculates the shape that would be produced if an elastic band were stretched around these nails.

The major disadvantage of this algorithm is that, as the name suggest, the final shape consists only of convex angles, meaning that the actual shape of the perimeter of the points may be lost, as shown in Figure 4.3.

Concave Hull Algorithm

An alternative to the convex hull algorithm is the significantly more complex *concave* hull algorithm [Moreira and Santos, 2007]. Instead of simply finding the outermost shape that will enclose all of the points, the concave hull algorithm finds that shape that fits the outer boundary of the points more closely. The steps involved are listed below.

1. Find the most southerly point to be the first point in the hull.
2. The algorithm looks at the distance to each of the nearest n points, where n is specified beforehand.
3. From these possibilities, it chooses the closest to be added to the "hull".

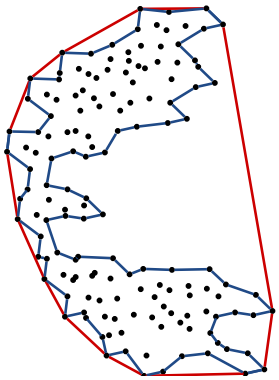


Figure 4.3: *Convex versus concave hull algorithms. The red line shows the convex hull found for this set of points, which misses much of the detail and will report the area as being significantly above its actual value. The blue line is the concave hull.*

4. The new line segment that would be added by joining the existing hull to the new point is examined to ensure that it does not cross the existing hull at all.
5. If it does cross, this point is removed as a possibility of being a part of the hull and the next nearest point is checked.
6. When the initial starting point is reached again, the algorithm will have generated a closed loop joining a number of points together.
7. This closed loop is checked to ensure that all points lie either on the hull or are enclosed within it.
8. If there are any points that fall outside the hull, the results are abandoned and the algorithm is re-run with the number of neighbours to check, n, incremented by one.
9. When a hull is found that encloses all points, the algorithm terminates.

Most implementations of the concave hull algorithm require an argument specifying the number of neighbours to examine when choosing the next segment. A higher number of neighbouring points that are checked, the less the final hull will include small incursions into the shape and so the hull will be smoother. The blue line in Figure 4.3 shows the concave hull identified for a cluster with a neighbouring number of 4.

Delaunay Triangulation

Another, more complex to implement, alternative is Delaunay Triangulation [Delaunay, 1934]. This is a method first proposed in 1934 by Boris Delaunay for use in geographical applications.

> The Delaunay triangulation is a triangulation which is equivalent to the nerve of the cells in a Voronoi diagram, i.e., that triangulation of the convex hull of the points in the diagram in which every circumcircle of a triangle is an empty circle.
>
> [Okabe et al., 2009]

This means that the result is a densely connected graph where every point is joined to exactly two other points such that there are no points within the circle joining any three connected points. This has the side effect of calculating the convex hull and makes calculating the area of the cluster simple—the areas of the triangles are summed.

This method is not used to calculate the perimeter or the area since the implementation is generally more complex than other methods and provides more information than is useful, and hence takes longer to calculate. Finally the area of the cluster that is identified is convex and so shows the same issues as the convex hull algorithm above.

Cluster Area

If using a hull-based algorithm to find the perimeter of the cluster, once the perimeter hull has been identified, it is simple to calculate the area enclosed by that cluster. Since the hull consists only of straight lines connecting single points, the area can be calculated by splitting the polygon into regular triangles, and finding each of their areas individually. The area of the whole cluster is then given by the sum of these areas. An algorithm for calculating the area of the irregular polygon produced with this method is shown in Listing 4.1.

```
function polygonArea(ps, numPoints) {

   area = 0;
   j = numPoints - 1;

   /* Sum the area contained between each pair of
    * points and the y-axis on the right hand side
    * of the shape and subtract the area between
    * pairs of points and the y-axis on the left
    * hand side.
    */
   for (i=0; i < numpoints; i++)
      area += (ps[j].getX() + ps[i].getX()) *
              (ps[j].getY() - ps[i].getY());
      j = i;

   return area/2;
}
```

Listing 4.1: *Code to find the area of an irregular polygon. Adapted from [Finley, 2006]*

4.2.3 Dilate/Erode Alternative

As an alternative to the other methods described above, a fast technique for finding the perimeter and area of the points that takes advantage of the fact that the points are represented in an image in ImageJ was developed. With the points that represent the cluster to find the details for, and using the built-in commands in ImageJ, several successive dilate filters are applied to the points followed by the same number of erode filters.

Dilate Filter A dilate filter, also known as a grow or expand filter, is a technique in mathematical morphology to extend the boundaries of a shape by adding pixels to the outer edge. It can be performed only on a binary image since the pixels that are added must all have the same value. The filter adds new white foreground pixels to the boundary of existing foreground regions with the effect that shapes get larger and holes in those shapes are reduced.

Erode Filter An erode filter, also called a shrink or reduce filter, is a very similar process to dilation above, but in reverse. Instead of adding to foreground regions, pixels are removed. Another way to consider the erode filter is as an identical filter to dilating but performed on the black background pixels.

In order to get the useful information using this method, the dilate filter is applied N times. This has the effect of filling in holes that exist between points and combining the cluster into a single conglomerate that can be manipulated as a one object, rather than a number of distinct points. Next an erode filter is applied to the shape $N + 1$ times. This reduces the shape back to the original size of the points. The reason for the extra erode is to remove outlying points that were not joined in to the main shape and so would skew the results. Finally, an additional dilation is performed to reverse the effect of the extra erode.

Figure 4.4 shows these steps on a number of points that have been identified, (a), which is dilated twice, (b) and (c), before being eroded again, (d) to (f) and then dilated again, (g). It can be seen from the image that the shape in (g) is a good approximation of the outline of the points in (a) with the single outlying point removed.

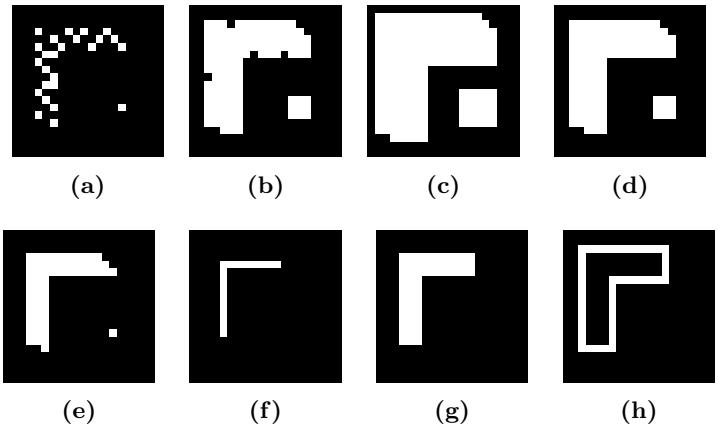


Figure 4.4: *An alternative perimeter algorithm implemented using dilate and erode filters which are provided natively in ImageJ. Image (a) shows an example of a few points that are to be analysed. Images (b) to (g) show the steps of the algorithm to find the area and (h) is the final step to find the perimeter of the resulting shape.*

To find the area now becomes a simple case of counting the number of white pixels that are left in the image and then scaling this according to the scale factor used to draw the image (page 27). A final step is performed to get the perimeter of the shape. This time an outline filter is applied. This has the effect of creating a one pixel wide outline of the shape. The perimeter is then the length of this outline which, again, can be found by counting the pixels it consists of and scaling for the size of the pixels.

Once the cluster information has been calculated, the list of clusters is examined and any that fall outside the limits set by the user ('Cluster Size', page 27) are removed and are not displayed either in the table or in the image. The final results are then displayed in a results table. This results table is sorted by the number of points within each of the clusters. At this point, the clustering algorithm is completed and the user is free to export the data using the built-in functionality of the ImageJ results table or re-run the plugin with altered settings.

Chapter 5

Evaluation

The aim of this project was to develop a plugin for the ImageJ software to perform cluster analysis of sub-diffraction limit data. This was achieved with a functioning piece of software in the form of a plugin that is able to perform well with a number of different testing situations. An evaluation of the extent to which this plugin satisfies the requirements and the development process are described below.

5.1 Testing

Since the absolute definition of the clusters that the plugin looks for are somewhat subjective, testing the algorithm for correctness is difficult. The individual functions and smaller components of the algorithms are tested using the unit testing framework JUnit [Tahchiev et al., 2010], and the overall effectiveness of the clustering algorithms is validated with generated data files and acceptance testing.

5.1.1 Generated Test Files

To try to test the algorithm and get some more quantitative results, it was performed on a number of simulated test files. These data sets, shown in Figure 5.1 were produced by Karypis et al. [1999] for testing their Chameleon clustering algorithm. The algorithm was found to identify 4 clusters in image (a) and a single cluster image (b), but performed poorly in the final two.

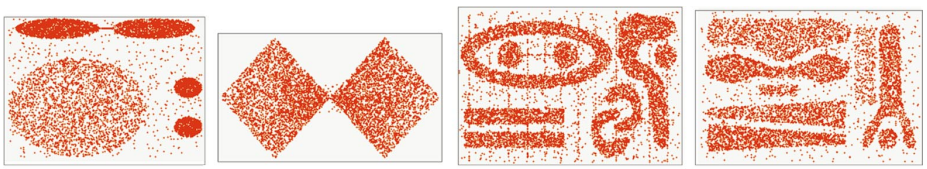


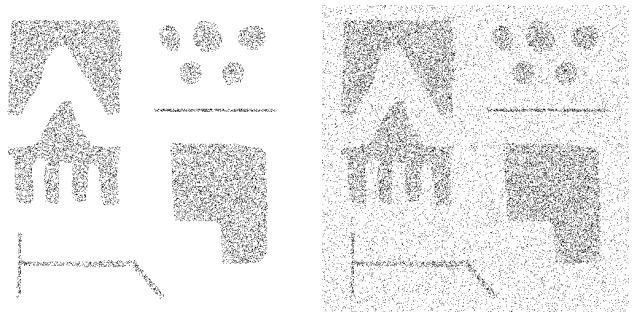
Figure 5.1: *Manually generated data sets for testing. The images (a), (b), (c) and (d) above are used to test the algorithm for this project and for a number of other studies in clustering.*

These particular datasets have been used to test a number of clustering algorithms and so they provide a helpful benchmark with which these algorithms can be compared. It is unfortunate that this algorithm fails to cope with the complex structures that are present in the second two images, but shows that it is more specifically tailored than the

more general algorithms that use these as tests. The clusters that the algorithm failed to identify lie too close together meaning that the propagation step is able to cross from one cluster to another.

The same effect is seen in images (a) and (b) where there is a link between what would, otherwise, be separate clusters. This means the algorithm reports these as a single object.

Another testing data set was created, similar to the final two images above, but with more spacing to see how the algorithm would cope. An image of this set is shown in Figure 5.2.



(a) **(b)**

Figure 5.2: *Custom generated data set for testing clustering algorithm. This data set is similar to that shown above, but has more space between clusters. The number of points originally in this data set is appropriately* 14 000*, (a), and random noise is added up to a ratio of* 1.4 : 1*, (b).*

The results were better with this data set since the propagation step of the algorithm was halted by the clear space between the clusters. When there is no noise at all in the data, all of the clusters are located. The amount of noise can be increased by simply adding random data points which are distributed evenly across the image space. This is done progressively, adding appropriately 2000 points at a time. The clustering algorithm stops finding the clusters properly when roughly 10 000 random points have been added, Figure 5.2b. This represents when the signal to noise ratio is noise 1.4 : 1.

5.1.2 Volunteer Validation

The first stage of acceptance testing took the form of a sample set of data that was plotted onto an image using the discrete grid method. This data was then presented to a number of volunteers who were asked to count the number of clusters they could identify and then highlight the clusters that they were able to observe in order of precedence, most defined through to least defined.

Once the volunteer had identified the clusters, the algorithm was performed on the same set of data and the results compared to the predicted results. Finally, when the algorithm identified clusters that the volunteers had not, they were asked to identify which of these they did not consider to be valid clusters and which were valid clusters that they had missed. The number of false positive clusters identified by the quadtree algorithm was found, by this method, to be low.

When using the settings described in Section 4.1.2, 100% of the clusters identified by the volunteers were able to be located by the algorithm. Of the clusters that the algorithm found but the volunteers did not, around 60% were considered to be valid by the volunteers. These results suggest that the algorithm is performing well at being able to identify

clusters, but requires some intervention to get the best results (see Section 5.4) and that, in general, the algorithm does not produce that many false positive results.

5.1.3 Researcher Validation

The final stage of acceptance testing was performed to confirm the type of clusters identified conformed with the actual objects that are being imaged. This was performed by a researcher working in the medical imaging field who produced the sample data used in this project.

The feedback from the researcher was very positive with relation to the performance of the clustering algorithm. In particular the speed of the plugin to produce results was commented as being much better than existing utilities that have been written in the MatLab suit [MATLAB, 2010]. This means that a user can make changes to the settings a number of times to get the results desired without spending long periods of time waiting for the process to complete.

The clusters that were identified, when given some sample data from current research, were also confirmed to be in line with the biological theory, something that only someone with good knowledge in the area could confirm. With some explanation of the effects of the different settings, the researcher was able to produce desirable results quickly.

There were a number of suggestions for improvements that were identified. Many of these would not require large changes to the code base. These are summarised below.

Provide a better mapping between the clusters highlighted in the image and the results table which summarises the clusters.

The image is colored with each cluster that is found having a different colour which then corresponds to a line in the results table. This mapping is not clear since the results table does not allow any contents other than strings of text. A better solution would be to label each of the clusters in the image with a numerical value which can then be identified in the table by the cluster id.

This would require a single extra step during the drawing phase to add a text label with the number of the cluster.

Allow choosing the scale of the image to correspond to the real size of the object which is imaged.

A setting in the main UI that allows the user to input a scale would mean that the results that are presented in the table would have more immediate value. This value would be the dimensions of the image which would be able to act as a multiplication factor for the other results.

Use different clustering algorithms from the same plugin.

As discussed in more detail in Section 5.4, the ability to choose from a number of different algorithms to perform the clustering would make the plugin more general purpose, in the event that one of them did not cope well with the particular data being examined. Adding other algorithms would, currently, require some refactoring of the code to separate some of the functionality from the algorithm. This would be required so that adding other algorithms was simple.

5.2 Requirements Satisfaction

With the plugin written, the requirements outlined in Section 1.4 are examined to check which have been fulfilled and which were not able to be completed. Each of the requirements below refer to a requirement from Section 1.4. In each case, the requirement is given with a note to say whether it was completed and if so, what component is the fulfillment of it and if not, why not.

5.2.1 Functional Requirements

1.1. Select a data file to process.
Completed with file chooser dialogue.

1.2. Select the appropriate column separator for the file.
Completed with column chooser integrated into the file chooser.

1.3. Select which coloumn the x- and y-coordinates appear in.
Completed with column chooser integrated into the file chooser.

1.4. Adjust parameters relating to the process of analysing the data file.
Completed with sliders and checkboxes for options in main user interface.

2.1. Create an image of the points from the selected file using naitve ImageJ functionality.
Completed with simple grid implementation which simply adds the points to the grid and draws the image to the screen, no analysis of the clusters is performed.

2.2. Create an image of the clusters found using native ImageJ functionality.
Completed with an ImageJ image of clusters found, colour-coded based on order in which the clusters were found.

3.1. Perform a clustering algorithm on the data in the chosen file.
Completed with the quadtree based clustering algorithm.

3.2. Choose from alternative clustering algorithms to perform on the data.
Not completed The simple grid method was implemented but found to perform too badly to be used as an alternative clustering algorithm. Limitations in time prevented the implementation of existing clustering algorithms.

4.1. Generate perimeter information for each of the clusters found.
Completed with dilate/erode or concave hull based perimeter methods.

4.2. Generate area information for each of the clusters found.
Completed with dilate/erode or concave hull based area methods.

4.3. Display a results table showing summary of information about each of the clusters found.
Completed with ImageJ results table shown after cluster analysis is completed.

4.4. Limit the clusters drawn to the image based on the size of the cluster.
Completed with limit chosen by user from main user interface.

4.5. Limit the clusters included in the results table based on the size of the cluster.
Completed with limit chosen by user from main user interface.

5.1. Export cluster information by selecting appropriate cluster from the results table.
Not completed since integration with the results table (which is defined and maintained by ImageJ) does not allow selecting a line and passing that line to the

plugin. To implement this functionality would require a custom implementation of the results table.

5.2. Export data points contained in cluster by selecting appropriate cluster from the results table.
Not completed since ImageJ does not allow making clickable selections from images as this would interfere with manipulation of the image itself.

5.2.2 Non-Functional Requirements

1.1. Handle input data files with up to 100 000 data points in under 10 seconds.
Completed, though the time to process files is very much dependant on the hardware used, the algorithms developed are quite efficient.

2.1. Be easy to use without instruction.
Completed with a small number of test users who were allowed to use the plugin and comment on its usability.

2.2. Have customisation ability for more advanced uses.
Not completed, a number of options are provided to customise the running of the algorithm but these are required to work and so there is no other customisation.

3.1. Be platform independent, as long as ImageJ is present.
Completed with ImageJ and Java being platform independent so the plugin should work wherever ImageJ and Java are installed.

5.3 Development Evaluation

The development process of this project followed an agile methodology. There was a distinction in steps between the initial design of the data structures and algorithms and the development of the program and the plugin for ImageJ. There were stages of development of the algorithms and the plugin which were revisited a number of times during the development. At each stage, the plugin was tested and any bugs fixed before new features or the next algorithms were added.

5.3.1 Development Process

The estimated schedule that was proposed prior to the start of development are listed in Table 5.1. This was followed reasonably closely throughout the project, though some deviations were encountered when the development of algorithms took longer than was expected. In particular, issues when designing the modified Gray code quadtree ordering slowed development significantly. See Section 3.2.2 for the reasons. To account for these changes, other stages were allocated slightly less time than is recorded here.

5.4 Possible Improvements

Though the primary aims of this project were fulfilled, there are a few areas that could be improved, or features that could be added that would improve the plugin and make it easier to use. These features were not developed during the course of this project because the core functionality was more important and time constraints prevented them from being added after the existing features were complete.

Stage	Tasks	Date
1	Research existing methods of cluster analysis and identify short comings.	20th June 2014
2	Build implementation of Uniform Discrete Cell method.	27th June 2014
3	Build implementation of Quadtree method.	11th July 2014
4	Test and compare previous algorithms. Perform timing and resource usage analysis.	18th July 2014
5	Build first iteration ImageJ plugin using chosen method.	25th July 2014
6	Using chosen method, implement cluster analysis algorithms.	1th August 2014
7	Add cluster analysis to ImageJ plugin.	8th August 2014
8	Performance and ease of use testing of plugin.	15th August 2014
9	Write-up of background research and current implementations investigation	18th July 2014
10	Write-up of data structure algorithms.	1st August 2014
11	Write-up of cluster analysis algorithms.	22nd August 2014
12	Final write-up of processes, improvements and end results of project.	29th August 2014

Table 5.1: *An initially proposed schedule for the development process. The dates were not followed exactly, but the tasks were performed in this order.*

Alternative algorithms

The algorithm that was developed for this project performs well under a range of different conditions. This was tested through the use of exemplar data sets and data sets constructed for use in testing to simulate different conditions. However, there may be some situations that were not tested for which it might not perform well. To avoid this, a number of alternative algorithms could be included so that, if the default algorithm fails, the user could switch to a different approach which might handle the data better.

One or more of the existing algorithms discussed in Part 2 could be implemented as alternatives which could then be selected from the main user interface if the quadtree based method failed.

Better starting location selection

The logic in the current implementation that selects new starting locations where clusters are propagated from is reasonably simple and so can fail to determine the best next location. This would be improved by a method that considered more than simply the quadtree structure, such as the placement of existing clusters and the depth of node surrounding the possible starting location.

Better algorithm halting

Similarly to above, the rules that stops the algorithm looking for more starting locations once a cluster has finished being propagated are not always sufficient. For example, in a data set with few clusters, from zero to six, the algorithm will continue to look for more and so might identify the background noise as a cluster that spans most of the image.

Calculation of optimum settings

Since the particular settings that are required vary a lot between different data sets, the user must often alter the settings before any useful results are found. To automate this process would require some significant changes to the algorithm so that it did not take a prohibitive amount of time to produce results, but would make the plugin far easier to use.

Use of additional fields from data file

The STORM and PALM imaging processes produce a considerable amount of data more than just the x- and y- coordinate information that is used by the plugin. A summary of the headers of the columns from a file is given in Appendix B. A number of these extra parameters for each data point could be used to enhance the algorithms to produce better results. For example, the `width` field, which represents the full width at half maximum of the point, could be used to classify the data point and maybe give some indication about whether it is interesting or is noise.

References

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

Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997.

Werner Bailer. Writing imagej plugins—a tutorial. *Upper Austria University of Applied Sciences, Austria*, 2006.

C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

Boris Delaunay. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.

Levent Ertoz, Michael Steinbach, and Vipin Kumar. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on Clustering High Dimensional Data and its Applications at 2nd SIAM International Conference on Data Mining*, pages 105–115, 2002.

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

Nicholas Fang, Hyesog Lee, Cheng Sun, and Xiang Zhang. Sub–diffraction-limited optical imaging with a silver superlens. *Science*, 308(5721):534–537, 2005.

Darel Rex Finley. Ultra-easy polygon area algorithm with c code sample. `www.alienryderflex.com/polygon_area`, 2006. [Online; accessed 18-Aug-2014].

Michael F Goodchild and Andrew W Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proceedings of Auto-Carto*, volume 6, pages 400–407, 1983.

Frank Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

Samuel T Hess, Thanu PK Girirajan, and Michael D Mason. Ultra-high resolution imaging by fluorescence photoactivation localization microscopy. *Biophysical journal*, 91(11):4258–4272, 2006.

David Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Gesammelte Abhandlungen*, pages 1–2. Springer, 1970.

Gisli R Hjaltason and Hanan Samet. Speeding up construction of pmr quadtree-based

spatial indexes. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(2):109–137, 2002.

Marta Indulska and Maria E Orlowska. Gravity based spatial clustering. In *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 125–130. ACM, 2002.

Raymond A Jarvis and Edward A Patrick. Clustering using a similarity measure based on shared near neighbors. *Computers, IEEE Transactions on*, 100(11):1025–1034, 1973.

George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

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

Marc Loy, Robert Eckstein, Dave Wood, James Elliott, and Brian Cole. *Java swing.* " O'Reilly Media, Inc.", 2002.

MATLAB. *version 7.10.0 (R2010a).* The MathWorks Inc., Natick, Massachusetts, 2010.

Mohamed F Mokbel, Walid G Aref, and Ibrahim Kamel. 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, 2002.

Adriano Moreira and Maribel Yasmina Santos. Concave hull: A k-nearest neighbours approach for the computation of the region occupied by a set of points. 2007.

GM Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM, Ottawa, Canada*, 1966.

James D Murray and William VanRyper. Encyclopedia of graphics file formats. *Sebastopol: O'Reilly,— c1996, 2nd ed.*, 1, 1996.

NIH. *ImageJ API Documentation.* US National Institutes of Health, 1.48t edition, March 2014.

Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

Dylan M Owen, Carles Rentero, Jérémie Rossy, Astrid Magenau, David Williamson, Macarena Rodriguez, and Katharina Gaus. PALM imaging and cluster analysis of protein heterogeneity at the cell surface. *Journal of biophotonics*, 3(7):446–454, 2010.

Oxford English Dictionary. Oxford English Dictionary Online, 2nd edition. `http://www.oed.com/`, August 2014. [Online; accessed 28-Aug-2014].

Cai Qian, Karel Zak, and Heiko Carstens. lscpu from util-linux. `https://git.kernel.org/cgit/utils/util-linux/util-linux.git/`, 2006. [Online; accessed 25-Aug-2014].

WS Rasband. ImageJ, US National Institutes of Health. *Bethesda, Maryland, USA*, 2012, 1997.

Michael J Rust, Mark Bates, and Xiaowei Zhuang. Sub-diffraction-limit imaging by stochastic optical reconstruction microscopy (STORM). *Nature methods*, 3(10):793–796, 2006.

Hanan Samet. Applications of spatial data structures. *Computer Graphics, Image Processing, and GIS*, 1990.

Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in action*. Manning Publications Co., 2010.

Peter van Oosterom. Spatial access methods. *Geographical information systems*, 1:385–400, 1999.

David J Williamson, Dylan M Owen, Jérémie Rossy, Astrid Magenau, Matthias Wehrmann, J Justin Gooding, and Katharina Gaus. Pre-existing clusters of the adaptor lat do not participate in early t cell signaling events. *Nature immunology*, 12(7): 655–662, 2011.

Chenyi Xia, Wynne Hsu, Mong Li Lee, and Beng Chin Ooi. Border: efficient computation of boundary points. *Knowledge and Data Engineering, IEEE Transactions on*, 18(3): 289–303, 2006.

Jiang Zhong, Longhai Liu, and Zhiguo Li. A novel clustering algorithm based on gravity and cluster merging. In *Advanced Data Mining and Applications*, pages 302–309. Springer, 2010.

John Zukowski. *Java AWT reference*, volume 3. O'Reilly, 1997.

Appendix A

Running the Plugin

A.1 Contents of CD

The files contained on the CD accompanying this report are as follows (⊢ represents a directory and ▷ represents a file):

- ▷ `report.pdf`: PDF version of this report,
- ⊢ `plugin`: Directory containing nessessary file to compile and run the plugin.
  - ⊢ `src`: Directory containing the Java source files for the ImageJ plugin.
  - ▷ `build.xml` Apache Ant build file.
- ⊢ `sampledata`: Directory containing a number of sample data files for testing the plugin.

A.2 Compiling and Running the Plugin

The steps required to compile the plugin are given below. These steps require Apache Ant to be installed with a relatively recent version of the JavaVM.

1. Edit the `build.xml` file to change the third line with the `imagej-plugin-location` variable to point to the "jars" folder within the plugin section of the ImageJ executable folder.
2. Run the command `ant compile` from the same folder as the `build.xml` file to compile the Java source and generate the Java jar file ready for installation.
3. If there were no errors, run the command `ant imagej` to copy the generated jar file to the relevant folder to be run by ImageJ.
4. The command `ant imagej` can be run to both compile and copy the file.

To run the plugin from ImageJ, follow the steps below. Note that a minimum ImageJ version of `v1.48` is required to run the plugin. ImageJ can be updated by clicking on `Help >> Update ImageJ`. The most recent version (as of 5th Sept. 2014) is `v1.49g`.

1. If ImageJ is already running, refresh the menus so that the latest version of the plugin by clicking `Help >> Refresh Menus`.
2. Run the plugin by clicking `Plugins >> jars >> Cluster Analysis`.

Appendix B

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 items of data, separated by newlines.

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

- `Channel Name`
- `X`
- `Y`
- `Xc`
- `Yc`
- `Height`
- `Area`
- `Width`
- `Phi`
- `Ax`
- `BG`
- `I`
- `Frame`
- `Length`
- `Valid`
- `Z`
- `Zc`
- `Photons`
- `Lateral Localisation Accuracy`
- `Xw`
- `Yw`
- `Xwc`
- `Ywc`

Only the `Xc` and `Yc` columns, which represent the x- and y- coordinate of the data point, are used.

Appendix C

Benchmarking Hardware

The benchmarking of algorithms to get time to run information is performed on hardware with the following specifications, as recorded by the `lscpu` utility from the util-linux package [Qian et al., 2006].

Architecture:	x86_64
CPU op-mode (s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU (s):	4
On-line CPU (s) list:	0–3
Thread (s) per core:	1
Core (s) per socket:	4
Socket (s):	1
NUMA node (s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	60
Stepping:	3
CPU MHz:	800.000
BogoMIPS:	6385.81
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	6144K
NUMA node0 CPU (s):	0–3

Appendix D

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.

$$R_{\text{circle}} = 1 \tag{D.1}$$

The value must be unit-less, so divide area by perimeter squared,

$$A = \pi r^2 \tag{D.2}$$
$$p = 2\pi r \tag{D.3}$$
$$\frac{A}{p^2} = \frac{\pi r^2}{4\pi^2 r^2} \tag{D.4}$$

Remove constants to give formula,

$$R = 4\pi \times \frac{\pi r^2}{4\pi^2 r^2} \tag{D.5}$$
$$= \frac{4\pi A}{p} \tag{D.6}$$

Test for regular shapes.

Circle:

$$A = \pi r^2 \tag{D.7}$$
$$p = 2\pi r \tag{D.8}$$
$$R = 4\pi \times \frac{\pi r^2}{(2\pi r)^2} = 1 \tag{D.9}$$

Square:

$$A = h^2 \tag{D.10}$$
$$p = 4h \tag{D.11}$$
$$R = 4\pi \times \frac{h^2}{(4h)^2} = \frac{\pi}{4} \approx 0.785 \tag{D.12}$$

Equilateral Triangle:

$$A = \frac{\sqrt{3}}{4}a^2 \tag{D.13}$$

$$p = 3h \tag{D.14}$$

$$R = 4\pi \times \frac{\frac{\sqrt{3}}{4}a^2}{(3a)^2} = \frac{\sqrt{3}\pi}{9} \approx 0.605 \tag{D.15}$$

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

$$A = 6.28 \tag{D.16}$$

$$p = 9.69 \tag{D.17}$$

$$R = \frac{6.28}{9.69} \approx 0.648 \tag{D.18}$$

Appendix E

Diagrams

E.1 Add-point Flow Diagram

Figure E.1 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.

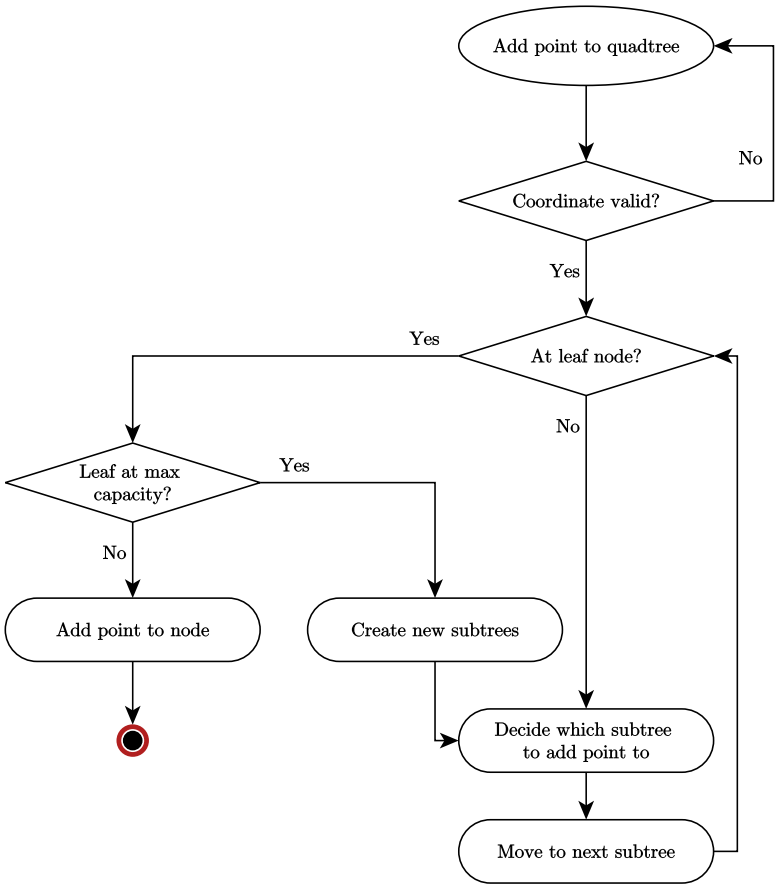
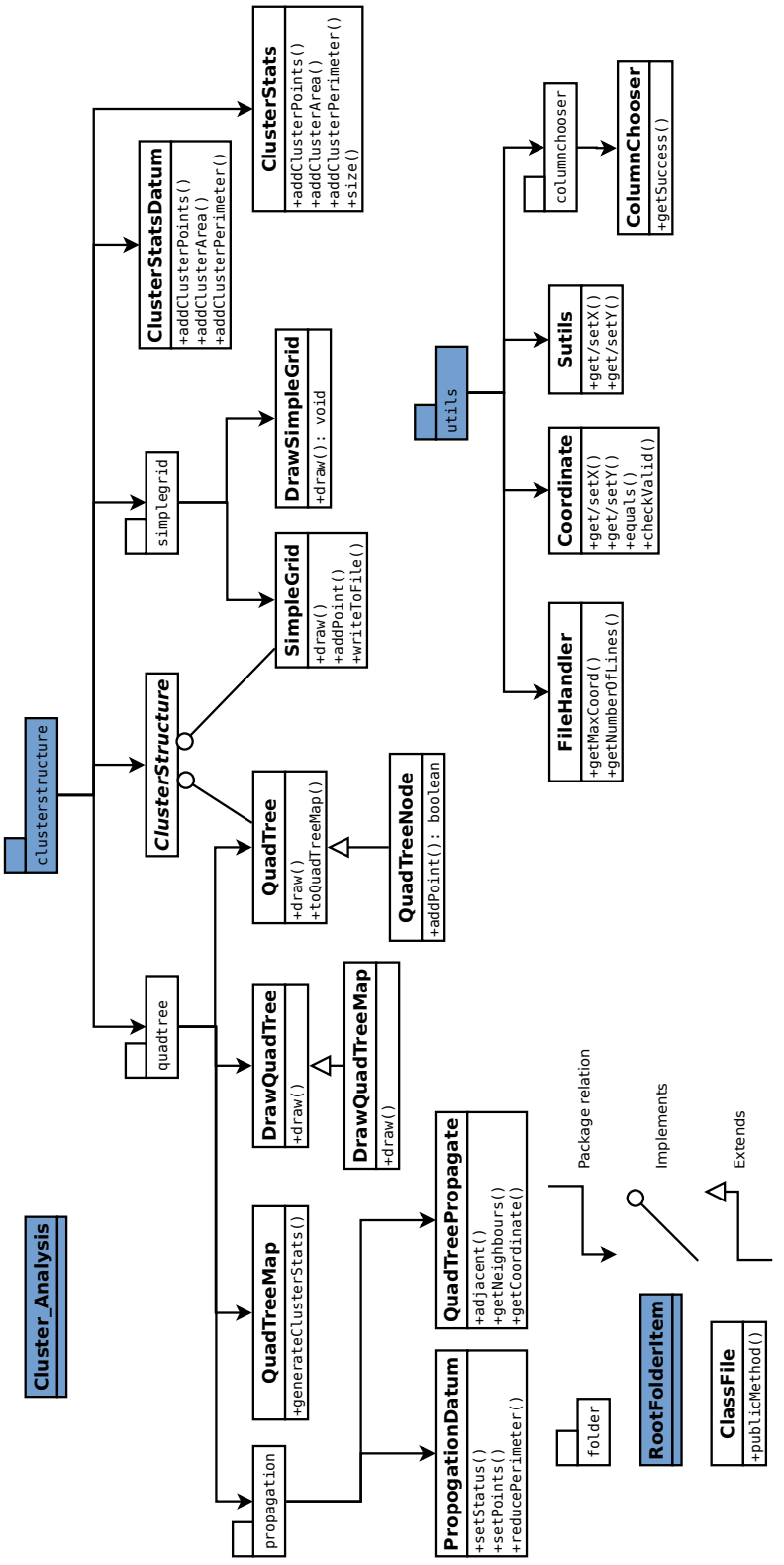


Figure E.1: *Class diagram for the Cluster Analysis ImageJ plugin written for this project.*

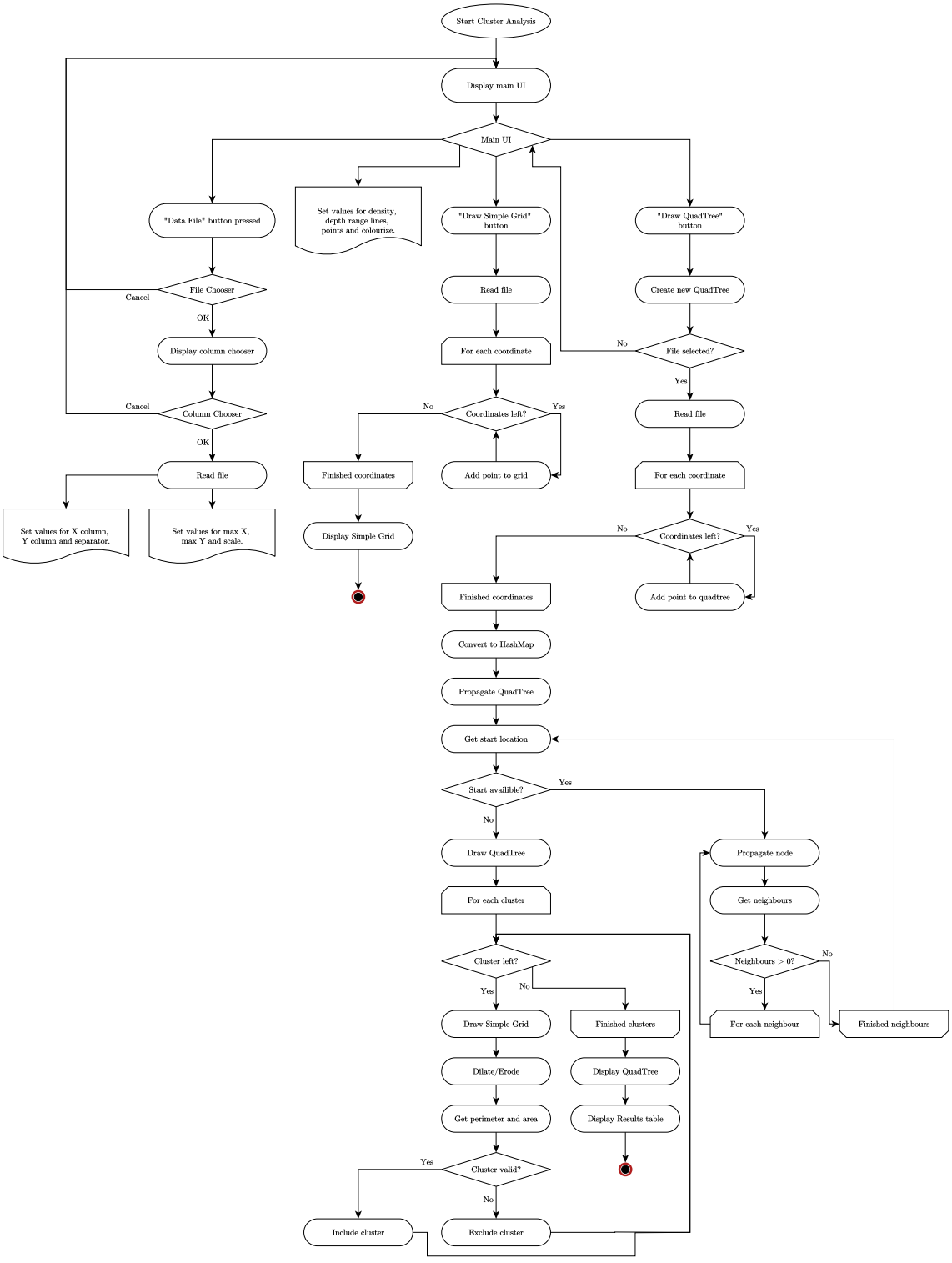
E.2 Class Diagram

Figure E.2 is a class diagram showing the structure of the ImageJ plugin.



E.3 Plugin Flow Diagram

Figure E.3 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.



Appendix F

Data Set Generator

A small C++ utility was written to aid the generation of data sets for testing. This program takes an image file and produces data based on the pixels in the image. The input must have the same format as when saving from ImageJ in the `text image` format. The file will then be formed of a decimal value, from 0 to 255, for each pixel in the image, separated by tab characters.

The utility code is shown in Listing F.1. It is used by calling `gen-coords FILENAME` at the command line and outputs the coordinates to standard output.

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <stdlib.h>

using namespace std;

int main(int argc, char* argv[]) {

   if (argc != 2) {
      cout << "Usage: " << argv[0] << " FILENAME" << endl;
      exit (EXIT_FAILURE);
   }

   string filename = argv[1];
   ifstream scores(filename.c_str());
   string line = "";

   // For each line, for each tab separated field, output coord if 0.
   int y = 0;
   while(getline(scores, line)) {

      string delimiter = "\t";
      size_t pos = 0;
      string token;
      int x = 0;
      while ((pos = line.find(delimiter)) != string::npos) {

         token = line.substr(0, pos);
         if (token == "0") {
            cout << x << "\t" << y << endl;
         }
         x++;
         line.erase(0, pos + delimiter.length());
      }

      y++;
   }
}
```

Listing F.1: *Utility to gen data sets from images.*