LONDON'S GLOBAL UNIVERSITY

UCL

COMPUTING AND STATISTICAL DATA ANALYSIS IN 3D IMAGING AND ANALYSIS OF A HUMAN CHROMOSOME

AUTHOR: SHERMAN LEIGH IP (1018526)

UNIVERSITY COLLEGE LONDON (UCL)

MSCI PHYSICS

PHASM201: PHYSICS PROJECT (MASTERS LEVEL)

SUPERVISOR: PROFESSOR IAN ROBINSON

SUPERVISOR: DR. GRAEME MORRISON

DATE: WEDNESDAY 2ND APRIL 2014

Chromosomes store and compress DNA using a series of loops and coils called nucleosomes and it was suggested that they form a 30 nm chromatin fibre. A series of serial section images of human chromosomes were taken using serial block face SEM. The objective of this project was to analyse a stack of serial block face SEM images, of a human chromosome, using computational tools to identify the external shape and find statistically significant internal structure in chromosomes, as well to model how the 30 nm chromatin fibres were arranged.

The images were statistically analysed and were segmented using a threshold in ImageJ. The marching cubes algorithm was programmed, in Processing, to produce a volume visual using the threshold images to see the external shape of the chromosome.

ImageJ maxima search was used to find significant internal structure. Evidence of 30 nm chromatin fibres may have been found because the same value of 30 nm corresponded to the mean distance to the nearest neighbour for each maxima. Each maxima was modelled as a sphere to see if they form coils and to estimate the number of base pairs in the chromosome.

CONTENTS

I. TABLE OF FIGURES

II. LITERATURE REVIEW

All living organisms are made up of basic biological units called cells. [1, p. 49] [2, p. 3] Organisms, like bacteria, can be unicellular, meaning made up of only one cell, while complex organisms, like humans, are made of many cells and are therefore multicellular. [1, p. 49]

Cells produce a good environment for chemical reactions to take place to sustain life of the organism. [2, p. 3] They replicate themselves by dividing to produce two genetically identical daughter cells. [1, p. 81] [2, p. 83] [3] This process is called mitosis which is part of the cell cycle. [1, p. 81] [2, p. 83] [3]

Figure 3 shows the structure of the cell. The cell surface, called the plasma membrane, is a partially permeable wall which separates internal chemical reactions with the environment outside the cell. [2, p. 4] The most important features inside the cell, for this project, are cytoplasm, centrioles and the nucleus. Cytoplasm is a solution of many substances in water, mainly proteins, in which many chemical reactions take place. [1, p. 49]

The nucleus is the largest structure inside the cell. [2, p. 11] The surface of the nucleus is the nuclear envelope, consisting of two membranes. [1, p. 66] [2, p. 11] There are some gaps in the nuclear envelope, called nuclear pores, and these controls the exchange of biological substances between the nucleus and cytoplasm outside the nucleus. [1, p. 66] [2, p. 11] Genetic materials, deoxyribonucleic acid (DNA) and chromosomes, are stored inside the nucleus, to prevent damage to it. [1, p. 64] [2, p. 12]

A molecule of DNA, in humans, is stored within 46 different chromosomes inside the nucleus. [1, p. 88] [2, p. 80] DNA can replicate itself perfectly, before cell mitosis, and pass on its genetic material to the two daughter cells created during cell mitosis. [1, p. 82] [2, p. 67] DNA is extremely important as it provides instructions to cells, for example what proteins to produce, and therefore determines the characteristics of the cell and organism. [1, p. 82] [2, p. 70]

DNA consists of polymers, meaning that they are made up of molecular structures. [2, p. 65] These molecular structures are called nucleotides. [1, p. 97] [2, p. 66] Nucleotides are made up of 3 substances, a pentose sugar called deoxyribose, a phosphate group and one of the 4 nitrogen containing bases: adenine (A), thymine (T), guanine (G) and cytosine (C), as shown in Figure 1. [1, p. 97] [2, p. 67] [4]

A polynucleotide strand can be formed by linking many nucleotides to make a long chain as shown in Figure 2. [4] [5, p. 198] DNA is made of two polynucleotide strands, running anti-parallel to each other, wounded in a double helix with 10 nucleotides per complete turn, held together by hydrogen bonds between the complementary base pairs A-T and C-G, as shown in Figure 4. [4] [5, p. 198]

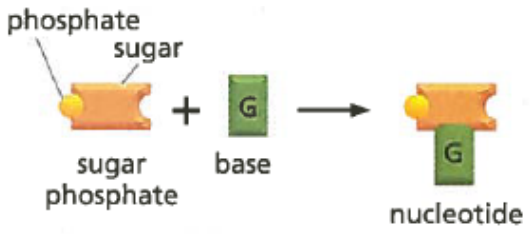


Figure 1 Nucleotides are the building blocks of DNA, consisting of phosphate, sugar and a nitrogen containing base. [5, p. 198]

Figure 2 A polynucleotide strand is formed by linking together a sequences of phosphate and sugar with one of the four nitrogen-containing bases, adenine, thymine, guanine and cytosine, attached to each sugar. [5, p. 198]

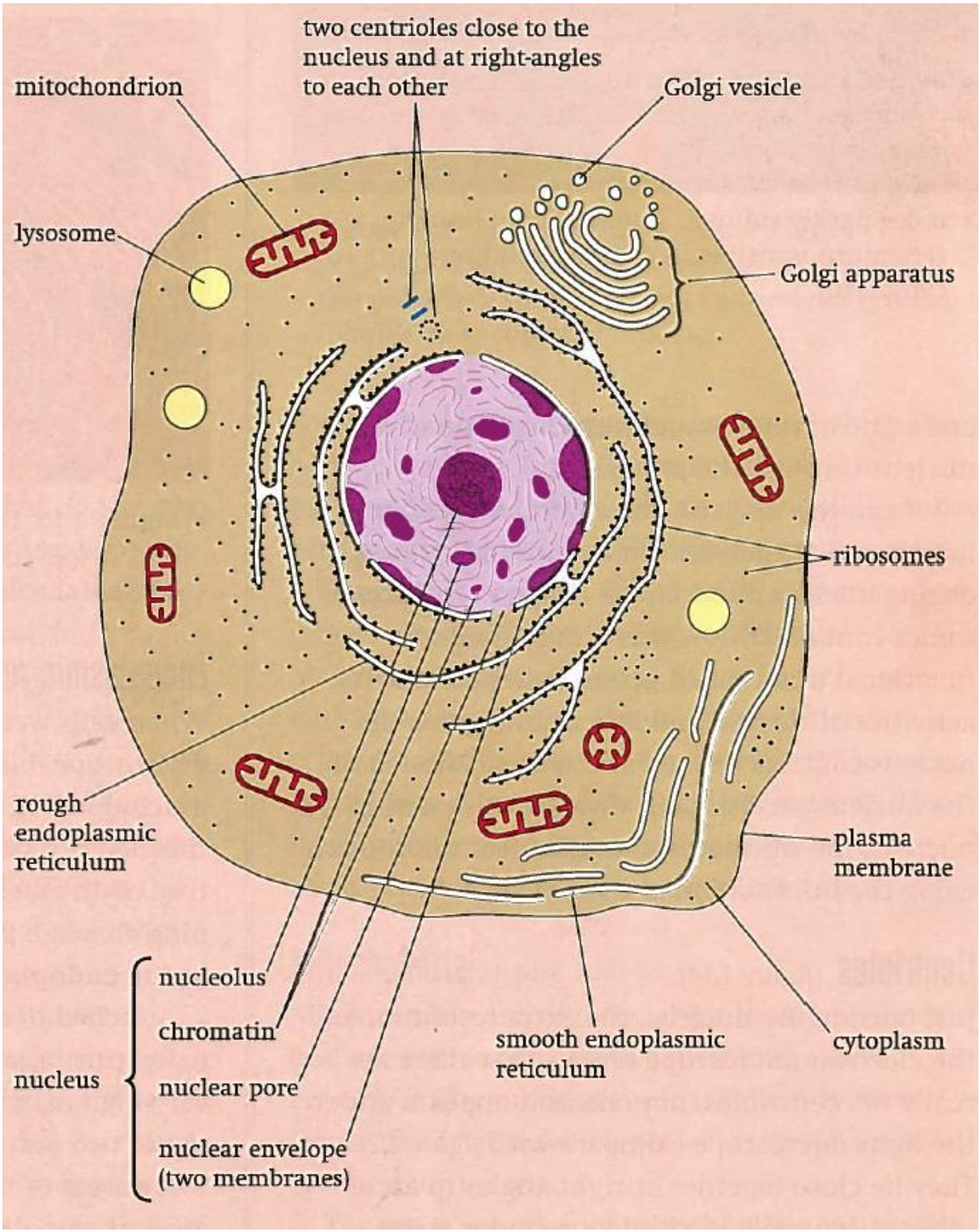


Figure 3 The internal features and structure of a typical animal cell. [2, p. 11]

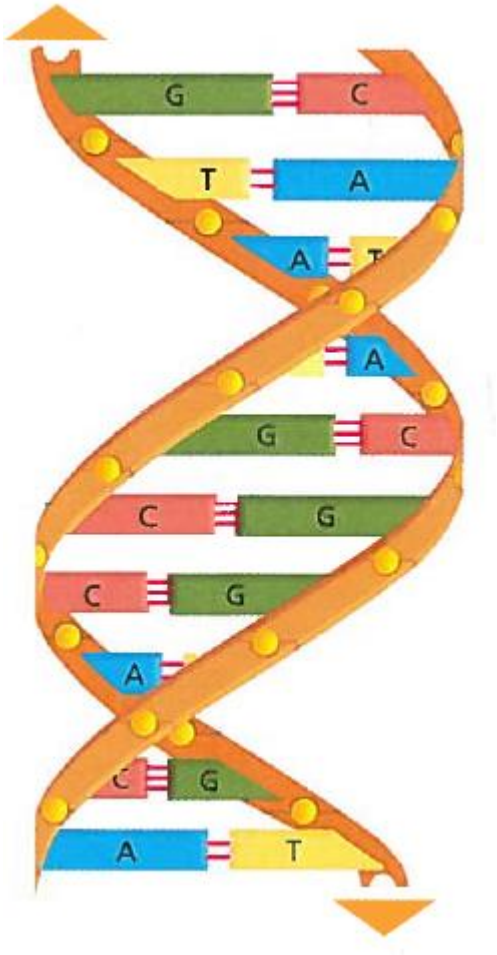


Figure 4 A pair of anti-parallel polynucleotide strands wound together in a double helix and held together by hydrogen bonds between complementary base pairs to form DNA. [5, p. 198]

It is the sequences of these base pairs which provide instructions for the production of proteins. [1, p. 98] [2, p. 70] A group of three nucleotides, called a triplet, codes for an amino acid. [1, p. 98] [2, p. 71] By linking amino acids together, proteins are produced. The length of DNA which produces a single protein is called a gene. [1, p. 98] The 64 possible combinations of triplets can code for 20 different amino acids. [1, p. 98]

By using the results from Rosalind Franklin, James Watson and Francis Crick in 1953 deduced the double helix structure of DNA, and this discovery was significant in how DNA can replicate itself perfectly every time. [2, p. 67] [4] The hydrogen bonds between the two polynucleotide strands can be easily broken so that the polynucleotide stands can unwind. [1, p. 97] Free nucleotides then can pair up with their respective complementary bases in each polynucleotide strand to form a new polynucleotide strand, as shown in Figure 5. [1, p. 97] [2, p. 67] [5, p. 3]

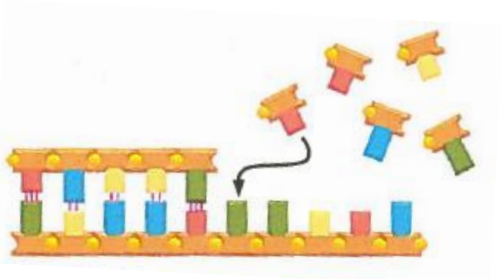


Figure 5 Free polynucleotide can form new DNA by pairing up with their respective complementary base pairs on an unwound polynucleotide strand. [5, p. 3]

The result is two identical DNA, one new DNA was formed from the old DNA. [1, p. 97] [2, p. 67] [5, p. 3] This method of replication is known as semi-conservative replication because half of the new DNA molecules comes from the old DNA. [1, p. 97] [2, p. 67]

The complete set of genetic material, or DNA, of an organism is called a genome. [5, p. 200] A human genome contains about 3.2 billion base pairs, stored across 46 chromosomes. [5, p. 200]

There are two types of chromosomes, autosomes and sex chromosomes. [1, p. 134] [2, p. 80] There are two sex chromosomes, X and Y, and they determine the sex or gender of the organism. [1, p. 134] [2, p. 80] A male human will have an X and Y chromosome and a female will have two X chromosomes. [1, p. 134] [2, p. 80] The remaining 44 chromosomes are called autosomes and they always comes in homologous pairs, where one comes from the mother and another from the father of the organism. [2, p. 80]

Chromosomes are most visible in mitosis form in which the chromosomes are at their densest and in the process of replicating itself. [5, p. 243] A technique called DNA hybridization distinguish the chromosomes, in mitosis form, by 'painting' them different colours and arranging them in order. [5, p. 202] A karyotype, a photograph of all the human chromosomes is shown in Figure 6. [5, p. 203] [6]

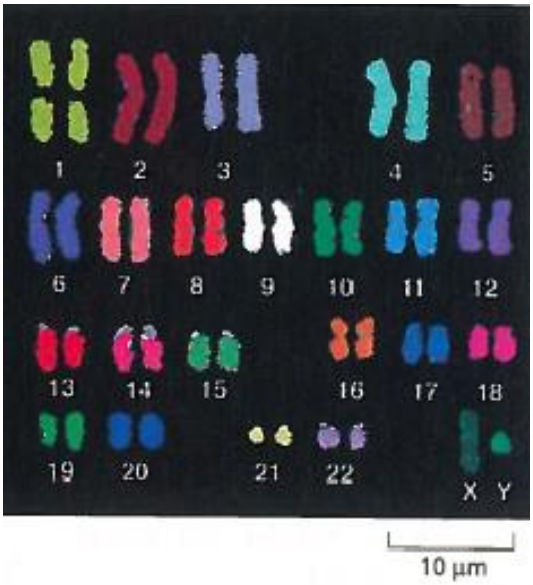


Figure 6 A karyotype of male human chromosomes. DNA hybridization is a technique used to 'paint' each homologous pairs of chromosomes different colours, making them distinguishable. [5, p. 203]

Another technique, to distinguish the different chromosomes, involves staining the chromosomes with a dye which produces very clear and regular bands on chromosomes. [5, p. 203] [7] It is not well understood why these bands are as regular but it as shown in Figure 8. [5, p. 203] [7]

The structure of a mitosis chromosome is as shown in Figure 7. [5, p. 243] The most important features are the two sister chromatids, which contain an identical DNA each, held together by the centromere, which contains a protein complex called kinetochore that is used to separate the two chromatids during mitotic. [5, p. 210]

Otherwise chromosomes can be invisible with the light microscopic when in interphase form. [5, p. 209] Cells goes through a cycle in which sometimes chromosomes are condensed and sometimes they are not. [5, p. 208]

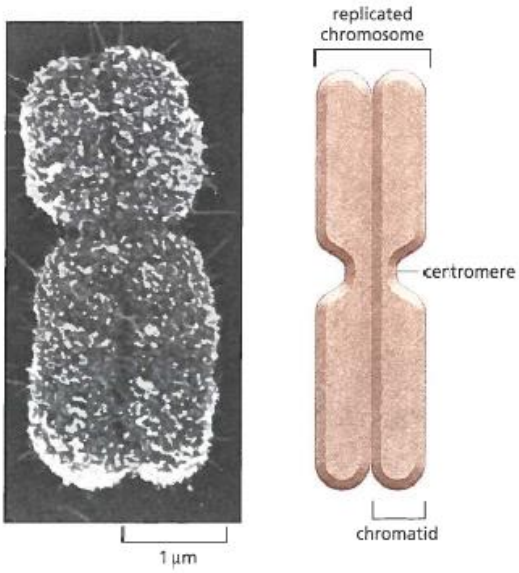


Figure 7 (Left) A scanning electron micrograph of a mitosis chromosome. (Right) A drawing of a mitosis chromosome which the chromatid and centromere were labelled. [5, pp. 209, 243]

The cell cycle is the period of time from the birth of a daughter cell to the replication of itself. [2, p. 83] At the start of the cycle called interphase, the cell has just been formed and separated from its sister. [1, p. 81] [2, p. 83] The cell grows to its normal size and carries out normal functions just like its parent. [2, p. 83] At this stage, the interphase chromosomes exist as long tangled threads of a single DNA. [5, p. 209] The cell spends most of its time in interphase and this is when genetic information is being read out. [5, p. 209] However when a signal is received for the cell to divide, the DNA replicates so that the nucleus contains two strands of identical DNA. [2, p. 83]

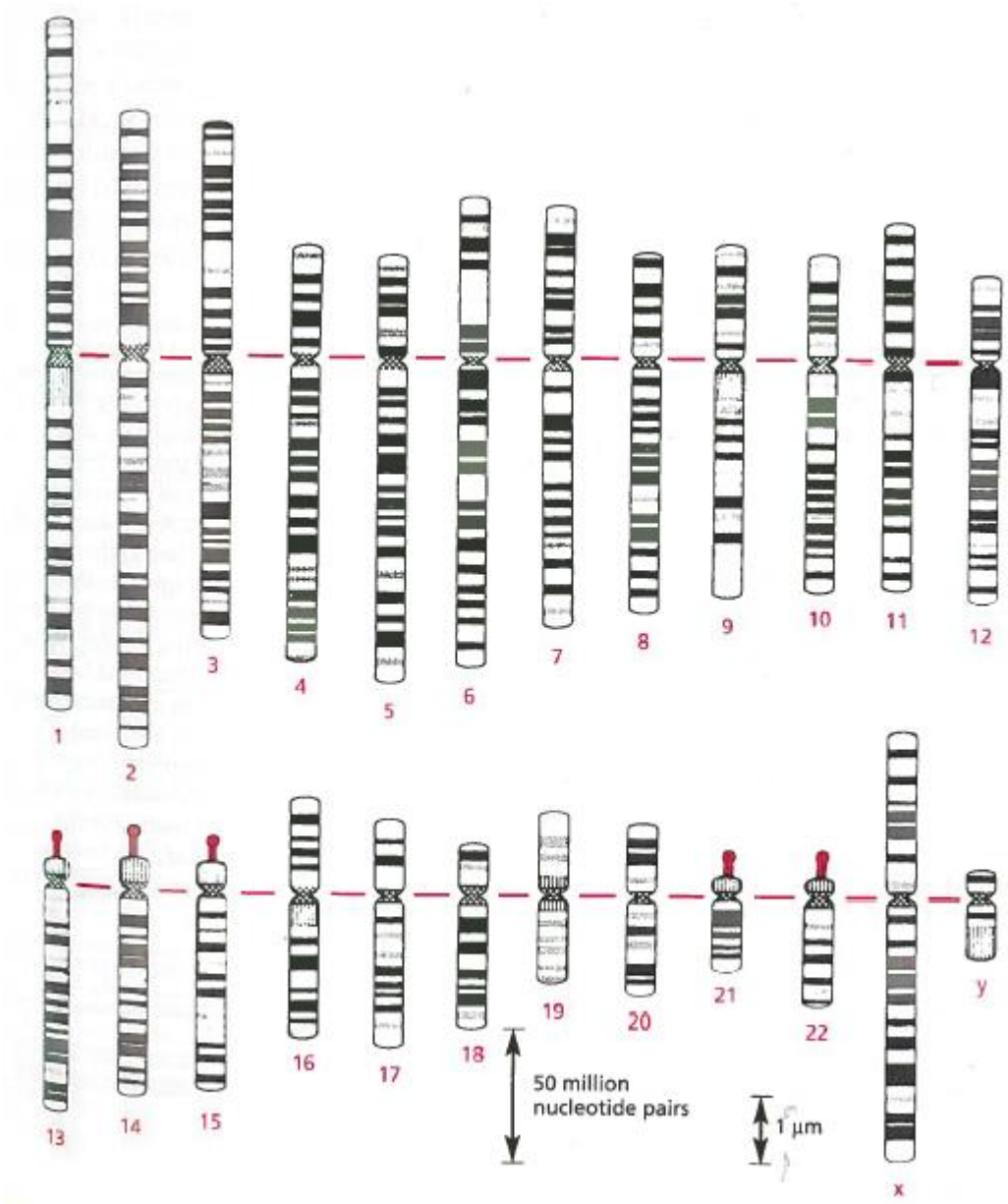


Figure 8 By staining the chromosomes, very clear bands can be seen. The 24 chromosomes shown are lined up by their centromere. [5, p. 203]

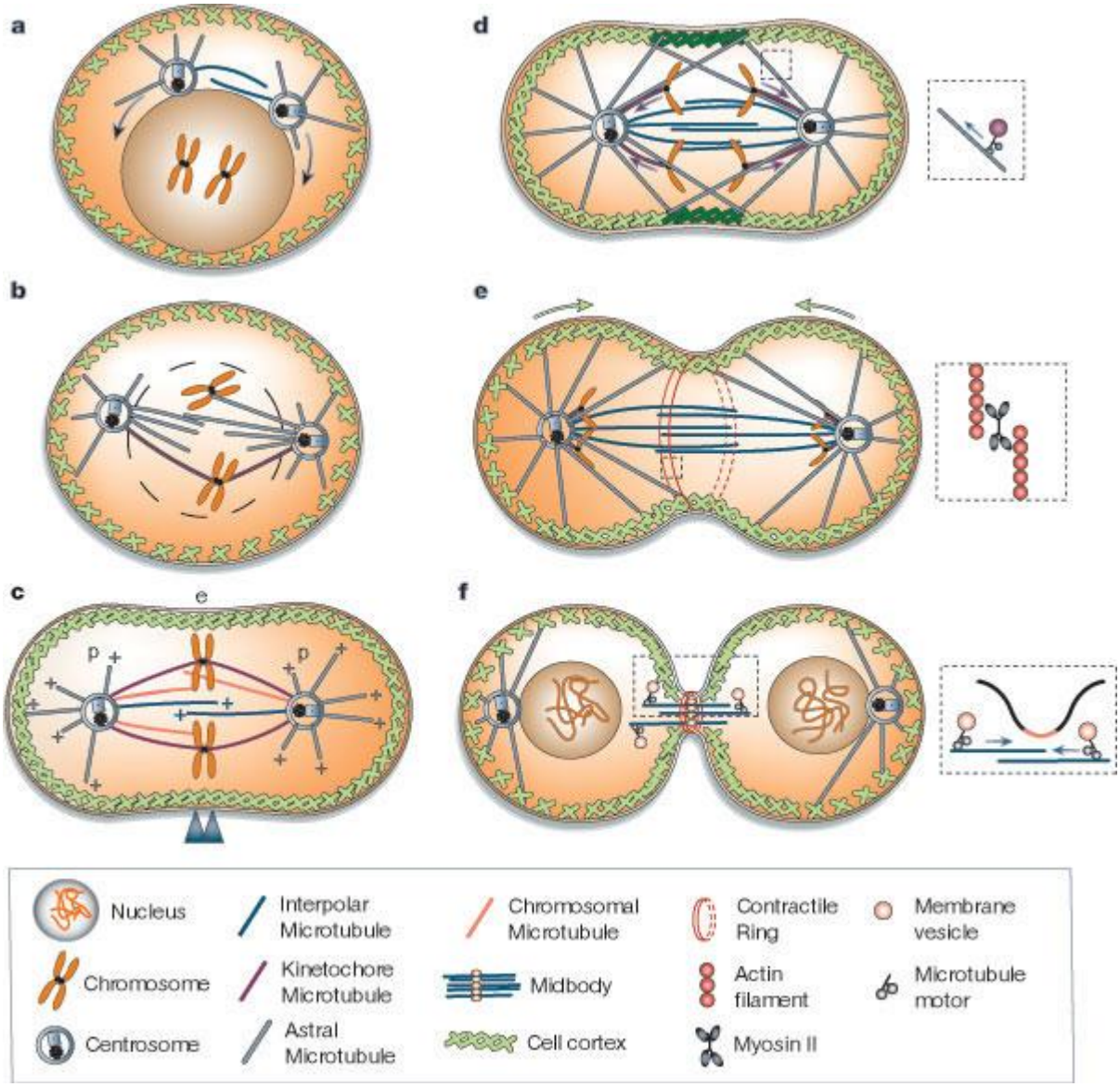


Figure 9 The stages of division of a mitosis cell. ***a)*** *Prophase: the replicated centrioles (surrounded by preicentriolar material to form centrosomes) move to opposite ends of the nucleus.* ***b and c)*** *Metaphase begins and the centrioles create spindles, made of protein microtubules, to organise the mitosis chromosomes.* ***d and e)*** *Anaphase: the centrioles pull the chromatids apart and drag them to the poles.* ***f)*** *Telophase: The nuclear envelope reform to segregate the nucleus. The chromatids in each nuclei begins to uncoil to form interphase chromosomes. [3]*

Mitosis follows interphase and this is when the nucleus starts to divide to give birth to two identical daughter nuclei. [1, p. 81] [2, p. 84] Mitosis can be split into 4 continuous stages, prophase, metaphase, anaphase and telophase, as shown in Figure 9. [1, p. 81]

At prophase, the centriole replicates and starts moving to opposite ends, or the poles, of the nucleus. [2, p. 85] [3] The chromosomes replicate to form sister chromatids joined in the middle by a centromere, where each chromatids contains a molecule of DNA. [3] [5, p. 209] The nuclear envelope breaks down to allow the chromosomes for movement. [2, p. 85] [3]

Once the pair of centrioles have reached to the poles, metaphase begins and they create the spindle apparatus, made of protein microtubules which attach to each centromere, organising the mitosis chromosomes such that they line up at the equator of the nucleus. [2, p. 85] [3] [5, p. 208] At this stage, this is when chromosomes are most visible. [1, p. 82] [5, p. 208]

Then quite suddenly, the centrioles pull the chromatids apart by their centromeres and drag them to the poles by the spindles. [1, p. 82] [3] This process is called anaphase. [1, p. 82] [2, p. 85] [3]

At telophase, there are 46 chromatids and a centriole at each pole. [2, p. 85] The nuclear envelope reform at each pole, creating two nuclei. [2, p. 85] [3] The remains of spindles break down and cytokinesis, the separation of cells, begins. [2, p. 85] [3] The chromatids in each nuclei beings to uncoil, to form interphase chromosomes, to start interphase. [1, p. 82] [2, p. 85] [3] [5, p. 208]

Chromosomes in the cell cycle do a good job of not only organising replicated DNA so that each daughter cell is genetically identical, but also storing and compressing the human genome into a tiny volume. [5, p. 202] The nucleus is about 6 μm long in diameter but laying out the human genome in a straight line would reach 2 meters long. [5, p. 202]

Chromosomes are mostly made up of proteins, called histones, and DNA. [5, p. 211] The complex of histones and DNA is called chromatin. [5, p. 211] It was discovered in 1974 that DNA is packaged into chromosomes by nucleosomes. [5, p. 211] [6] Nucleosomes are a series of loops and coils in which DNA is tightly wrapped around histones. [5, p. 211] [6] Because there are about 30 million nucleosomes in a human cell, on average there about 650 thousand nucleosomes per chromosome. [5, p. 211]

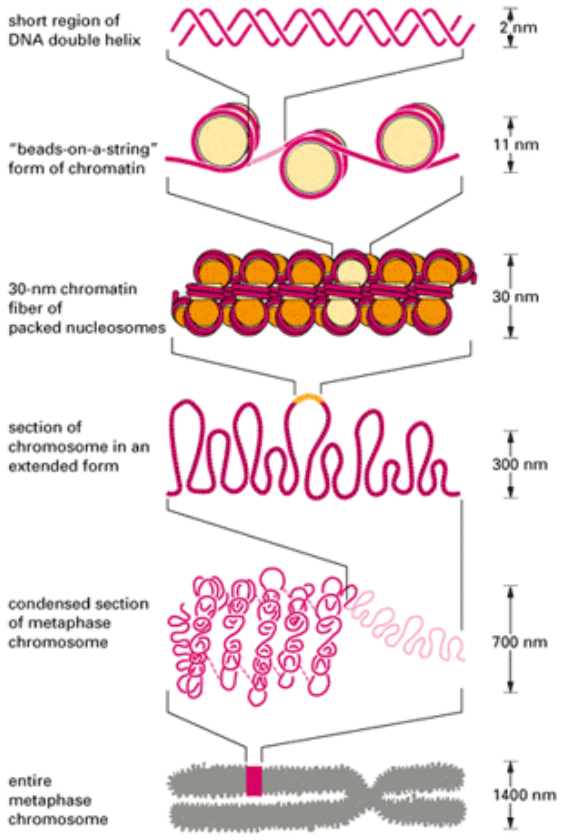


Figure 10 DNA is wrapped around histones to form nucleosomes. The nucleosomes form a 30 nm chromatin fibre to produce a very dense mitotic chromosome. [5, p. 244]

It can be shown by treating nucleosomes such that they fold out, they appear to be like ‘beads on a string,’ as shown in Figure 10. [5, p. 211] [6] [8] These ‘beads’ are the nucleosome cores which consist of an octameric of two of each of these histones: H2A, H2B, H3, H4. [5, p. 211] [6] [8] [9] These nucleosome cores have diameter of about 11 nm and DNA wraps itself around each nucleosome core 1.7 times in a left handed coil, which is 147 base pairs, as shown in Figure 11. [5, p. 212] [8] [9] The strands of DNA between nucleosome cores is called linker DNA, which consists of between 0 to 80 base pairs. [9]

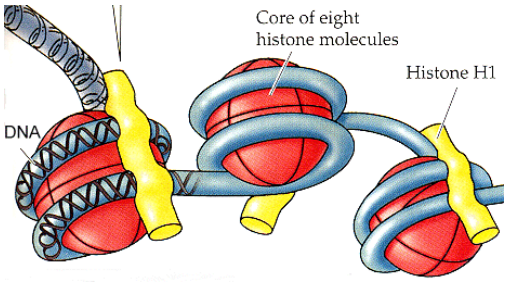


Figure 11 DNA is wrapped around each nucleosome core, consisting of histones, 1.7 times in a left handed coil. [6]

It was suggested that nucleosomes form a 30 nm diameter chromatin fibre as shown in Figure 10. [5, p. 216] [9] It is not well known how the nucleosomes are arranged or packed in the 30 nm chromatin fibre but many experimental data have produced many suggestions and models about how nucleosomes are arranged in the 30 nm chromatin fibre. [5, p. 217] An experiment using x-ray crystallography suggests that nucleosomes are arranged in a zigzag model while another experiment using cryoelectron microscopy suggests a solenoid arrangements in the 30 nm chromatin fibre. [5, p. 217] [9] [10]

In the zigzag model, the linker DNA remains extended and straight throughout the whole arrangement in the chromatin fibres. [9] The nucleosomes are arranged such that they produce a zigzag formation in the chromatin fibres as shown in Figure 12 and Figure 13. [9] [10] In contrast the linker DNA in solenoid arrangements bends to produce a solenoid structure as shown in Figure 12 and Figure 14. [9] [10]

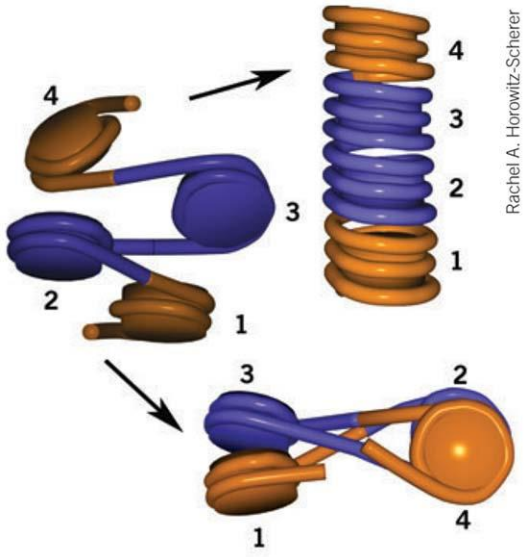


Figure 12 (Lef) Nucleosomes in an extended state. The linker DNA can remain extended and straight to form a zigzag formation (bottom right) or bends to form a solenoid (top right). [10]

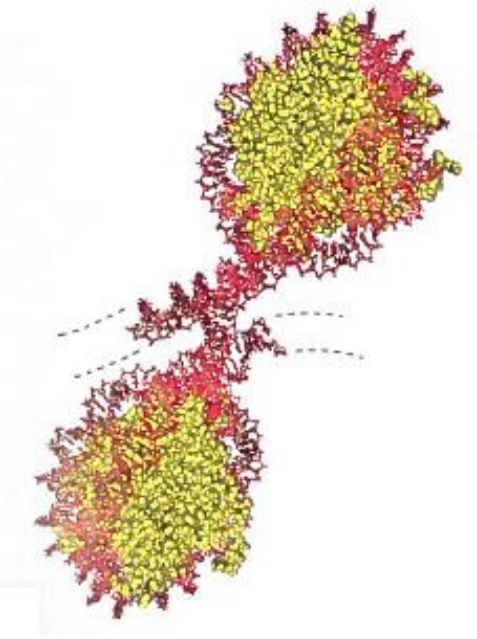


Figure 13 The zigzag formation of the chromatin fibres on one of the two chromatids using data from an x-ray crystallography. [5, p. 217]

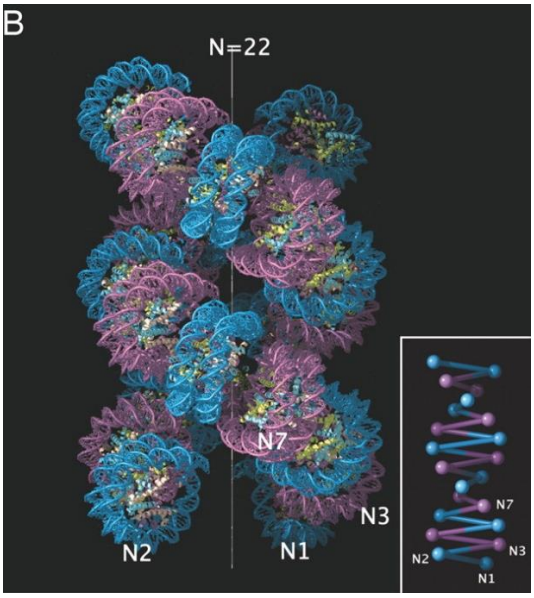


Figure 14 Solenoid arrangements of chromatin fibres in a mitotic chromosome using data from cryoelectron microscopy. [9]

However there is still some uncertainty in the arrangement of nucleosomes in chromatin fibres. [5, p. 217] One controversy is that the length of linker DNA varies throughout the arrangement and each model should be able to accommodate it. [9] The solenoid arrangements can easily accommodate for variable DNA length, but on the other hand there are some unanswered questions on how the zigzag model can deal with variable linker DNA length, whenever the size of the nucleosome fibres varies with linker DNA length. [9]

Many of the small scale biological substances discussed can be viewed using a range of equipment. [2, p. 3] One of the earliest equipment used is the light microscope. [2, p. 3] It works by focusing light onto the specimen, the light penetrates the specimen and then is refocused onto the eyepiece. [1, p. 51] The specimen must be thin enough to allow light to penetrate it and should be stained to allow transparent biological features to become visible under the light microscope. [1, p. 51] Often the specimens are submerged in liquid between a glass slide and a cover slip. [1, p. 51] The maximum resolution of the light microscope is about 200 nm, which is enough to see mitotic chromosomes. [1, p. 52] [2, p. 8]

To view objects smaller than 200 nm, the transmission electron microscope (TEM) or the scanning electron microscope (SEM) can be used. [1, p. 51] [2, p. 9] TEM works in the same principle as the light microscope but electrons are used instead of light, producing a monochromatic image where the brightness of a pixel corresponds to the energy of the electron detected in the fluorescent screen. [1, p. 51] Electrons are fired at the specimen and are focused using electromagnets. [1, p. 51] The specimen must be ultra-thin, thin enough for a fraction of electrons to penetrate it, and should be stained using metal ions. [1, p. 51] [11] Particular parts of the cell and other biological substances take in the metal ions, which are large and positively charged, so that they scatter electrons and will appear dark in the image. [1, p. 51] [2, p. 10] The specimen must also be in a vacuum to prevent air particles scattering electrons. [1, p. 51] [2, p. 11] As a result the specimen must be dehydrated therefore only dead material can usually be viewed in an electron microscope. [1, p. 51] [2, p. 11] In SEM, backscattered or reflected electrons are detected instead of transmitted electrons. [1] [2, p. 52] [12]

A three-dimensional (3D) image of the specimen can be reconstructed using a continuous parallel series of many TEM images of ultra-thin serial sections of the specimen. [13] Serial sections can be obtained by placing the specimen in an ultramicrotome and using a diamond knife to cut the specimen. [13] [14] Each image of serial sections can be stacked together as a series using the appropriate software to produce a 3D reconstruction of the specimen. [13] This method of serial

sectioning imaging is called serial-section TEM (ssTEM). [14]

ssTEM, however, can be very time consuming and labour intensive to obtain many fragile serial sections of a specimen in the ultramicrotome [11] [14]. By cutting the specimen using a diamond knife, the continuity between each section is destroyed and a lot of work is needed to precisely align each image of serial sections of the specimen. [14]

Another method of serial sectioning imaging is called serial block face SEM (SBFSEM). [11] [13] [14] Instead of viewing each serial section under a TEM, the surface of the specimen is viewed using a SEM. [15] An ultra-thin serial section is extracted, or sectioned, and discarded from the top of the specimen using a diamond knife which exposes a freshly cut block face. [15] The new face is then viewed and imaged using a SEM. [11] [15] This is done many times to obtain a stack of SEM images of the specimen surfaces after each serial section removal, as shown in Figure 15. [11] [15]

The main advantage of SBFSEM is hundreds of images can be generated in a few days and there is no need to handle fragile serial sections, they are simply discarded. [11] [14]

Experiments were done by J. Rouquette et al. to obtain a 3D reconstruction of chromatin arrangement of an interphase nuclei of rat hepatocytes using ssTEM and SBFSEM. [15] The specimen was treated, in addition to staining and dehydration as mentioned before, by embedding it in a plastic resin. [15]

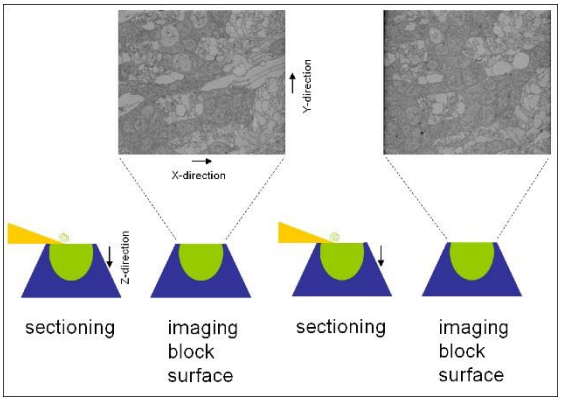


Figure 15 SBFSEM consist of a series of sectioning of the surface of the specimen and imaging the new exposed surface using SEM. [11]

There has been some work on using SBFSEM to image human interphase chromosomes. The objective of this project was to analyse SBFSEM images using computational tools to identify the external shape and find statistically significant internal structure of chromosomes. Depending on the scale of the images, it may be possible to look at how nucleosomes are arranged in 30 nm chromatin fibres and also how the 30 nm chromatin fibre is coiled up and compressed in an interphase chromosome.

III. DEVELOPMENT OF A 3D RENDERING PROGRAM

The objective of this section was to render a 3D view of a chromosome, using a stack of SBFSEM images of a chromosome, using open source software.

Using open source programming languages was an advantage because they are free to be used and shared so no costs were involved, compared to commercial software and programs which can require yearly contracts. [16] [17]

ImageJ and Fiji are open source software, written in Java, designed for scientific image processing. [18, pp. 1, 8] They support stacks, meaning it can process images obtained from SBFSEM, making it very suitable for this project. ImageJ and Fiji also have many procedures, functions and plugins for image processing, for example cropping, noise removing and thresholds. [18] Fiji is just an extension of ImageJ with more plugins and features, making Fiji more useful to work with. [18, p. 101]

Processing is an open source programming language built on Java. [17] The main purpose of Processing is to make a suitable environment for programmers to produce computer graphics. [19, p. 1] Processing makes use of OpenGL compatible graphics card, which enables good rendering of 2D and 3D graphics. [19, p. 528]

ImageJ and Processing were used together to produce a 3D rendering of a chromosome using a stack of SBFSEM chromosome slices.

A threshold interval was set to select pixels, from the data, with certain intensity values to produce binary images. In a binary image, each pixels can only have two possible intensity values. An intensity value of one was given to pixels in the threshold interval, otherwise an intensity value of zero was given. This method of abstraction or segmentation was used to identify the shape of the chromosome.

All 22 slices from the SBFSEM images of a human chromosome are shown in Figure 17. The colour depth of these images was 16-bit grayscale, meaning each pixel can have 65546 different intensity values. Each slice had dimensions of 153 x 168 pixels where each pixel represent a scale of 27 nm x 27 nm and each slice was separated by 100 nm.

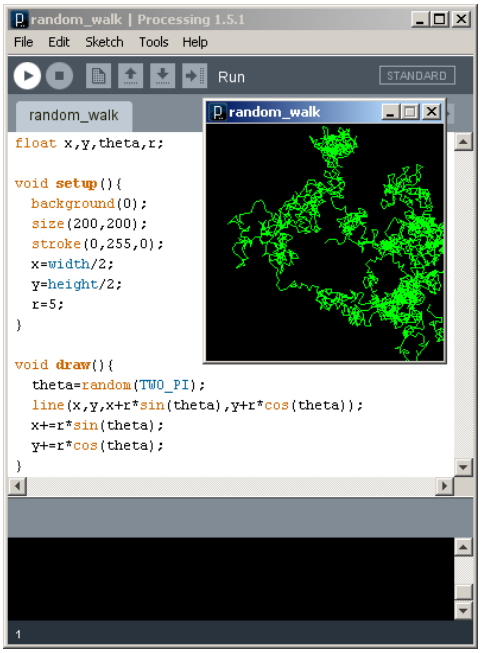


Figure 16 A screen shot of a simple Brownian motion simulation in Processing.

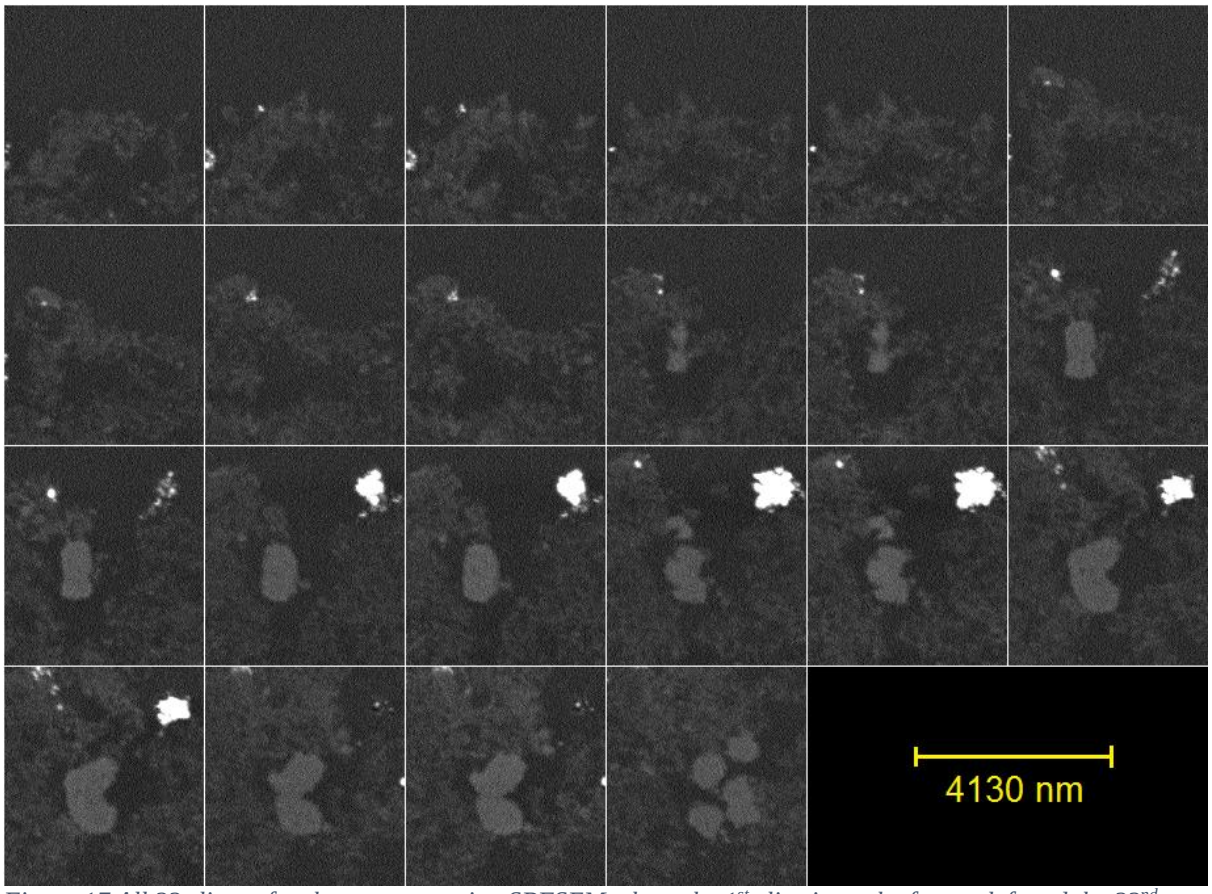


Figure 17 All 22 slices of a chromosome using SBFSEM where the 1st slice is on the far top left and the 22nd slice is the last image on the fourth row. The first sign of the chromosome is on slice 10. There were other signals in the images including a very bright object from slice 14 to slice 19. There was also another portion of a chromosome in slice 22.

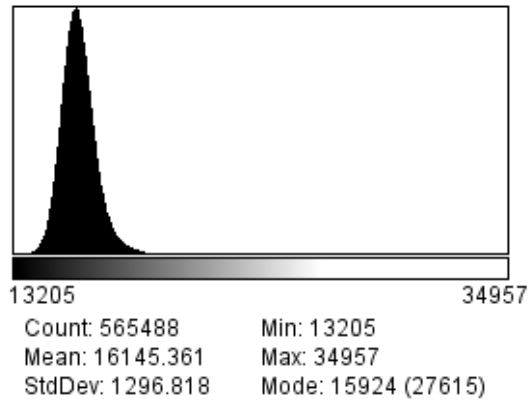


Figure 18 The histogram of all the intensity values of all slices. (x-axis is the intensity values and the y-axis is the frequency.)

Outside the context of this project, it is meaningless to have 16-bit grayscale as 8-bit grayscale (256 possible intensity values) is sufficient enough to cover the full range of grayscale a human eye can distinguish. [20, p. 118] However a 16-bit grayscale image in this project was extremely useful as each pixel contained more possible intensity values, and therefore more information, however the image will need more memory space to be stored. [18, p. 17]

The histogram in Figure 18 shows the distribution of each pixel intensity values in the stack. By inspection, the histogram looked very Normally distributed however a closer look showed there was a slight positive skewness in the distribution.

It was hypothesised that the skewness was the result of many lower intensity pixels in the background compared to the not many higher intensity pixels in the chromosome. Using the ImageJ histogram function, N = 10 samples of chromosome and background

intensity values were taken extracting the count n_i, sample mean $\bar{x}_i$ and the sample standard deviation s_i of the intensity values, as shown in Figure 19.

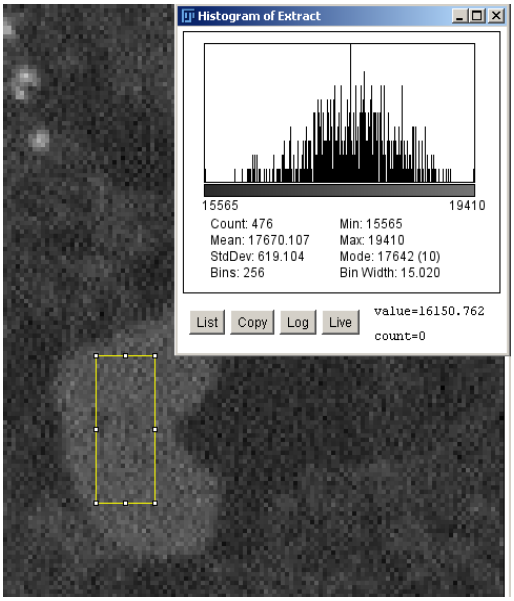


Figure 19 Using ImageJ histogram function to extract the count, sample mean and sample standard deviation of the intensity values in the selected area (yellow).

Equation 1

$$\bar{x} = \frac{\sum_{i=0}^{N} n_i \bar{x}_i}{\sum_{i=0}^{N} n_i}$$

Equation 2

$$s = \sqrt{\frac{\sum_{i=0}^{N} (n_i - 1) s_i^2}{(\sum_{i=0}^{N} n_i) - N}}$$

Equation 3

$$\bar{x} - \frac{\Phi^{-1}(97.5\%)s}{\sqrt{n}} < \mu < \bar{x} + \frac{\Phi^{-1}(97.5\%)s}{\sqrt{n}}$$

Equation 4

$$\frac{\nu s^2}{\sqrt{2\nu}\Phi^{-1}(97.5\%) + \nu} < \sigma^2 < \frac{\nu s^2}{\sqrt{2\nu}\Phi^{-1}(2.5\%) + \nu}$$

where $\nu = (\sum_{i=0}^{N} n_i) - N$

The population mean and population standard deviation, μ and σ respectively, were estimated using the estimators shown in Equation 1 and Equation 2. [21, p. 195] [22, p. 194]

Because the Normal distribution is a good approximation to the t_υ and χ_υ^2 distribution for large degrees of freedom υ, the 95% confidence interval of μ and σ were worked out using Equation 3 and Equation 4 respectively, where Φ^{-1} is the inverse cumulative distribution function of a standard Normal distribution. [21, pp. 195-198] [22, p. 194]

The estimated population mean and population standard deviation with their respective 95% confidence limits are shown in Figure 20.

	Chromosome	Background
Mean	17804 ± 24	16059 ± 5
Standard Deviation	647 ± 17	676 ± 4

Figure 20 95% confidence limits of the population mean and standard deviation chromosome and background intensity values.

Figure 20 shows more than sufficient evidence that the mean intensity for the background and chromosome were significantly different. This suggested that there are two distinct signals contributing to the histogram, such that the positive skewness in the histogram was due to the intensity values of the chromosome with mean 17804 and standard deviation 647.

A threshold interval was set centred at 17804, which was the estimated population mean of the chromosome intensity value, with a variable width. The events T, B and C were defined to be:

- T = The pixel intensity value is in the threshold interval

- C = The pixel represent the chromosome
- B = The pixel represent the background

Following from these definitions of these events, and assuming the intensity values of the background and chromosome pixels were Normally distributed with parameters from their estimations in Figure 20, the probability of a chromosome pixel or background pixel will be in a threshold interval, P(T|C) and P(T|B) respectively, were calculated as shown in Figure 21 and plotted in Figure 22.

P(T|C)	Threshold lower limit	Threshold upper limit	P(T|B)
70%	17133	18475	5.6%
75%	17059	18548	6.9%
80%	16974	18633	8.8%
85%	16872	18736	11.5%
90%	16739	18869	15.7%
95%	16535	19073	24.1%
99%	16136	19471	45.5%

Figure 21 P(T|C) and P(T|B) as the threshold centred at 17804 varies its width.

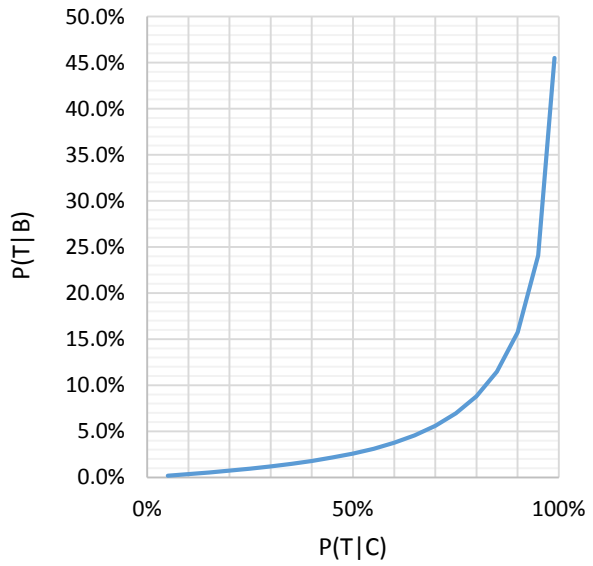


Figure 22 P(T|B) plotted against P(T|C).

Figure 21 showed that increasing P(T|C) will also increase P(T|B). It was a trade-off between obtaining as many chromosome pixels in the threshold as possible and obtaining as least background pixels in the threshold as possible. The threshold was set at 16974 - 18633 because P(T|B) was under 10% while P(T|C) is at 80%.

The assumption that the distribution of the intensity values, for the chromosome and background, are Normally distribution was a good assumption, but unjustified because the histogram looked Normal but not tested to see if the data fitted with the Normal distribution. A χ^2 goodness of fit hypothesis test could be conducted to test this assumption [23, p. 103] however it would be fruitless. The Normal distribution was only used to estimate the proportion of background or chromosome pixels which will be in the threshold interval.

A more useful statistics would be the portion of pixels in the threshold which are background pixels P(B|T) rather than P(T|B) . Bayes' theorem could be used to calculate P(B|T) as shown in Equation 5. [21, p. 20]

Equation 5

$$P(B|T) = \frac{P(B)P(T|B)}{P(T)}$$

The main problem with Equation 5 was that P(B), the proportion of pixels which are background in the stack, was needed. P(B) was estimated later on after analysing the threshold images. The threshold images are as shown in Figure 23. As expected, there were a few background pixels in the images. But by using the despeckle function and some manual editing in ImageJ, these were removed and the binary stack of the chromosome were restored, as shown in Figure 24.

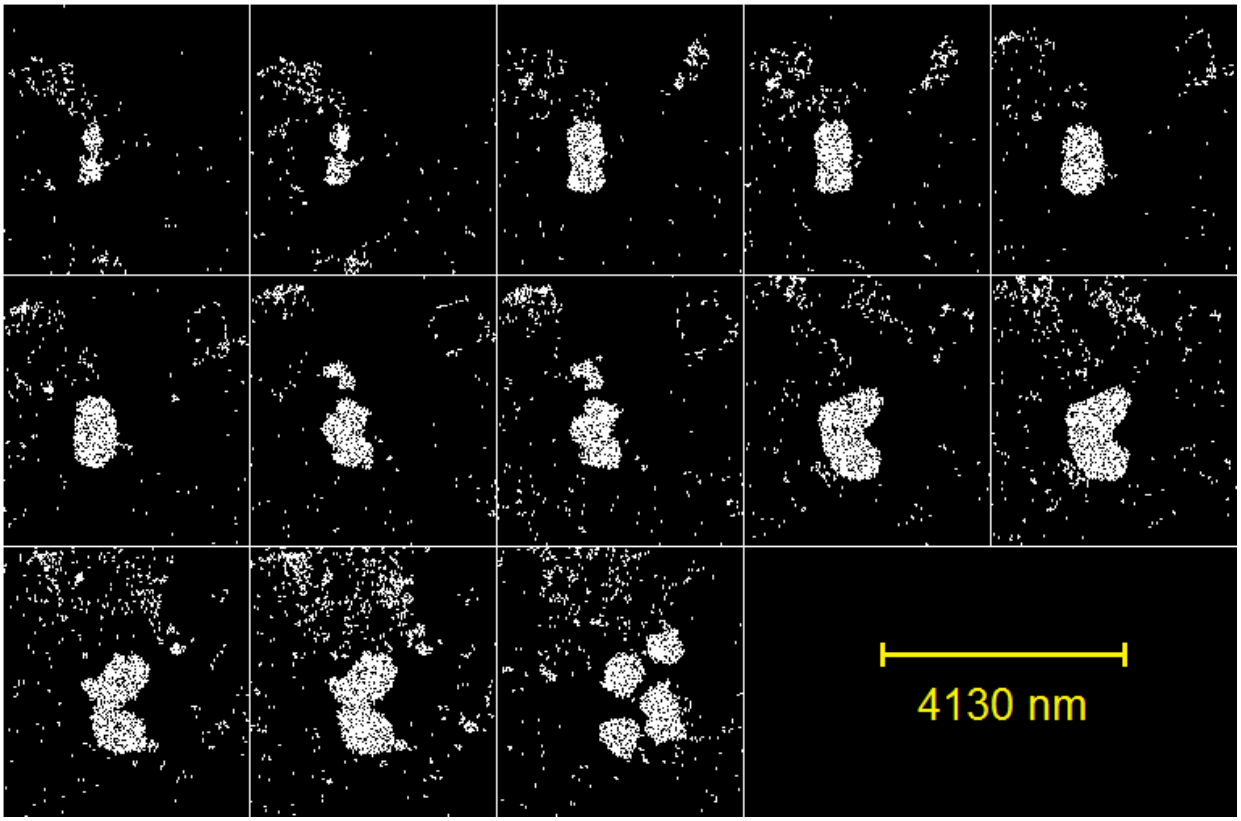


Figure 23 The result of a threshold with the interval 16974-18633.

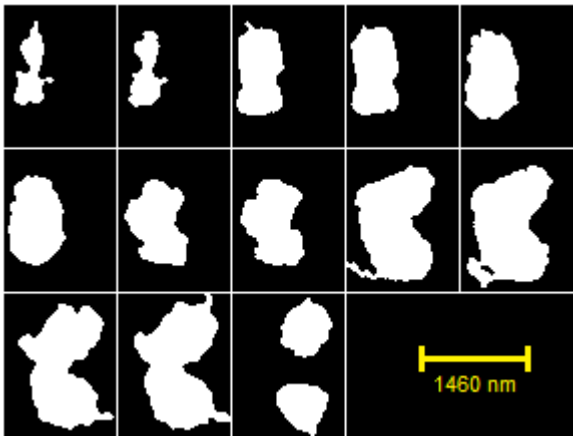


Figure 24 The result of the despeckle function. The slices were also cropped to only display the chromosome.

By treating each threshold pixel as a cube with dimensions 27 nm x 27 nm x 100 nm, the volume of the chromosome was estimated to be about 2.84 μm^3.

P(T|C)	P(T|B)	P(T)	P(B|T)
70%	5.6%	7.1%	5.4%
75%	6.9%	8.4%	5.7%
80%	8.8%	10.0%	6.1%
85%	11.5%	12.3%	6.5%
90%	15.7%	15.9%	6.8%
95%	24.1%	23.0%	7.2%
99%	45.5%	42.2%	7.4%

Figure 25 The proportion of threshold pixels which are background pixels P(B|T) were worked out for each P(T|C)

From the binary stack, P(B) was estimated to be 6.9% and this was used together with Equation 5 to work out P(B|T) as shown in Figure 25. For the chosen threshold interval, P(B|T) was worked out to be 6.1%. This meant 6.1% of the threshold pixel will be background and this seemed to be a reasonable result.

These threshold images were then used to produce a 3D render of the chromosome using voxels. A voxel is a 3D equivalent of a pixel, such that a voxel is a cube instead of a square and a 3D object can be made up of many voxels. [24]

Firstly using Processing, each binary slices were analysed and the intensity values were stored in a 3D Boolean array, as shown in Figure 26. Then for each `true` value in the 3D Boolean array, a cube was drawn at the corresponding position as shown in Figure 27. The resulting render is as shown in Figure 30.

The first problem was that the program had a slow frame rate, of about 30 frames per second (fps). The main cause for such a slow program was because too often too many objects were being drawn.

In the 3D render of the chromosome, only voxels on the surface of the chromosome could be seen. Therefore voxels drawn inside the chromosome were meaningless because they cannot be seen. An algorithm was written to remove voxels with 6 connecting neighbouring voxels as shown in Figure 28. Because these hidden voxels were removed, the computer drawn less voxels and yet produced the same result.

At every frame, the program had to calculate the geometry of each voxel for it to be drawn. [25] This kind of algorithm is called immediate mode. [25] A faster way to render the chromosome was to work out the geometry of each voxel once and save it in memory, called a Vertex Buffer Object in lower-level OpenGL syntax, and draw it from memory at each frame without recalculating the geometry of each voxel. [25] This style of rendering is called retained mode and the code snippet is as shown in Figure 29. [25]

The final result resulted in an identical 3D render of the chromosome but at a frame rate of about 60 fps.

```
//for every pixel in the image
for (int i=0; i<imageArray.get(z).pixels.length; i++){
  if (imageArray.get(z).pixels[i]==color(255)){ //if that pixel is white
    pixelGrid[x][y][z] = true; //put true in the pixelGrid
  }//end if
  else{
    pixelGrid[x][y][z] = false; //else put false
  }//end else

  //increase x and y co-od accordingly
  x++;
  if (x==imageWidth){
    x=0;
    y++;
  }//end if
}//end for
```

Figure 26 A code snippet which analyses the pixels on a slice. It puts a `true` value for white pixels and a `false` value otherwise in a 3D Boolean array called `pixelGrid`.

```
//for every slice
for (int z=0; z<nSlices; z++){
 //for every pixel
 for (int y=0; y<imageHeight; y++){
   for (int x=0; x<imageWidth; x++){
     if (pixelGrid[x][y][z]==true){ //if the pixelGrid contain true
       fill(0); //set fill to black
       stroke(0,255,0); //set stroke to green
       //draw box at corresponding co-od
       pushMatrix();
       translate(5*(x-imageWidth/2),5*(z-nSlices/2),5*(y-imageHeight/2));
       box(5);
       popMatrix();
     }//end if
   }//end for
 }//end for
}//end for
```

Figure 27 A code snippet which goes through all entries in `pixelGrid` and draws a voxel for every `true` entry.

```
//for each slice
for (int z=0; z<nSlices; z++){
  //for each pixel in the slice
  for (int y=0; y<imageHeight; y++){
    for (int x=0; x<imageWidth; x++){

      //if the pixel is on the border, the pixel remains unchanged
      if(x==0||x==imageWidth-1||y==0||y==imageHeight-1||z==0||z==nSlices-1){
        pixelGridHollow[x][y][z] = pixelGrid[x][y][z];
      }//end if

      else if(pixelGrid[x][y][z]==true){ //if the pixel is true
        //if the pixel has 6 true neighbouring pixels
        if(pixelGrid[x-1][y][z]==true&&pixelGrid[x+1][y][z]==true
        &&pixelGrid[x][y-1][z]==true && pixelGrid[x][y+1][z]==true
        &&pixelGrid[x][y][z-1]==true && pixelGrid[x][y][z+1]==true){
          pixelGridHollow[x][y][z]=false; //then that pixel shall be false
        }//end if
        else{
          pixelGridHollow[x][y][z]=true; //else the pixel will remain true
        }//end else
      }//end if

      else{
        //if the pixel is not true, it will be false
        pixelGridHollow[x][y][z]=false;
      }//end else

    }//end for
  }//end for
}//end for
```

Figure 28 A code snippet which goes through all the entries in `pixelGrid` and saves them in another 3D Boolean array `pixelGridHollow` but `true` values with 6 neighbouring `true` values were converted to `false`.

```
//chromosome is a group of voxel
chromosome = createShape(GROUP);

//for every slice
for (int z=0; z<nSlices; z++){
  //for every pixel
  for (int y=0; y<imageHeight; y++){
    for (int x=0; x<imageWidth; x++){

      //if the pixelGrid contain true save box at corresponding co-od

      if (pixelGridHollow[x][y][z]==true){
        PShape voxel;
        voxel = createShape(BOX,5);
        voxel.setFill(color(0)); //fill black
        voxel.setStroke(color(0,255,0)); //stroke green
        //translate voxel
        voxel.translate(5*(x-imageWidth/2),5*(z-nSlices/2),5*(y-imageHeight/2));
        chromosome.addChild(voxel); //add voxel to chromosome
      }//end if
    }//end for
  }//end for
}//end for
```

Figure 29 (14) A code snippet which goes through all entries in `pixelGridHollow` but saves a voxel as a `PShape`, instead of drawing it, for every `true` entry.

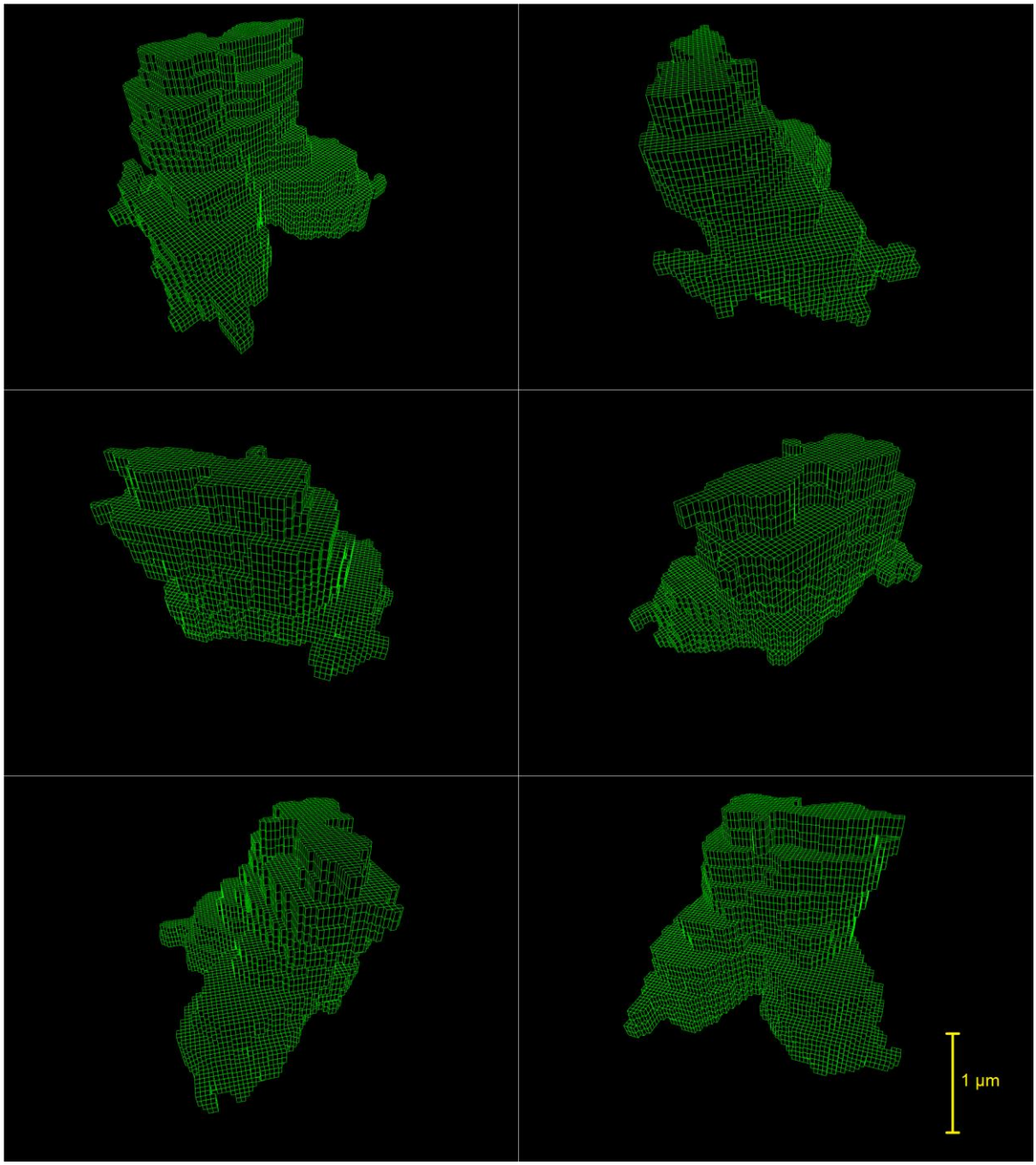


Figure 30 A 3D voxel render of a chromosome using voxels.

Voxels do produce a good 3D render of the chromosome however it was very crude to approximate a chromosome into many cubes. More work was done to obtain more details about the shape of the chromosome.

One way was to use the marching cubes algorithm. This algorithm approximates voxels into a triangular mesh and this produced a more detailed shape for the chromosome. [24] [26]

The marching cubes algorithm works by analysing 4 neighbouring voxels at a time, throughout the binary stack, and draws a polygon according to the values of the 4 voxels. [24] [26] Because each voxel can only have 2 possible values, there are a possible of 256 possible polygons to be drawn as shown in Figure 31. [24] [26]

It was more convenient and more organized to start using object ordinated programming to program the marching cubes algorithm. Figure 33 shows the class definitions of the 'marching cube,' called `Cube`, and of a polygon to be drawn, called `Triangle`, with their corresponding member variables and methods.

Knowing which polygon to draw was tackled by using a lookup table, written by Paul Bourke (see [26]), containing the vertices for all 256 possible polygons, with a lookup index corresponding to an 8-bit number where each bit shows the Boolean value of the voxel, as shown in Figure 32. [26] The code snippet used for working out the lookup index and rendering the corresponding polygon is shown in Figure 34

Figure 37 and Figure 38 shows a wire frame and coloured filled render using the marching cubes algorithm respectively. The results are very good because the shape of the chromosome was more defined and less 'voxelated'.

The main problem was that the colour in the coloured filled render of the chromosome was very flat and many multi-coloured spotlights were used to make the render look as 3D as possible. This was because the normal for each polygon were not defined, therefore light interacted with each polygon in the same way, regardless of the orientation of each polygon.

The normals were calculated using the cross product between two vertices of the polygon as shown in the code snippet in Figure 35. This resulted in an image in Figure 39, which clearly shows the vertices and edges of the chromosome, making it look 3D. However because the normals were defined using the cross product, the normals were perpendicular to the surface of the polygon, instead to the original surface. [26] This created a very triangular and unsmooth render of the chromosome.

Bourke's suggested approach was to use a weighted average of normals of the polygons sharing its vertex, where the weight is the reciprocal of the area of the polygon. This meant smaller polygons have greater weight because they may occur in regions of high surface curvature. [26] The code snippet using Bourke's approach is shown in Figure 36.

By averaging the normals, Figure 40 was produced. The vertices and edges were certainly much smoother than before.

Figure 41 shows the centromere and one of the chromosome's chromatid labelled in the 3D render of the chromosome.

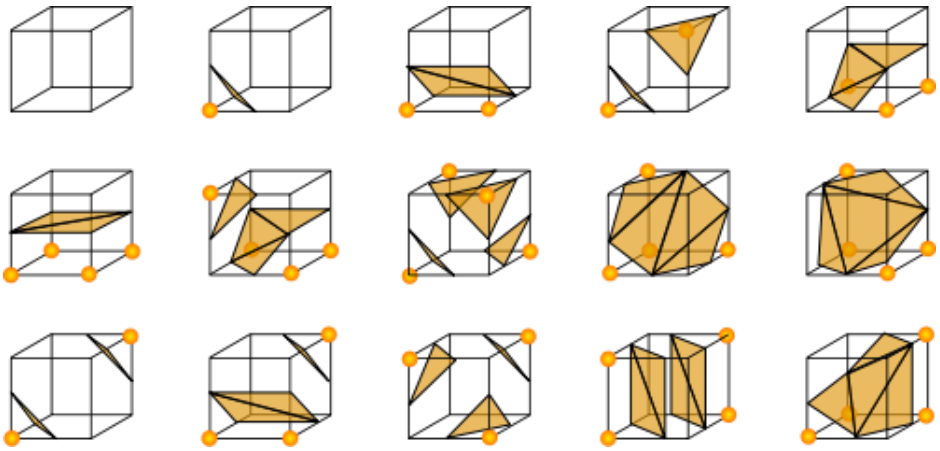


Figure 31 The 256 possible polygons which could be drawn in the marching cubes algorithm. These can be generalised into 15 polygons. The spheres represent a `true` value voxel in the binary stack of the chromosome. [27]

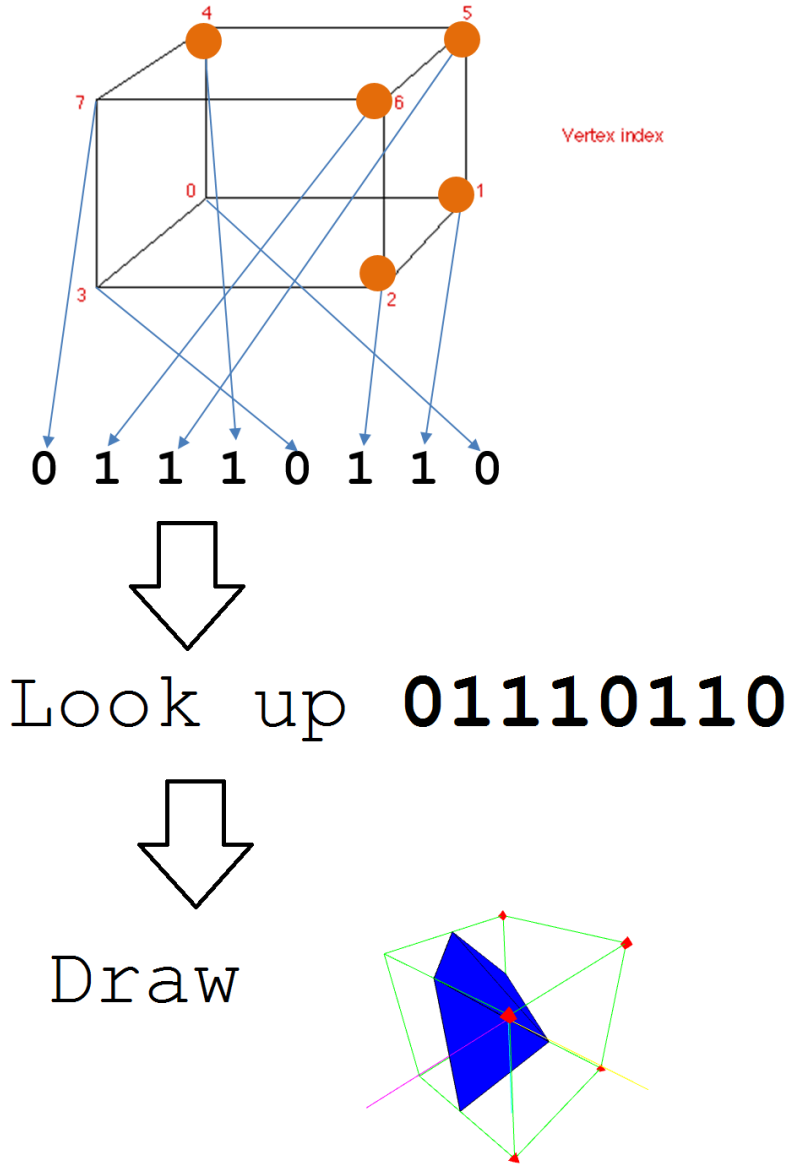


Figure 32 The Boolean value at each voxel was saved as an 8-bit binary number. This was then used to look up what polygon to draw. [26]

```
Class Cube{
   MEMBER VARIABLES
   • PVector position
         o Position vector of the centre of the marching cube
   • ArrayList <Triangle> triangleArray
         o Contains an arrayList of triangle objects in the marching
           cube
   CONSTRUCTOR ARGUMENTS
   o int x, int y, int z
         o position = new PVector(x,y,z);
   METHODS
   ➢ void addTriangles(verticesIndex)
         o Add a new triangle object, with vertices according to the
           verticesIndex, to triangleArray
   ➢ PShape getShape()
         o Returns the PShape of all the triangles in the marching cube
   ➢ ArrayList <Triangle> getTraingleArray (required for
     recalculateNormal())
         o Returns triangleArray
   ➢ void recalculateNormal ()
         o Recalculate the normals of all the triangles in the marching
           cube by taking the weighted average of the normals
}
Class Triangle{
   MEMBER VARIABLES
   • PVector a,b,c
         o Position vectors of the vertices of the triangle
   • PVector cube_centre
         o Position vector of the centre of the marching cube the
           triangle is in
   • PVector n
         o Normal vector to the surface of the triangle
   • PVector nBar
         o Weighted average normal
   • float area
         o The area of the triangle
   CONSTRUCTOR ARGUMENTS
   o verticesIndex, int x, int y, int z
         o Calculates a,b,c according to verticesIndex
         o Cube_centre = new PVector(x,y,z)
   METHODS
   ➢ PShape getShape()
         o Returns the PShape of the triangle
   ➢ PVector getNormal()
         o Returns n (required for recalculateNormal())
   ➢ void newNormal(PVector parameter)
         o Set nBar = parameter
   ➢ float getArea()
         o Returns area (required for recalculateNormal())
   ➢ PVector[] getVerticesVector()
         o Returns an array of position vectors of the 3 vertices of the
           triangle (required for recalculateNormal())
   ➢ boolean isShareVertex(Triangle that)
         o Returns true if this triangle shares at least one vertex with
           that triangle (required for recalculateNormal())
}
```

Figure 33 Class definitions of the marching cube, called `Cube`, and `Triangle`.

```
shape = createShape(GROUP);

//for every slice
for (int z=0; z<=nSlices; z++){

  //for every pixel in the slice
  for (int y=0; y<=imageHeight; y++){
    for (int x=0; x<=imageWidth; x++){

      //declare a variable called verticesIndex
      //this variable indicates which vertices has a true value…
      //…in the pixelGrid as an 8 bit binary number
      int verticesIndex = 0;
      //each digit in the binary number represent a vertex

      //work out verticesIndex by analysing each corner
      if(pixelGrid[x][y+1][z]==true){ //0
        verticesIndex += 1;
      }//end if
      if(pixelGrid[x+1][y+1][z]==true){ //1
        verticesIndex += 2;
      }//end if
      if(pixelGrid[x+1][y+1][z+1]==true){ //2
        verticesIndex += 4;
      }//end if
      if(pixelGrid[x][y+1][z+1]==true){ //3
        verticesIndex += 8;
      }//end if
      if(pixelGrid[x][y][z]==true){ //4
        verticesIndex += 16;
      }//end if
      if(pixelGrid[x+1][y][z]==true){ //5
        verticesIndex += 32;
      }//end if
      if(pixelGrid[x+1][y][z+1]==true){ //6
        verticesIndex += 64;
      }//end if
      if(pixelGrid[x][y][z+1]==true){ //7
        verticesIndex += 128;
      }//end if

      //look up what polygon to draw using vertices…
      //…and draw it at x,y,z
      drawPolygon(x,y,z,verticesIndex);

    }//end for
  }//end for
```

Figure 34 A code snippet used for working out a lookup index, called `verticesIndex`. This was then used to lookup what polygon to draw in the marching cube..

```
//CONSTRUCTOR
//Parameters: 3 edge indexes,
//and which edges the vertices of the triangle should be on
Triangle(int index1, int index2, int index3, float x, float y, float z){

  //convert the edge indexes to position vectors
  a = getEdgeCo(index1);
  b = getEdgeCo(index2);
  c = getEdgeCo(index3);

  //get the position vector of the center
  this.cube_center = new PVector(x,y,z);

  //work out the normal vector(n) and the area
  PVector r1,r2;
  r1 = PVector.sub(b,a);
  r2 = PVector.sub(b,c);
  n = r1.cross(r2);
  area = n.mag()/2;
  n.normalize();
  nBar = n; //nBar will be the averaged normal later on

  //default colouring
  wantStroke = false; //no stroke
  wantFill = true; //fill
  stroke = color(0,255,0); //green stroke
  fill = color(255); //white fill
}//end constructor
```

Figure 35 A code snippet, from the `Triangle` class constructor, used to work out and define the normal of the instantiated triangle.

```
//for each triangle in the cube
for (int i=0; i<triangleArray.size(); i++){
  //prepare this triangle and declare statististical variables
  Triangle triangle = triangleArray.get(i);
  PVector nBar; //average normal vector
  nBar = new PVector(0,0,0);
  float weight = 0; //the sum of all weights used for averaging

  //compare this triangle to the triangles in neighbouring cubes
  //for every neighbouring cube
  for (int u=xLower; u<=xUpper; u++){
    for (int v=yLower; v<=yUpper; v++){
      for (int w=zLower; w<=zUpper; w++){

      //get all the triangles in the neighbouring cube
      ArrayList<Triangle> thatTriangleArray =
cubeArray[x+u][y+v][z+w].getTriangleArray();

      //for every triangle in the cube
      for (int j=0; j<thatTriangleArray.size(); j++){
        Triangle that = thatTriangleArray.get(j);
        //if this triangle share a vertex with that triangle
        if(triangle.isShareVertex(that)){
          //update the normal statistics with the reciprocal area as the weight
          nBar.add(PVector.mult(that.getNormal(),1/that.getArea()));
          weight += 1/that.getArea();
        }//end if
      }//end for
    }//end for
  }//end for
}//end for
//work out the mean normal and update that triangle's normal
nBar.mult(1/weight);
triangle.newNormal(nBar);
```

Figure 36 A code snippet from the `calculateNormal()` method in the `Cube` class. It calculated the weighted averaged normal for each triangle in the instantiated cube.

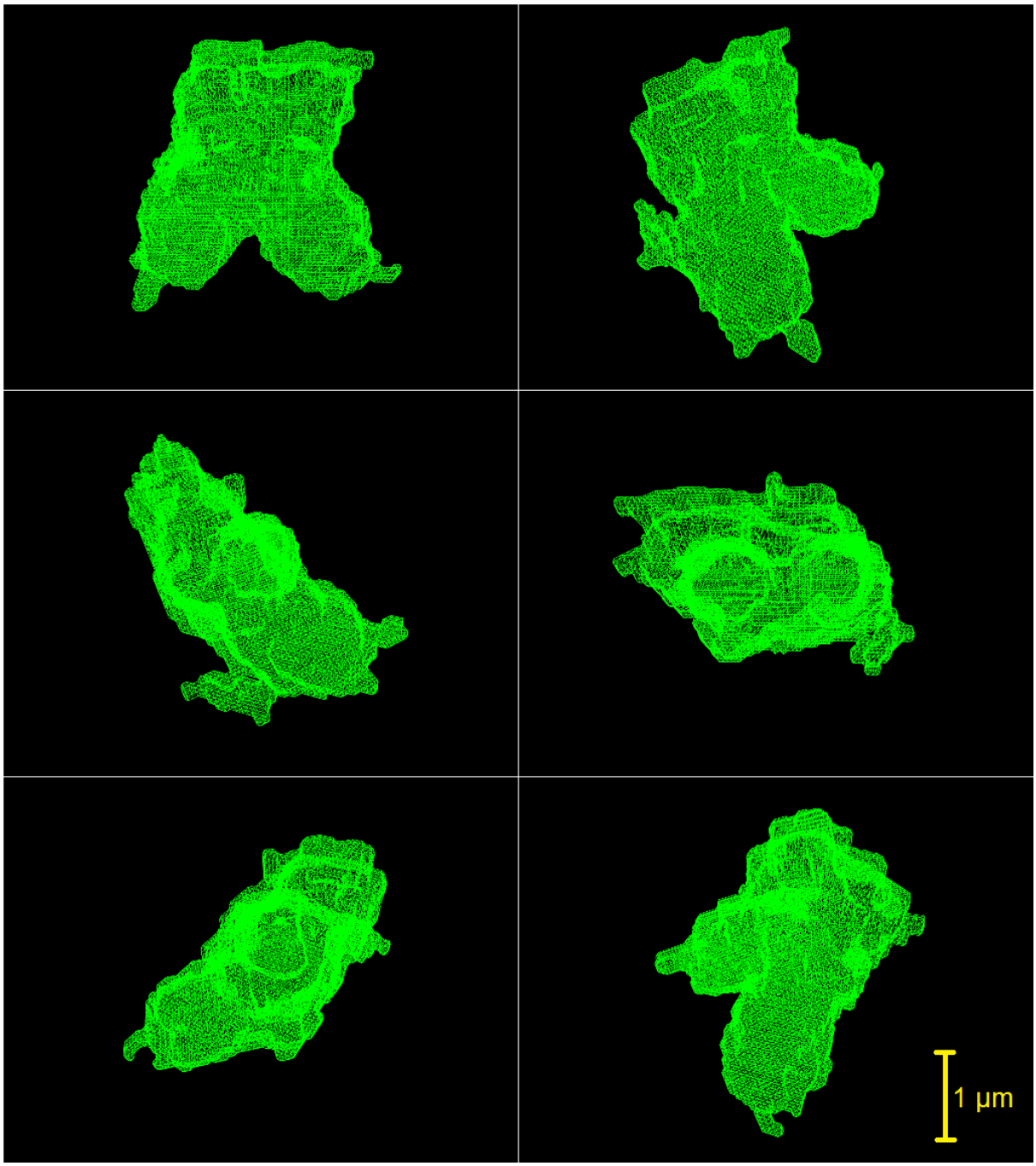


Figure 37 A wire frame render of the chromosome using the marching cubes algorithm.

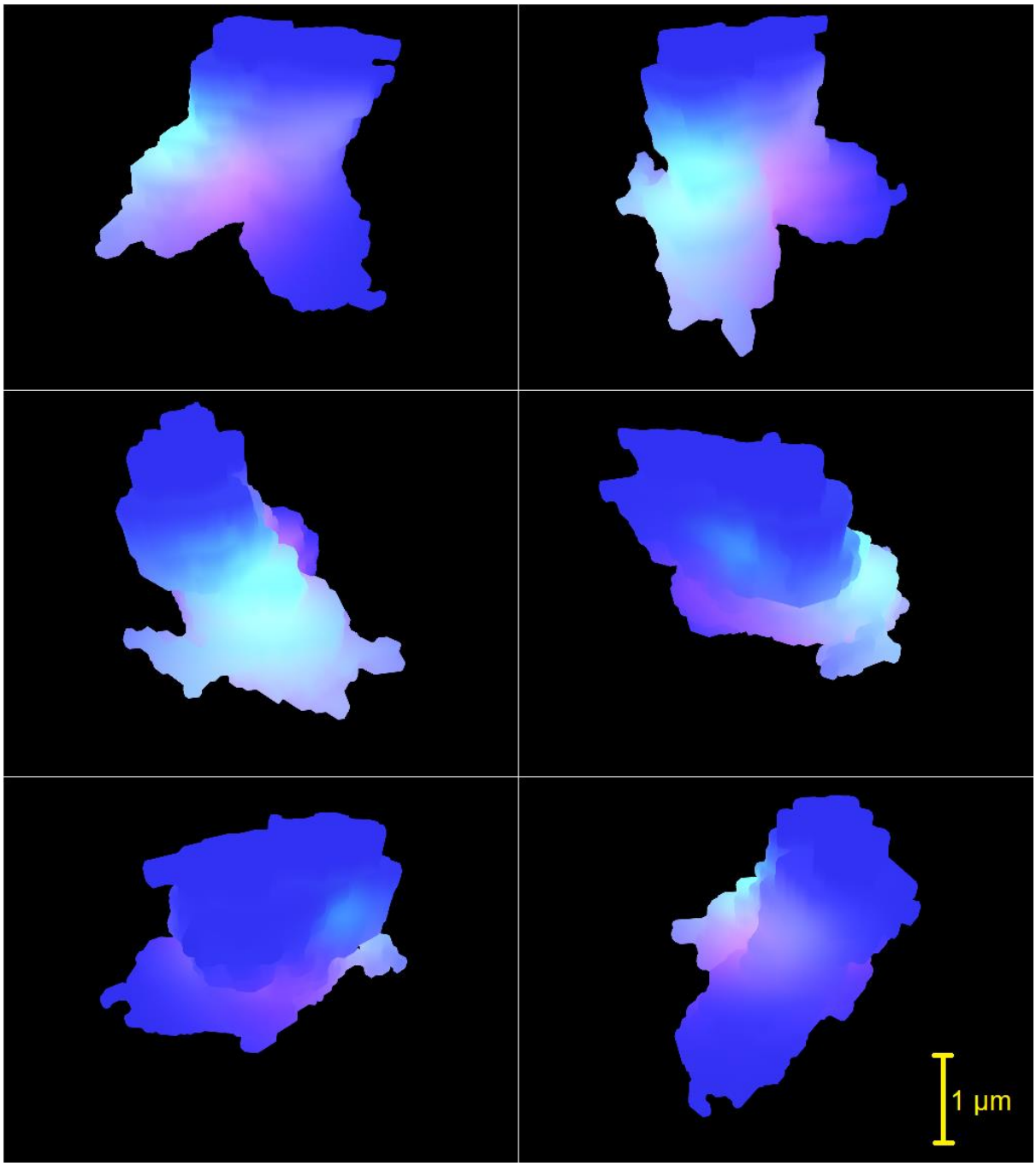


Figure 38 A render of the chromosome using the marching cubes algorthim with no defined normals.

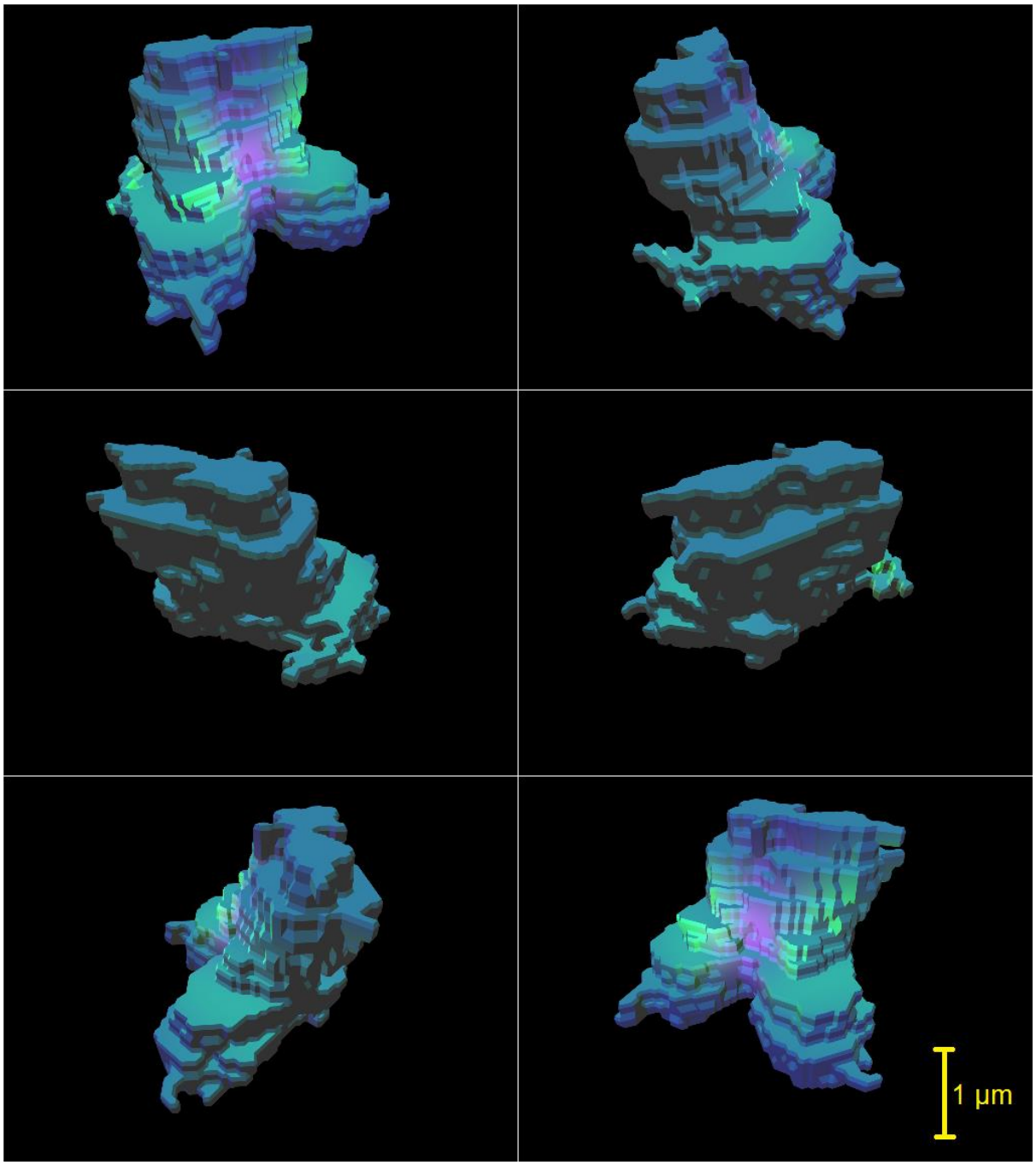


Figure 39 A render of the chromosome, using the marching cubes algorthim, with normals defined perpendicular to each triangle.

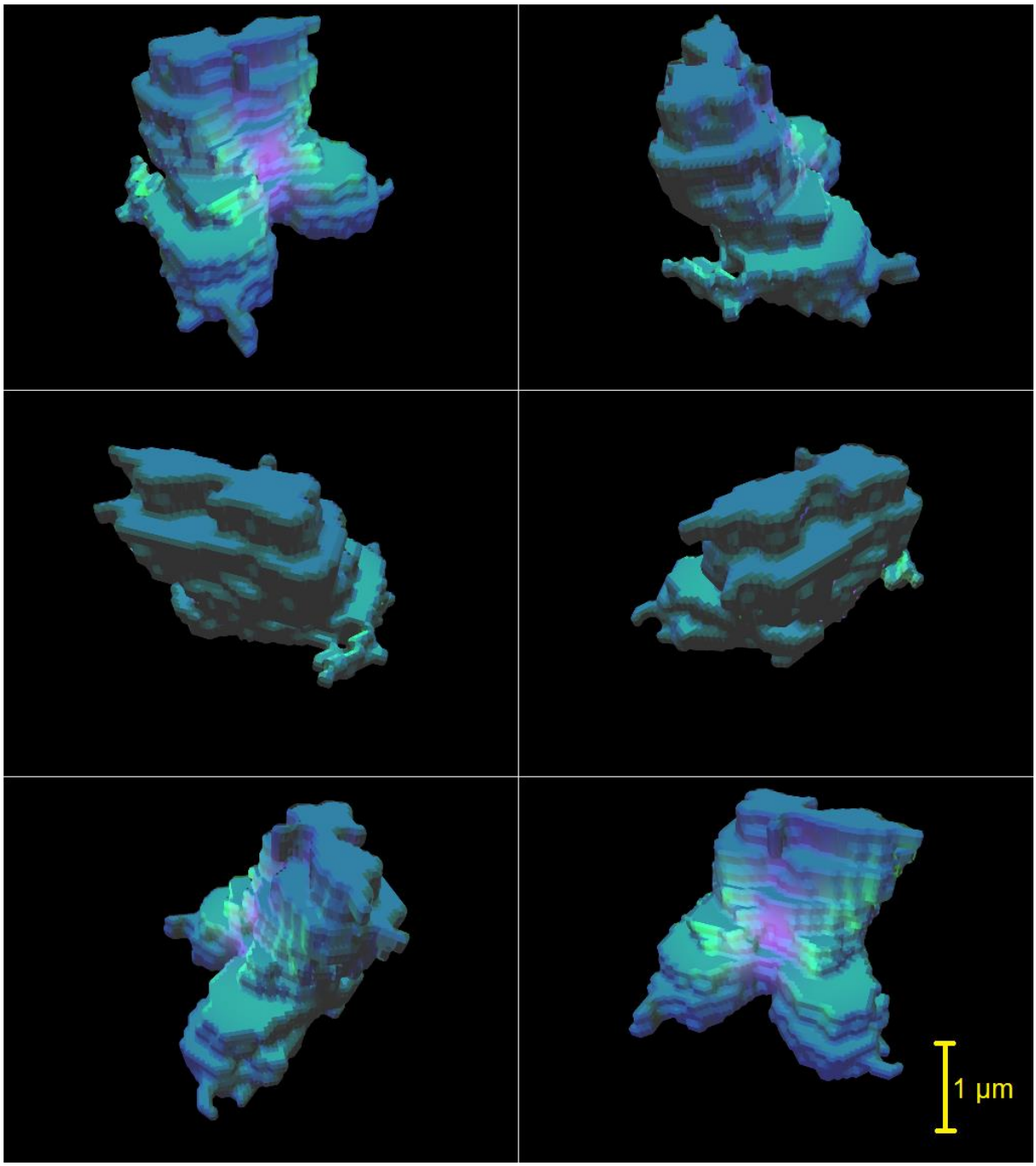


Figure 40 A render of the chromosome, using the marching cubes algorthim, with each normal defined as the weighted average normal for each polygon it shares it vertices with.

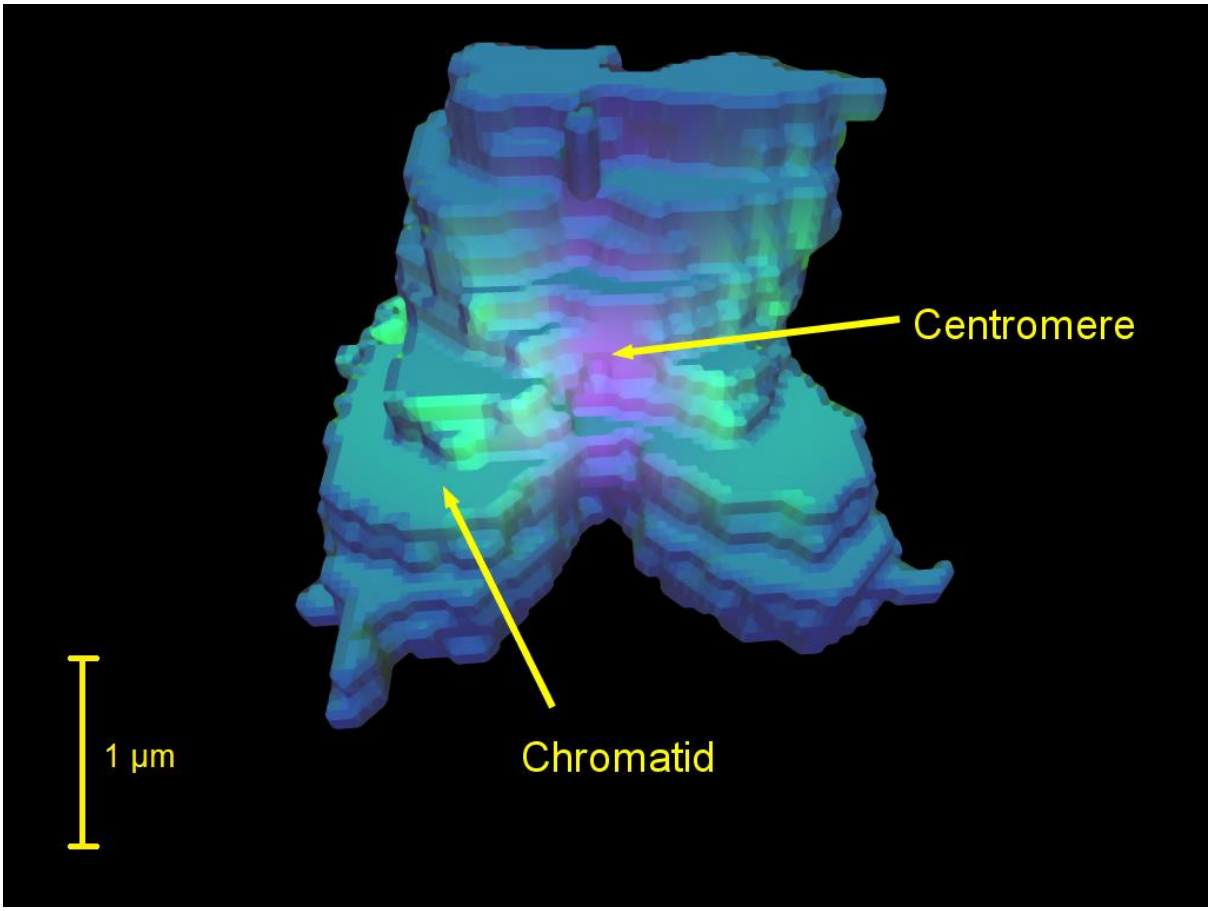


Figure 41 A render of the chromosome, using the marching cubes algorthim, with the centromere and one of its chromatid identified.

IV. ANALYSIS OF INTERNAL STUCTURES

Another stack of SBFSEM slices of a chromosome was worked on in the project, as shown in Figure 44. The stack had 122 slices which had dimensions of 266 x 314 pixels and, again, had a 16-bit grayscale colour depth. Each pixels represent a scale of 11 nm x 11nm and the slice separation is 20 nm. Platinum blue was used to dye the chromosome, embedded in an epoxy resin, which meant the chromosome will show up with low intensity in the obtained images as shown in Figure 44.

It was noted that a few slices at the beginning of the stack were distorted, as shown in Figure 44 by a red arrow. It was also noted there were noticeable consistent lines in the orthogonal views of the stack, as shown in Figure 45, and it looked as though each imaging of each surface took up more than just one slice. This may be the result of alignment issues during sectioning of the specimen using a diamond knife. However these problems were ignored because it would be extremely difficult to remove such distortions and alignment issues.

Figure 42 shows the histogram of all the intensities in the stack. From inspection, the distribution was very negatively skewed with 3 maximas. This suggested there may be 3, not necessary Normally distributed, random variables contributing to the overall distribution. There were 3 distinct signals in the stack: the background, the chromosome and the internal & external features of the chromosome as shown in Figure 46.

A sample of the background, chromosome and the internal features intensities were taken. Their population mean and population standard deviation were estimated as shown in Figure 43.

Figure 43 showed more than sufficient evidence that the 3 objects' mean intensity values were significantly different. Their estimated means were also at least 4 standard deviations away from each other, making them even more significantly different.

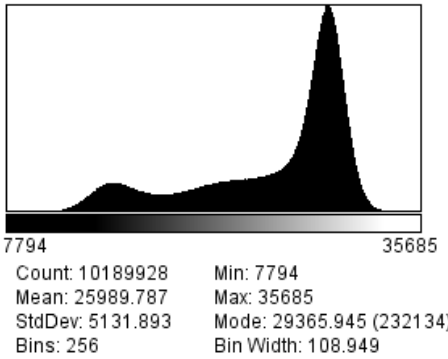


Figure 42 The distribution of all the intensity values in the stack. (x-axis is the intensities and the y-axis is the frequency)

	Chromosome	Internal features	Background
Mean	13872 ± 31	20402 ± 97	29410 ± 9
Standard deviation	976 ± 22	1630 ± 69	868 ± 7

Figure 43 95% confidence limits of the mean and standard deviation of the chromosome, internal features and background intensities.

The threshold interval were chosen to be 2 standard deviations away from the mean of the chromosome and the internal features intensity values. As a result, the threshold was set at 11920 - 23662, to obtain the external shape of the chromosome, results in Figure 47.

After threshold, it was noticed that quite a few features in the slices were fused together, causing them to look like one single object. Using the watershed function in ImageJ, these were separated as shown in Figure 52. The threshold binary stack was edited using the despeckle function to remove noise, the medium function to smooth the edges, and some manual editing in ImageJ, as shown in Figure 48.

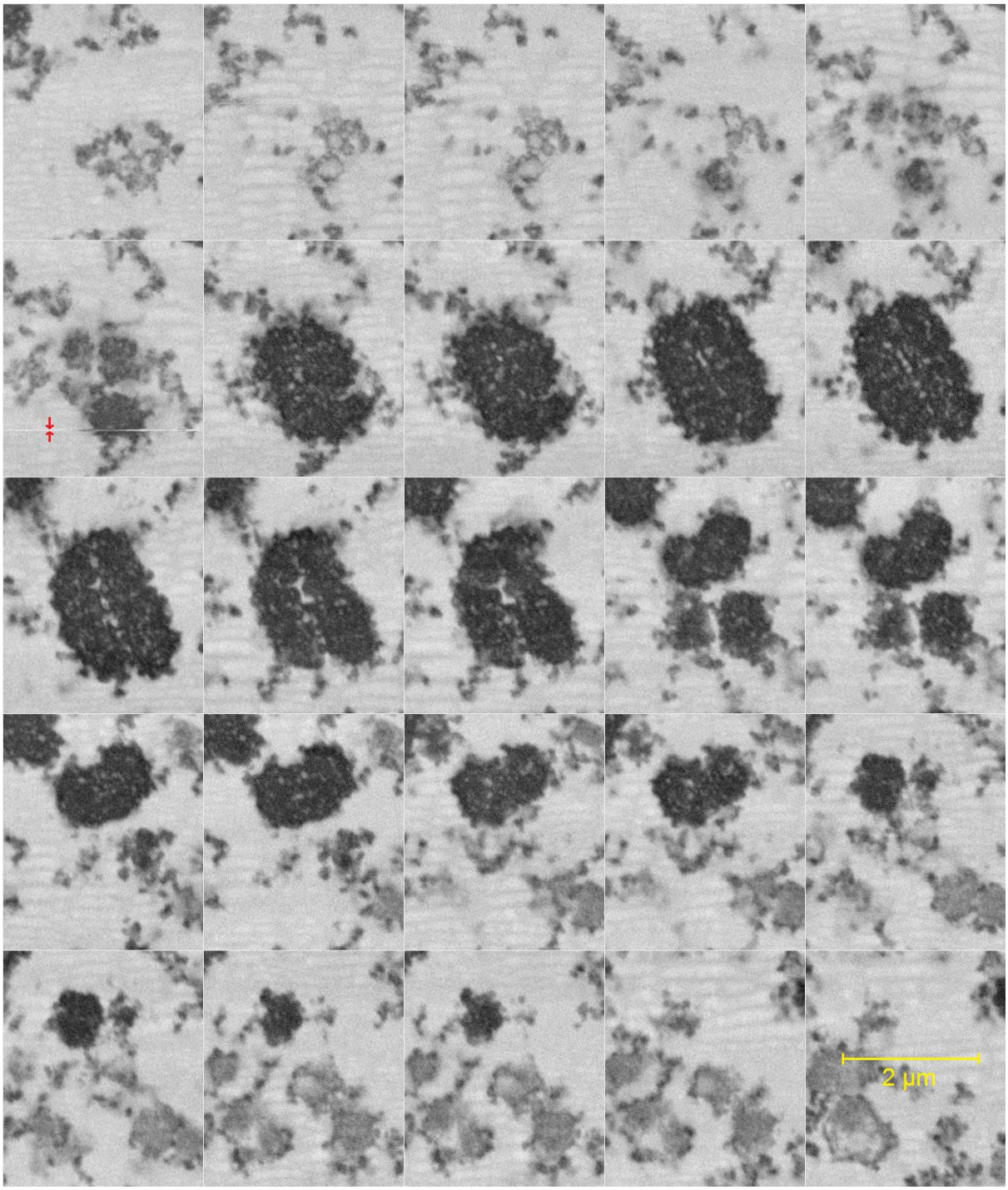


Figure 44 Slices 1,6,11,…,121 of a human chromosome taken using SBFSEM. A red arrow on slice 26 points to a distortion.

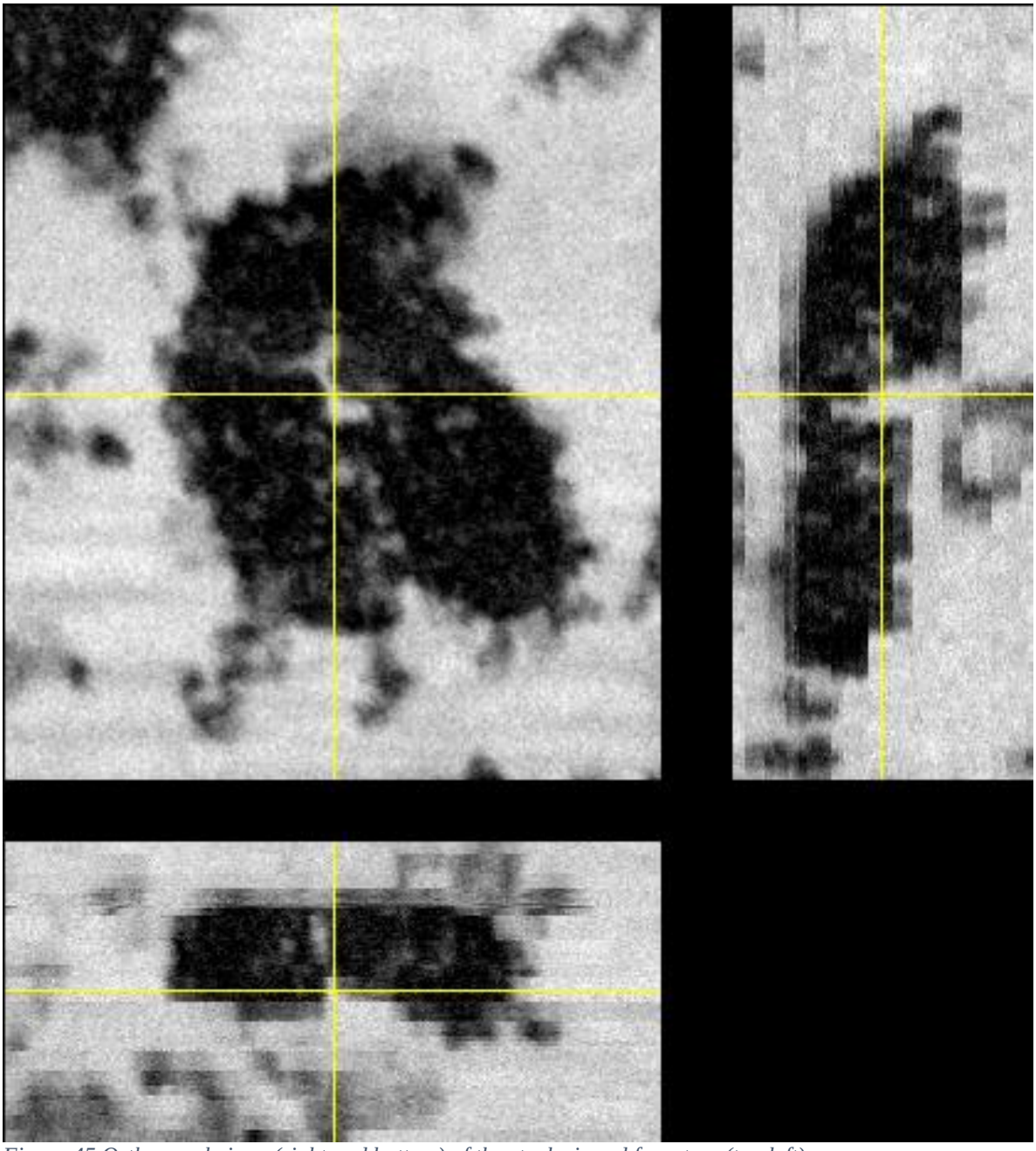


Figure 45 Orthogonal views (right and bottom) of the stack viewed from top. (top left).

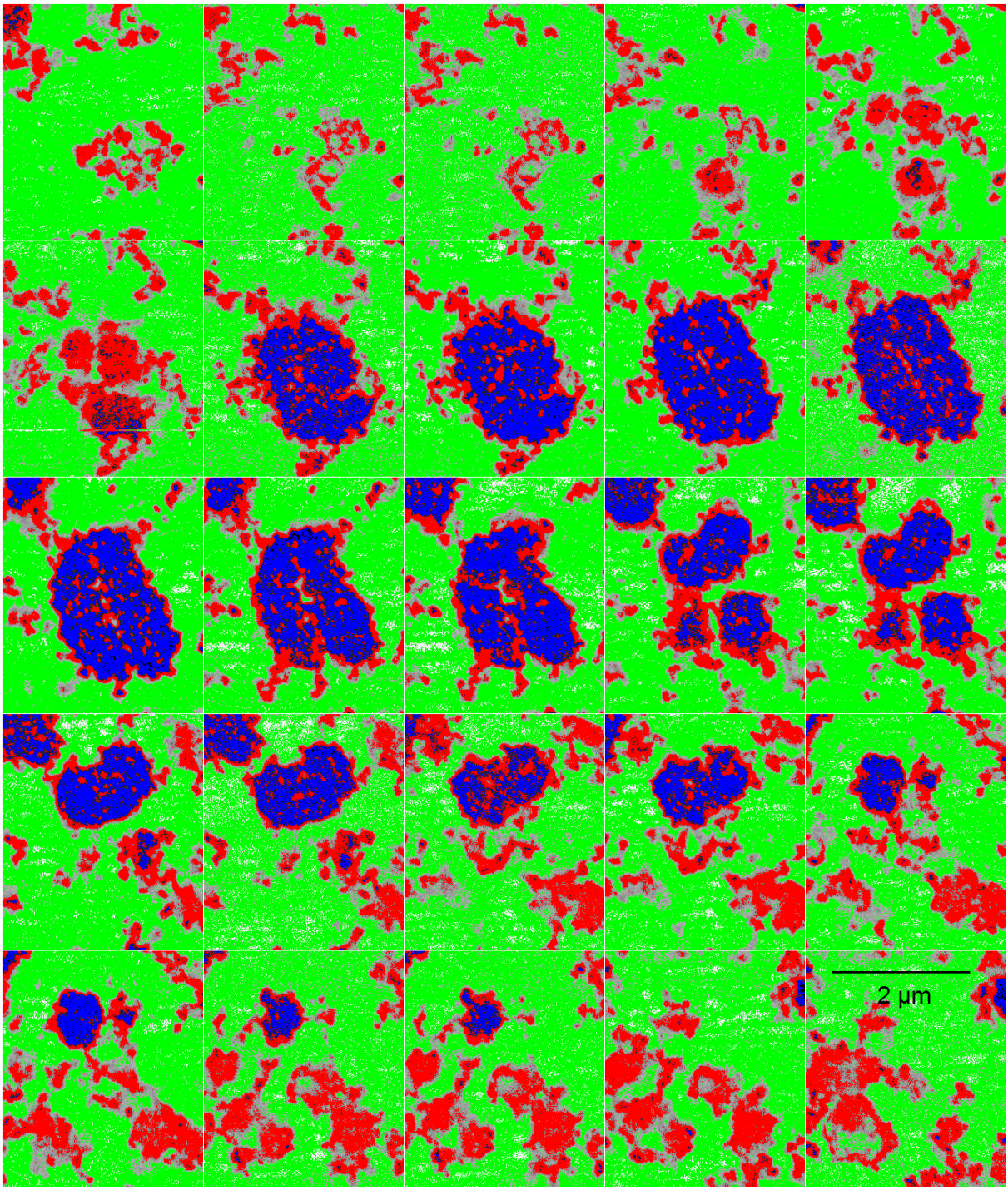


Figure 46 Three thresholds were applied to the stack to segment the chromosome (blue) and its features (red) and the background (green) with the following intensity values:

- *Blue: 11920 – 15824*
- *Red: 17142 – 23662*
- *Green: 27674 - 31146*

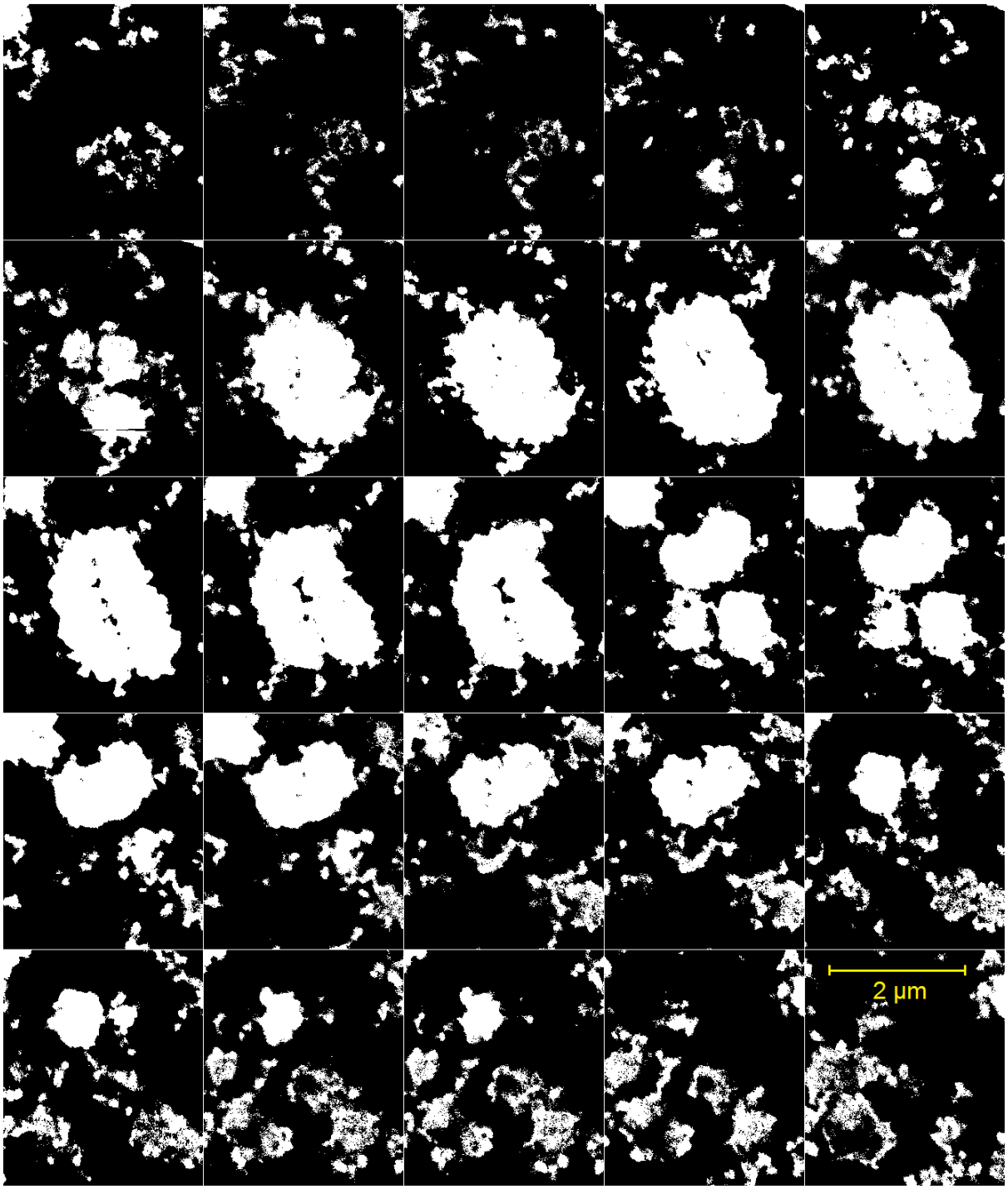


Figure 47 A binary stack as a result of a threshold set at 11920 – 23662 to segment the external shape of the chromosome.

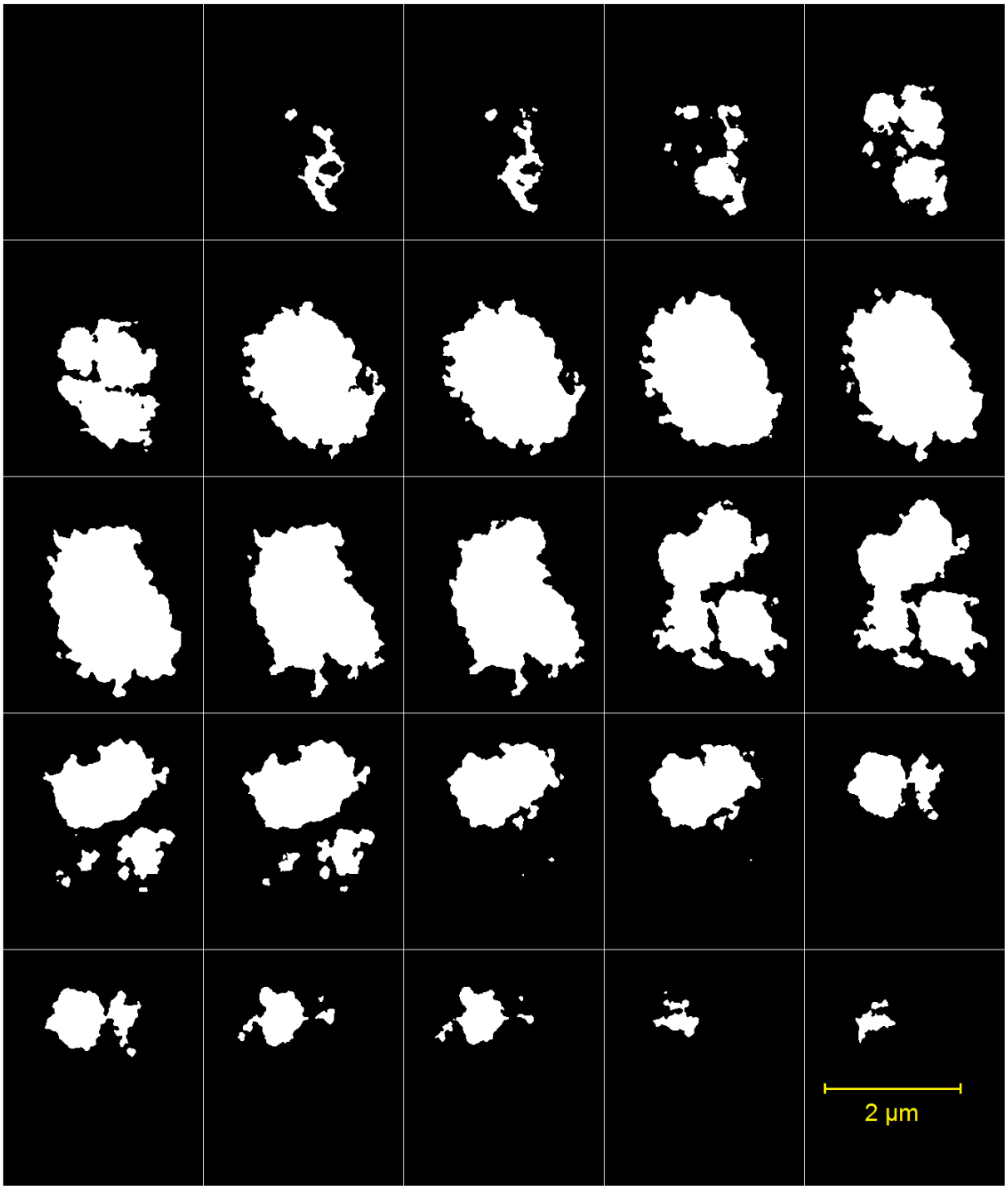


Figure 48 The binary stack in Figure 47 was manually edited using a medium filter, watershed algorthim and some manual editing.

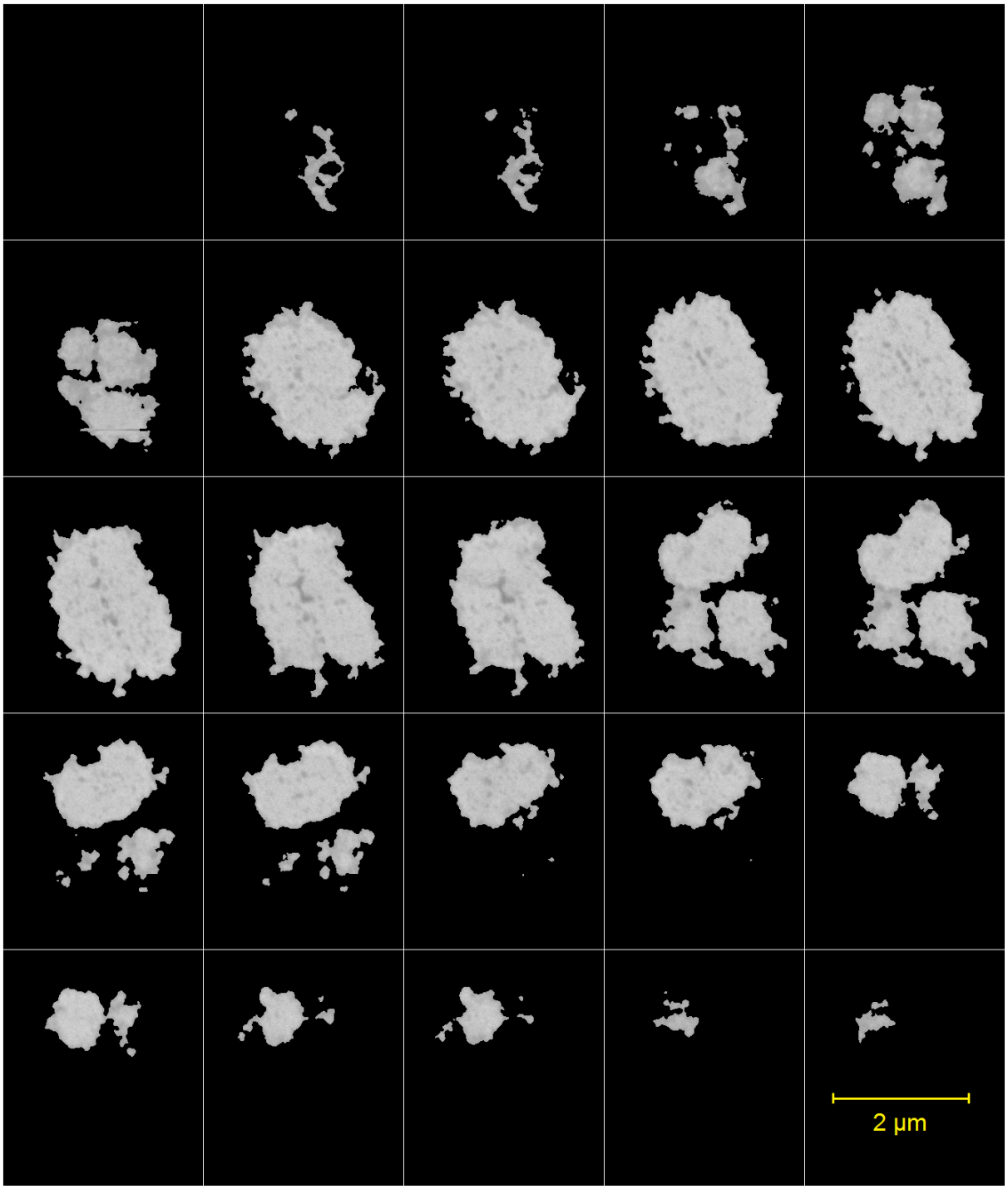


Figure 49 The binary stack in Figure 48 was used to crop out the chromosome intensity values from the original stack in Figure 44. The stack was also inverted so that the chromosome will show up as high intensities.

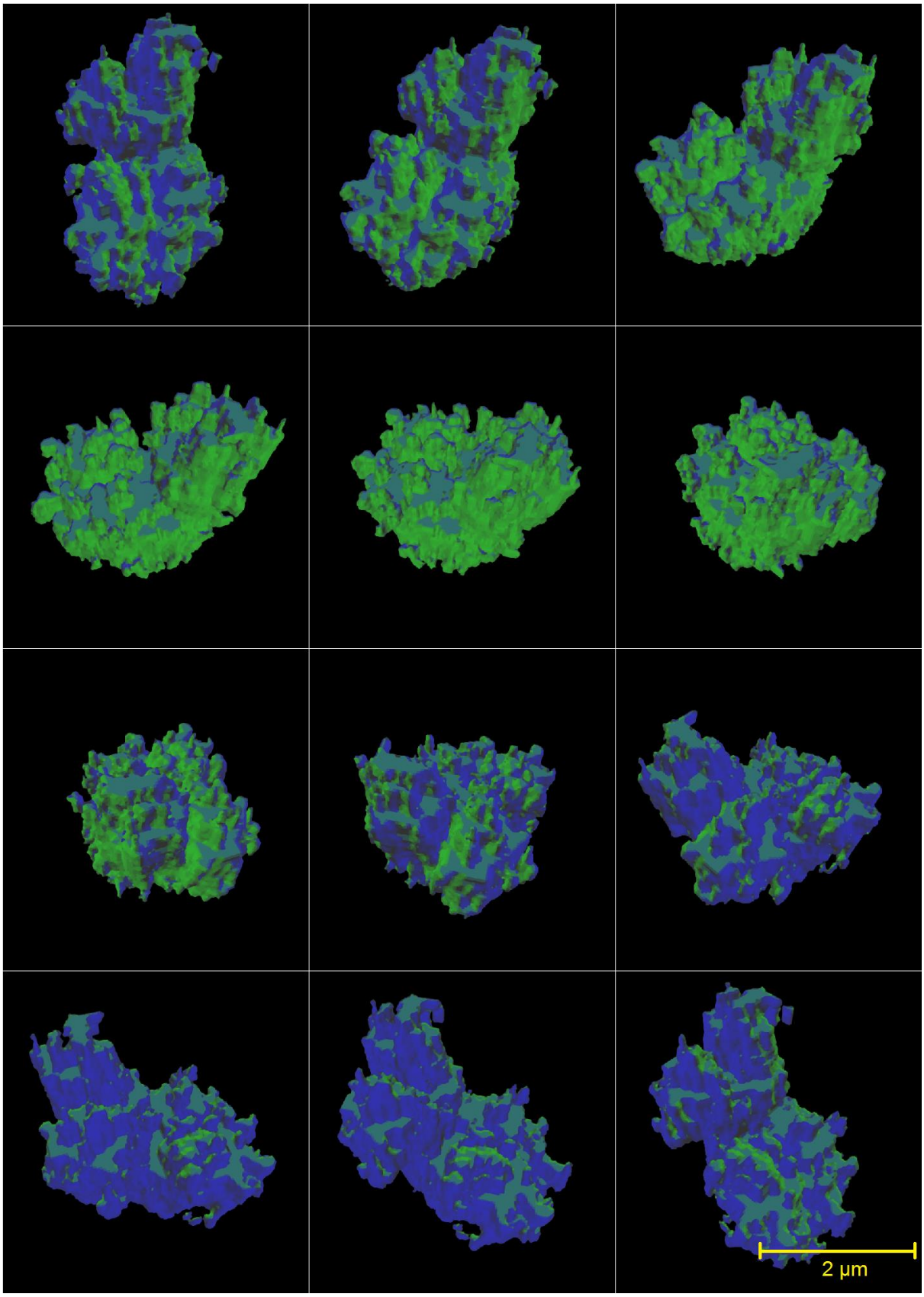


Figure 50 A 3D render of a human chromosome using the binary stack in Figure 48.

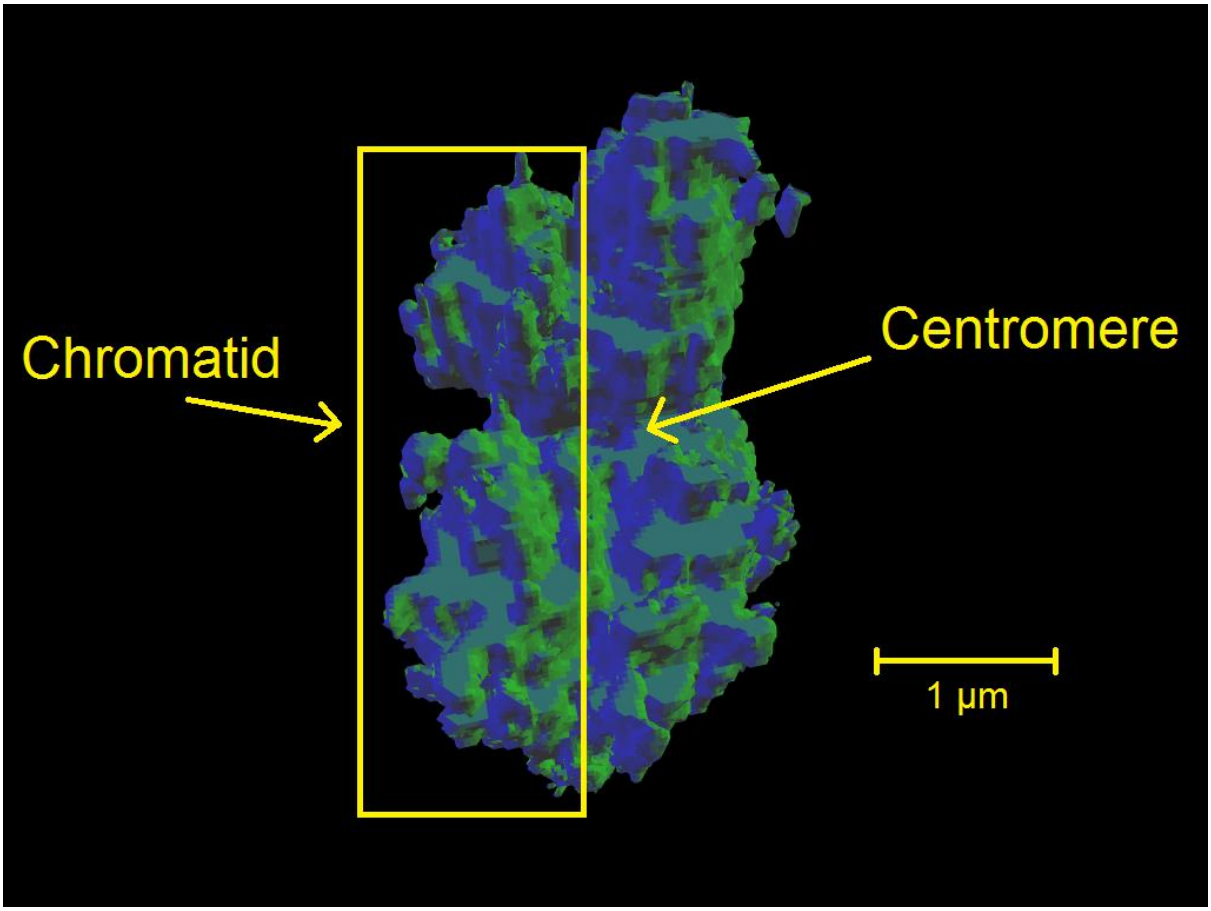


Figure 51 The centromere and chromatids were identified in this 3D render of a human chromosome.

Figure 52 The watershed algorithm was applied to the image on the left which resulted in an image on the right.

By using the binary stack in Figure 48, a 3D render of the chromosome was made using the marching cubes algorithm, as shown in Figure 50. The centromere and chromatid were easily identified, as shown in Figure 51. By summing the volume of all the threshold voxels, the volume of the chromosome was estimated to be about 3.95 μm^3. The binary stack was used to cut out the data with the shape of the chromosome as shown in Figure 49.

The scale of the slices was 11 nm per pixel and this unfortunately meant individual nucleosome, of diameter 11 nm, cannot be resolved because the scale is not small enough. It also meant the 30 nm chromatin fibre would be about 3 pixels thick in these images, which could be too small to resolve. However it may be possible to see structure, of size larger than the 30 nm chromatin fibre, in these images

ImageJ has a function called 'Find Maxima' which finds the positions of maximas in an image. [28] By using the 'Find Maxima' function on an inverted cropped stack, Figure 49, maximas were found which should represent parts of the chromosome which are dense. The noise tolerance, a value equal are greater than zero, could be adjusted such that maximas were only accounted for if its intensity is more than this value from the ridge to a higher maximum. [29]

As a test, slice 59 (Figure 53) was used to see what results the 'Find Maxima' function could produce. An ImageJ macro was written to see how the number of maximas found, using the 'Find Maxima' function, varied with noise tolerance. The results are shown in Figure 55.

By treating the number of maximas as a frequency, the noise tolerance was treated as a random variable with probability proportional to the number of maximas. As a result, the sample mean and sample standard deviation noise tolerance were calculated. These were used to estimate the shape and scale parameters of a Gamma distribution, using the method of moments, as shown in Equation 7 and Equation 8. [21, p. 263] These parameters were used to see if a Gamma distribution, with probability density function shown in Equation 6, could fit with the data as shown in Figure 55. [21, p. 258]

Equation 6

$$f(x) = \frac{\lambda^{\alpha} x^{\alpha-1} e^{-\lambda x}}{\Gamma(\alpha)}$$

(p.d.f. $f(x)$, noise tolerance x, shape parameter α, scale parameter λ)

Equation 7

$$\alpha = \bar{x}^2 / s^2$$

(estimated mean $\bar{x}$, estimated standard deviation s)

Equation 8

$$\lambda = \bar{x} / s^2$$

It was clear that visually a Gamma distribution did not fit with the data because the distribution tended to zero as the noise tolerance goes to zero, which did not agree with the data. As a result of the sample mean and sample standard deviation being significantly different, an exponential distribution, which is a special case of the Gamma distribution, would not fit the data anyway. [21, p. 53] As a result, the Gamma distribution cannot be used to model the number of maximas as a function of noise tolerance. Another model was used to compare the data with.

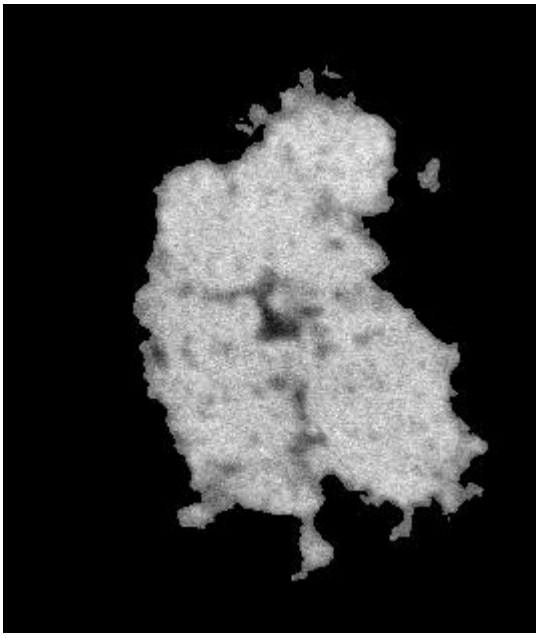


Figure 53 Slice 59 from the stack in Figure 49.

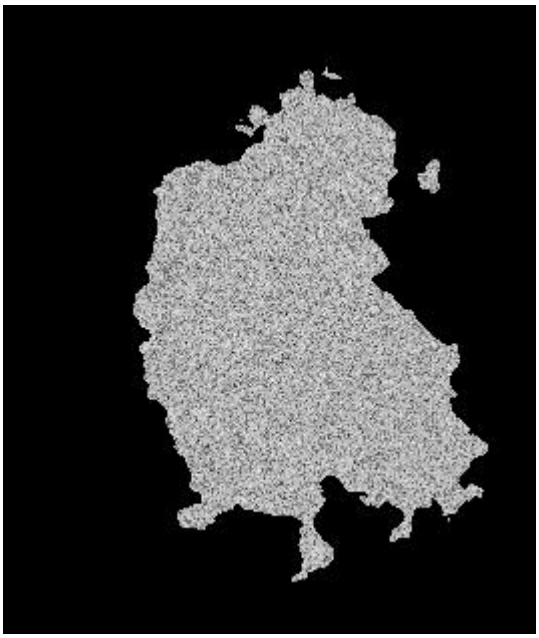


Figure 54 A Monte Carlo image which has the same shape as slice 59 The intensities were randomly generated by treating the histogram of intensities in slice 59 as a probability density function.

The noise tolerance should be chosen such that it produce results which are interesting or significantly different from expected. One way was to compare the image with a

Monte Carlo simulated image which has the same shape as the chromosome but with completely no structure. This was done by treating the histogram of intensity values of the data as a probability distribution function to simulate the intensity values each pixel will have in the Monte Carlo image. A macro was written to generate these images, as shown in Figure 54 and Figure 58.

The number of maximas in the Monte Carlo image, f_{MC}, was worked out and averaged over a sample of 5 Monte Carlo images. This was compared with the number of maximas in data, f_o, as shown in Figure 56. There was a significant difference between f_{MC} and f_o at low noise tolerances, however a statistic, for each noise tolerance, should be calculated to put a measure on how different the number of maximas is compared with the data and the Monte Carlo images.

One possible statistic, z_1, was to compare the difference between the number of maximas with the sample standard deviation of the number of maximas in the Monte Carlo images, s_{MC}, as shown in Equation 9. Another possible statistic, z_2, which could be calculated was to assume the occurrence of maximas, in the Monte Carlo image, is a Poisson process in space, thus the standard deviation of the number of maximas could be estimated as the square root of the number of maximas in the Monte Carlo images, as shown in Equation 10. [23, p. 61]

Equation 9

$$z_1 = \left(\frac{f_o - f_{MC}}{s_{MC}}\right)^2$$

Equation 10

$$z_2 = \frac{\left(f_o - f_g\right)^2}{f_{MC}}$$

Equation 9 and Equation 10 were chosen such that if they were summed over all possible values of noise tolerance and there was no significant difference between f_{MC} and f_o, it would be follow a χ^2 distribution. [23, pp. 61, 106] Therefore the noise tolerance with the highest of one of these statistics would contribute the most to the χ^2 statistic, therefore making it most significant, thus that noise tolerance should be selected. Figure 56 shows the two statistics, z_1 and z_2 plotted.

The two statistics were well behaved for high noise tolerances because the statistics tended to zero at high noise tolerance as expected. This was because there was little difference between the number of maximas compared with the data and Monte Carlo images at high noise tolerances. z_1 had much more random fluctuation than z_2 due to the statistical error in estimating the standard deviation of the number of maximas in a sample of Monte Carlo images. This can be seen in Figure 56 because the estimated standard deviation, shown as the thickness of the curve, varied randomly but decreased at higher noise tolerance. This statistical error could be reduced by taking a larger sample of Monte Carlo images, but at a cost of computing power and time.

z_2 was chosen to be worked with rather than z_1, even though the assumption of a Poisson process was not justified, because the curve of z_2 in Figure 56 was much smoother and made fitting an 8th order polynomial to the data much easier. By modelling z_2 as a function of noise tolerance as a polynomial, the noise tolerance with the highest z_2 was found by finding the maximum of the polynomial; this hopefully helped 'average' out random statistical fluctuations from estimating the mean of number of maximas in a sample of Monte Carlo images.

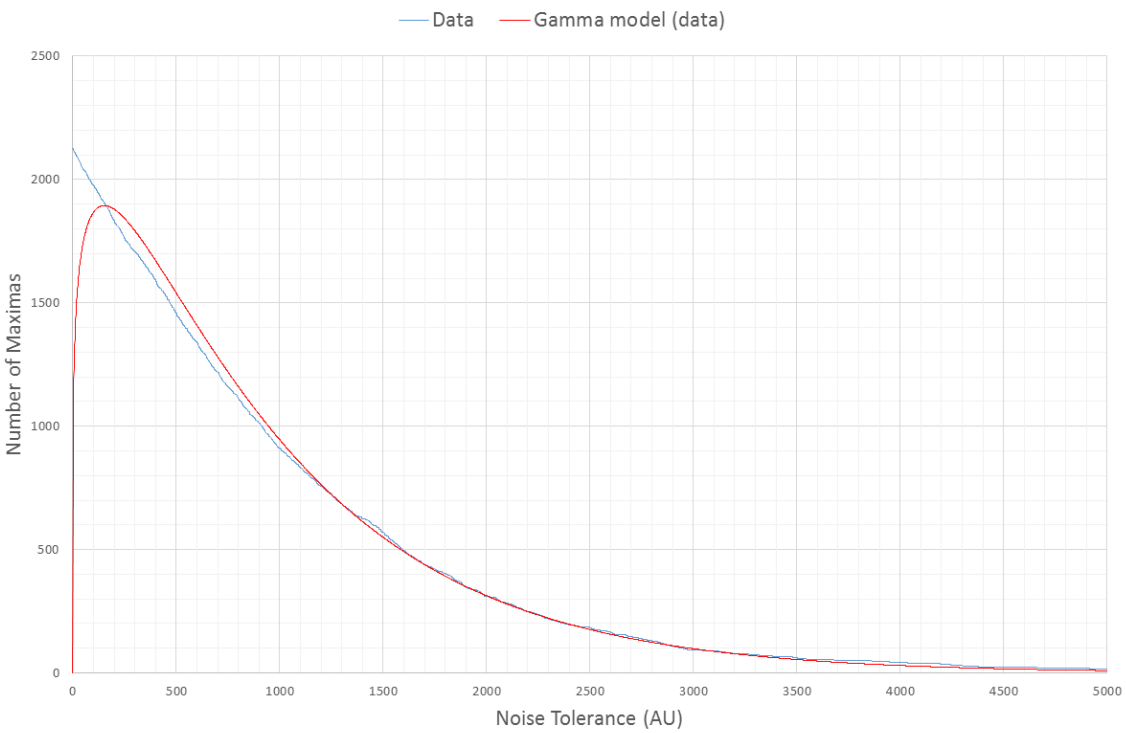


Figure 55 The number of maximas were worked out for each integer setting of noise tolerance on slice 59 (blue). By treating the number of maximas as a frequency, the mean and variance noise tolerance were estimated to attempt to fit a gamma distrubtuion to the data (red).

Mean = 961, Variance = 776941, α = 1.19, λ = 0.00124

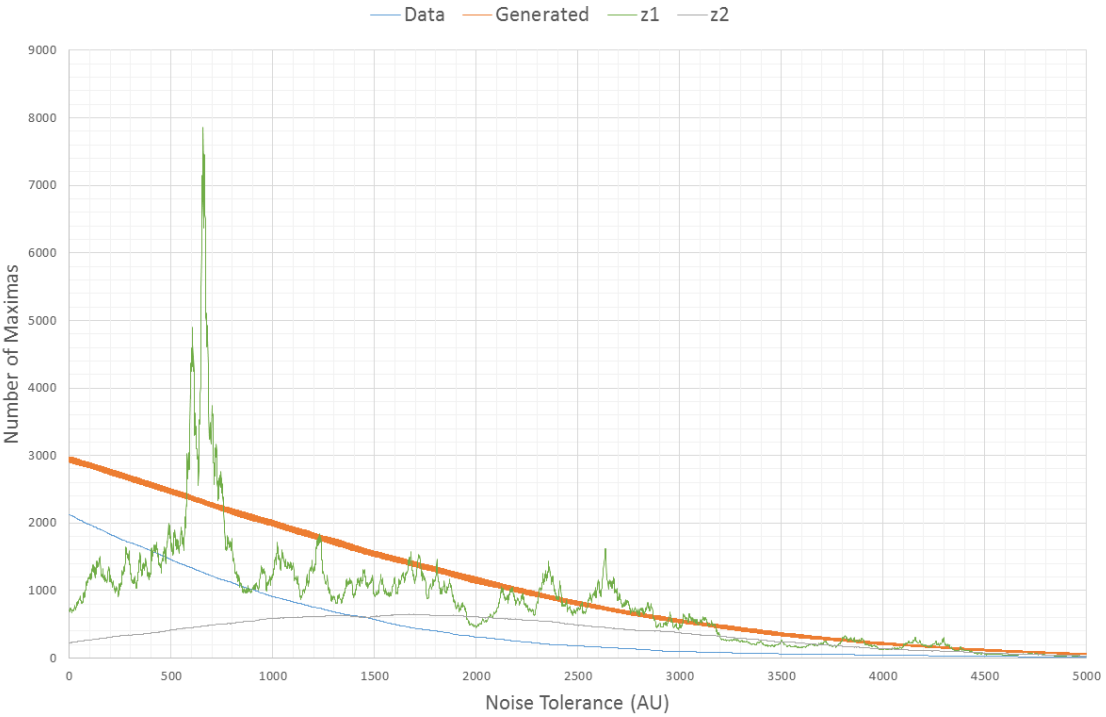


Figure 56 The number of maximas at different noise tolerances, for slice 59 from the data (blue) and the Monte Carlo simulation (orange) were compared. The thickness of the orange line corresponds to the estimated standard deviation. The statistics z_1 (green) and z_2 (gray) were also plotted.

An ImageJ plugin was written to compare the number of maximas, for each integer noise tolerance setting from 0 to 5000, for each and all 122 slices in Figure 49 with a sample of 10 Monte Carlo slices per slice (Figure 58). It fitted an 8th order polynomial on the z_2 statistic as a function of noise tolerance for each slice, so that the noise tolerance which maximizes the polynomial was used to calculate maximas in that slice.

The result of the plugin, which took about 4 hours to run, found 32748 maximas as shown in Figure 59. A watershed algorithm was used on the maximas to connect them together and then cropped to fit in the shape of the chromosome as shown in Figure 60. The problem with the watershed algorithm was that the algorithm only worked in 2D, thus only on each slice separately, therefore was unable to connect the whole stack of maximas together. A different approach was used to connect the maximas together.

A good statistics to look at is the estimated population mean and standard deviation of the distance to the nearest neighbour for each maxima. In Processing, an algorithm was written to calculate the distance to the nearest neighbour of every maxima in the stack. The algorithm was very naive because the algorithm calculated, for each maxima, the distances between every maxima, the smallest distance being the distance to the nearest neighbour. The histogram of the distances to the nearest neighbour is as shown in Figure 61.

The distribution of distances to the nearest neighbour in the histogram appeared to be very discrete and form bands on a certain range of distances. This was not a surprising result because this was the result of the discrete nature of voxels. Because voxels are arranged in a grid, this caused the distances to the nearest neighbour to have a discrete nature as well. All the possible distances to the nearest neighbour as are shown in Figure 57.

***z* (nm)**	***x* (nm)**	***y* (nm)**	***d* (nm)**
20	0	0	20.0
0	22	0	22.0
20	11	0	22.8
0	22	11	24.6
20	11	11	25.3
20	22	0	29.7
0	22	22	31.1
20	22	11	31.7
0	33	0	33.0
0	33	11	34.8
20	22	22	37.0
20	33	0	38.6
0	33	22	39.7
40	0	0	40.0

Figure 57 The smallest 14 possible distances to nearest neighbours (d) due to the discrete nature of the position of the voxels, the slice separation (z) of 20 nm and the pixel size (x and y) of 11 nm.

Equation 11

$$\bar{x} = \frac{\sum_{i=0}^{n} x_i}{n}$$

Equation 12

$$s = \sqrt{\frac{\sum_{i=0}^{n} x_i^2 - \left(\sum_{i=0}^{n} x_i\right)^2 / n}{n-1}}$$

Equation 3

$$\bar{x} - \frac{\Phi^{-1}(97.5\%)s}{\sqrt{n}} < \mu < \bar{x} + \frac{\Phi^{-1}(97.5\%)s}{\sqrt{n}}$$

Equation 13

$$\frac{\nu s^2}{\sqrt{2\nu}\Phi^{-1}(97.5\%) + \nu} < \sigma^2 < \frac{\nu s^2}{\sqrt{2\nu}\Phi^{-1}(2.5\%) + \nu}$$

where $\nu = n - 1$

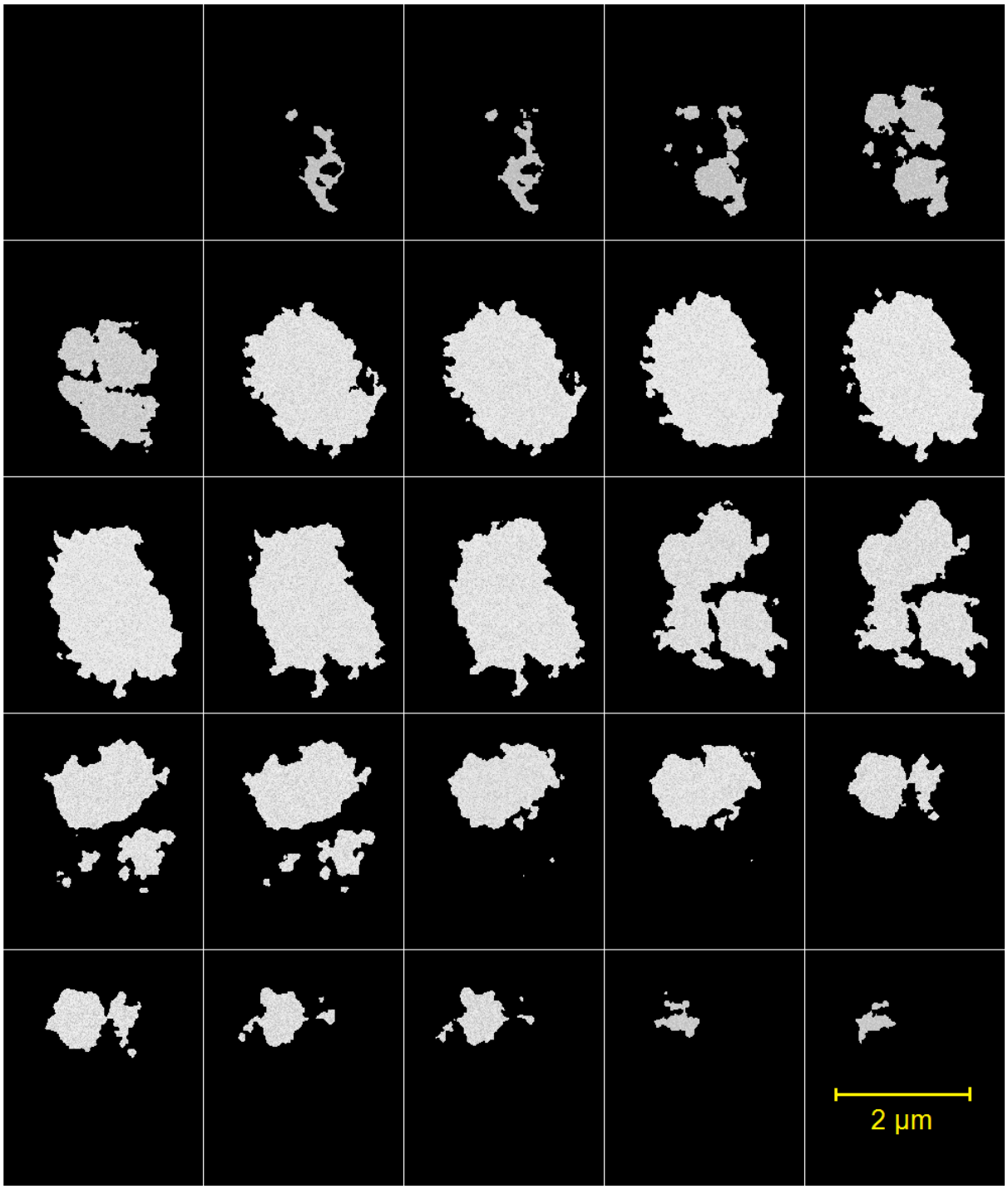


Figure 58 Slices 1,6,11,…,121 of the Monte Carlo stack.

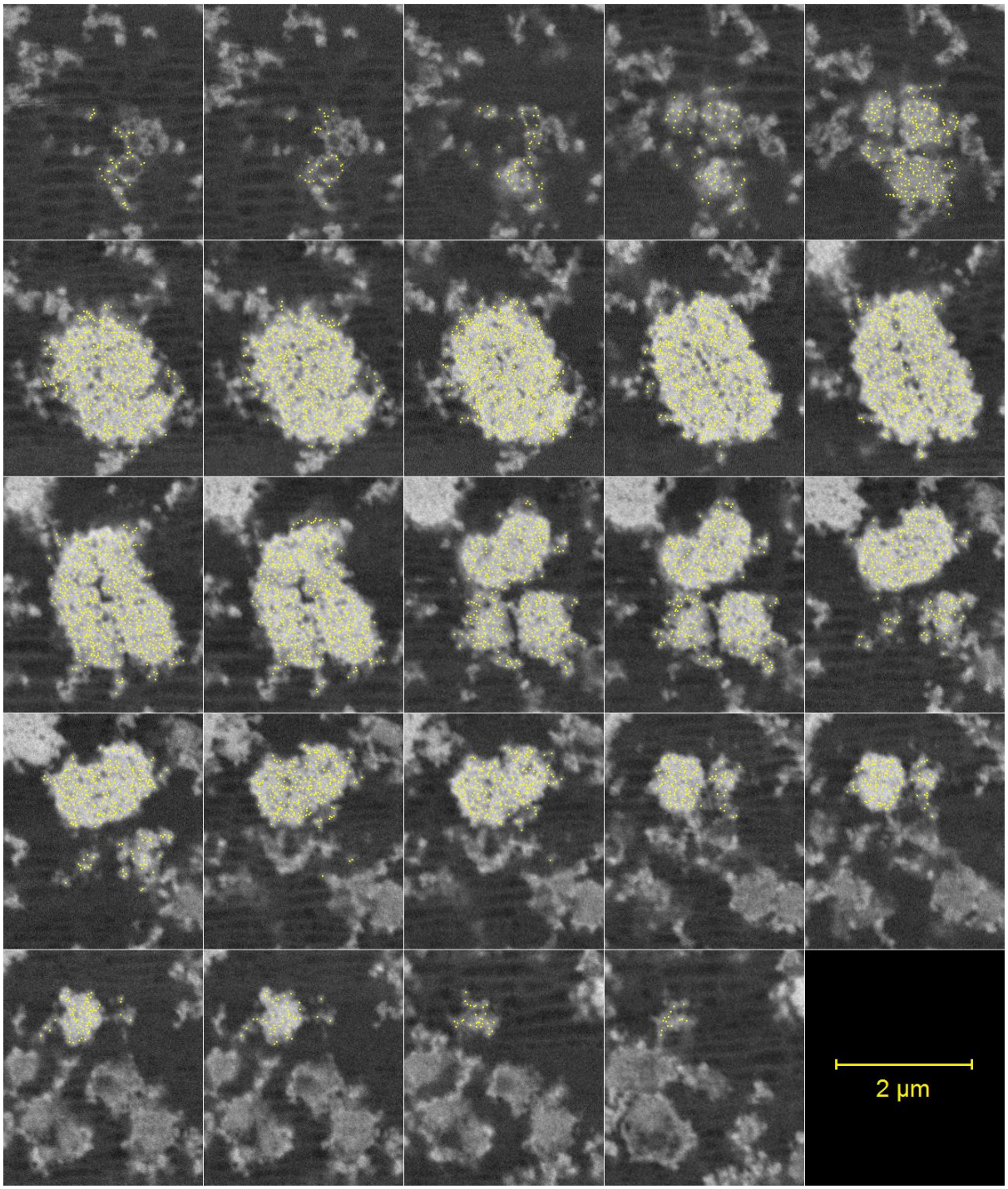


Figure 59 The result of a plugin of slice 5,10,15,…,120 which found maximas, shown as yellow dots, by using a noise tolerance setting which maximizes the z_2 statistic.

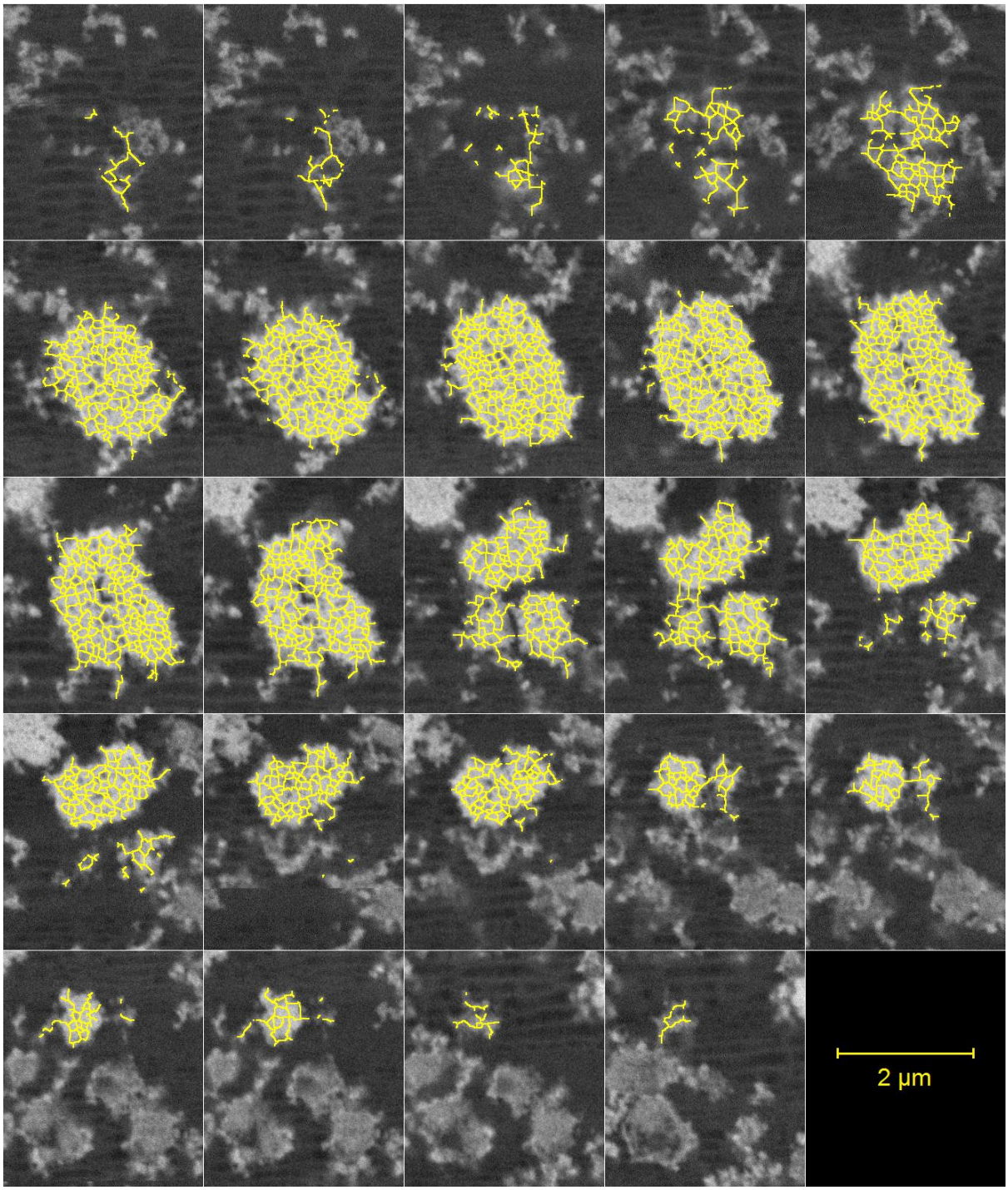


Figure 60 The watershed algorthim was applied and cropped on the points in Figure 59.

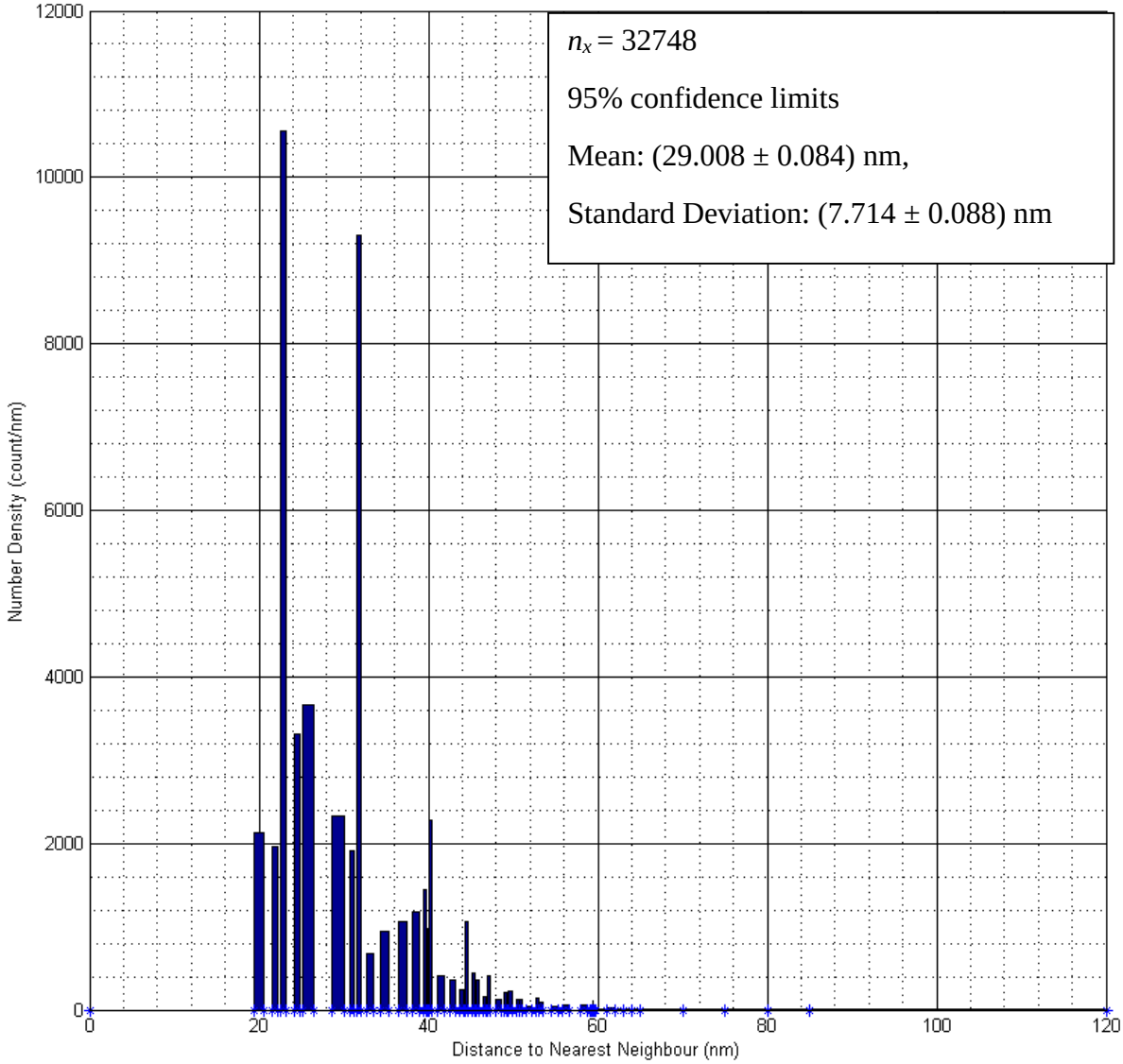


Figure 61 A histogram of the distances to nearest neighbour using the maximas obtained from the orginal data stack..

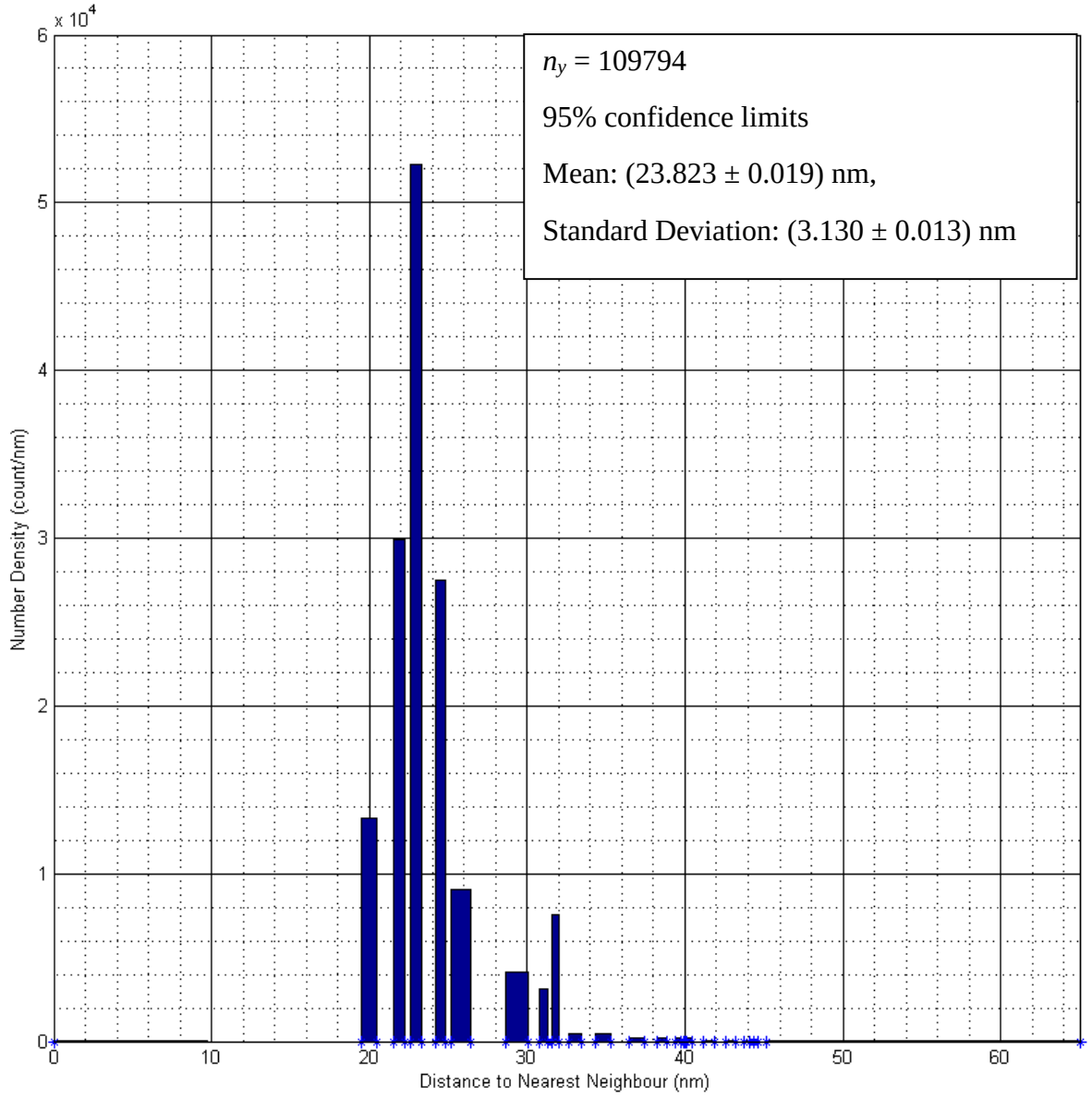


Figure 62 A histogram of the distances to nearest neighbour using the maximas obtained from the Monte Carlo stack.

The sample mean, $\bar{x}$, and sample standard deviation, *s*, of the distances to the nearest neighbour, were calculated using Equation 11 and Equation 12 respectively. [21, p. 195]

Because a large sample of distances to the nearest neighbour was obtained, 32748, the sample mean and sample standard deviation approximately has a Normal distribution. [21, p. 181] As a result, the 95% confidence limit of the population mean and population standard deviation were calculated using Equation 3 and Equation 13 respectively.

The 95% confidence limit of the mean distance to the nearest neighbour is (29.008 ± 0.084) nm, which could be evidence of 30 nm chromatin fibre. However the result could be just a fluke and could be as a result of the resolution of the stack, for example maximas must be as least one voxel apart and this puts a lower limit on the distances to the nearest neighbour.

The same statistics, the mean and standard deviation of the distances to the nearest neighbour, were estimated using the maximas calculated from the Monte Carlo stack (Figure 58), and the 95% confidence limit of the mean distance to the nearest neighbour was calculated to be (23.823 ± 0.019) nm. The histogram of the data is shown in Figure 62. The mean distances to the nearest neighbour in the original data, *x*, and the Monte Carlo stack, *y*, were compared by calculating a *z* statistics, which has a standard Normal distribution, as shown in Equation 14. [22, p. 149]

Equation 14

$$z = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y}}}$$

The *z* statistic was calculated to be ~119 and this was evidence that the mean distance to the nearest neighbour, obtained from the Monte Carlo stack, was extremely significantly different from the mean value obtained from the data.

Following from this, the justification that the obtained mean distance to the nearest neighbour from the original data being evidence of 30 nm chromatin fibre, was slightly strengthen because the statistic obtained from the data is significantly different from the simulation, ruling out the possibility that the statistics obtained was the result of the resolution of the stack. The results, however, are not yet reliable because the same method should be repeated and applied to other SBFSEM images, to see if the same results are obtained.

Another statistics calculated was the Pearson's correlation coefficient, *r*, between the distances to the nearest neighbour with the distances to the centre of mass of the maximas. If evidence of such correlation existed, this could suggest that the maximas are denser closer to the centre of mass. Again, an algorithm was written in Processing to analyse the stack of maximas and to work out the distances to the centre of mass and the nearest neighbour. A scatter graph of the correlation using the data and the Monte Carlo stack is as shown in Figure 63 and Figure 64 respectively. Even though the values of *r* were calculated to have magnitude of two decimal places, the degrees of freedom, which is essentially the number of data points *n*, must be considered to reach a statistical conclusion. [30, p. 56] Equation 15 was used to transform, *r*, to a random variable *z* which is approximately Normally distributed with mean 0 and unit variance. [31] Following from this, the *z* statistics were calculated as shown in Figure 65.

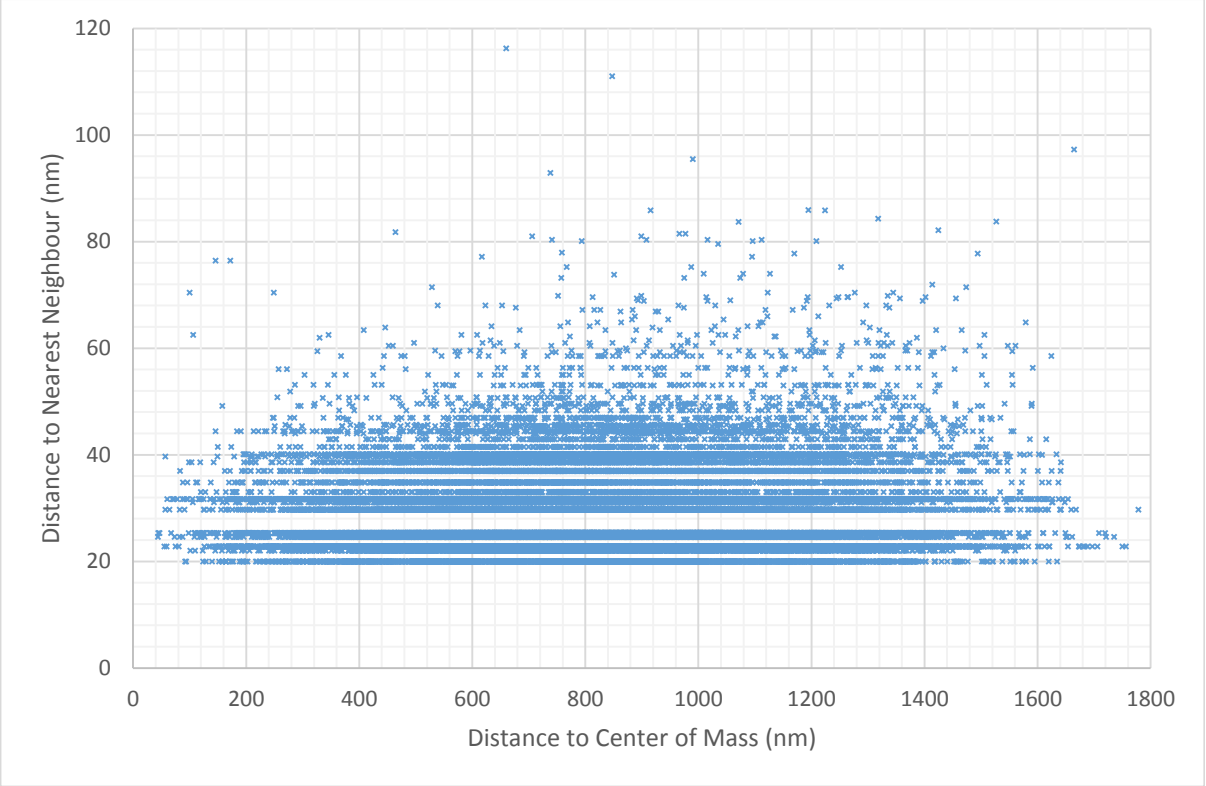


Figure 63 A scatter graph of the each maxima distance to nearest neighbour and distance to the center of mass using maximas from the data.

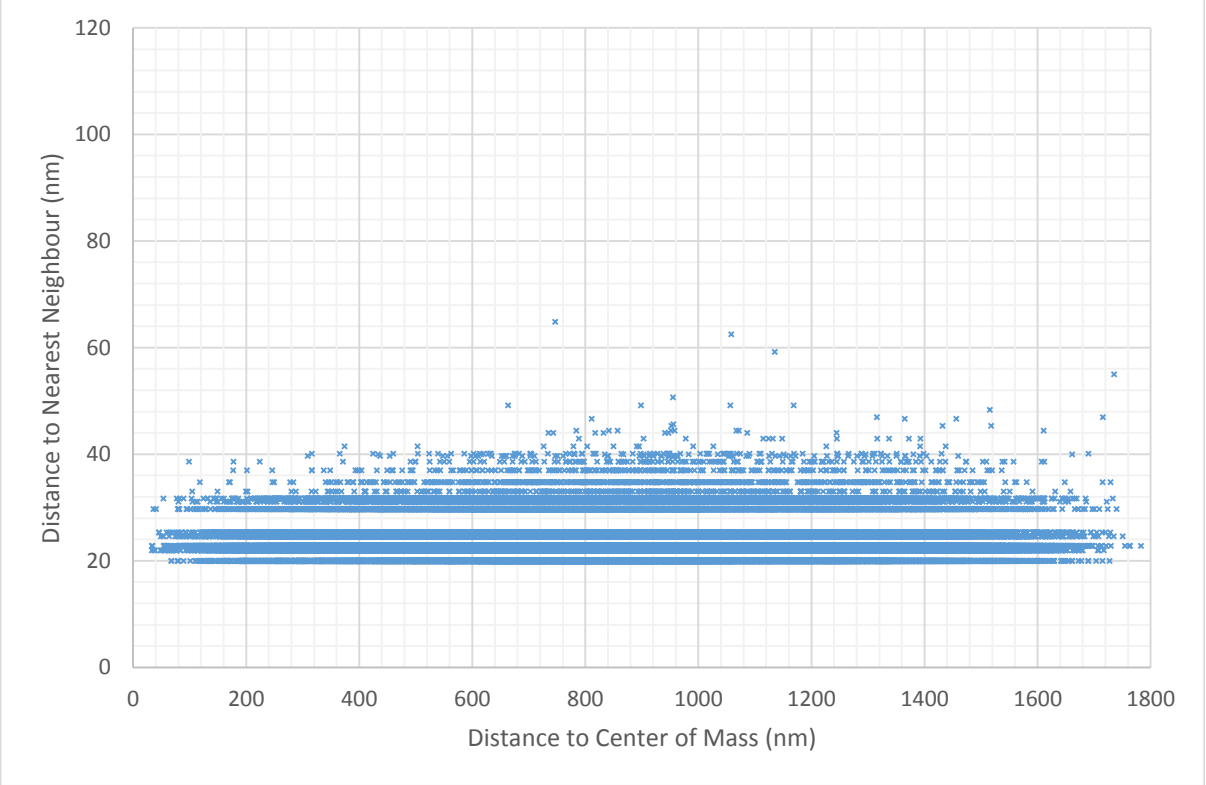


Figure 64 A scatter graph of each maxima distance to enarest neighbour and distance to the center of mass using maximas from the Monte Carlo simulation.

Equation 15

$$z = \sqrt{n-3}\,\operatorname{arctanh}(r)$$

	Data	Monte Carlo
n	32748	109794
r	0.076011	0.0212027
z	13.781	7.02649

Figure 65 Correlation statistics between the distance to nearest neighbour with the distance to the centre of mass using maximas from the data and the Monte Carlo simulation. The higher the value of z is, the stronger the evidence that there is a correlation.

The values of *z* shows that there was strong evidence of correlation between the distance to the nearest neighbour with the distance to the centre of mass. Although a correlation was found, nothing about the cause between the two random variables was found. Some suggestions to the cause of the correlation could be that the maximas are denser closer to the centre of mass or that maximas closer to the centre of mass have more neighbours than maximas on the surface on the chromosome. More sophisticated approaches should be used to further look into how the maximas are arranged inside the chromosome, rather than just statistically analysing random variables.

An approach used to connect maximas together was to fit 29.008 nm diameter spheres at every maxima, using Processing. Of course, it will be computationally intense if 32748 spheres were drawn, therefore only a sample of maximas were worked on in the project. The result is shown in Figure 66. At a glance, the 3D render did appeared to be spheres placed randomly. However after closer examination, some spheres overlap each other consistently to form a coil, as shown in Figure 67, Figure 68 and Figure 69. But at most times the spheres merged with each other to form a very tight cluster rather than a coil. The results could be evidence of 30 nm chromatin fibres, however more work or more accurate and precise data would be needed to find the structure or the arrangement of the maximas.

Finally, another approach was used to connect maximas together by simply drawing lines between maximas if the distances between maximas are in the order of (29.0 ± 7.7) nm. An algorithm was written in Processing to connect the maximas together as shown in Figure 70 and Figure 71.

The result was excellent because the X shape of the chromosome was much clearer than the result used from the marching cubes algorithm. The centromere and the pair of chromatids was easily identified.

Because the centromere was much more easily recognized, the position of the centromere could be estimated using Figure 70. The ratio of the length of the short chromatin over the length of the chromosome was estimated to be about ~0.4. This information could be used to predict the specific chromosome which was worked on.

The volume of the chromatin fibres could be estimated by summing the volume of all 32748 29.008 nm diameter spheres. The diameter of the spheres, *D*, was treated as a random variable with mean *d* nm and standard deviation σ_d nm, and as a result the volume of chromatin fibres, *V*, was also a random variable as shown in Equation 16. Equation 17 and Equation 18 is an approximate estimation for the mean and standard deviation of a function of a random variable, where dashes are derivatives with respect to *D*. [21, p. 162]

These equations were used to approximately estimate the mean and standard deviation of *V* as shown in Equation 19 and Equation 20 respectively, where *N* is the number of spheres.

Equation 16

$$V(D) = \frac{1}{6}N\pi D^3$$

Equation 17

$$\mathrm{E}(V) = V(d) + \frac{1}{2}\sigma_d^2 V''(d)$$

Equation 18

$$\sigma_V = V'(d)\sigma_d$$

Equation 19

$$\mathrm{E}(V) = \frac{N\pi d^3}{6}\left(1 + \frac{3\sigma_d^2}{d^2}\right)$$

Equation 20

$$\sigma_V = \frac{\sigma_d N\pi d^2}{2}$$

Taking the values of the mean and standard deviation of *D* as 29.008 nm and 7.714 nm respectively, as shown in Figure 61, the mean and standard deviation of *V* was estimated to be (0.51 ± 0.33) μm^3.

This was used as a model for the volume of the chromatin fibres in the chromosome. Following from this, the number of base pairs, or pairs of nucleotides, in the chromosome was estimated. By modelling nucleosomes as an 11nm diameter cylindrical disc with thickness 5.7 nm, [32] the number of nucleosomes was estimated by calculating the number of nucleosomes in the chromatin fibres, modelled by spheres. This was done, naively, by dividing the volume of the chromatin fibres with the volume of a nucleosome. It was estimated that there are (940 ± 620) thousands nucleosomes in the chromosome. This was an overestimation because the chromatin fibres consist of other material than just nucleosomes.

For each nucleosome, there are 147 base pairs wrapped around each histone with an additional 20 base pairs bounded by the linker histone H1/H5. [9] Furthermore, each nucleosome are separated by linker DNA which varies in length between 0 and 80 base pairs. [9] The distribution of the length of linker DNA, between each nucleosomes, was modelled with a uniform distribution from 0 to 80 base pairs, thus has mean 40 base pairs and standard deviation 23 base pairs. Following from this, it was estimated there are (207 ± 23) base pairs per nucleosome.

Equation 21

For $Y = X_1X_2$ and X_1 independent of X_2
and X_i has mean x_i and standard deviation σ_i

$$\sigma_Y = y\sqrt{\left(\frac{\sigma_1}{x_1}\right)^2 + \left(\frac{\sigma_2}{x_2}\right)^2}$$

By assuming the number of base pairs per nucleosome and the number of nucleosomes in the chromosome are independent, the mean and standard deviation of the number of base pairs, in the chromosome, were estimated, with the aid of the error propagation formula Equation 21. [23, p. 22] It was estimated there are (190 ± 130) million base pairs in the chromosome.

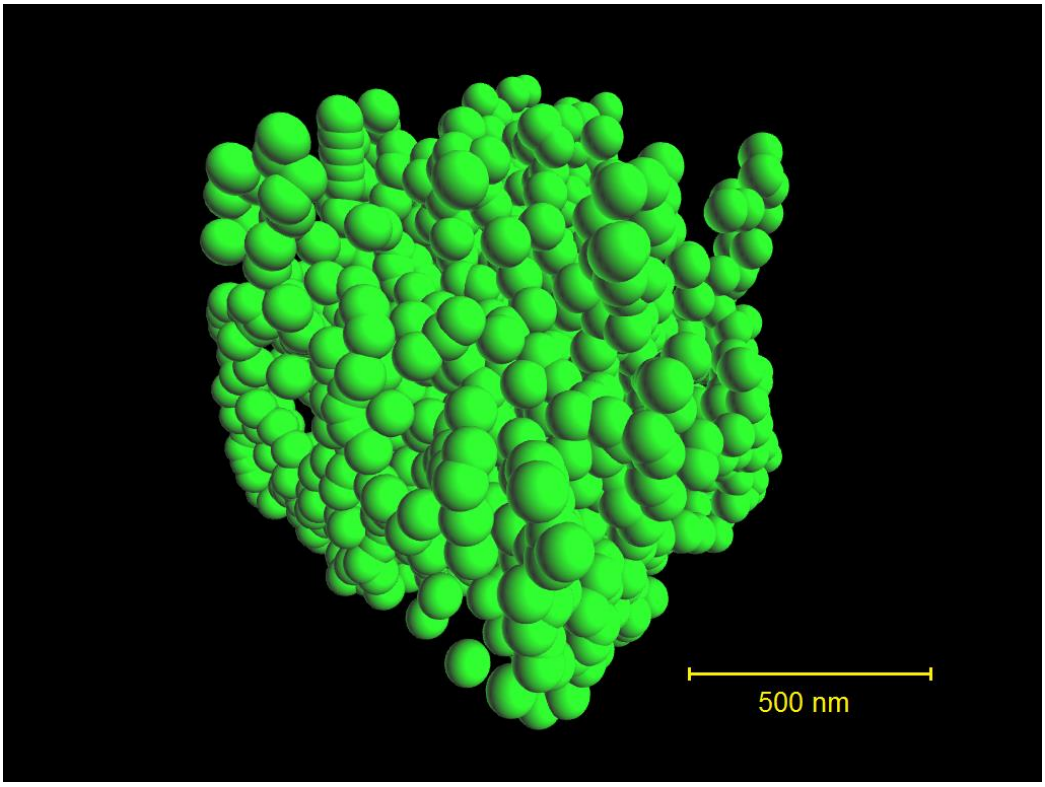


Figure 66 A sample of maximas were used to model each maxima as a 29.0 nm diamter sphere.

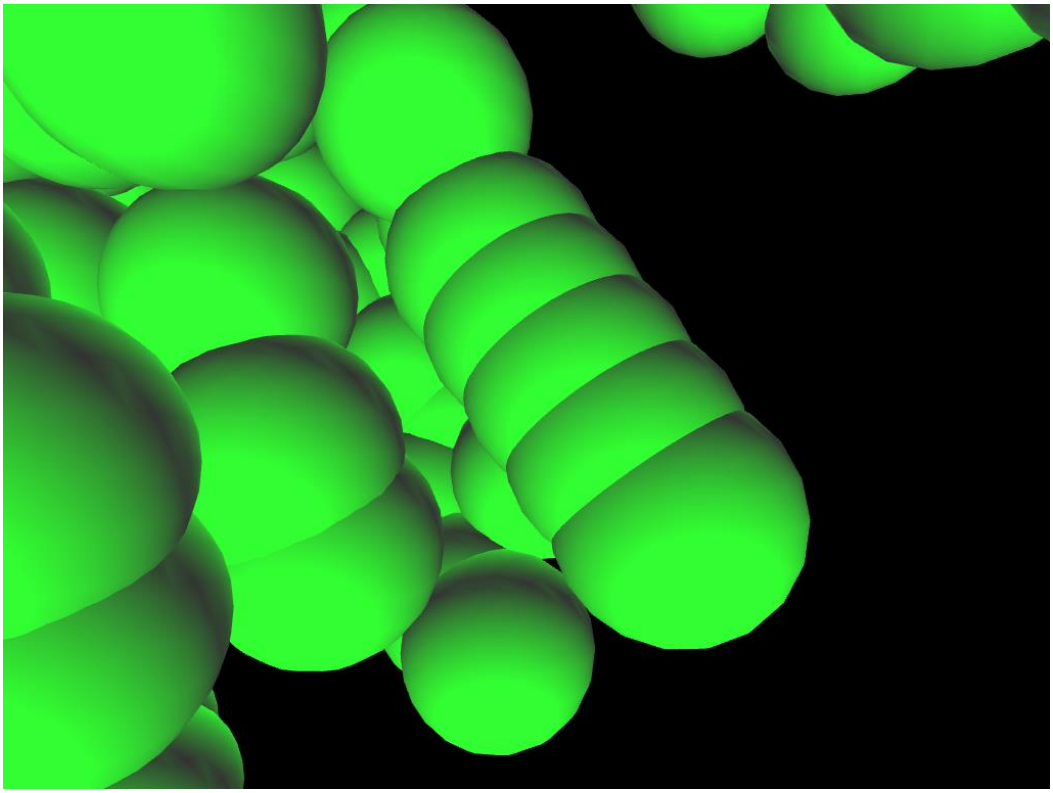


Figure 67 Some spheres form a straight line.

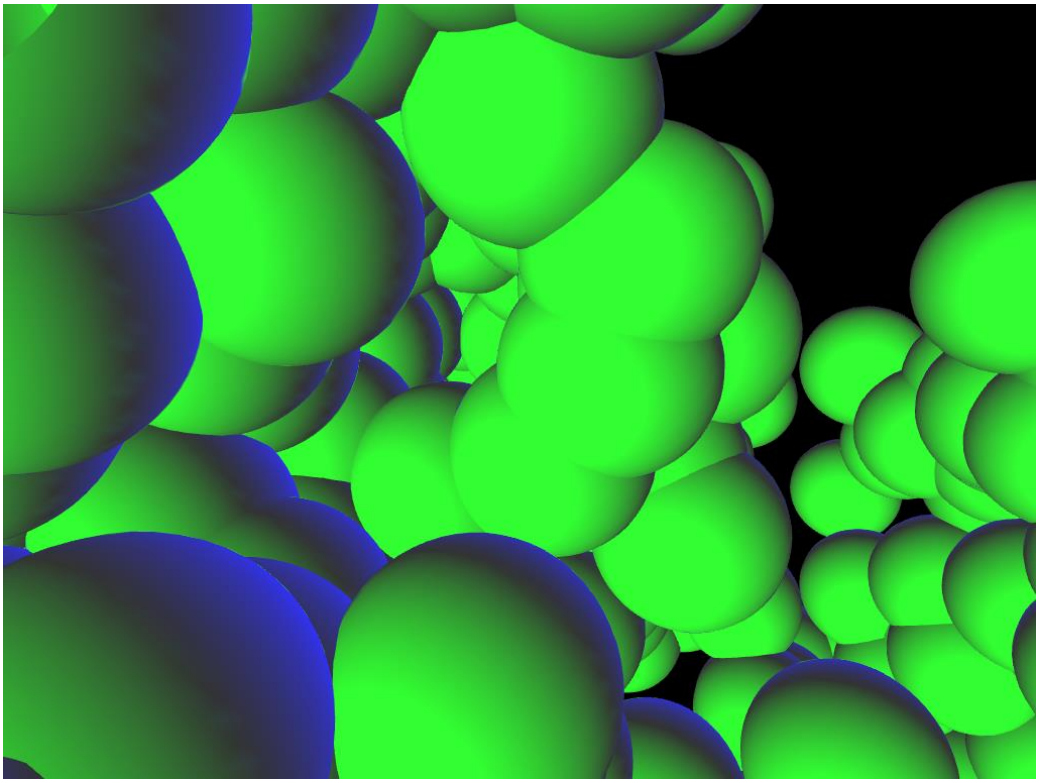


Figure 68 Some spheres from a smooth curve.

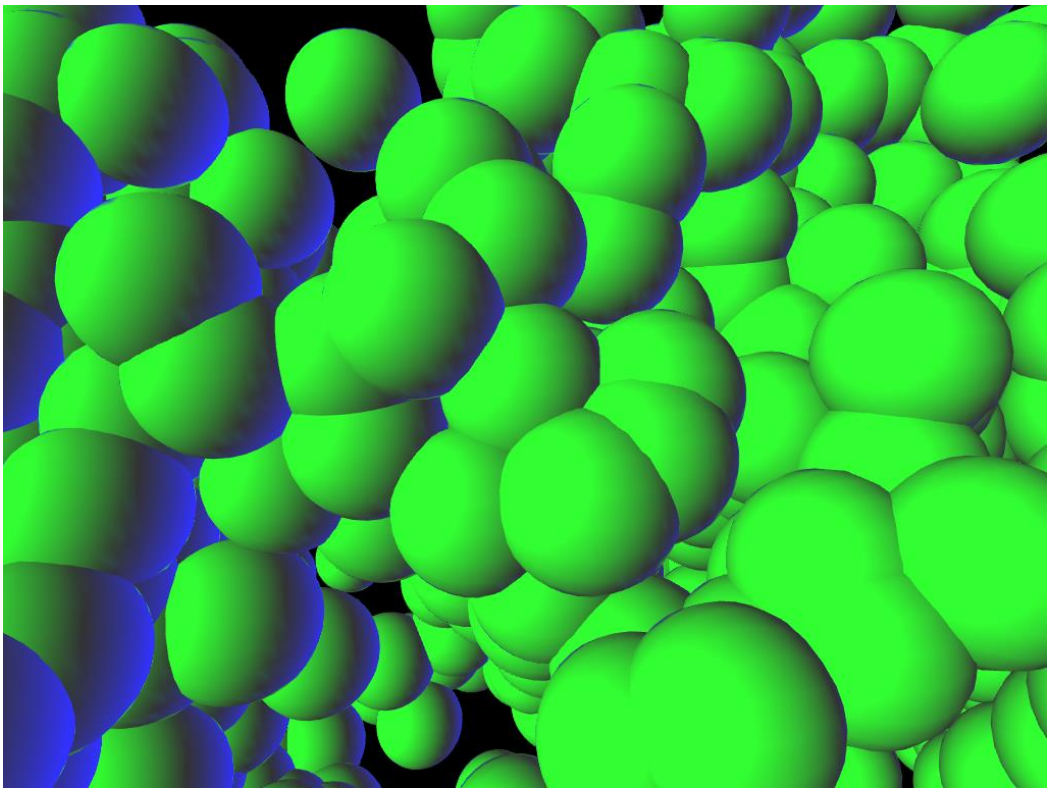
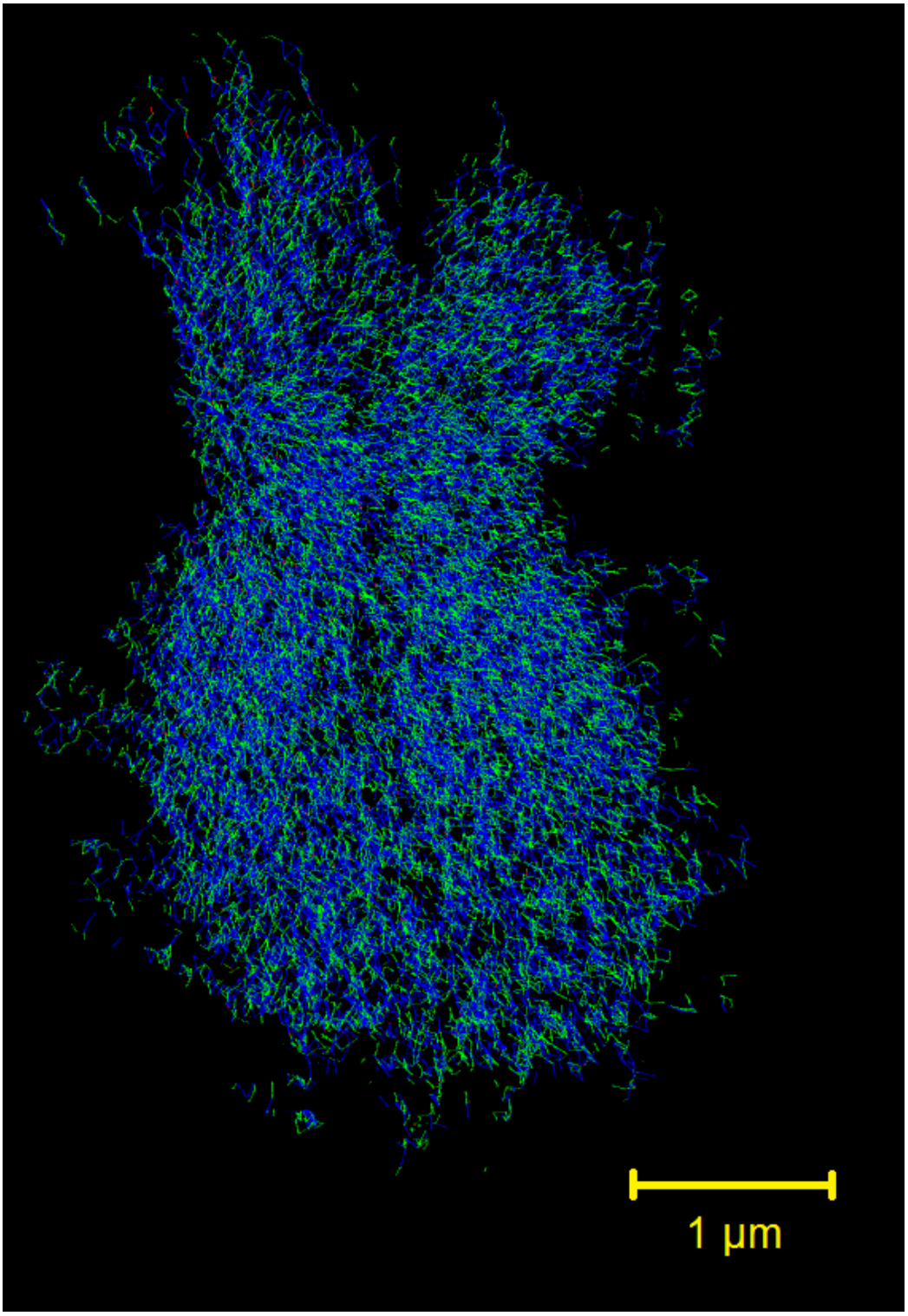
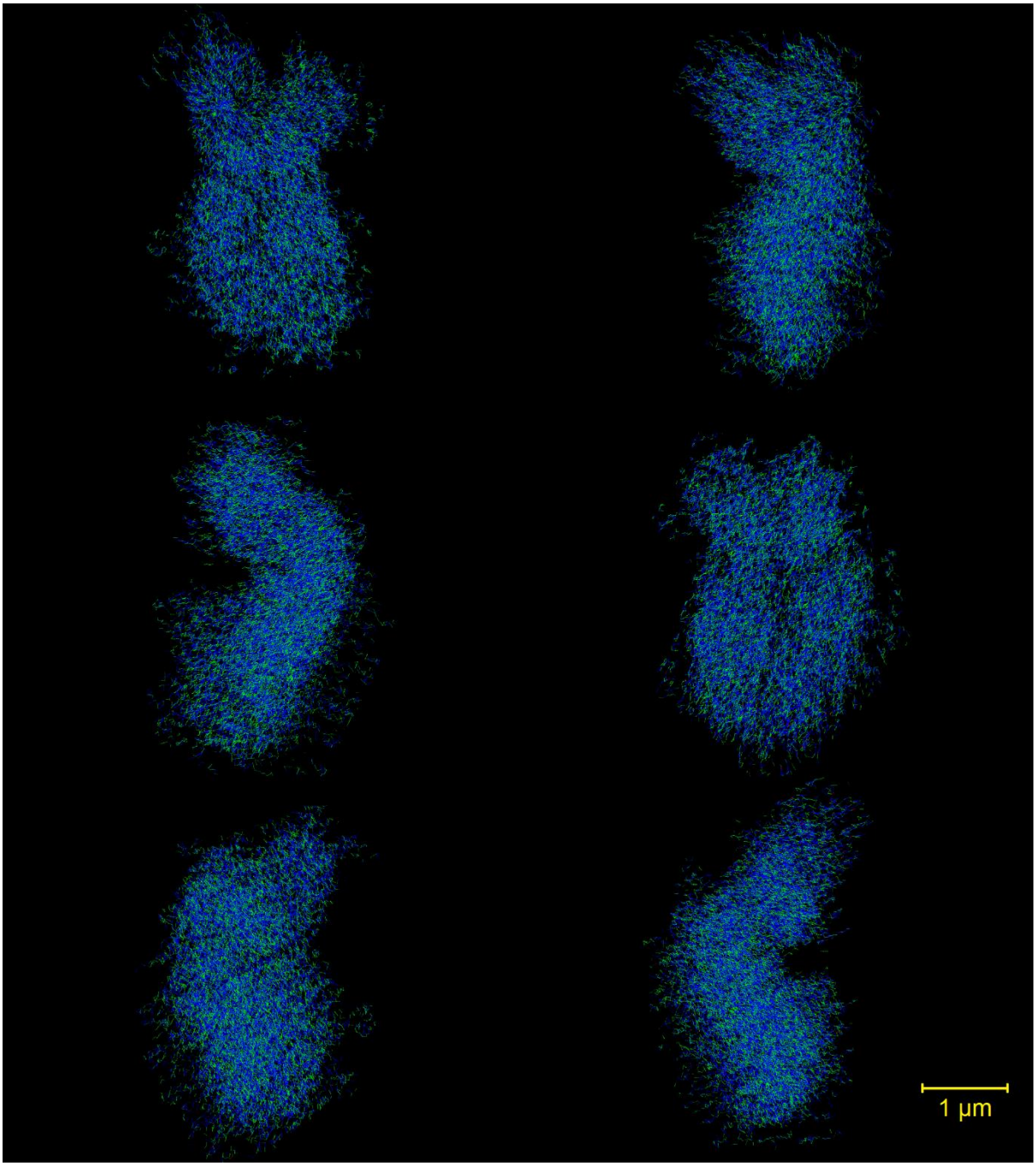


Figure 69 Some spheres form a tight coil.



Red: (0.0 – 21.3) nm
Green: (21.3 – 36.7) nm
Blue: (36.7 – 44.4) nm

Figure 70 Different coloued lines were drawn connecting maximas together if the distances between them are in the order of (29.0 ± 7.7) nm.



Red: (0.0 – 21.3) nm
Green: (21.3 – 36.7) nm
Blue: (36.7 – 44.4) nm

Figure 71 The links between maximas viewed at different angles.

V. CONCLUSION

The SBFSEM images were segmented by a threshold. Firstly, samples of intensity values of different features or signals of the images were taken and were used to estimate the mean and standard deviation of the intensity values of each signal. Three significant signals were found: the chromosome, the background and other signals inside and around the chromosome (see Figure 46 p. 42). Fortunately the intensity values of the different signals were significantly different and a threshold was set to cover the range of intensity values within two standard deviations away from the mean intensity values of the chromosome and its features.

This was then used to estimate the volume of the chromosome to be about 3.95 μm^3. The volume calculation was dependent on the threshold used. The value calculated was suggested to be an overestimate because the distinguishable intensity values inside the chromosome may be vacancies, and should be accounted for in the volume calculation.

Following from the threshold, the volume of the chromosome was rendered in 3D using the marching cubes algorithm. [24] [26] The algorithm was programmed from scratch using an open source programming language called Processing. [19] This saved a lot in terms of cost, but more importantly a program specific for this project was built. The results were excellent and the centromere and chromatids were easily identified in the renders (see Figure 50 p. 46 and Figure 51 p. 47). The next step in visualization of the data should be looking at commercial software like Avizo [33] and Imaris [34]. Both software process 3D, and even 4D, images, using state of the art features. [33] [34]

Using commercial software would take the methodology of this project in a different direction. The marching cubes algorithm was programmed from first principles in this project, but using commercial software would require training to use the software instead.

Internal structures of the chromosome were attempted to be found using the ImageJ maxima finder. [29] The number of maximas found were compared using different noise tolerances and Monte Carlo images, which has the same shape as the data and uses the histogram of the intensity values of the data as a probability distribution function to generate an image (see Figure 58 p. 52). The noise tolerance was chosen which had the highest χ^2 statistics (see Equation 10 p. 49) under the assumption that maximas occurring, in the Monte Carlo simulation, was a Poisson process in space.

The assumption was unjustified; for a Poisson process in space, each pixel in the image must have equal probability of a maxima occurring and independent of other pixels and maximas. [35, pp. 313, 327] This was, of course, not true because each maxima must be at least two pixels away. However, even under that assumption, the algorithm had produced smooth results which a polynomial can be fitted with (see Figure 56 p. 50), used to find the noise tolerance with the highest χ^2 statistics. For the assumption to be dropped and Equation 9 (p. 49) to be used, it would need a lot of computing power to simulate and analyse a larger sample of Monte Carlo images. Larger samples of Monte Carlo images would produce more accurate and precise values of the mean and standard deviation of the number of maximas and thus should decrease the fluctuation seen in the statistics for different noise tolerances,

making finding the noise tolerance with the highest χ^2 statistics easier.

The mean distance to the nearest neighbour was found for each maxima to be (29.008 ± 0.084) nm. This could be good evidence for 30 nm chromatin fibres. This was then used to model each maxima as a 29 nm sphere. The results do show coils forming (see Figure 68 p. 62) however not many. This was most likely due to the alignment problem with the diamond knife (see Figure 45 p. 41). Better data, without alignment issues, would be needed to produce more precise and accurate models on how the 30 nm chromatin fibres were structured.

By modelling the spheres as chromatin fibres, it was estimated that there are (190 ± 130) million base pairs in the chromosome. The standard deviation in the value was too big to make a judgement on what chromosome it is. Chromosome 4 also has about 190 million base pairs however chromosome 1, the biggest, has about 250 million pairs [36] which is within range of one standard deviation. The main source of error was from cubing the distance to the nearest neighbours to find the volume of the sphere (see Equation 16 p. 60).

A better statistics to use is the average of the distance to the first 2 nearest neighbours because for a chain, or fibre, to be formed, each point must be connected by two neighbouring points. This statistics should give a more accurate, and larger, estimation for the diameter of the 30 nm chromatin fibre.

Lines were drawn between maximas, where the distances between them are in the order of (29.0 ± 7.7) nm (see Figure 71 p. 64). The pair of chromatins were easily identified with most of the connections at the centromere. Using the figure, it was estimated that the ratio the length of the short arm over the length of the chromosome was ~0.4. This corresponds to chromosome 2 which has a ratio of 0.389, [37] suggesting the chromosome which was worked on may be chromosome 2.

VI. APPENDIX

The table below shows the information of the source code used in the project shown in the appendix.

Name	Page Number	Language and Environment	Data Files	Classes
3D RENDER OF A CHROMOSOME USING VOXELS	68	Processing	Binary threshold .png images See Figure 24 p. 22	
3D RENDER OF A CHROMOSOME USING THE MARCHING CUBES ALGORTHIM	72	Processing	Binary threshold .png images See Figure 48 p. 44	Material Cube Triangle
MAXIMA FINDER BY USING NOISE TOLERANCES WHICH MAXIMIZES CHI-SQUARED	98	Java in an ImageJ Plugin environment	Cropped and inverted chromosomes .tiff stack See Figure 49 p. 45	
NEAREST NEIGHBOUR SEARCH AND DISTANCE TO CENTER OF MASS	105	Processing	Binary .png images of the maximas	PointVector
MODELLING MAXIMAS AS SPHERES	110	Processing	Binary .png images of the maximas	
CONNECTING MAXIMAS TOGETHER USING LINES	113	Processing	Binary .png images of the maximas	

3D RENDER OF A CHROMOSOME USING VOXELS

```
//import PeasyCam
import peasy.*;
PeasyCam cam;

//PShape for chromosome
PShape chromosome;

//declare a 3D array of booleans, true for white pixel
boolean[][][] pixelGrid;
boolean[][][] pixelGridHollow;
int imageWidth, imageHeight; //dimension of image
int nSlices = 26; //26 slices

//SETUP
void setup(){
  //setup drawing
  size(1024,768,P3D);
  cam = new PeasyCam(this, 500); //makes a new camera and set distances

  //create the arrayList of slices
  ArrayList <PImage> imageArray = new ArrayList<PImage>();

  //for the fist 10 times, put the image into the arrayList
  for (int i=0; i<10; i++){
    imageArray.add(loadImage("X0"+i+".png")); //load image
  }//end for
  //for the last set of image, put the image into the arrayList
  for (int i=10; i<nSlices; i++){
    imageArray.add(loadImage("X"+i+".png")); //load image
  }//end for

  //get the dimensions of the image
  imageWidth = imageArray.get(0).width;
  imageHeight = imageArray.get(0).height;

  //create the pixelGrid of the correct size
  pixelGrid = new boolean[imageWidth][imageHeight][nSlices];
```

```
  pixelGridHollow = new boolean[imageWidth][imageHeight][nSlices];

  //fill in pixelGrid using the images
  getPixelGrid(imageArray);
  //hollow out the pixelGrid by removing voxels with 6 neighbours
  hollowOutGrid();
  //set up the PShape of chromosome
  setupChromosome();

}//end setup


//DRAW
void draw(){
  background(0); //set background to black
  scale(1,50.0/27.0,1); //scale the maxtrix
  shape(chromosome); //show the chromosome
  println(frameRate); //print the frameRate
}//end draw


//Reads the binary images in imageArray and puts true in the pixelGrid for every white pixel
void getPixelGrid(ArrayList<PImage> imageArray){

  //for every image in imageArray
  for (int z=0; z<nSlices; z++){

    //load the pixels in the image
    imageArray.get(z).loadPixels();

    //declare x and y co-od
    int x = 0;
    int y = 0;

    //for every pixel in the image
    for (int i=0; i<imageArray.get(z).pixels.length; i++){
      if (imageArray.get(z).pixels[i]==color(255)){ //if that pixel is white
        pixelGrid[x][y][z] = true; //put true in the pixelGrid
      }//end if
```

```
      else{
        pixelGrid[x][y][z] = false; //else put false
      }//end else

      //increase x and y co-od accordingly
      x++;
      if (x==imageWidth){
        x=0;
        y++;
      }//end if
    }//end for
  }//end for
}


//remove voxels with 6 neighbours from pixelGrid
void hollowOutGrid(){
  //for each slice
  for (int z=0; z<nSlices; z++){
    //for each pixel in the slice
    for (int y=0; y<imageHeight; y++){
      for (int x=0; x<imageWidth; x++){

        //if the pixel is on the border, the pixel remains unchanged
        if(x==0||x==imageWidth-1 || y==0||y==imageHeight-1 || z==0||z==nSlices-1){
          pixelGridHollow[x][y][z] = pixelGrid[x][y][z];
        }//end if

        else if(pixelGrid[x][y][z]==true){ //if the pixel is true
          if(pixelGrid[x-1][y][z]==true && pixelGrid[x+1][y][z]==true //if the pixel has 6 true neighbouring pixels
          &&pixelGrid[x][y-1][z]==true && pixelGrid[x][y+1][z]==true
          &&pixelGrid[x][y][z-1]==true && pixelGrid[x][y][z+1]==true){
            pixelGridHollow[x][y][z]=false; //then that pixel shall be false
          }//end if
          else{
            pixelGridHollow[x][y][z]=true; //else the pixel will remain true
          }//end else
        }//end if
```

```
        else{
          pixelGridHollow[x][y][z]=false; //if the pixel is not true, it will be false
        }//end else

      }//end for
    }//end for
  }//end for
}//end hollowOutGrid


//setup the PShape for the Chromosome
void setupChromosome(){

  //chromosome is a group of voxel
  chromosome = createShape(GROUP);

  //for every slice
  for (int z=0; z<nSlices; z++){
    //for every pixel
    for (int y=0; y<imageHeight; y++){
      for (int x=0; x<imageWidth; x++){

        if (pixelGridHollow[x][y][z]==true){ //if the pixelGrid contain true
          //draw box at corresponding co-od
          PShape voxel;
          voxel = createShape(BOX,5);
          voxel.setFill(color(0)); //fill black
          voxel.setStroke(color(0,255,0)); //stroke green
          voxel.translate(5*(x-imageWidth/2),5*(z-nSlices/2),5*(y-imageHeight/2)); //translate voxel
          chromosome.addChild(voxel); //add voxel to chromosome
        }//end if
      }//end for
    }//end for
  }//end for
}//end setUp chromosome
```

3D RENDER OF A CHROMOSOME USING THE MARCHING CUBES ALGORTHIM

Main.pde (functions and procedures)

```
//import PeasyCam
import peasy.*;

//GLOBAL VARIABLES
PeasyCam cam;
Material chromosome; //PShape for chromosome

//SETUP
void setup(){
  size(1024,768,P3D);
  //make a new camera and set distance
  cam = new PeasyCam(this, 500);

  //image scale by 0.4

  //set up materials
  chromosome = new Material(58,"Z",true);
  chromosome.setStroke(false);
  chromosome.setFill(true);
  chromosome.saveShape();
}//end setup


//DRAW
void draw(){
  background(0); //set background to black
  scale(1,20.0/11.0,1); //scale the chromosome
  lighting(); //set up the lights
  shape(chromosome.getShape()); //draw the chromosome
  println(frameRate); //print the frame rate
}//end draw


//lighting
void lighting(){
```

```
  ambientLight(50,50,50);
  directionalLight(0,128,0,1,-1,0); //green light
  directionalLight(0,0,128,-1,-1,0); //blue light
}//end lighting()
```

Lookup.pde (functions and procedures)

```
//GET EDGE COORDINATES
//Converts the edge index to a position vector (returned)
PVector getEdgeCo(int index){
  switch(index){
    case 0: return new PVector(0.0,5.0,-5.0);//0
    case 1: return new PVector(5.0,5.0,0.0);//1
    case 2: return new PVector(0.0,5.0,5.0);//2
    case 3: return new PVector(-5.0,5.0,0.0);//3
    case 4: return new PVector(0.0,-5.0,-5.0);//4
    case 5: return new PVector(5.0,-5.0,0.0);//5
    case 6: return new PVector(0.0,-5.0,5.0);//6
    case 7: return new PVector(-5.0,-5.0,0.0);//7
    case 8: return new PVector(-5.0,0.0,-5.0);//8
    case 9: return new PVector(5.0,0.0,-5.0);//9
    case 10: return new PVector(5.0,0.0,5.0);//10
    case 11: return new PVector(-5.0,0.0,5.0);//11
    default: return new PVector(0.0,0.0,0.0);
  }//end switch
}//end getEdgeCo


//FUNCTION: GET EDGES
//EXTRACTED FROM: http://paulbourke.net/geometry/polygonise/
//Polygonising a scalar field
//return a row from the lookup table for which 3 edges the triangles&apos; 3 vertices should be on based on the inde
x
//the index shows which vertices has a voxel
//a -1 will indicate there are no more trianlges to be rendered
int[] getEdges (int index){
  return edgeLookup[index];
}//end get edges

int [][] edgeLookup =
  {{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  {0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
  {1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
```

```
{1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1},
{3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 11, 2, 8, 11, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 2, 1, 9, 11, 9, 8, 11, -1, -1, -1, -1, -1, -1, -1},
{3, 10, 1, 11, 10, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 10, 1, 0, 8, 10, 8, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{3, 9, 0, 3, 11, 9, 11, 10, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 3, 0, 7, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 1, 9, 4, 7, 1, 7, 3, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 4, 7, 3, 0, 4, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 9, 0, 2, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
{2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, -1, -1, -1, -1},
{8, 4, 7, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 4, 7, 11, 2, 4, 2, 0, 4, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 1, 8, 4, 7, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
{4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, -1, -1, -1, -1},
{3, 10, 1, 3, 11, 10, 7, 8, 4, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, -1, -1, -1, -1},
{4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, -1, -1, -1, -1},
{4, 7, 11, 4, 11, 9, 9, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 5, 4, 8, 3, 5, 3, 1, 5, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 1, 2, 10, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
{5, 2, 10, 5, 4, 2, 4, 0, 2, -1, -1, -1, -1, -1, -1, -1},
{2, 10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, -1, -1, -1, -1},
{9, 5, 4, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 11, 2, 0, 8, 11, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
{0, 5, 4, 0, 1, 5, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
```

```
{2, 1, 5, 2, 5, 8, 2, 8, 11, 4, 8, 5, -1, -1, -1, -1},
{10, 3, 11, 10, 1, 3, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 5, 0, 8, 1, 8, 10, 1, 8, 11, 10, -1, -1, -1, -1},
{5, 4, 0, 5, 0, 11, 5, 11, 10, 11, 0, 3, -1, -1, -1, -1},
{5, 4, 8, 5, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1},
{9, 7, 8, 5, 7, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 3, 0, 9, 5, 3, 5, 7, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 7, 8, 0, 1, 7, 1, 5, 7, -1, -1, -1, -1, -1, -1, -1},
{1, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 7, 8, 9, 5, 7, 10, 1, 2, -1, -1, -1, -1, -1, -1, -1},
{10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, -1, -1, -1, -1},
{8, 0, 2, 8, 2, 5, 8, 5, 7, 10, 5, 2, -1, -1, -1, -1},
{2, 10, 5, 2, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1},
{7, 9, 5, 7, 8, 9, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, -1, -1, -1, -1},
{2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, -1, -1, -1, -1},
{11, 2, 1, 11, 1, 7, 7, 1, 5, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, -1, -1, -1, -1},
{5, 7, 0, 5, 0, 9, 7, 11, 0, 1, 0, 10, 11, 10, 0, -1},
{11, 10, 0, 11, 0, 3, 10, 5, 0, 8, 0, 7, 5, 7, 0, -1},
{11, 10, 5, 7, 11, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 1, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 1, 9, 8, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 5, 2, 6, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 5, 1, 2, 6, 3, 0, 8, -1, -1, -1, -1, -1, -1, -1},
{9, 6, 5, 9, 0, 6, 0, 2, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, -1, -1, -1, -1},
{2, 3, 11, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 0, 8, 11, 2, 0, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 1, 9, 2, 9, 11, 2, 9, 8, 11, -1, -1, -1, -1},
{6, 3, 11, 6, 5, 3, 5, 1, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 11, 0, 11, 5, 0, 5, 1, 5, 11, 6, -1, -1, -1, -1},
{3, 11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, -1, -1, -1, -1},
{6, 5, 9, 6, 9, 11, 11, 9, 8, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 3, 0, 4, 7, 3, 6, 5, 10, -1, -1, -1, -1, -1, -1, -1},
```

```
{1, 9, 0, 5, 10, 6, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
{10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, -1, -1, -1, -1},
{6, 1, 2, 6, 5, 1, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, -1, -1, -1, -1},
{8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, -1, -1, -1, -1},
{7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, -1},
{3, 11, 2, 7, 8, 4, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 4, 7, 2, 4, 2, 0, 2, 7, 11, -1, -1, -1, -1},
{0, 1, 9, 4, 7, 8, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1},
{9, 2, 1, 9, 11, 2, 9, 4, 11, 7, 11, 4, 5, 10, 6, -1},
{8, 4, 7, 3, 11, 5, 3, 5, 1, 5, 11, 6, -1, -1, -1, -1},
{5, 1, 11, 5, 11, 6, 1, 0, 11, 7, 11, 4, 0, 4, 11, -1},
{0, 5, 9, 0, 6, 5, 0, 3, 6, 11, 6, 3, 8, 4, 7, -1},
{6, 5, 9, 6, 9, 11, 4, 7, 9, 7, 11, 9, -1, -1, -1, -1},
{10, 4, 9, 6, 4, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 10, 6, 4, 9, 10, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1},
{10, 0, 1, 10, 6, 0, 6, 4, 0, -1, -1, -1, -1, -1, -1, -1},
{8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1, 10, -1, -1, -1, -1},
{1, 4, 9, 1, 2, 4, 2, 6, 4, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, -1, -1, -1, -1},
{0, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 3, 2, 8, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1},
{10, 4, 9, 10, 6, 4, 11, 2, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 2, 2, 8, 11, 4, 9, 10, 4, 10, 6, -1, -1, -1, -1},
{3, 11, 2, 0, 1, 6, 0, 6, 4, 6, 1, 10, -1, -1, -1, -1},
{6, 4, 1, 6, 1, 10, 4, 8, 1, 2, 1, 11, 8, 11, 1, -1},
{9, 6, 4, 9, 3, 6, 9, 1, 3, 11, 6, 3, -1, -1, -1, -1},
{8, 11, 1, 8, 1, 0, 11, 6, 1, 9, 1, 4, 6, 4, 1, -1},
{3, 11, 6, 3, 6, 0, 0, 6, 4, -1, -1, -1, -1, -1, -1, -1},
{6, 4, 8, 11, 6, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 10, 6, 7, 8, 10, 8, 9, 10, -1, -1, -1, -1, -1, -1, -1},
{0, 7, 3, 0, 10, 7, 0, 9, 10, 6, 7, 10, -1, -1, -1, -1},
{10, 6, 7, 1, 10, 7, 1, 7, 8, 1, 8, 0, -1, -1, -1, -1},
{10, 6, 7, 10, 7, 1, 1, 7, 3, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, -1, -1, -1, -1},
{2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, -1},
{7, 8, 0, 7, 0, 6, 6, 0, 2, -1, -1, -1, -1, -1, -1, -1},
{7, 3, 2, 6, 7, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 11, 10, 6, 8, 10, 8, 9, 8, 6, 7, -1, -1, -1, -1},
```

```
{2, 0, 7, 2, 7, 11, 0, 9, 7, 6, 7, 10, 9, 10, 7, -1},
{1, 8, 0, 1, 7, 8, 1, 10, 7, 6, 7, 10, 2, 3, 11, -1},
{11, 2, 1, 11, 1, 7, 10, 6, 1, 6, 7, 1, -1, -1, -1, -1},
{8, 9, 6, 8, 6, 7, 9, 1, 6, 11, 6, 3, 1, 3, 6, -1},
{0, 9, 1, 11, 6, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 8, 0, 7, 0, 6, 3, 11, 0, 11, 6, 0, -1, -1, -1, -1},
{7, 11, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 1, 9, 8, 3, 1, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
{10, 1, 2, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 3, 0, 8, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
{2, 9, 0, 2, 10, 9, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
{6, 11, 7, 2, 10, 3, 10, 8, 3, 10, 9, 8, -1, -1, -1, -1},
{7, 2, 3, 6, 2, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 0, 8, 7, 6, 0, 6, 2, 0, -1, -1, -1, -1, -1, -1, -1},
{2, 7, 6, 2, 3, 7, 0, 1, 9, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, -1, -1, -1, -1},
{10, 7, 6, 10, 1, 7, 1, 3, 7, -1, -1, -1, -1, -1, -1, -1},
{10, 7, 6, 1, 7, 10, 1, 8, 7, 1, 0, 8, -1, -1, -1, -1},
{0, 3, 7, 0, 7, 10, 0, 10, 9, 6, 10, 7, -1, -1, -1, -1},
{7, 6, 10, 7, 10, 8, 8, 10, 9, -1, -1, -1, -1, -1, -1, -1},
{6, 8, 4, 11, 8, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 6, 11, 3, 0, 6, 0, 4, 6, -1, -1, -1, -1, -1, -1, -1},
{8, 6, 11, 8, 4, 6, 9, 0, 1, -1, -1, -1, -1, -1, -1, -1},
{9, 4, 6, 9, 6, 3, 9, 3, 1, 11, 3, 6, -1, -1, -1, -1},
{6, 8, 4, 6, 11, 8, 2, 10, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 3, 0, 11, 0, 6, 11, 0, 4, 6, -1, -1, -1, -1},
{4, 11, 8, 4, 6, 11, 0, 2, 9, 2, 10, 9, -1, -1, -1, -1},
{10, 9, 3, 10, 3, 2, 9, 4, 3, 11, 3, 6, 4, 6, 3, -1},
{8, 2, 3, 8, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1},
{0, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, -1, -1, -1, -1},
{1, 9, 4, 1, 4, 2, 2, 4, 6, -1, -1, -1, -1, -1, -1, -1},
{8, 1, 3, 8, 6, 1, 8, 4, 6, 6, 10, 1, -1, -1, -1, -1},
{10, 1, 0, 10, 0, 6, 6, 0, 4, -1, -1, -1, -1, -1, -1, -1},
{4, 6, 3, 4, 3, 8, 6, 10, 3, 0, 3, 9, 10, 9, 3, -1},
{10, 9, 4, 6, 10, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
```

```
{4, 9, 5, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 4, 9, 5, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 0, 1, 5, 4, 0, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
{11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, -1, -1, -1, -1},
{9, 5, 4, 10, 1, 2, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
{6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5, -1, -1, -1, -1},
{7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2, -1, -1, -1, -1},
{3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6, -1},
{7, 2, 3, 7, 6, 2, 5, 4, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, -1, -1, -1, -1},
{3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, -1, -1, -1, -1},
{6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, -1},
{9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7, -1, -1, -1, -1},
{1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, -1},
{4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10, -1},
{7, 6, 10, 7, 10, 8, 5, 4, 10, 4, 8, 10, -1, -1, -1, -1},
{6, 9, 5, 6, 11, 9, 11, 8, 9, -1, -1, -1, -1, -1, -1, -1},
{3, 6, 11, 0, 6, 3, 0, 5, 6, 0, 9, 5, -1, -1, -1, -1},
{0, 11, 8, 0, 5, 11, 0, 1, 5, 5, 6, 11, -1, -1, -1, -1},
{6, 11, 3, 6, 3, 5, 5, 3, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 9, 5, 11, 9, 11, 8, 11, 5, 6, -1, -1, -1, -1},
{0, 11, 3, 0, 6, 11, 0, 9, 6, 5, 6, 9, 1, 2, 10, -1},
{11, 8, 5, 11, 5, 6, 8, 0, 5, 10, 5, 2, 0, 2, 5, -1},
{6, 11, 3, 6, 3, 5, 2, 10, 3, 10, 5, 3, -1, -1, -1, -1},
{5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, -1, -1, -1, -1},
{9, 5, 6, 9, 6, 0, 0, 6, 2, -1, -1, -1, -1, -1, -1, -1},
{1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, -1},
{1, 5, 6, 2, 1, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 3, 6, 1, 6, 10, 3, 8, 6, 5, 6, 9, 8, 9, 6, -1},
{10, 1, 0, 10, 0, 6, 9, 5, 0, 5, 6, 0, -1, -1, -1, -1},
{0, 3, 8, 5, 6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{10, 5, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 5, 10, 7, 5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 5, 10, 11, 7, 5, 8, 3, 0, -1, -1, -1, -1, -1, -1, -1},
{5, 11, 7, 5, 10, 11, 1, 9, 0, -1, -1, -1, -1, -1, -1, -1},
{10, 7, 5, 10, 11, 7, 9, 8, 1, 8, 3, 1, -1, -1, -1, -1},
{11, 1, 2, 11, 7, 1, 7, 5, 1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2, 11, -1, -1, -1, -1},
{9, 7, 5, 9, 2, 7, 9, 0, 2, 2, 11, 7, -1, -1, -1, -1},
```

```
{7, 5, 2, 7, 2, 11, 5, 9, 2, 3, 2, 8, 9, 8, 2, -1},
{2, 5, 10, 2, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1},
{8, 2, 0, 8, 5, 2, 8, 7, 5, 10, 2, 5, -1, -1, -1, -1},
{9, 0, 1, 5, 10, 3, 5, 3, 7, 3, 10, 2, -1, -1, -1, -1},
{9, 8, 2, 9, 2, 1, 8, 7, 2, 10, 2, 5, 7, 5, 2, -1},
{1, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 7, 0, 7, 1, 1, 7, 5, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 3, 9, 3, 5, 5, 3, 7, -1, -1, -1, -1, -1, -1, -1},
{9, 8, 7, 5, 9, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{5, 8, 4, 5, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1},
{5, 0, 4, 5, 11, 0, 5, 10, 11, 11, 3, 0, -1, -1, -1, -1},
{0, 1, 9, 8, 4, 10, 8, 10, 11, 10, 4, 5, -1, -1, -1, -1},
{10, 11, 4, 10, 4, 5, 11, 3, 4, 9, 4, 1, 3, 1, 4, -1},
{2, 5, 1, 2, 8, 5, 2, 11, 8, 4, 5, 8, -1, -1, -1, -1},
{0, 4, 11, 0, 11, 3, 4, 5, 11, 2, 11, 1, 5, 1, 11, -1},
{0, 2, 5, 0, 5, 9, 2, 11, 5, 4, 5, 8, 11, 8, 5, -1},
{9, 4, 5, 2, 11, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 5, 10, 3, 5, 2, 3, 4, 5, 3, 8, 4, -1, -1, -1, -1},
{5, 10, 2, 5, 2, 4, 4, 2, 0, -1, -1, -1, -1, -1, -1, -1},
{3, 10, 2, 3, 5, 10, 3, 8, 5, 4, 5, 8, 0, 1, 9, -1},
{5, 10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, -1, -1, -1, -1},
{8, 4, 5, 8, 5, 3, 3, 5, 1, -1, -1, -1, -1, -1, -1, -1},
{0, 4, 5, 1, 0, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, -1, -1, -1, -1},
{9, 4, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 11, 7, 4, 9, 11, 9, 10, 11, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 4, 9, 7, 9, 11, 7, 9, 10, 11, -1, -1, -1, -1},
{1, 10, 11, 1, 11, 4, 1, 4, 0, 7, 4, 11, -1, -1, -1, -1},
{3, 1, 4, 3, 4, 8, 1, 10, 4, 7, 4, 11, 10, 11, 4, -1},
{4, 11, 7, 9, 11, 4, 9, 2, 11, 9, 1, 2, -1, -1, -1, -1},
{9, 7, 4, 9, 11, 7, 9, 1, 11, 2, 11, 1, 0, 8, 3, -1},
{11, 7, 4, 11, 4, 2, 2, 4, 0, -1, -1, -1, -1, -1, -1, -1},
{11, 7, 4, 11, 4, 2, 8, 3, 4, 3, 2, 4, -1, -1, -1, -1},
{2, 9, 10, 2, 7, 9, 2, 3, 7, 7, 4, 9, -1, -1, -1, -1},
{9, 10, 7, 9, 7, 4, 10, 2, 7, 8, 7, 0, 2, 0, 7, -1},
{3, 7, 10, 3, 10, 2, 7, 4, 10, 1, 10, 0, 4, 0, 10, -1},
{1, 10, 2, 8, 7, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 1, 4, 1, 7, 7, 1, 3, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, -1, -1, -1, -1},
```

```
{4, 0, 3, 7, 4, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 8, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 9, 3, 9, 11, 11, 9, 10, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 10, 0, 10, 8, 8, 10, 11, -1, -1, -1, -1, -1, -1, -1},
{3, 1, 10, 11, 3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 11, 1, 11, 9, 9, 11, 8, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 9, 3, 9, 11, 1, 2, 9, 2, 11, 9, -1, -1, -1, -1},
{0, 2, 11, 8, 0, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 8, 2, 8, 10, 10, 8, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 10, 2, 0, 9, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 8, 2, 8, 10, 0, 1, 8, 1, 10, 8, -1, -1, -1, -1},
{1, 10, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 3, 8, 9, 1, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 9, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 3, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}};
```

Material.pde (class)

```
class Material{
  //contains an array of marching cubes, use getShape to get a PShape of all the polygons of the marching cubes

  //MEMBER VARIABLES

  PShape shape; //PShape
  Cube [][][] cubeArray; //array of  marching cubes
  int imageWidth,imageHeight,nSlices; //dimensions of the stack


  //CONSTRUCTOR
  Material(int nSlices, String filename, boolean wantAverageNormal){

    //LOAD IMAGES--------------------------
    //create an arraylist of PImages
    ArrayList <PImage> imageArray = new ArrayList<PImage>();

    //load all the images and add them in imageArray
    for (int i=0; i<10; i++){
      imageArray.add(loadImage(filename+"0"+i+".png")); //load image
    }//end for
    for (int i=10; i<nSlices; i++){
      imageArray.add(loadImage(filename+i+".png")); //load image
    }//end for

    //get imageWidth, imageHeight, nSlices
    this.imageWidth = imageArray.get(0).width;
    this.imageHeight = imageArray.get(0).height;
    this.nSlices = nSlices;

    PVector center = new PVector(imageWidth/2,imageHeight/2,nSlices/2); //work out the center of the image


    //CREATE ARRAY OF MARCHINGS CUBES---------

    //create the pixelGrid and cubeArray of size imageWidth+2 x imageHeight+2 x nSlices+2
    //fill pixelGrid with all entries as false
```

```
//fill cubeArray with cube with the position relative to the center
boolean[][][] pixelGrid = new boolean[imageWidth+2][imageHeight+2][nSlices+2];
cubeArray = new Cube [imageWidth+2][imageHeight+2][nSlices+2];
//fill the pixelGrid with falses
for (int i=0; i<imageWidth+2; i++){
  for (int j=0; j<imageHeight+2; j++){
    for (int k=0; k<nSlices+2; k++){
      pixelGrid[i][j][k] = false;
      cubeArray[i][j][k] = new Cube(10*(i-center.x),10*(k-center.z),10*(j-center.y));
    }//end for
  }//end for
}//end for

//for every image in imageArray
for (int z=0; z<nSlices; z++){
  PImage slice = imageArray.get(z);
  //load the pixels in the image
  slice.loadPixels();

  //declare x and y co-od
  int x = 0;
  int y = 0;
  //for every pixel in the image
  for (int i=0; i<slice.pixels.length; i++){
    if (slice.pixels[i]==color(255)){ //if that pixel is white
      pixelGrid[x+1][y+1][z+1] = true; //put true in the pixelGrid
    }//end if
    //increase x and y co-od according
    x++;
    if (x==imageWidth){
      x=0;
      y++;
    }//end if
  }//end for
}//end for


//SETUP PSHAPE-------------------------------------------------------------------------------
//set the shape is a group of PShapes
```

```
    shape = createShape(GROUP);

    //for every slice
    for (int z=0; z<=nSlices; z++){

      //for every pixel in the slice
      for (int y=0; y<=imageHeight; y++){
        for (int x=0; x<=imageWidth; x++){

          //declare a variable called verticesIndex
          int verticesIndex = 0; //this variable indicates which vertices has a true value in the pixelGrid as an 8
bit binary number
          //each digit in the binary number represent a vertex

          //work out verticesIndex by analyzing each corner
          if(pixelGrid[x][y+1][z]==true){ //0
            verticesIndex += 1;
          }//end if
          if(pixelGrid[x+1][y+1][z]==true){ //1
            verticesIndex += 2;
          }//end if
          if(pixelGrid[x+1][y+1][z+1]==true){ //2
            verticesIndex += 4;
          }//end if
          if(pixelGrid[x][y+1][z+1]==true){ //3
            verticesIndex += 8;
          }//end if
          if(pixelGrid[x][y][z]==true){ //4
            verticesIndex += 16;
          }//end if
          if(pixelGrid[x+1][y][z]==true){ //5
            verticesIndex += 32;
          }//end if
          if(pixelGrid[x+1][y][z+1]==true){ //6
            verticesIndex += 64;
          }//end if
          if(pixelGrid[x][y][z+1]==true){ //7
            verticesIndex += 128;
          }//end if
```

```
        //look up what polygon to draw using vertices and draw it at x,y,z
        drawPolygon(x,y,z,verticesIndex);

      }//end for
    }//end for

    //tell the user the progress of rendering each slice
    println("Analyzing "+filename+" images: "+((float)z)*100/((float)nSlices)+"%");

  }//end for

  //if we want averaging the normals
  if (wantAverageNormal){
    //for every slice
    for (int z=0; z<=nSlices; z++){
      //for every pixel in the slice
      for (int y=0; y<=imageHeight; y++){
        for (int x=0; x<=imageWidth; x++){
          //recalculate the normal in the cube
          cubeArray[x][y][z].recalculateNormal(x,y,z,imageWidth,imageHeight,nSlices,cubeArray);
          //add the cube shape to the shape
        }//end for
      }//end for
      //tell the user the progress of rendering each slice
      println("Calculating "+filename+" normals: "+((float)z)*100/((float)nSlices)+"%");
    }//end for
  }//end if
}//end constructor


//METHODS


//saveShape
public void saveShape(){
  shape = null;
  shape = createShape(GROUP);
  for (int z=0; z<=nSlices; z++){
```

```
      //for every pixel in the slice
      for (int y=0; y<=imageHeight; y++){
        for (int x=0; x<=imageWidth; x++){
          shape.addChild(cubeArray[x][y][z].getShape());
        }//end for
      }//end for
    }//end for
  }//end saveShape


  //getShape
  public PShape getShape(){
    return shape;
  }//end getShape


  //drawPolygon at (x,y,z) by looking up what triangles to draw using verticesIndex
  private void drawPolygon(int x, int y, int z, int verticesIndex){
    //make an array of edgeIndexs from the lookup table
    //this contains which 3 edges the 3 vertices of the triangles are on, each group of 3 entries are for each
triangle
    //the edgeIndexArray will contain a -1 to indicate when there are no more triangles to be drawn
    int [] edgeIndexArray = getEdges(verticesIndex);

    //pointer points to entries in edgesArray
    int pointer = 0;

    boolean trianglesToRender = true; //boolean to indicate when there is a trianle to render

    //while there are triangles to render
    while (trianglesToRender){
      if (edgeIndexArray[pointer]!=-1){ //if the edgeIndex is not -1
        //add a new triangle with 3 vertices on the 3 edges in the edgeIndexArray to the cube
        cubeArray[x][y][z].addTriangle(edgeIndexArray[pointer], edgeIndexArray[pointer+1], edgeIndexArray[pointer+2]
);
        //increase the pointer by 3
        pointer += 3;
      }//end if
      else{ //else there are no more trianlges to render
```

```
      trianglesToRender = false;
    }//end else
  }//end while
 }//end drawPolygon


//COLOUR METHODS
 public void setStroke(boolean wantStroke){
   for (int z=0; z<=nSlices; z++){
     //for every pixel in the slice
     for (int y=0; y<=imageHeight; y++){
       for (int x=0; x<=imageWidth; x++){
         cubeArray[x][y][z].setStroke(wantStroke);
       }//end for
     }//end for
   }//end for
 }//end stroke

 public void setFill(boolean wantFill){
   for (int z=0; z<=nSlices; z++){
     //for every pixel in the slice
     for (int y=0; y<=imageHeight; y++){
       for (int x=0; x<=imageWidth; x++){
         cubeArray[x][y][z].setFill(wantFill);
       }//end for
     }//end for
   }//end for
 }//end fill

 public void setStroke(color colour){
   for (int z=0; z<=nSlices; z++){
     //for every pixel in the slice
     for (int y=0; y<=imageHeight; y++){
       for (int x=0; x<=imageWidth; x++){
         cubeArray[x][y][z].setStroke(colour);
       }//end for
     }//end for
   }//end for
 }//end fill
```

```
  public void setFill(color colour){
    for (int z=0; z<=nSlices; z++){
      //for every pixel in the slice
      for (int y=0; y<=imageHeight; y++){
        for (int x=0; x<=imageWidth; x++){
          cubeArray[x][y][z].setFill(colour);
        }//end for
      }//end for
    }//end for
  }//end fill

}//end class
```

Cube.pde (class)

```
class Cube{

  //CUBE CLASS
  //A marching cube analyse 4 voxels and store/draw triangles according to the state of the voxels
  //The marching cube contains triangles to represent the surface of the chromosome according if the voxel is white
or not.


  //MEMBER VARIABLES
  private PVector position; //the position of the cube
  private ArrayList<Triangle> triangleArray; //arraylist of Triangles


  //CONSTRUCTOR
  //Position of the cube (x,y,z)
  Cube(float x, float y, float z){
    position = new PVector(x,y,z);
    triangleArray = new ArrayList<Triangle>();
  }//end constructor


  //METHODS

  //addTriangle to the marching cube
  public void addTriangle(int index1, int index2, int index3){
    //add triangle object to the triangle arraylist
    triangleArray.add(new Triangle(index1, index2, index3, this.position.x, this.position.y, this.position.z));
  }//add triangle


  //get the PShape for all the triangles in the cube
  public PShape getShape(){
    //create a PShape for the marching cube
    PShape cube;
    cube = createShape(GROUP);
    //for every trianlge in the arraylist
    for (int i=0; i<triangleArray.size(); i++){
```

```
      //add the triangle to the cube PShape
      cube.addChild(triangleArray.get(i).getShape());
    }//end for
    //translate the cube according to its position and return it
    cube.translate(position.x, position.y, position.z);
    return cube;
  }//end getShape


  //get the array of triangles
  public ArrayList<Triangle> getTriangleArray(){
    return triangleArray;
  }//end getTriangleArray


  //recalculate the normals of each triangle in the marching cube
  public void recalculateNormal(int x, int y, int z, int imageWidth, int imageHeight, int nSlices, Cube[][][]
cubeArray){
    //recalculate normals for the triangles by taking the average of the normals of triangles sharing its vertices

    //set the boundary on which neighbouring cubes to analyze
    int xLower,xUpper, yLower, yUpper, zLower, zUpper;
    xLower = -1;
    yLower = -1;
    zLower = -1;

    xUpper = 1;
    yUpper = 1;
    zUpper = 1;

    //if the marching cube is on the boundary, than it has less neighbours
    if (x==0){
      xLower = 0;
    }//end if
    else if (x==imageWidth){
      xUpper = 0;
    }//end elif

    if (y==0){
```

```
  yLower = 0;
}//end if
else if (y==imageHeight){
  yUpper = 0;
}//end elif

if (z==0){
  zLower = 0;
}//end if
else if (z==nSlices){
  zUpper = 0;
}//end elif

//for each triangle in the cube
for (int i=0; i<triangleArray.size(); i++){
  //prepare this triangle and declare statististical variables
  Triangle triangle = triangleArray.get(i);
  PVector nBar; //average normal vector
  nBar = new PVector(0,0,0);
  float weight = 0; //the sum of all weights used for averaging

  //compare this triangle to the traingles in neighbouring cubes
  //for every neighbouring cube
  for (int u=xLower; u<=xUpper; u++){
    for (int v=yLower; v<=yUpper; v++){
      for (int w=zLower; w<=zUpper; w++){

        //get all the triangles in the neighbouring cube
        ArrayList<Triangle> thatTriangleArray = cubeArray[x+u][y+v][z+w].getTriangleArray();

        //for every triangle in the cube
        for (int j=0; j<thatTriangleArray.size(); j++){
          Triangle that = thatTriangleArray.get(j);
          //if this triangle share a vertex with that triangle
          if(triangle.isShareVertex(that)){
            //update the normal statisticts with the reciprocal area as the weight
            nBar.add(PVector.mult(that.getNormal(),1/that.getArea()));
            weight += 1/that.getArea();
          }//end if
```

```
            }//end for
          }//end for
        }//end for
      }//end for
      //work out the mean normal and update that triangle's normal
      nBar.mult(1/weight);
      triangle.newNormal(nBar);
    }//end for
  }//end recalculateNormal

  //COLOUR METHODS
  public void setStroke(boolean wantStroke){
    for (int i=0; i<triangleArray.size(); i++){
      triangleArray.get(i).setStroke(wantStroke);
    }//end for
  }//end stroke

  public void setFill(boolean wantFill){
    for (int i=0; i<triangleArray.size(); i++){
      triangleArray.get(i).setFill(wantFill);
    }//end for
  }//end fill

  public void setStroke(color colour){
    for (int i=0; i<triangleArray.size(); i++){
      triangleArray.get(i).setStroke(colour);
    }//end for
  }//end stroke

  public void setFill(color colour){
   for (int i=0; i<triangleArray.size(); i++){
      triangleArray.get(i).setFill(colour);
    }//end for
  }//end fill

}//end class
```

Triangle.pde (class)

```
class Triangle{
  //TRIANGLE CLASS
  //An instance from this class can return a triangle PShape according to the 3 edgeIndexes inputted in the
constructor

  //MEMBER VARIABLES
  private PVector a,b,c; //position vectors of the vertices of the triangle relative to cube_center
  private PVector cube_center; //the co-od of the center of the cube
  private PVector n; //normal of the triangle
  private PVector nBar; //the mean normal of the trianle
  private float area; //the area of the triangle
  private boolean wantStroke, wantFill;
  private color stroke, fill;

  //CONSTRUCTOR
  //Parameters: 3 edge indexes, which edges the vertices of the triangle should be on
  Triangle(int index1, int index2, int index3, float x, float y, float z){

    //convert the edge indexes to position vectors
    a = getEdgeCo(index1);
    b = getEdgeCo(index2);
    c = getEdgeCo(index3);

    //get the position vector of the center
    this.cube_center = new PVector(x,y,z);

    //work out the normal vector(n) and the area
    PVector r1,r2;
    r1 = PVector.sub(b,a);
    r2 = PVector.sub(b,c);
    n = r1.cross(r2);
    area = n.mag()/2;
    n.normalize();
    nBar = n; //nBar will be the averaged normal later on

    //default colouring
    wantStroke = false; //no stroke
```

```
  wantFill = true; //fill
  stroke = color(0,255,0); //green stroke
  fill = color(255); //white fill
}//end constructor


//METHODS


//getShape of the triangle
public PShape getShape(){

  //create a PShape
  PShape triangle;
  triangle = createShape();

  //set the colour of the stroke
  if(wantStroke){
    triangle.setStroke(stroke);
  }//end if
  else{
    triangle.setStroke(false);
  }//end else

  //set the colour of the fill
  if(wantFill){
    triangle.setFill(fill);
  }//end if
  else{
    triangle.setFill(false);
  }//end else

  //draw the triangle and return it
  triangle.beginShape();
  triangle.normal(nBar.x,nBar.z,nBar.y);
  triangle.vertex(a.x,a.z,a.y);
  triangle.vertex(b.x,b.z,b.y);
  triangle.vertex(c.x,c.z,c.y);
  triangle.vertex(a.x,a.z,a.y);
```

```
  triangle.endShape(CLOSE);
  return triangle;
}//end display


//get the normal vector
public PVector getNormal(){
  return n;
}//end

//update nBar, the averaged normal
public void newNormal(PVector normal){
  nBar = normal;
  nBar.normalize();
}//end newNormal


//GET AREA of triangle
public float getArea(){
  return area;
}//end getArea


//get array of vertices vectors of this triangle
public PVector[] getVerticesVectors(){
  PVector [] verticesArray;
  verticesArray = new PVector[3];
  verticesArray[0] = PVector.add(cube_center,a);
  verticesArray[1] = PVector.add(cube_center,b);
  verticesArray[2] = PVector.add(cube_center,c);
  return verticesArray;
}//end getVertices


//IsShareVertex
//does this traingle share a vertex with that triangle?
public boolean isShareVertex(Triangle that){
  boolean share = false; //assume false
```

```
  //get the 2 triangles vertices vectors
  PVector [] vertexArray1, vertexArray2;
  vertexArray1 = this.getVerticesVectors();
  vertexArray2 = that.getVerticesVectors();

  PVector r1, r2;

  //for every vertex
  for (int i=0; i<3; i++){
    //compare this vertex
    r1 = vertexArray1[i];
    //for every that vertex
    for (int j=0; j<3; j++){
      r2 = vertexArray2[j];
      //if the cood of both vertex is the same, then they share the same vertex
      if(r1.x==r2.x && r1.y==r2.y && r1.z==r2.z){
        share = true;
        //break the loop
        j=3;
        i=3;
      }//end if
    }//end for
  }//end for
  return share;
}//end


//COLOUR METHODS
public void setStroke(boolean wantStroke){
  this.wantStroke = wantStroke;
}//end

public void setFill(boolean wantFill){
  this.wantFill = wantFill;
}//end

public void setStroke(color colour){
  this.stroke = colour;
  this.wantStroke = true;
```

```
  }//end

  public void setFill(color colour){
    this.fill = colour;
    this.wantFill = true;
  }//end

}//end class
```

MAXIMA FINDER BY USING NOISE TOLERANCES WHICH MAXIMIZES CHI-SQUARED

```
import ij.*;
import ij.process.*;
import ij.measure.*;
import ij.gui.*;
import ij.plugin.filter.*;
import java.lang.*;
import java.util.*;
import java.awt.*;

public class MaximaAnalysisStack_Plugin implements PlugInFilter {
	ImagePlus imp;

	//GLOBAL VARAIBLES
	private int background; //binary value of a background pixel
	private int width,height,nSlices; //dimensions of the image
	private int numberOfSimulations; //number of simulated images to be produced per slice
	private int maximumNoiseTolerance; //the maximum noise tolerance setting for the maxima function
	private double t; //t statistics

	//GLOBAL OBJECTS
	Random random = new Random(); //random number generator
	MaximumFinder maximum_finder = new MaximumFinder(); //maxima_finder object: finds the maximas of an image

	//SETUP
	public int setup(String arg, ImagePlus imp) {
		this.imp = imp;

		//set global variables
		width = imp.getDimensions()[0];
		height = imp.getDimensions()[1];
		nSlices = imp.getDimensions()[3];
		background = 0;
		maximumNoiseTolerance = 5000;

		//t-statistics
		numberOfSimulations = 10;
```

```
            t = 2.262; //2.5% percentage point

            return DOES_16; //does only 16 bit images
      }//end setup

      //RUN
      public void run(ImageProcessor image) {
            ImageStack maximaStack = new ImageStack(width,height); //stack of maxima point images
            ByteProcessor sliceMaximas; //imageProcessor object for each slice

            //for every slice
            for (int i=1; i<=nSlices; i++){
                  //obtain the optimized maxima point image of a slice and add it to the stack
                  sliceMaximas = getMaximas(image,i);
                  maximaStack.addSlice(sliceMaximas);
                  //save a copy of the maxima point image
                  IJ.save(new
ImagePlus(String.valueOf(i),sliceMaximas),"C://Users/Sherman/Documents/chromosomes/Z/term1/maxima_analysis/data/"+i+
".tif");
            }//end for

            //create an image of the stack, show it and save it
            ImagePlus result = new ImagePlus("Result",maximaStack);
            result.show();
            IJ.save(result,"C://Users/Sherman/Documents/chromosomes/Z/term1/maxima_analysis/maximas.tif");

      }//end run

      //GET MAXIMAS
      //Returns an image of the maxima points using a noise tolerance setting which maximizes chi-squared
      //Arguments: stack to work on, which slice to work on
      private ByteProcessor getMaximas(ImageProcessor image, int slice){

            //set the slice on the stack
            IJ.setSlice(slice);

            //Create an array of simulated images
            ImageProcessor [] simulationArray = new ImageProcessor[numberOfSimulations];
            for (int i=0; i<numberOfSimulations; i++){
```

```
                simulationArray[i] = simulate(image);
            }//end for

            /*//DEBUG: Show all simulation images
            for (int i=0; i<numberOfSimulations; i++){
                ImagePlus imp = new ImagePlus(String.valueOf(i),simulationArray[i]);
                imp.show();
            }//end for*/

            //create arrays to store data
            double [] noiseTolerance = new double[maximumNoiseTolerance+1]; //noise tolerance settings
            double [] dataCount = new double[maximumNoiseTolerance+1]; //number of maximas in the data
            double [] simulationCount = new double[maximumNoiseTolerance+1]; //average number of maximas in the
simulated images
            double [] simulationCountError = new double[maximumNoiseTolerance+1]; //std of maximas in the simulated
images
            double [] sumSimulation = new double[maximumNoiseTolerance+1]; //sum(x)
            double [] sumSquaresSimulation = new double[maximumNoiseTolerance+1]; //sum(x^2)
            double [] chiSquared = new double[maximumNoiseTolerance+1]; //poison chi-squared
            double [] chiSquaredError = new double[maximumNoiseTolerance+1]; //estimated standard deviation of chi-
squared

            //for every noise tolerance setting, find the number of maximas on the data image and put it in the
array dataCount
            for (int i=0; i<=maximumNoiseTolerance; i++){
                dataCount[i] =  maximum_finder.getMaxima(image,(double)i,false).npoints;
                noiseTolerance[i]=i;
            }//end for

            //for every noise tolerance setting, find the number of maximas on the simulated images and update
statistical arrays
            double value;
            for (int j=0; j<numberOfSimulations; j++){ //repeat simulations for n times
                for (int i=0; i<=maximumNoiseTolerance; i++){
                    value = maximum_finder.getMaxima(simulationArray[j],(double)i,false).npoints;
                    sumSimulation[i] += value;
                    sumSquaresSimulation[i] += value*value;
                }//end for
            }//end for
```

```
            //calculate statistics
            for (int i=0; i<=maximumNoiseTolerance; i++){
                  //mean
                  simulationCount[i] = sumSimulation[i]/numberOfSimulations;
                  //standard deviation
                  simulationCountError[i] = Math.sqrt((sumSquaresSimulation[i]-
Math.pow(sumSimulation[i],2)/numberOfSimulations)/(numberOfSimulations-1));
                  //chi-squared
                  chiSquared[i] = Math.pow(simulationCount[i]-dataCount[i],2)/simulationCount[i];
                  //estimated standard deviation of chi-squared
                  chiSquaredError[i] =
chiSquared[i]*simulationCountError[i]*(simulationCount[i]+dataCount[i])/(simulationCount[i]*Math.abs(simulationCount
[i]-dataCount[i]));
            }//end for

            //develop a curve fitter which fits a curve for noiseTolerance VS chiSquared
            CurveFitter curve = new CurveFitter(noiseTolerance,chiSquared);
            curve.doFit(CurveFitter.POLY8);
            double [] parameters = curve.getParams(); //array of parameters

            //create a new results table which will display statistics
            ResultsTable results = new ResultsTable();

            //declare variables to find the optimal noise tolerance
            double maxChiSquared = 0;
            double optimalNoiseTolerance = 0;
            double f; //value of curve at a certain point

            //get the data from arrays and put it on the ResultsTable...
            //...find the optimal noise tolerance setting...
            //...while there is data in the arrays
            int i;
            do {
                  //increment the counter and store it in i => noise tolerance = i-1
                  results.incrementCounter();
                  i = results.getCounter();

                  //put data into results
```

```
                  results.addValue("Noise tolerance",i-1);
                  results.addValue("Data",dataCount[i-1]);
                  results.addValue("Simulation",simulationCount[i-1]);
                  results.addValue("Standard Deviation",simulationCountError[i-1]);
                  results.addValue("95% confidence error",t*simulationCountError[i-
1]/Math.sqrt(numberOfSimulations));
                  results.addValue("Chi-squared",chiSquared[i-1]);
                  results.addValue("+/-",chiSquaredError[i-1]);
                  f = curve.f(parameters,i-1);
                  results.addValue("Curve fit",f);

                  //update optimalNoiseTolerance if found a new maxChiSquared
                  if (f>maxChiSquared){
                        maxChiSquared = f;
                        optimalNoiseTolerance = i-1;
                  }//end if

            } while(results.getCounter()<=maximumNoiseTolerance);
            //end do-while

            //show results table and save it
            results.addValue("Optimal", optimalNoiseTolerance);
            results.show("Results");
            try{

      results.saveAs("C://Users/Sherman/Documents/chromosomes/Z/term1/maxima_analysis/data/results"+slice+".csv");
            }//end try
            catch(Exception exception){
                  IJ.showStatus("Cannot save results in slice "+slice);
            }//end catch

            //show image of points of maximas
            return
maximum_finder.findMaxima(image,optimalNoiseTolerance,ImageProcessor.NO_THRESHOLD,MaximumFinder.SINGLE_POINTS,false,
false);
      }//end get maximas

      //SIMULATE
      //Creates a Monte Carlo simulated image
```

```
//Each non-background pixel is arranged randomly WITH REPLACEMENT
private ImageProcessor simulate(ImageProcessor image){

	//count number of signal pixels n
	int n=0;
	for (int x=0; x<width; x++){
		for (int y=0; y<height; y++){
			if (image.getPixel(x,y)!=background){
				n++;
			}//end if
		}//end for
	}//end for

	//store pixel values in array of length n
	int [] pixelArray = new int[n]; //array of pixel values in the original image
	int i=0;
	for (int x=0; x<width; x++){
		for (int y=0; y<height; y++){
			if (image.getPixel(x,y)!=background){
				pixelArray[i]=image.getPixel(x,y);
				i++;
			}//end if
		}//end for
	}//end for

	//replace signal pixels with a random signal pixel in the array
	ImageProcessor result = new ShortProcessor(width,height);
	for (int x=0; x<width; x++){
		for (int y=0; y<height; y++){
			//If not background, put random pixel from array
			if (image.getPixel(x,y)!=background){
				result.putPixel(x,y,pixelArray[random.nextInt(n)]);
			}//end if
			//else put a background pixel
			else{
				result.putPixel(x,y,background);
			}//end else
		}//end for
	}//end for
```

```
            //return the image
            return result;
        }//end procedure

}//end class
```

NEAREST NEIGHBOUR SEARCH AND DISTANCE TO CENTER OF MASS

Main.pde (functions and procedures)

```
//SETUP
void setup(){
  //make csv output
  PrintWriter output = createWriter("data.csv");

  float xScale = 11; //11 nm per pixel
  float yScale = 11; //11 nm per pixel
  float zScale = 20; //20 nm per pixel

  int nSlices = 117; //number of slices (117)

  //save slices in array
  PImage [] imageArray;
  imageArray = new PImage[nSlices];
  //for every slice, load and save image in the array
  for (int i=0; i<10; i++){
    imageArray[i] = loadImage("maximas00"+i+".png");
  }//end for
  for (int i=10; i<100; i++){
    imageArray[i] = loadImage("maximas0"+i+".png");
  }//end for
  for (int i=100; i<nSlices; i++){
    imageArray[i] = loadImage("maximas"+i+".png");
  }//end for

  //set size of image
  size(imageArray[0].width,imageArray[0].height);

  ArrayList<PointVector> PointVectorArray = new ArrayList<PointVector>(); //array list of PointVectors
  PVector COM = new PVector(0,0,0); //center of mass
  int n = 0; //number of points

  //for every slice
  for (int z=0; z<nSlices; z++){
    PImage slice = imageArray[z]; //get slice
```

```
  //load the pixels in the slice
  slice.loadPixels();
  color colour;
  //for every pixel in the slice
  for (int x=0; x<slice.width; x++){
    for (int y=0; y<slice.height; y++){
      //get the colour of that pixel
      colour = slice.pixels[x+y*width];
      //if white
      if (colour == color(255)){
        //make new PointVector
        PointVectorArray.add(new PointVector(x*xScale,y*yScale,z*zScale));
        //update COM statistics
        COM.x += x*xScale;
        COM.y += y*yScale;
        COM.z += z*zScale;
        n++;
      }//end if
    }//end y for
  }//end x for
}//end  z for

//compute the center of mass
COM.div(n);

//find distance to nearest neighbours
//for every pointVector
for (int j=0; j<PointVectorArray.size(); j++){
  //find and print the distance to the nearest neighbour
  output.print(PointVectorArray.get(j).findDistanceToNearestNeighbour(PointVectorArray));
  output.print(", ");
  //find and print the distance to the center of mass
  output.println(PointVectorArray.get(j).distanceTo(COM));
}//end for

//flush and close output
output.flush();
output.close();
```

```
  println("done");
}//end setup

void draw(){
}//end draw
```

PointVector.pde (class)

```
class PointVector{

  //MEMBER VARIABLES
  private PVector r; //position vector

  //CONSTRUCTOR
  PointVector(float x, float y, float z){
    this.r = new PVector(x,y,z);
  }//end constructor

  //METHOD

  private PVector getPVector(){
    return r;
  }//end getPVector

  public float distanceTo(PVector u){
    return r.dist(u);
  }//end distanceTo

  private float distanceTo(PointVector u){
    return r.dist(u.getPVector());
  }//end distanceTo

  //find distance to nearest neighbour using an array of PointVector in the parameter
  public float findDistanceToNearestNeighbour(ArrayList<PointVector> array){
    float dMin = pow(width,2)+pow(height,2); //distance to nearest neighbour, set to to the length of the diagonal
squared first
    //for every maxima
    for (int i=0; i<array.size(); i++){
      //get the position vector of that maxima
      float d = this.distanceTo(array.get(i));
      if (d < dMin && d!=0){ //if d is less than dMin and not equal to zero
        dMin = d;
      }//end if
    }//end for
    return dMin; //return the distance to nearest neighbour
```

```
  }//end findDistanceToNearestNeighbour
}//end class
```

MODELLING MAXIMAS AS SPHERES

```
//import camera
import peasy.*;


//GLOBAL VARIABLES
int nSlices = 29; //number of slices
PeasyCam cam; //camera
PShape chromosome; //PShape for chromosome
float R = 29.0078145178338; //mean distance to nearest neighbour
float xScale = 11; //scale
float yScale = 11; //scale
float zScale = 20; //scale


//SETUP
void setup(){
  size(1024,768,P3D);
  cam = new PeasyCam(this,1000); //camera
  ArrayList<PVector> maximaArray = new ArrayList<PVector>(); //arrayList of position vectors of maximas

  //save the slices in an array
  PImage [] imageArray; //array of images
  imageArray = new PImage[nSlices]; //nSlices in the array

  //save the slices in an array
  for (int i=0; i<10; i++){
    imageArray[i] = loadImage("maximas0"+i+".png");
  }//end for
  for (int i=10; i<nSlices; i++){
    imageArray[i] = loadImage("maximas"+i+".png");
  }//end for

  //get dimensions of the image
  int imageWidth = imageArray[0].width;
  int imageHeight = imageArray[0].height;

  //for each slice
```

```
  for (int z=0; z<nSlices; z++){

    //get PImage of the slice and load the pixels
    PImage slice = imageArray[z];
    slice.loadPixels();

    //for every pixel
    for (int x=0; x<slice.width; x++){
      for (int y=0; y<slice.height; y++){
        //if that pixel is white
        if (slice.pixels[x+y*slice.width] == color(255)){
          maximaArray.add(new PVector(x,y,z)); //add the position vector to maximaArray
        }//end if
      }//end y
    }//end x
  }//end for

  //create PShape for chromosome
  chromosome = createShape(GROUP);
  sphereDetail(20); //set level of detail to be used to draw spheres

  //for every maxima
  for (int i=0; i<maximaArray.size(); i++){
    noStroke(); //noStroke
    PShape sphere = createShape(SPHERE,R); //create a sphere
    sphere.setFill(color(255,255,255)); //fill the sphere white
    PVector r = maximaArray.get(i); //get position vector of the maxima
    sphere.translate(r.x*xScale,r.y*yScale,r.z*zScale); //translate the sphere according to the position vector
    //translate the sphere relative to the centre
    sphere.translate(-imageWidth*xScale/2, -imageHeight*yScale/2, -((float)nSlices)*zScale/2);
    chromosome.addChild(sphere); //add the sphere to the chromosome PShape
  }//end for
}//end setup


//DRAW
void draw(){
 background(0); //black background
```

```
 //lights
 ambientLight(50,50,50);
 directionalLight(255, 255, 255, 0, 1, 1);
 directionalLight(255, 255, 255, 0, -1, -1);

 //draw chromosome
 shape(chromosome);
 //print frameRate
 println(frameRate);
}//end draw
```

CONNECTING MAXIMAS TOGETHER USING LINES

```
//import camera
import peasy.*;


//GLOBAL VARIABLES
int nSlices = 117; //number of slices (117)
PeasyCam cam; //camera
PShape chromosome; //PShape for chromosome
float R = 29.0078145178338; //mean distance to nearest neighbour
float STD = 7.71405915186672; //standard deviation distance to nearest neighbour
float xScale = 11; //scale
float yScale = 11; //scale
float zScale = 20; //scale


//SETUP
void setup(){
  size(1024,768,P3D);
  cam = new PeasyCam(this,1000); //camera
  ArrayList<PVector> PVectorArray = new ArrayList<PVector>(); //array list of position vectors of maximas

  //save the slices in an array
  PImage [] imageArray; //array of images
  imageArray = new PImage[nSlices]; //nSlices in the array

  //load all the slices
  for (int i=0; i<10; i++){
    imageArray[i] = loadImage("maximas00"+i+".png");
  }//end for
  for (int i=10; i<100; i++){
    imageArray[i] = loadImage("maximas0"+i+".png");
  }//end for
  for (int i=100; i<nSlices; i++){
    imageArray[i] = loadImage("maximas"+i+".png");
  }//end for

  //get imageWidth and imageHeight
```

```
float imageWidth = imageArray[0].width;
float imageHeight = imageArray[0].height;

//for every slice
for (int z=0; z<nSlices; z++){
  PImage slice = imageArray[z]; //get slice
  slice.loadPixels(); //load the pixels in the slice
  color colour;

  //for every pixel in the slice
  for (int x=0; x<imageWidth; x++){
    for (int y=0; y<imageHeight; y++){
      //get the colour of that pixel
      colour = slice.pixels[x+y*slice.width];
      //if white, add maxima position vector to PVectorArray
      if (colour == color(255)){
        PVectorArray.add(new PVector(x*xScale,y*yScale,z*zScale));
      }//end if
    }//end y for
  }//end x for
}//end  z for

//create PShape chromosome
chromosome = createShape(GROUP);

//for every pair of maximas
for (int i=0; i<PVectorArray.size()-1; i++){
  for (int j=i+1; j<PVectorArray.size(); j++){
    //get position vectors of the maximas
    PVector a = PVectorArray.get(i);
    PVector b = PVectorArray.get(j);
    float r = a.dist(b); //get distance between maximas

    //if less than 2 standard deviations away from the mean
    if (r<=R+2*STD){
      if (r<R-STD){ //if less than mean-1*standardDeviation
        stroke(255,0,0); //set stroke to read
      }//end if
      else if (R-STD<=r && r<=R+STD){ //if within one standard devivation from mean
```

```
          stroke(0,255,0); //set stroke to green
        }//end elif
        else { //else (between one and two standard deviationas away from mean)
          stroke(0,0,255); //set stroke to blue
        }//end else
        PShape line = createShape(LINE,a.x,a.y,a.z,b.x,b.y,b.z);
        line.translate(-imageWidth*xScale/2, -imageHeight*yScale/2, -((float)nSlices)*zScale/2);
        chromosome.addChild(line);
      }//end if
    }//end j for
  }//end i for
}//end setup


//DRAW
void draw(){
  background(0); //set background to black
  shape(chromosome); //draw chromosome
  println(frameRate); //print the frameRate
}//end draw
```

VII. REFERENCES

[1] M. Jones and G. Jones, Advanced Biology, Cambridge University Press, 1997.

[2] M. Jones, R. Fosbery and D. Taylor, Biology 1, Cambridge University Press, 2000.

[3] J. M. Scholey , I. Brust-Mascher and A. Mogilner, "Cell Division," *Nature,* vol. 422, no. 6933, p. 746, 2003.

[4] J. D. Watson and F. H. C. Crick, "Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid," *Nature,* vol. 224, no. 5218, p. 470, 1969.

[5] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts and P. Walter, Molecular Biology of the Cell, 5th ed., Garland Science, 2007.

[6] G. R. Kantharaj, "Chromosomal Nature-Before, During and After Gene Activation," [Online]. Available: http://mol-biol4masters.masters.grkraj.org/html/Gene_Expression_II13-Chromosomal_Nature_Before_During_and_After_gene_Gene_Activation.htm. [Accessed October 2013].

[7] S. R. Patil, S. Merrick and H. A. Lubs, "Identification of Each Human Chromosome with a Modified Giemsa Stain," *Science,* vol. 173, no. 3999, pp. 821-822, 1971.

[8] C. J. Stoeckert, M. Beer, J. Wiggins and J. C. Wierman, "Histone Positions Within the Nucleosome Using Platinum Labeling and the Scanning Transmission Electron Microscope," *Journal of Molecular Biology,* vol. 177, no. 3, pp. 483-505, 1984.

[9] P. J. J. Robinson, L. Fairall, V. A. T. Huynh and D. Rhodes, "EM Measurments Define the Dimensions of the "30-nm" Chromatin Fibre: Evidence for a Compact, Interdigitated Structure," *Proceedings of the National Academy of Sciences of United States of America,* vol. 103, no. 17, pp. 6506-6511, 2006.

[10] L. C. Woodcock, "A Milestone in the Odyssey of Higher-Order Chromatin Structure," *Nature Structural & Molecular Biology,* vol. 12, no. 8, pp. 639-640, 2005.

[11] W. Denk and H. Horstmann, "Serial Block-Face Scanning Electron Microscopy to Reconstruct Three-Dimenstional Tissue Nanostructure," *PLOS Biology,* vol. 2, no. 11, pp. 1900-1909, 2004.

[12] T. E. Everhart and T. L. Hayes, "The Scanning Electron Microscope," *Scientific American,* vol. 226, no. 1, pp. 55-69, 1972.

[13] B. Chen, M. Guizar-Sicairos, G. Xiong, L. Shemilt, A. Diaz, J. Nutter, N. Burdet, S. Huo, J. Mancuso, A. Monteith, F. Vergeer, A. Burgess and I. Robinson, "Three-Dimensional Strucure Analysis and Precolation Properties of a Barrier Marine Coating," *Scientific Reports,* vol. 3, no. 1177, 2013.

[14] T. Starborg, N. S. Kalson, Y. Lu, A. Mironov, T. F. Cootes, D. F. Holmes and K. E. Kadler, "Using Transmission Electron Microscopy and 3View to Determine Collagen Fibril Size and Three-Dimensional Organization," *Nature America,* vol. 8, no. 7, pp. 1433-1448, 2013.

[15] J. Rouquette, C. Genoud, G. H. Vazquez-Nin, B. Kraus, T. Cremer and S. Fakan, "Revealing the High-Resolution Three-Dimensional Network of Chromatin and Interchromatin Space: A Novel Electron-Microscopic Approach to Reconstructing Nuclear Architecture," *Chromosome Resarch (Springer Science + Business Media),* vol. 17, no. 6, pp. 801-810, 2009.

[16] H. Baytiyeh and J. Pfaffman, "Open source software: A community of altruists," *Computers in Human Behavior,* vol. 26, no. 6, pp. 1345-1354, 2010.

[17] "FAQ - Processing," [Online]. Available: http://wiki.processing.org/w/FAQ. [Accessed September 2013].

[18] J. M. Mateos-Pérez and J. Pascau, Image Processing with ImageJ, Packt Publishing Ltd., 2013.

[19] C. Reas and B. Fry, Processing: A Programming Handbook for Visual Designers and Artists, Massachusetts Institute of Technology Press, 2007.

[20] K. Bond and S. Langfield, AQA Computing AS, Nelson Thornes, 2008.

[21] J. A. Rice, Mathematical Statistics and Data Analysis, 3rd ed., Brooks/Cole, Cengage Learning, 2007.

[22] M. Davies, B. Francis, B. Gibson and G. Goddall, Statistics 4, 3rd ed., Hodder Murray, 2005.

[23] G. Cowan, Statistical Data Analysis, Oxford University Press, 1998, 2002.

[24] M. Cirne and H. Pedrini, "Marching Cubes Technique for Volumetric Visualization Accelerated with Graphics Processing Units," *Journal of the Brazilian Computer Society,* vol. 19, no. 3, pp. 223-233, 2013.

[25] C. Reas and B. Fry, "PShape \ Processing.org," Processing, [Online]. Available: http://www.processing.org/tutorials/pshape/. [Accessed September 2013].

[26] P. Bourke, "Polygonising a scalar field (Marching Cubes)," May 1994. [Online]. Available: http://paulbourke.net/geometry/polygonise/. [Accessed September 2013].

[27] M. Fisher, "Matt's Webcorner - Marching Cubes," 2012. [Online]. Available: http://graphics.stanford.edu/~mdfisher/MarchingCubes.html. [Accessed September 2014].

[28] T. Ferreira, "ImageJ User Guide - IJ 1.46r | Process Menu," 24 June 2012. [Online]. Available: http://rsb.info.nih.gov/ij/docs/guide/146-29.html. [Accessed January 2014].

[29] "MaximumFinder (ImageJ API)," Research Services Branch - National Institutes of Health, [Online]. Available: http://rsbweb.nih.gov/ij/developer/api/ij/plugin/filter/MaximumFinder.html. [Accessed January 2014].

[30] D. V. Lindley and W. F. Scott, New Cambridge Statistical Tables, 2nd ed., Cambridge University Press, 1995.

[31] R. Fouladi and J. Steiger, "The Fisher Transform of the Pearson Product Moment Correlation Coefficient and Its Square: Cumulants, Moments, and Applications," *Communications in Statistics - Simulation and Computation,* vol. 37, no. 5, pp. 928-944, 2008.

[32] T. J. Richmond, J. T. Flinch, B. Rushton, D. Rhodes and A. Klug, "Structure of the Nucleosome Core Rarticle at 7 A Resolution," *Nature,* vol. 311, no. 5986, pp. 532-7, 1984.

[33] "Avizo | FEI Visualization Sciences Group," Visualization Sciences Group, 2013. [Online]. Available: http://www.vsg3d.com/avizo/overview. [Accessed March 2014].

[34] "Imaris | 3D and 4D Real-Time Interactive Data Visualization | Bitplane," Bitplane AG, [Online]. Available: http://www.bitplane.com/imaris/imaris. [Accessed March 2014].

[35] S. M. Ross, Introduction to Probability Models, 10th ed., Elsevier Inc., 2010.

[36] "Vega Genome Browser 55: Homo sapiens - Description," Wellcome Trust Sanger Institute, January 2014. [Online]. Available: http://vega.sanger.ac.uk/Homo_sapiens/Info/Index.

[37] "Giemsa-staining method - Radiation Effects Research Foundation," Radiation Effects Research Foundation, 2007. [Online]. Available: http://www.rerf.jp/dept/genetics/giemsa_3_e.html. [Accessed March 2014].