DME, University of Southern Denmark
Jost Adam and Matiyas Korsa

Final Project

Handed out: Wednesday, December 1, 2021
Return by: Sunday, December 19, 2021, by 23:59

Group Number:	3	(Number)
Pass/Fail:		(Prof.)
Comments:		(Prof.)

Announcements:
Please **fill out your group number** on this sheet and **use it as a cover sheet for your project report**. Hand in the report and your source code (as one zipped file via Black Board), prior to the return date stated above. Please hand in **one report per group**, prepared by **all group members**.

Optimization and Image Processing

Group 3

Agni Biswas

Agbis21@student.sdu.dk

Andri Egilsson

Anegi17@student.sdu.dk

Björn Andreas Aumayr

Bjaum17@student.sdu.dk

Tobias Jacobsen

Tojac19@student.sdu.dk

Abstract

With the growing needs of the material analysis world, especially in nanoparticles, there is a strong need for fast, simple, and efficient characterization of particles and particle clusters, often produced by Scanning Electron Microscopy. The alignment particles in such clusters dictate material properties, henceforth necessitating a formalized approach capable of generating models for the data. The following research is an analysis of the data generated by three types of SEM images and an optimized strategy for reducing inaccuracies of the image processing algorithms that have been implemented. This allows for generating particle counts in the cluster by directly correlating particle counts to the number of foreground pixels.

Contents

Introduction

The following report is an experimental driven analysis of molecular clusters, intending to identify and model the data to aid with identifying properties of the materials. This research demonstrates an analysis of the types of groups, the inaccuracies faced, and the involved procedures and ultimately, how optimizing the data can affect the

This report explains our approach to determining different cluster types in three different images. We assume these images either contain Rod clusters, Circular clusters or Triangle clusters. It can be safely assumed that the images will only have one type of cluster, e.g. if it is an image with Circular sets, it is safe to say that all the clusters in the image will be Circular clusters.

Once the cluster type is determined, the next objective is to determine the number of objects in the clusters and then further determine the size of the particles in pixels.

Finally, the Particle swarm optimizer runs on the generated image and then creates a Lorentzian fit for the foreground pixels per cluster

All data will be presented with either Histogram, Box Plots or Violin plots and then further discussed.

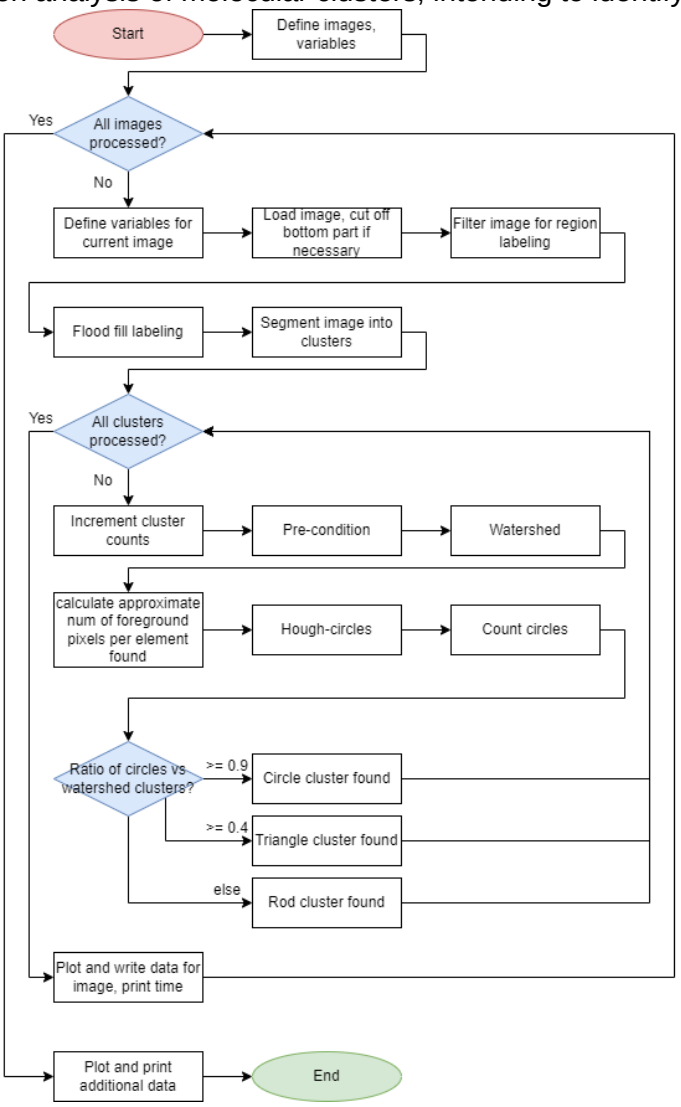


Figure 1- Overall Flowchart

Image Processing

Thresholding

Observations show that the images have a high amount of noise embedded. A simple python dash app was created to analyze the images, creating histograms for the selected area. This showed the following:

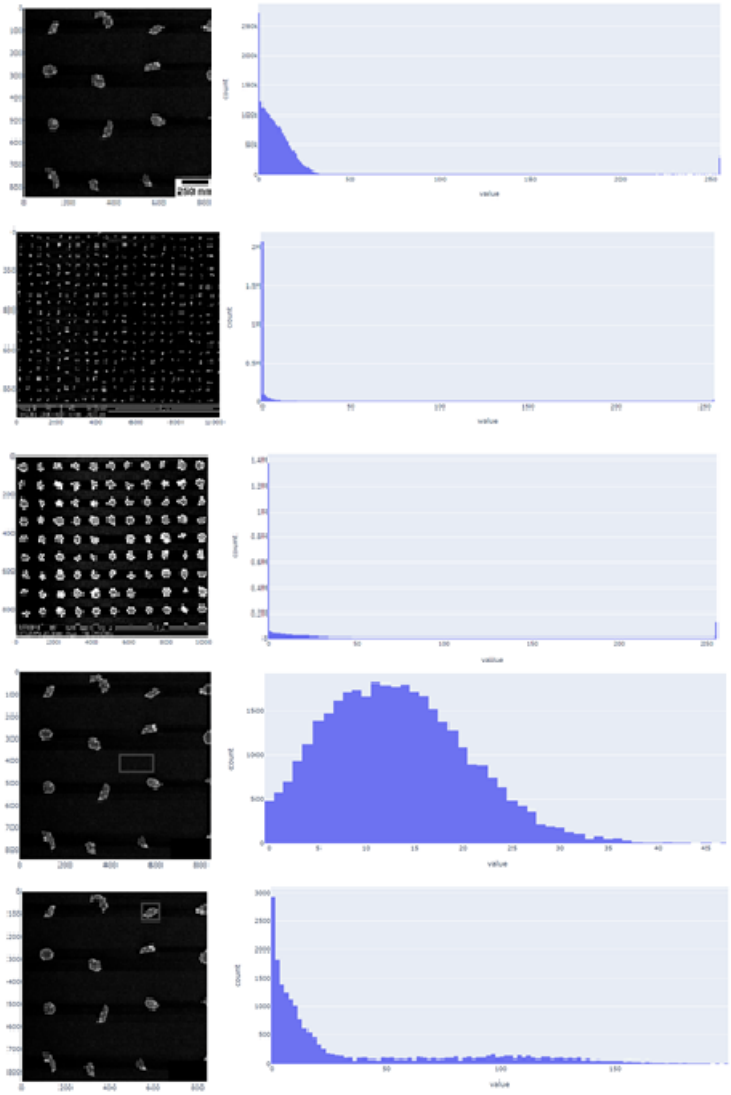


Figure 2 - General histograms

1. There is a bimodality in the images, with two sets of independent regional maxima: One towards the bright end and the other towards the darker end.
2. There exists sufficient bias in the images towards the darker mode.
3. The variance in the blocks which have noise is high.

We propose an "adaptive" threshold following Otsu’s Law.

We apply the algorithm by doing the following manually:

1. Assume a threshold
2. Segment the image into the two parts
3. Calculate the weight of the back/foreground, which is the average number of pixels (wb/wf)
4. Form normalised weighted average intensities by multiplying intensity with the number of particles for the background and the foreground respectively normalised over the total particles of that type (ub/uf)
5. Calculate the between-class variance $(wb * wf(ub - uf)^2)$
6. Repeat the steps 2-6 for the between-class variance for all intensities to maximise the value
7. Select the intensity value that corresponds to the max variance as the new threshold
8. Perform thresholding to binarise the image; this will make the image genuinely bimodal

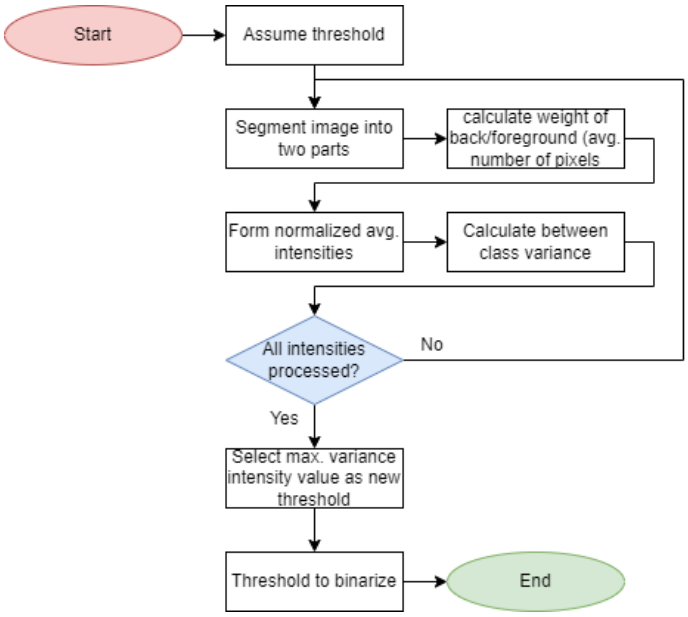


Figure 3 Flowchart Otsu'sThresholding

The function auto_thresh() was created using this method to allow for automatic thresholding.

The algorithm is known to perform sub-optimally with complete images with varying degrees of average brightness, so it is essential to run specific functionality on a focused image segment, identified by region labelling. This algorithm's primary advantage is that the newly generated threshold removes any manual labour involved in objectively thresholding the image for further operations.

```python
def mean_class_variance(hist, thresh):
    wf = 0.0
    uf = 0.0
    ub = 0.0
    wb = 0.0
    total = np.sum(hist)
    # print(total)
    if len(hist) > 256:
        raise IndexError("object image depth too large, recheck input")
    for i in range(len(hist)):
        if i < thresh:
            wb += hist[i]
            ub = ub + (i * hist[i])
        elif i >= thresh:
            wf += hist[i]
            uf = uf + (i * hist[i])
    if wb == 0:
        ub = 0
    else:
        ub = ub / wb
    if wf == 0:
        uf = 0
    else:
        uf = uf / wf
    wb = wb / total
    wf = wf / total

    variance = float(wb * wf * ((ub - uf) ** 2))
    return variance, thresh

def auto_thresh(img, mode="Thresholding"):
    hst = hist256(img)
    variance_list = []
    for thresh in range(len(hst)):
        if thresh > 1:
            variance_list.append(mean_class_variance(hst, thresh))
    # print(variance_list)
    # print("MAX tuple: ", max(variance_list))
    var, thresh = max(variance_list)
    # print("MAX tuple: ", thresh)
    if mode == "Thresholding":
        return threshold(img, thresh)
    elif mode == "UpperSave":
        img[img <= thresh] = 0
        return img
    elif mode == "LowerSave":
        img[img >= thresh] = 255
        return img
```

Figure 3 - Code example of Therholding

Pre-processing

The first step is to load the images into an array before cutting unnecessary parts of the picture, like the border seen at the bottom of figure 5. This is done for the circle and rod clusters because the segmentation algorithm only expects the raw image with the clusters.

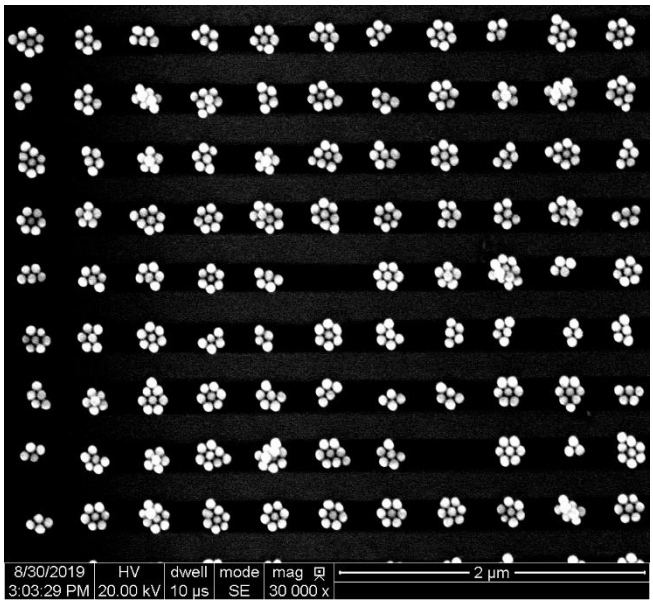


Figure 4 - Un-edited picture

Segmentation

In this part, the goal is to segment the images such that there only is one cluster per image. This should work on the different kinds of clusters, and it must cut out the clusters accurately, due assumption that a cluster from the segmentation is indeed only one cluster.

The idea for segmentation is to label each cluster. For this to be viable, each element in the clusters needs to be connected, and the image needs to be binarised. The images have a good amount of shot noise; this needs to be reduced for the threshold only to pick up the clusters. This is done by blurring the image to make the shot noise peak values go down, such that the threshold will only pick up the clusters. The binarized image is then dilated to ensure each part of a cluster is connected. Then it's ready to be labelled using a modified flood fill algorithm which will also return the maximum and minimum pixel indices for locating where the cluster is in the image. Figure 6 shows the process on the image with circular clusters.

The pixel indices are then used to cut out the segmented parts from the original picture. This is only done if the pixel indices are not at the end of the image because this means the cluster might be partial. All the cut-out clusters are saved in a list for individual processed feature extraction and classification.

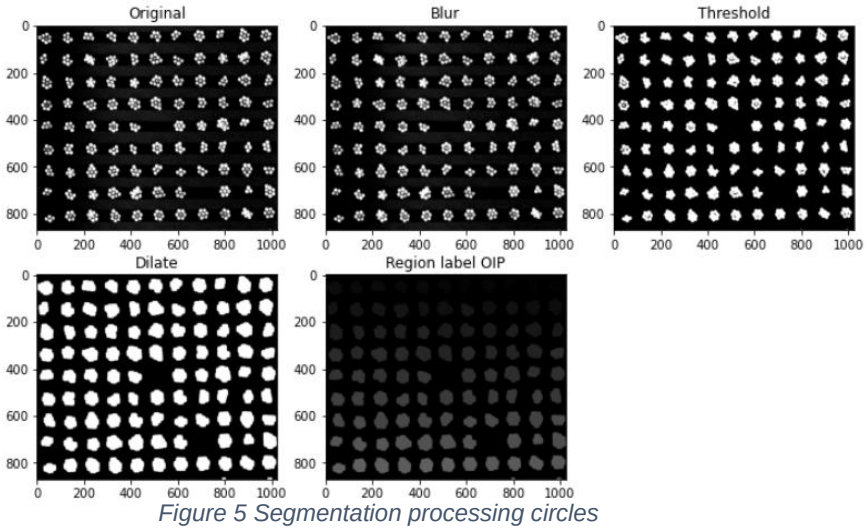


Figure 5 Segmentation processing circles

Watershed

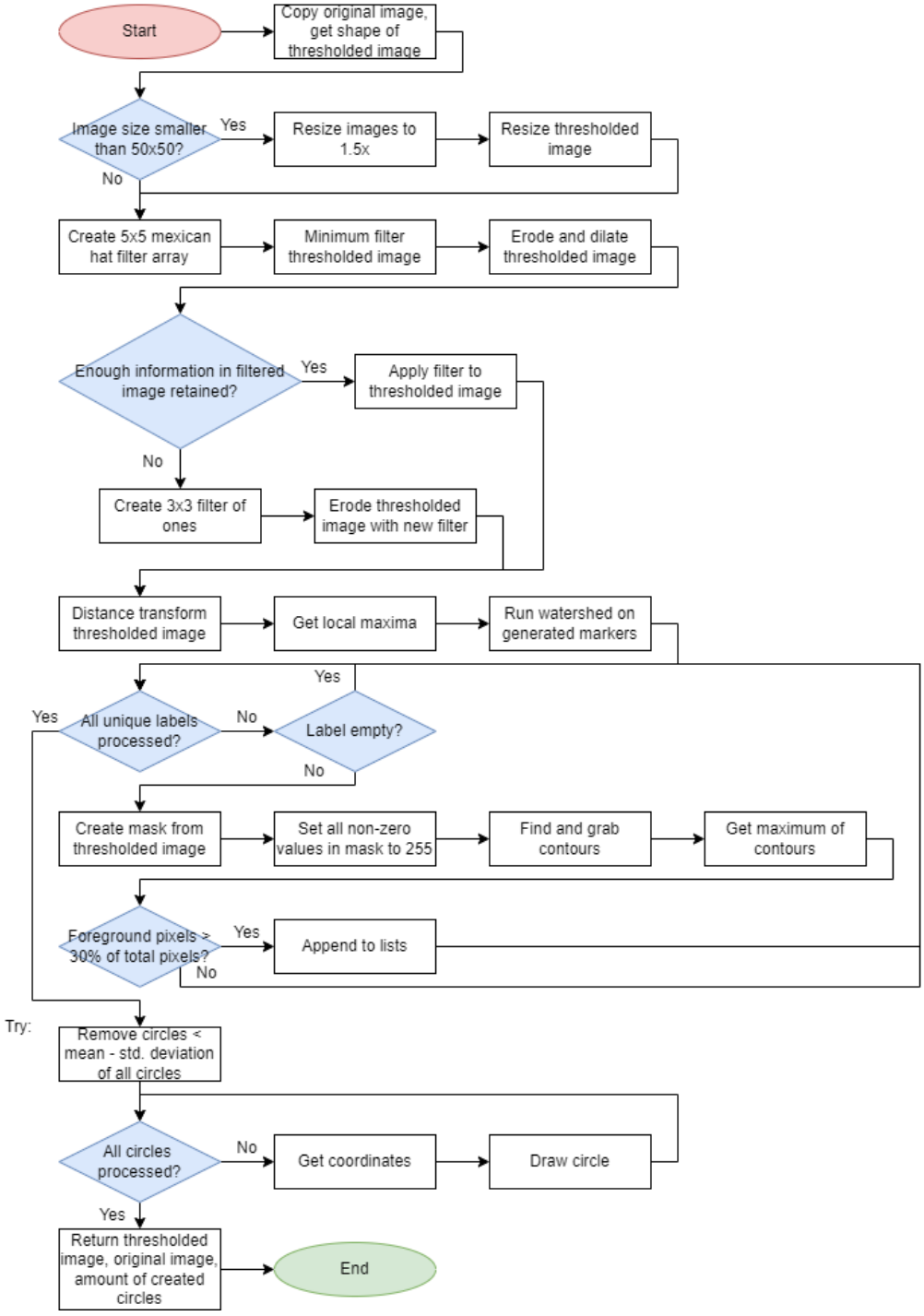


Figure 6 - Watershedding flowchart

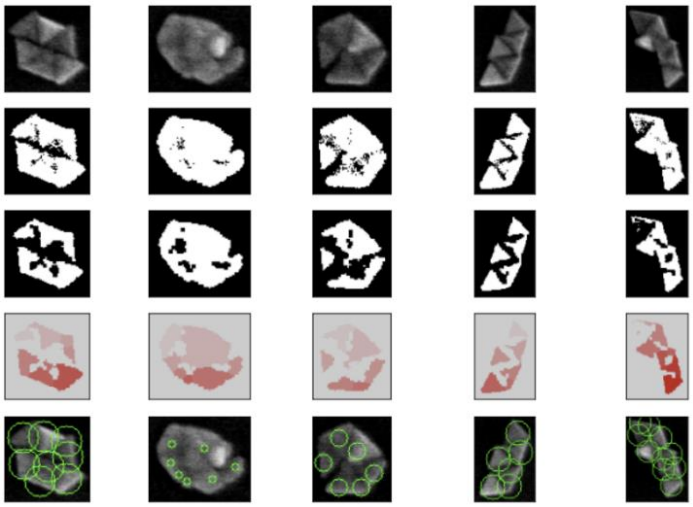


Figure 7 - Watershedding example

The watershed process was mildly modified to serve as a general feature generation; the three significant modifications made were:

1. Gaussian filtering for enclosing circles: All minimum enclosing circles that possess a radius less than $mean - standard\ deviation$ are directly eliminated; this prevents smaller blobs from counting as labels.
2. Circles that possess less than 10% foreground pixels are eliminated.
3. A minimum filter was run with a morphological opening, which activates only if the shrinkage in the foreground pixels was more than 10%, otherwise it erodes the image with a 3x3 unity matrix.

The locwatershed() function takes the image and the thresholded image (steps 1 and 2) and returns the modified thresholded image, the overlay of the minimum area circles with the original image (3 and 5 respectively) and the labels count as an output. The label count is stored in a list in the main script. The watershed method has limited accuracy with the rods, owing to the scale of original image which would fundamentally leave larger pixelated spots as jagged edges which would lead to multiple watershed blobs.

Feature generation and classification for identifying cluster-types

The fundamental aim is to be able to generate ample data which could be used to describe the three types of images we have, i.e. rods, triangles, and circles. Features are generated by running a common set of algorithms that are not tuned for specificity, which means that they provide a generic enough data set to allow for some degree of freedom across all the different types of inputs.

Once the image has been split into clear segments, the next step is to do general pre-conditioning on each segment, since the three different images can be of different types and contain different amounts of noise, the filtering chosen was auto-contrasting to sharpen up the image before then running in through auto-thresholding. After the thresholding, the image is then passed through edge detection resulting in clusters as seen in figure 9, these clusters are then sent towards type detection. It should be noted that further filtering could be done at this stage, but since the filtering was sufficient for type detection, it was kept to a minimum, since the image quality varies widely.

Figure 8 - Thresholder and Edge detected cluster

The simplest way to determine the cluster's type is to run it through the most reliable object determining that is available and then making an educated guess from there. In this case, the most reliable way was to first run Hough-Circle detection and then to run Watershedding to determine the number of circles vs. the number of watershed objects. If the number of circles vs. objects is close to 1:1 then the picture must contain circle clusters while, if the image includes almost no circles vs. objects, the picture must be rods. If the Hough-Circle detection picks up some circles, but at a much lower circle-to-object ratio than a circle image, it must be a triangle picture.

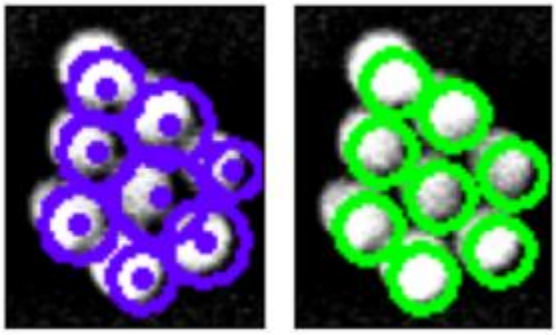


Figure 9 - HoughCircle vs WaterShedd

Circle processing

The circle counting is done in the same manner as in the identifying stage. The segmented image is passed through light pre-filtering and Hough-Circle detection is used on the cluster.

The Hough-Circle detection picks up 596 individual circles in 97 clusters with an accuracy of about 99%, as seen in figure 11, with 5 to 7 particles detected in most clusters.

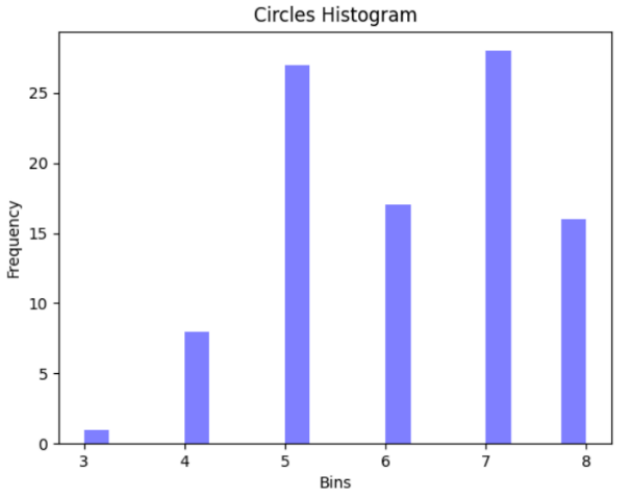


Figure 10 - Circle Histogram

Triangle Processing

After the cluster type has been identified as a triangle cluster, the number of particles found by the watershedding is appended to an array.

In the provided sample image, the algorithm manages to find 74 individual triangles in 12 clusters. As seen in figure 13, the triangle count varies greatly, which may be due sample size being the smallest out of all the pictures.

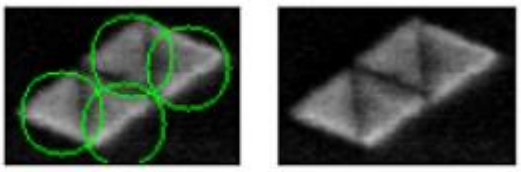


Figure 11 - Watershed triangles

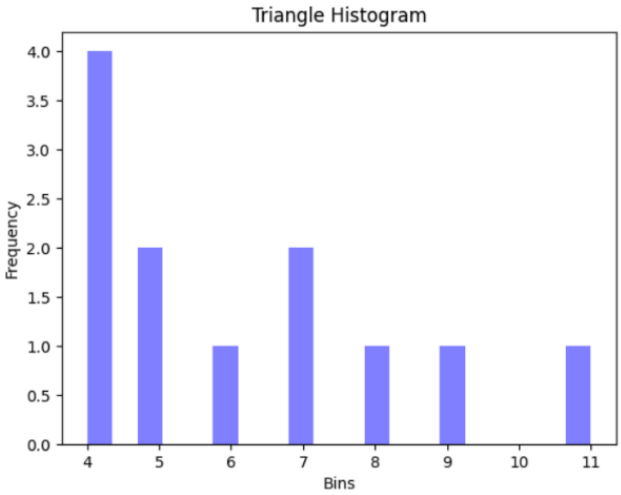


Figure 12 - Triangle Historgram

Rod Processing

The counting of rods presented major difficulties and went through multiple iterations, including the use of Hough line detection, and watershed. Pre-processing the image for use with the Hough line transform was an issue due to the low intensity of the rod clusters combined with very unsharp edges, especially between individual particles. For the rods to be shown clearly in the binary image, a Mexican hat filter was used in two iterations. This makes some parts of the low intensity rods also show up on the binary image. The problem with the approach is that it would also highlight some noise in the picture outside of the cluster.

To get rid of this noise, a mask was made by blurring and thresholding the original image. This mask ideally only has the cluster pixels as non-zero pixels. It can then be combined with the filtered image using a logical AND operation. The mask can be applied to the image before or after the Mexican hat filtration; the output would be very similar. See figure 14 when looking atIn the first row of Figure 13 the low-intensity middle rod is being amplified by the Mexican hat filter, compared to the threshold of the original image. In the second row it can be observed that the separation of the two rods is amplified.

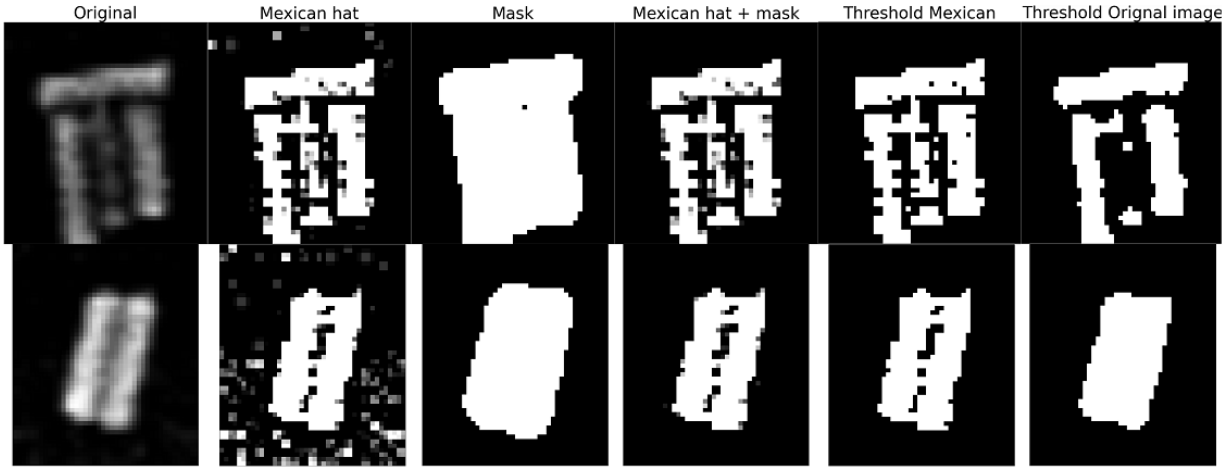


Figure 13 Preprocessing for rod counting

Additional processing is done prior to attempting to detect lines, in order to find the middle of the rods using distance transform. This will return higher values in the middle of the foreground pixels. The local maximums are then found. These points ideally lie in the middle of the rods. Dealing with single pixels, it can be hard to find a line that lies exactly on the maximums found. Therefore, the maximums are dilated to make it easier to find lines (Figure 14).

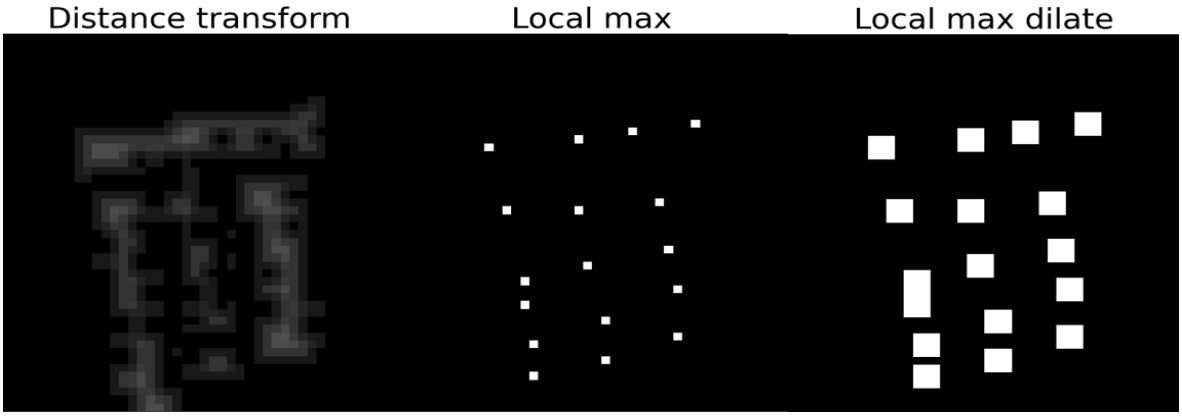


Figure 14 Distance transform local max

After Hough lines transform is applied, the problem arises as to which lines should be chosen, as it will also pick up lines which are not present. This is a hard problem and multiple solutions were attempted. First, if the angle and the distance that describe the line are similar, discard the line with the lowest point in the accumulator matrix. On top of that it, lines intersected and have similar angles, the lowest would be discarded. This worked well on some clusters, while on others it gave a lot of lines which did not represent any rods. The final way this was solved was by taking the highest line in the accumulator matrix and drawing a 0-intensity line on top of the binary image. This was done until no more lines could be found. This had the opposite effect to the first way of choosing lines, meaning it would sometimes find less lines than rods. This is because, when a line is drawn on the image it might draw over a maximum from another rod (Figure 15). But (Figure 15), but it was found to be a better and more accurate way to count the rods, as it undershoots a bit, while the initial option could overshoot by a lot.

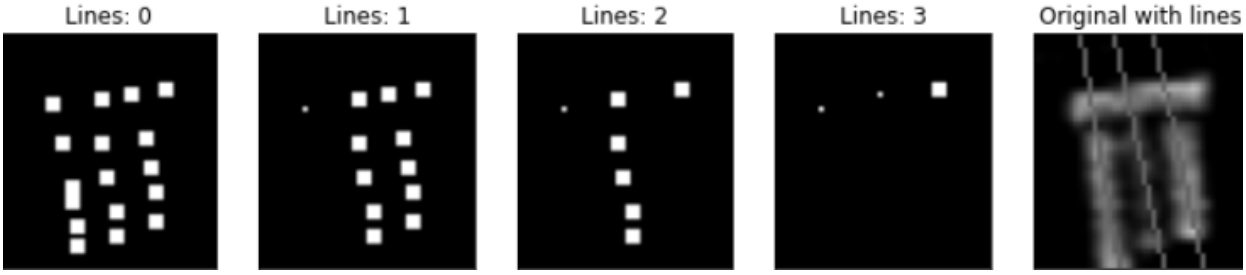


Figure 15 Finding lines drawing on top

The flow chart for the rod processing can be found in appendix: A.2 - A.2 - Flow chart rod processing.

Image Processing Results

```
----------------------- Text Printout ------------------


Total number of Clusters :  474


Picture with most ammount of Circles :  ./pictures/001_002.tif
Number of Clusters in picture :  97
Number of Circles in picture :  596
Average number of circles in clusters :  6.144329896907217


Picture with most ammout of lines :  ./pictures/R001_001.tif
Number of Clusters in picture :  365
Number of Lines in picture :  994
Average number of lines in clusters :  2.723287671232877


Picture with most ammount of Triangles :  ./pictures/T001.png
Number of Clusters in picture :  12
Number of Triangles in clusters :  74
Average number of Triangles in clusters :  6.166666666666667
Mean number of a cluster particles :  6.17  Median number of a cluster particles :  5.5
Standard Deviation of cluster particles :  2.19  Variance of cluster particles :  4.81


----------------------- Time Data ------------------
Tottal run time :  313.7770926952362
Average run time :  104.5923642317454
Shortest run time :  8.451897382736206
Longest run time :  272.1808638572693
Longest run time picture :  ./pictures/R001_001.tif


--------------------------------------------------------
```

Figure 16- Text data printout

As seen in figure 179, the data for each picture looks pretty promising with a total cluster count of 474 with the fewest being in the triangle picture and the most being in the rods picture, the run time also corresponds this, though it has to be taken into account that not only does the rod picture have the largest amount of clusters, but the method used to determine the rod count is also the most resource intensive out of the three methods.

Results Circle

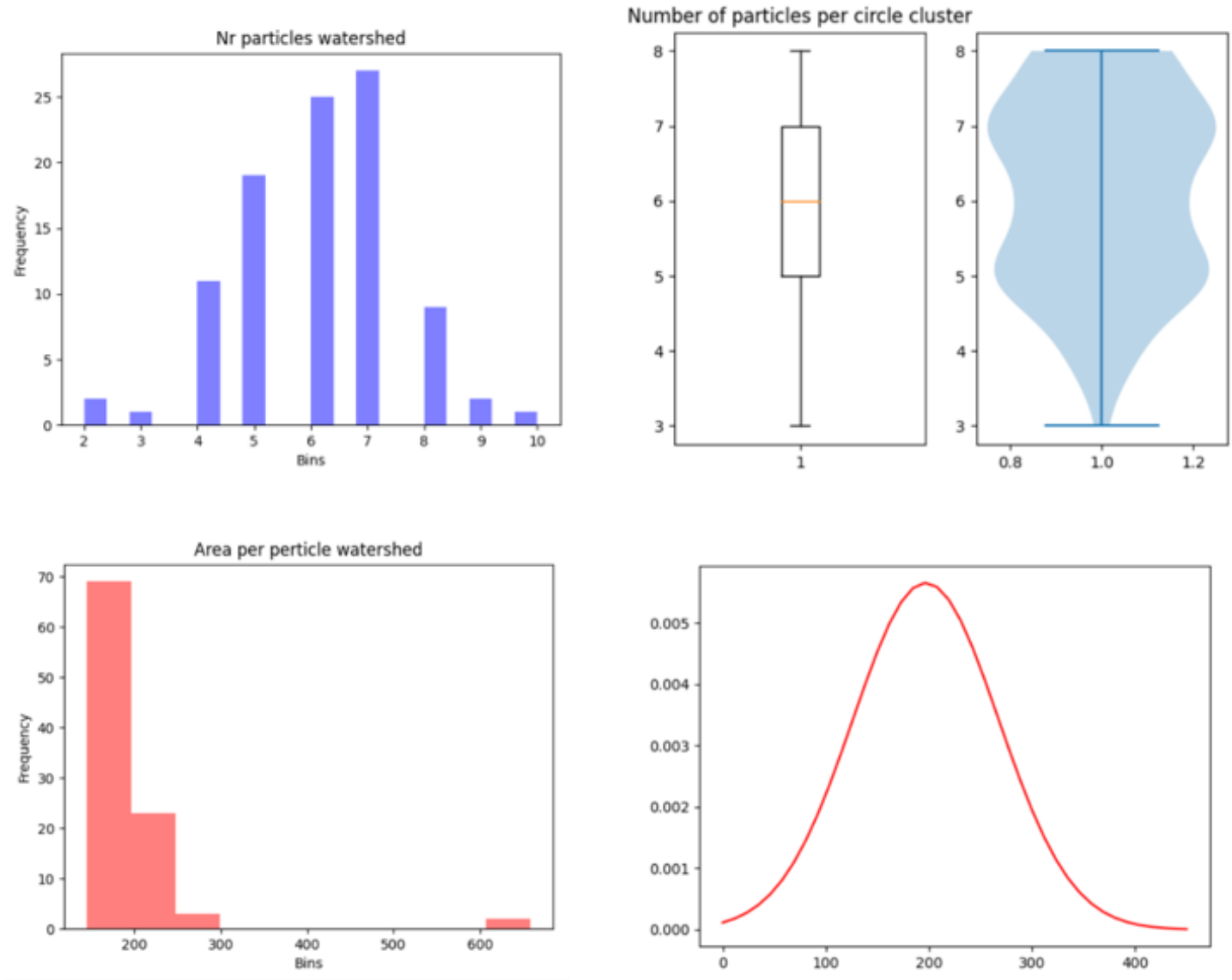


Figure 17 - Circle results from image processing
Top left : particles per cluster. Top right: box and violin plots for images in the cluster
Bottom left: area per particle. Bottom right: gaussian Distribution of foreground pixels per particle

As seen in figure 18, the graphs for the circle image processing are as expected, the number of particles watershed, area per particle watershed and the curve were found using watershedding while the violine and boxplot were found using Hough circle detection.

Results Triangle

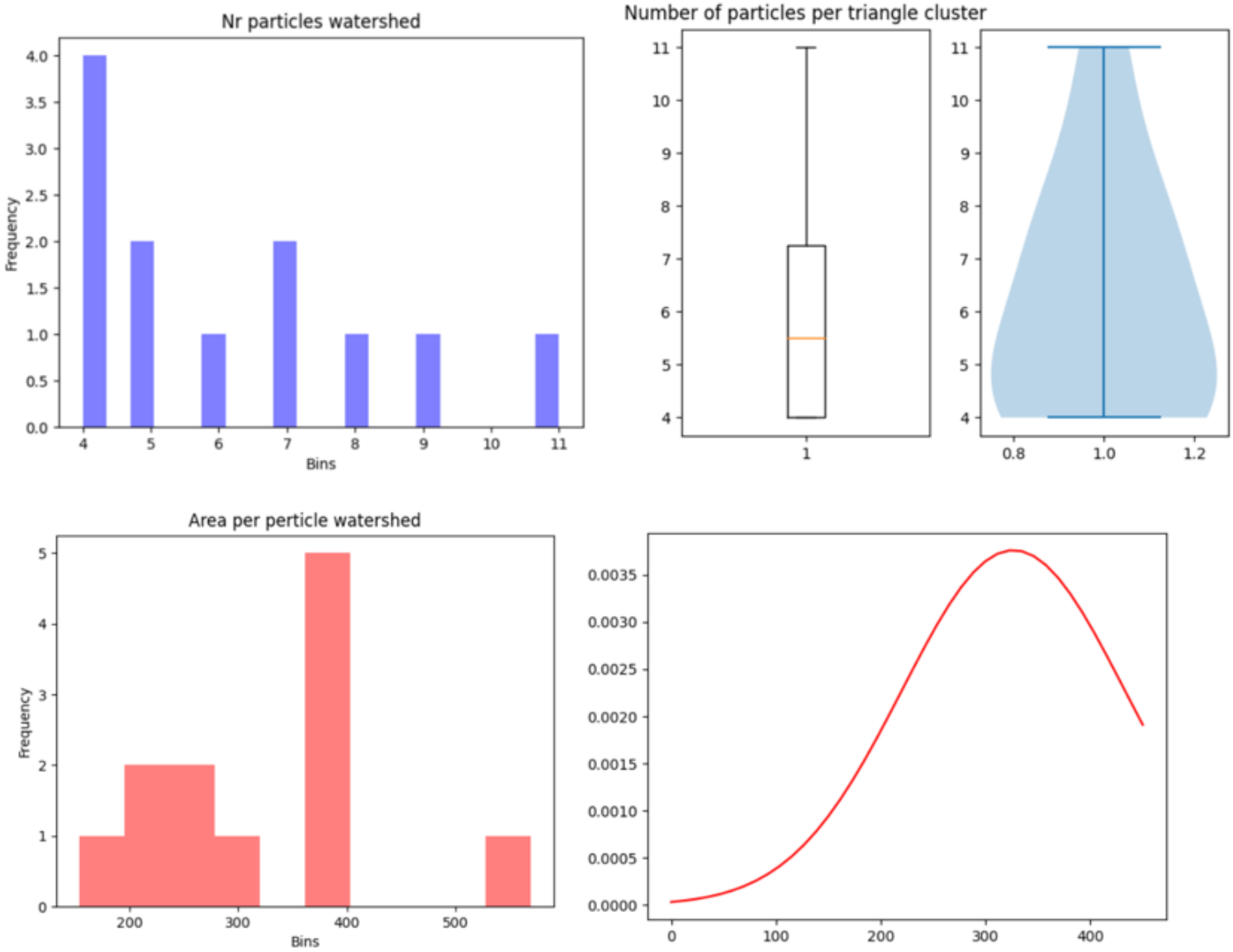


Figure 18 - Triangle results from image progressing
Top left : particles per cluster. Top right: box and violin plots for images in the cluster
Bottom left: area per particle. Bottom right: gaussian Distribution of foreground pixels per particle

All graphs in figure 19 were found using watershedding.

Results Rods

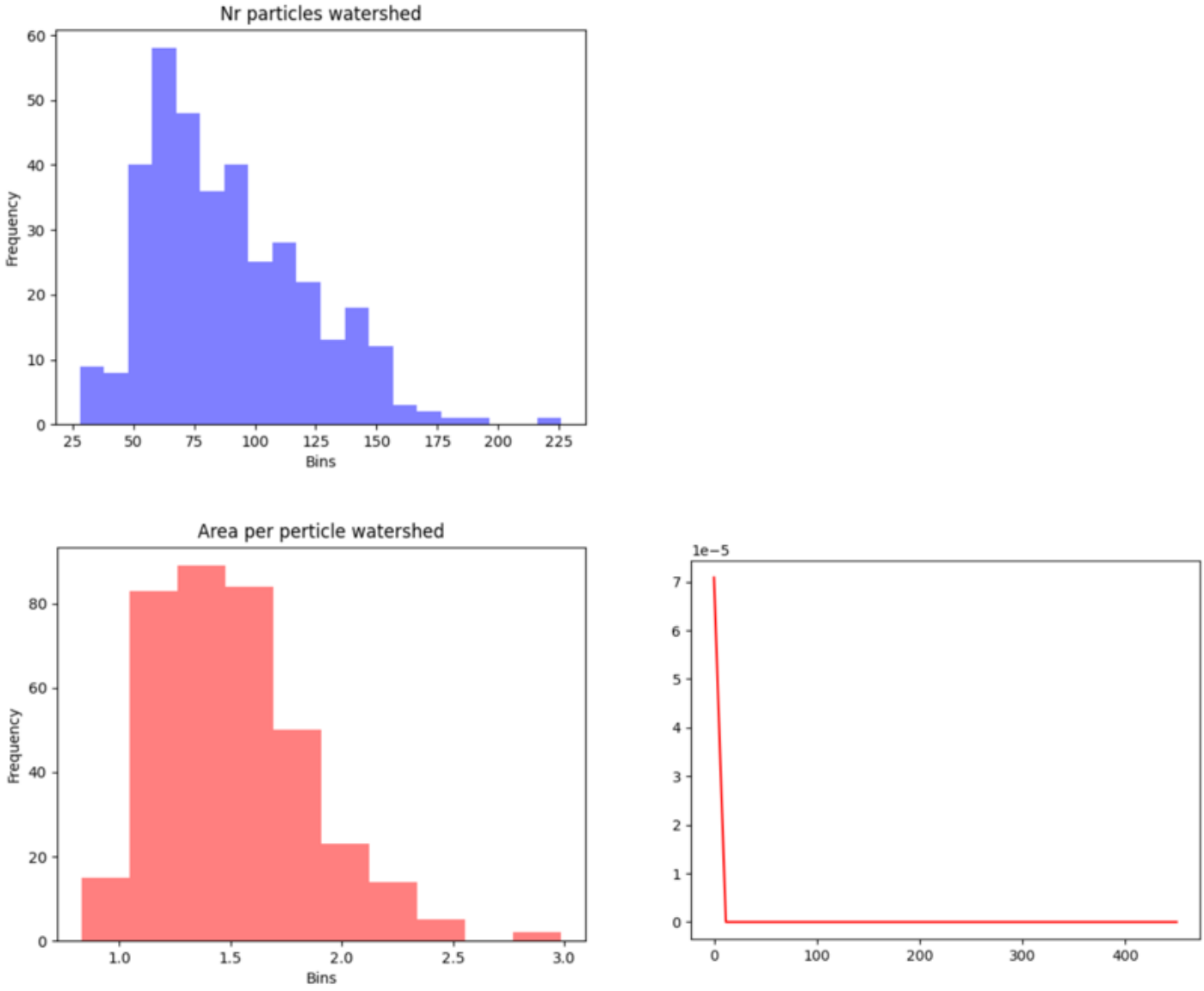


Figure 19 - Rod results using watersheding
Top left : particles per cluster.
Bottom left: area per particle. *Bottom right: gaussian Distribution of foreground pixels per particle*

As seen in figure 20, the watershedding is picking up an unusual number of particles in the rods picture, so the images above do not represent it as well. Instead it would be better to use the values from the Rod Processing function, as seen in figure 21 on the next page.

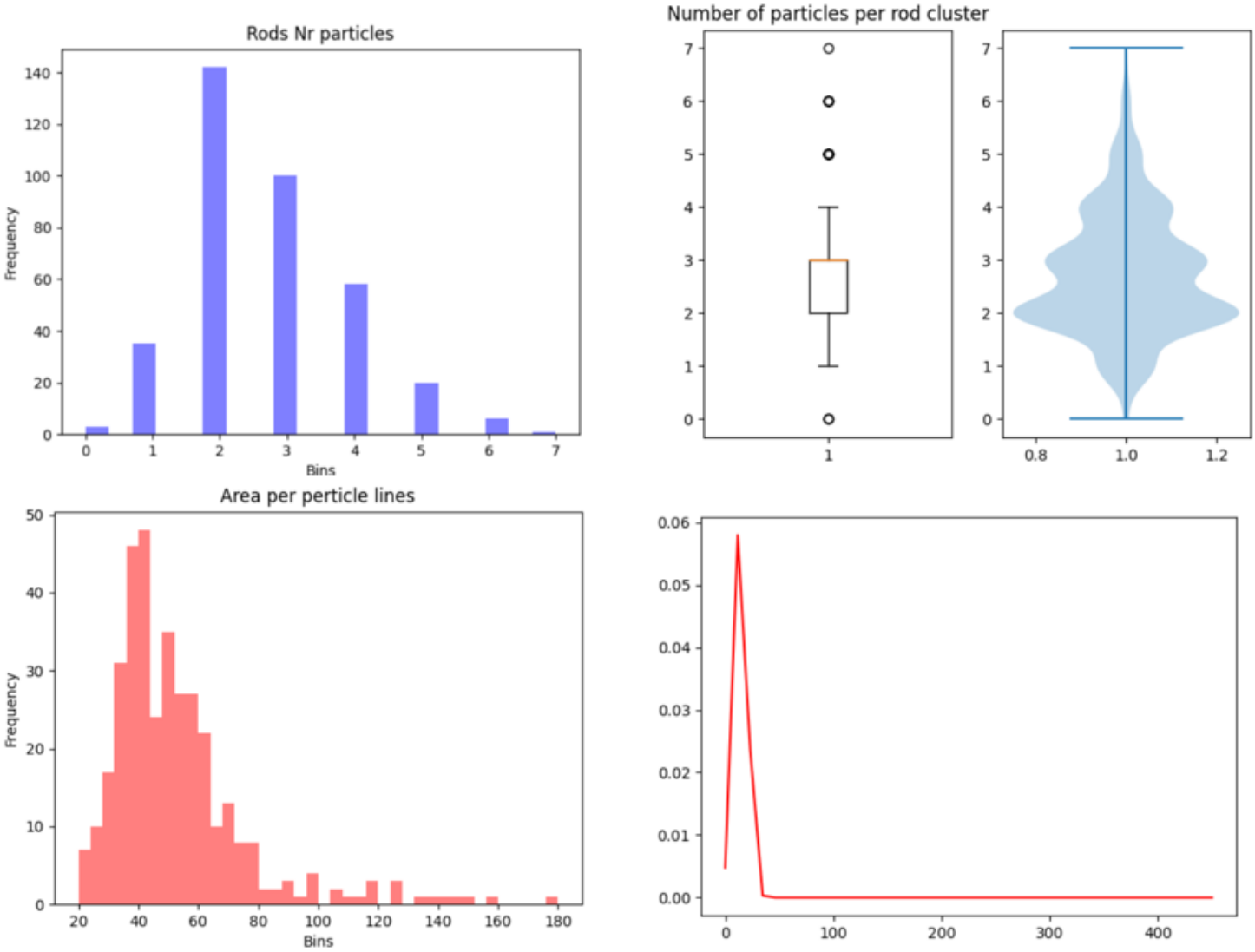


Figure 20 - Results using Distance Transform and Hough Lines
Top left : particles per cluster. Top right: box and violin plots for images in the cluster
Bottom left: area per particle. Bottom right: gaussian Distribution of foreground pixels per particle

As seen in figure 21 the values started to become more realistic as soon as the function described in Rod Processing was used to determine the number of rods.

Optimization

Formulation

The optimization problem being pursued is the identification and curve-fitting for the distribution of the particle size data, so as to improve accuracy of the particle distribution.

Assuming that the particle distributions are from a single type of classified particle with uniform dimensions, it is safe to predict that the particles would adhere to a single mode with a certain degree of variation subject to 2-dimensional projection of the particle in the images.

This implies that the curve fitting is fundamentally limited to a single peak Lorentzian model.

Lorentzian

A sum of Lorentzians can be used to make a curve which will later be optimized for fitting to a dataset.

$$f(x_n) = \sum_{k=1}^{K} \frac{y_k \sigma_k^2}{(x_n - x_{kCenter})^2 + \sigma_k^2}$$

y_k peak height, σ_k peak width and $x_{kCenter}$ center position.

In our case K=1 gives

$$f(x_n) = \frac{y_1 \sigma_1^2}{(x_n - x_{1Center})^2 + \sigma_1^2}$$

Gaussian

A sum of Gaussians can be used to make a curve, which will later be optimized for fitting a dataset.

$$f(x_n) = \sum_{k=1}^{K} a_k * e^{-\frac{(x_n - x_{kCenter})^2}{2\sigma_k^2}}$$

a_k amplitude, σ_k standard diviation and $x_{kCenter}$ center position

In our case K=1 gives

$$f(x_n) = a_1 * e^{-\frac{(x_n - x_{1Center})^2}{2\sigma_1^2}}$$

Least squares

To optimize our function the least square method is used. This gets the error between the fitting function and the real data and squares it, making sure that the error wil be positive. This value J is what will be minized. N is the number of datapoints, y_n is the data's y value at x_n and $f(x_n)$ is the fitting function's y value at x_n.

$$J = \sum_{n=1}^{N} [y_n - f(x_n)]^2$$

Algorithm

Particle swarm optimization was chosen owing to its random nature, which ensures that the probability of falling into a global minimum is maximized. In addition, particle swarm optimization is easier to run on discrete data without forcefully interpolating the data into a curve which could potentially alter the behavior of the original data. This was done due to the severe limitations in the quantity of image data, which resulted in 97 samples of circles, 12 samples of triangles and 365 samples of the rod-like image clusters.

Particle swarm optimization operates by creating a random population of particles that possess a "social awareness" and an "individual awareness" of the maxima/minima. Every iteration the particles move in a direction that is a function of the globally communicated maxima, the personal maxima/minima, and its current velocity. Given enough iterations, particles in the swarm tend to converge towards the globally detected maxima/minima assuming at least one particle in the full pool of the particles had previously managed to move into/towards the true maxima/ minima.

The standard operating procedure of particle swarm optimization is demonstrated in the flow chart in Figure 22.

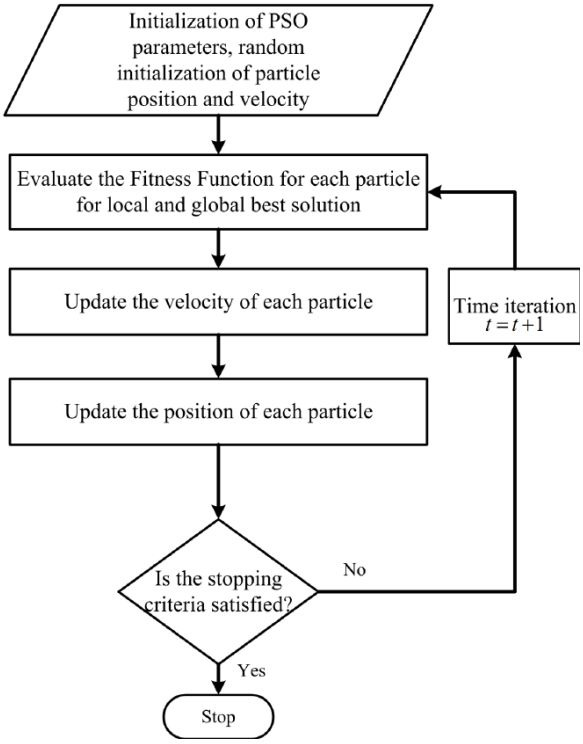


Figure 21 – Particle Swarm flowchart

Implementation:

Particle swarm optimization is done in a separate file, so as to not slow down the main image processing. Data is taken from a text file written to by the main.py process.

In the code's main, the data is read and particle number, number of iterations and initial PSO parameters are set. The maximum x and y values are found and the maximum particle velocity are initialized.

```
plt.scatter(*data.T, marker='o', c=np.random.rand(len(data)))
```

An approximation of the mean value is used along with the maximum y value and an evenly spaced array of 200 values, from 0 to 100, to test the model function.

The PSO function is now called, taking the parameters initialized in the beginning.

```
best_point, best_value, particles = PSO_testfunction(particle_number, Iterations, c1, c2, om, k, t, xmin, xmax)
```

The particle positions are set to random values in the range of xmin to xmax and the personal best values are initialized to these positions. The particle fitness values are found using the

calc_error() function, which in turn uses the model function to calculate the error. The fitness values are then appended to a list.

```
for i in range(n - 1):
    Par_Val.append(calc_error(pos[i]))
```

The global best position is set, following which the while loop for the PSO iterations is started. This loop continues until the set number of iterations has been achieved.

In each iteration, the particle velocities and positions are updated, and a for loop is used to once again evaluate the fitness of each particle.

```
# step 2a
# Update Particle Velocity
v = update_velocity_testfunction(p_best, g_best, pos)
# Step 2b
# Update Particle Position
pos = update_position_testfunction(v, pos)
# step 2c
# Evaluate fitness
for i in range(n):
    Par_Val.insert(i, calc_error(pos[i]))
```

The personal bests and global best are updated and printed, and the iteration value is incremented.

```
# Step 2d
# Updating the personal best
# if
p_best = update_personal_best_testfunction(p_best, Par_Val, pos)
particles.append(p_best)
# Step 2e
# Updating the global best
# if
g_best = update_global_best_testfunction(g_best, Par_Val, pos)
g_best_value = calc_error(g_best)
```

After all iterations have finished, the global best position and index, as well as the array of personal bests, are returned.

In the main, the model function is then used to plot out the optimized curve.

Each image's data was read from the stored file and the results were obtained the following pictures are the results of 600 iterations for 150 particles for rods, triangles, and triangles.

```
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  595
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  596
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  597
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  598
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  599
(150, 3) (3,)
The best position is:  [41.43181164 14.40774872 41.37142796]  Value:  649.0274952526805  in iteration number  600
```

Figure 22 - PSO printout of Rods

```
(150, 3) (3,)
The best position is:  [ 11.13987878 -19.90750051 179.39386454]  Value:  47.55496088270713  in iteration number  596
(150, 3) (3,)
The best position is:  [ 11.13987878 -19.90750051 179.39386454]  Value:  47.55496088270713  in iteration number  597
(150, 3) (3,)
The best position is:  [ 11.13987878 -19.90750051 179.39386454]  Value:  47.55496088270713  in iteration number  598
(150, 3) (3,)
The best position is:  [ 11.13987878 -19.90750051 179.39386454]  Value:  47.55496088270713  in iteration number  599
(150, 3) (3,)
The best position is:  [ 11.13987878 -19.90750051 179.39386454]  Value:  47.55496088270713  in iteration number  600
```

Figure 23 - PSO printout of Circles

```
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  595
(150, 3) (3,)
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  596
(150, 3) (3,)
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  597
(150, 3) (3,)
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  598
(150, 3) (3,)
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  599
(150, 3) (3,)
The best position is:  [  5.10392029 -86.5034109  380.4357367 ]  Value:  3.11071047034343  in iteration number  600
```

Figure 24 - PSO printout of Triangles

Results

All testing was done using the particle swarm algorithm described above with 600 iterations and 150 Particles.

The procedure that was followed after the Lorentzian fit curve generation was to use the original image, and in every cluster divide the total foreground pixels (white pixels on the thresholded image) with the 3rd parameter in the “Best Position ”output of the PSO, which is assumed to be the new particle count of the cluster leading to a recalibrated histogram of particle counts per cluster for each image. For results, see figures below.

Circle Results

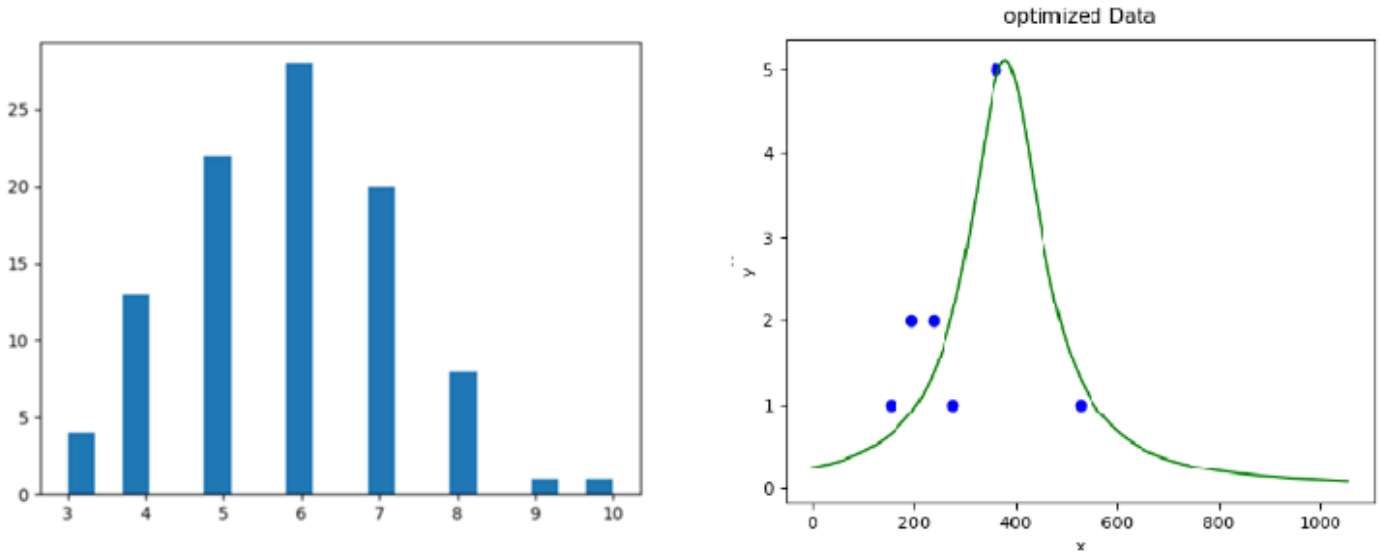


Figure 25 - Optamized Histogram and Graph

As seen in Figure 26 the optimized data plot is as expected if we compare it to the actual data plot from the image processing, as seen in Figure 27, and the optimized histogram has shifted more to the left and centered around 6 particles per cluster.

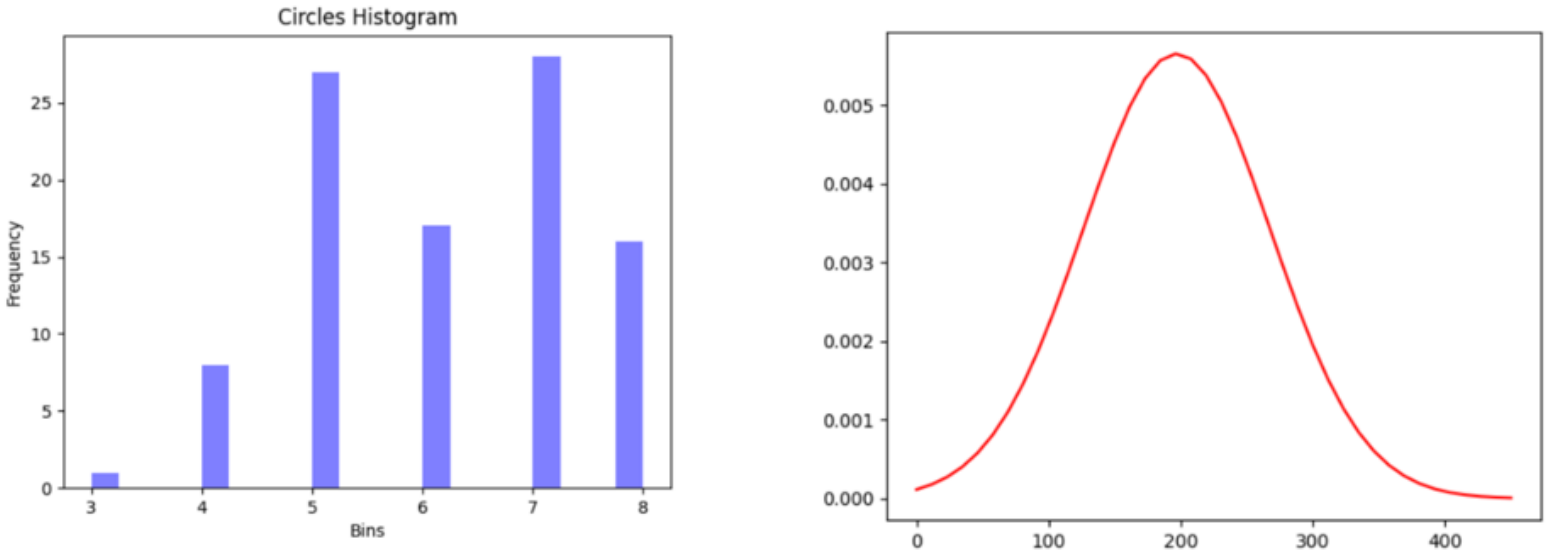


Figure 27 - Histogram and Graph from image processing

Triangle Results

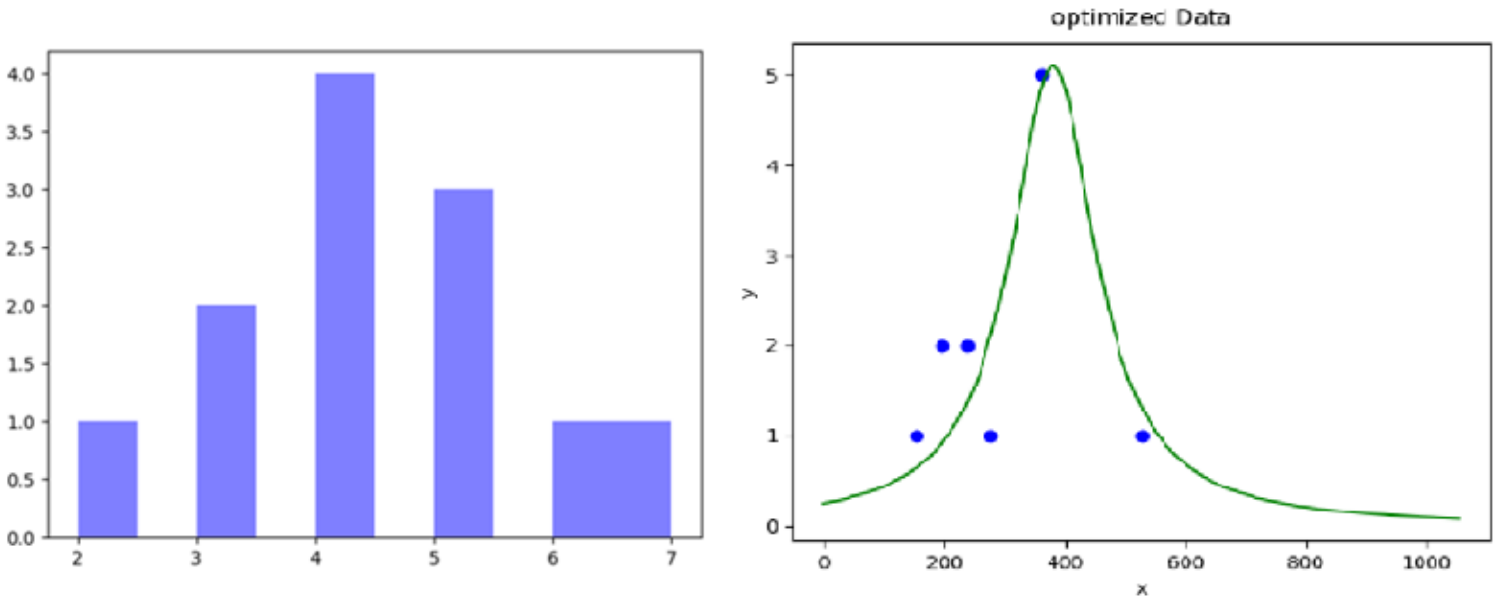


Figure 26 – Optimized Histogram and Graph

As seen, the triangle image processing is not as well optimized as the circle image processing, but still relatively accurate, considering a cluster sample size of 12 for the triangles vs a cluster sample size of 97 for the circles.

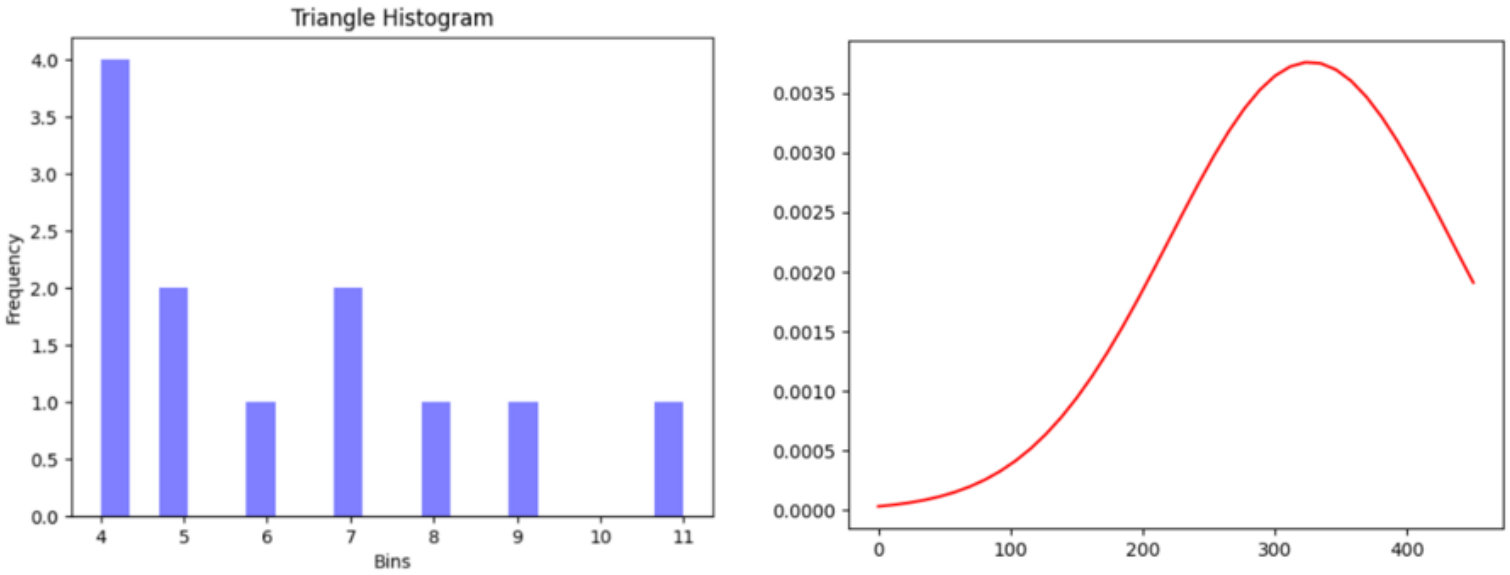


Figure 29 - Histogram and graph from image processing

Rods Results

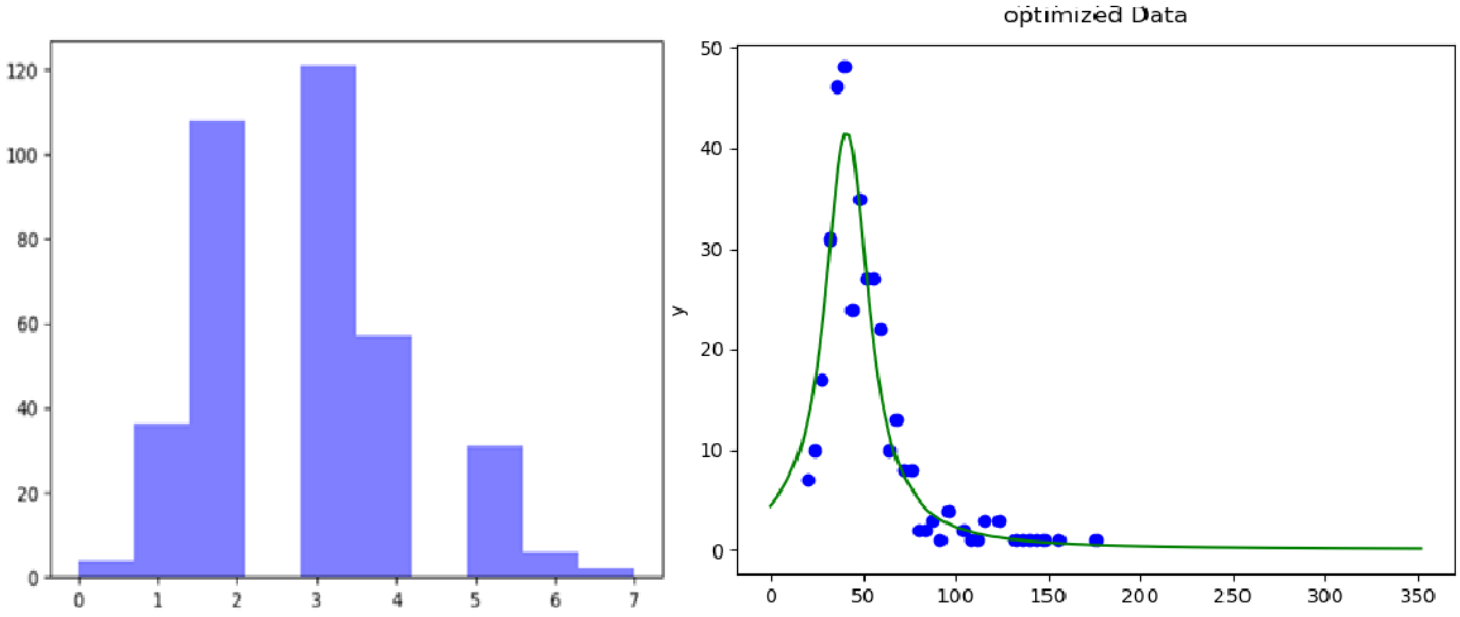


Figure 30 - Optimized Histograms and Graph

It seems like the rods could also be optimized a some more, which was expected, as the rods were the most problematic to detect and count, due to do both low image quality and lack of reliable line prediction.

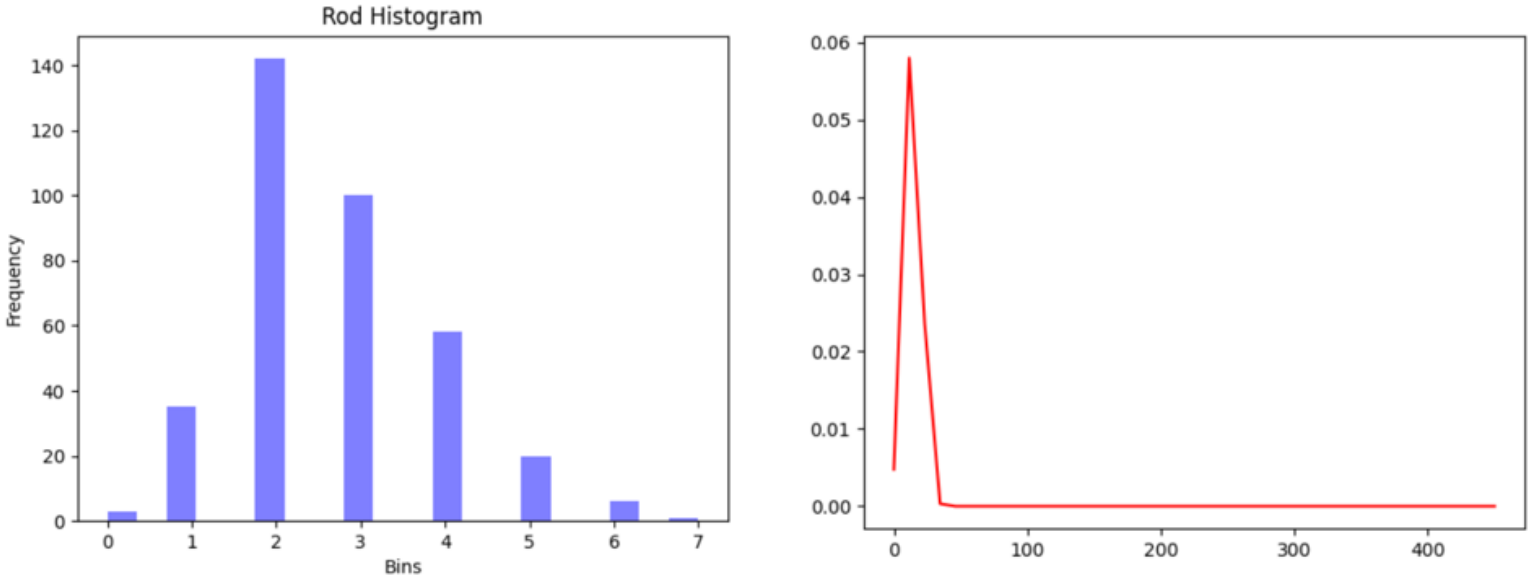


Figure 31 - Histogram and Graph from Image processing

Discussion

As detailed above in the Image Processing portion of the report extracting all relevant data from the pictures is not an issue and even some data that was not asked for in the project description was included though some of the data could be better fine-tuned if more sample pictures were available.

Code Architecture and Threading:

Overall the project was a success, the original goals were all completed.

While the application was designed for being used with a generic set of images with rods/triangles and spheres, the following improvements would result in more versatility, however owing to time and resource constraints were not implemented:

1. Modularization: Isolated test modules would allow for a deeper understanding of the behavior of the different code segments allowing for easier integration of processes specific to a new cluster type as was done with the rods
2. Threading: Modularized code would lead to smaller work units that can easily be threaded for e.g. The output of the region segmentation could be individually worked on, which would lead to faster individual operations both for particle counting and feature generation for the cluster classification
3. Object oriented Programming: A generalized class structure would help with a cleaner more modular approach which would aid in a cleaner data acquisition and data visualization processes

Image processing:

1. Errors in watershedding: Watershedding, despite all the added filters, to attempt to remove noise fails on the rods, would imply that there is a need for some sort of optimization on the parameters, however, this would need a more in-depth analysis of the watershedding function and the image
2. Advanced Filtering: submodules of OpenCV provide cleaner image filtering which was not completely explored for e.g. Gaussian mean shift filtering, we would have in general liked to spend more time studying the shot noise in the image and its removal techniques.
3. KNN on Watershed: KNNs can be natively applied with the Sci kit learn library to watershed images, this was not explored any further owing to time constraints
4. Probabilistic Hough lines: OpenCV has a native Probabilistic Hough line transform, which we did not use but seemed more promising than the generic Hough lines method that we used
5. Canny edge detection: Canny edge detection uses a sequence of filters, to generate a cleaner more appropriate image boundary, but was not explored further, owing to limitations of time

6. Poly curve fitting: it would have been interesting to attempt to attempt to classify images based on the polygon that had the highest degree of fit with the clusters, however, we could not figure out a method to implement it in the given time

Classification:

Our classifier is heavily reliant on the fact that the rods image processing leads to a rather extreme set of easily identifiable feature which is a ridiculous ratio of watershed clusters per particle as opposed to a nearly 1:1 ratio of particle to watershed clusters in the other 2 images, this forms an extrema, while circles on the other hand which have a nearly 1:1 ratio of the watershed clusters to Hough circles form a second extrema, the thresholding is completely manual and would require a larger data set to actually validate.

As explained and shown in the image processing portion of the report, extracting all relevant data from the pictures is not an issue and even some data that was not asked for in project description was included.

Some areas, such as the the threshold of the Hough-circle detection could have been fine-tuned more, but this is difficult to achieve without a greater sample size.

While machine learning was briefly considered we found that, in a project of this scale the possibility of predicting the number of clusters and reliably acquiring other features is not feasible with how few features arethe low sample size currently being generated and if it would available, but some simple code could be plausible to testin an attempt to train the generated program to be more in line with the optimized output, thus potentially increasing the reliability of the algorithm with the current quantity of samples.

Optimization

Every part of the optimization aspect of the project description was covered and accounted for and in the end, the. The data found was displayed and discussed. The main takeaway from thattakeaway was that both the triangle and rod pictures could have been optimized more, which would be possible with the help of more advanced OpenCV functionality, but it was considered more appropriate to stay within the bounds of the course contents.

Conclusion

Throughout this project the group was able to deepen their understanding of Python-based programming and become more proficient in categorizing image processing and optimization-related problems using common image processing and optimization algorithms.

As seen from the results, the peaks of the optimized images correspond somewhat to the peaks of the processed image, but the difference is still substantial enough so massively shift the histogram. As seen in the images the following inferences can be drawn:

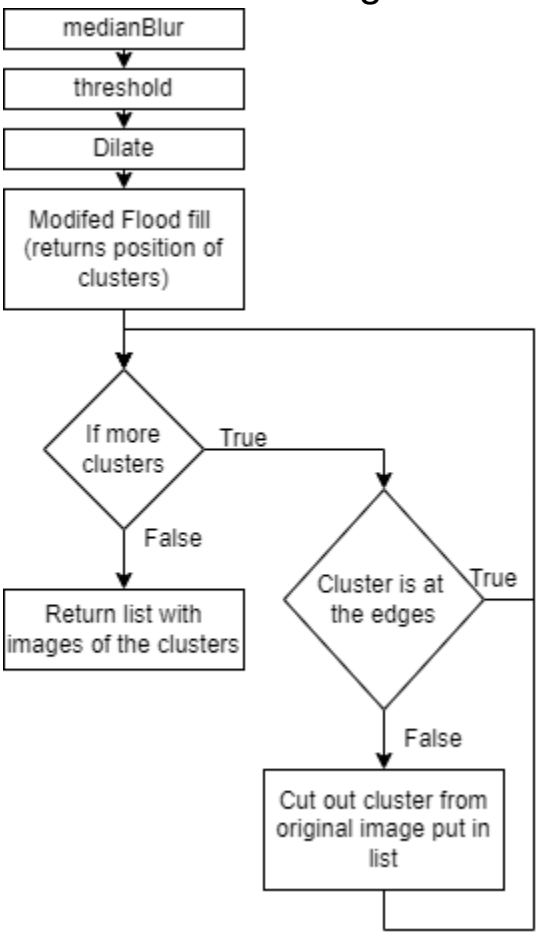
1. Circles : Watershed was occasionally missing or counting some extra particles while trying to count particles in the image, however when optimization was run, and the mean of the plot (the value of x at Ymax of the generated lorentzian) was used to determine the total particles in the clusters, the erratic counting seems to have been drastically reduced.

2. Triangles: Watershed, despite all the filtering, was detecting a significant amount of extra particles in the image, which were clearly segments of particles that could not be completely retained post processing, leading to outliers which were clearly visible. On recalibration of the image with the optimized curve the outliers seem to be reduced. The errors in the result can be attributed to having only 12 samples for handling the data

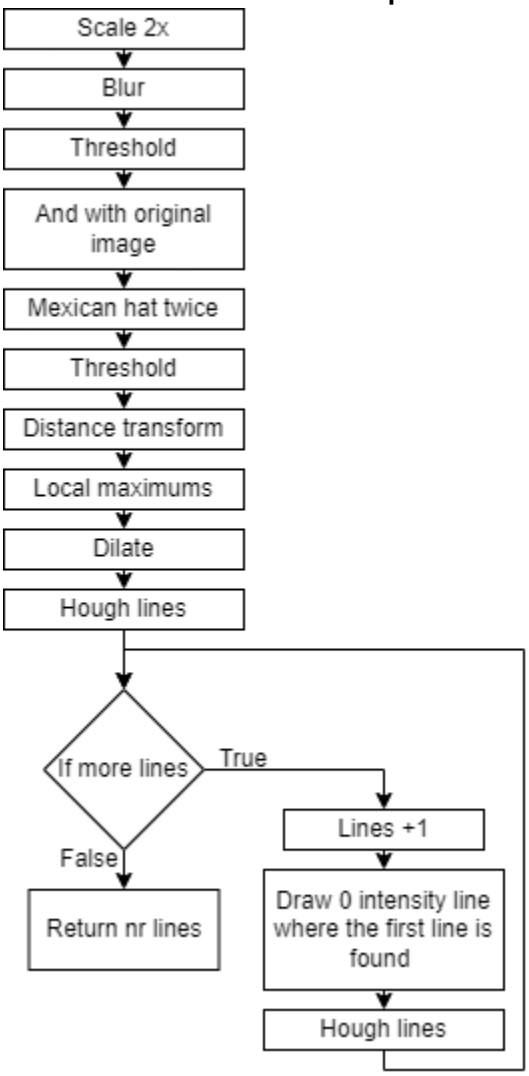
3. Rods: Watershed completely failed. With the modified line counting the algorithm added a negative bias to the original histogram, which on being recalibrated seems to have shifted the histogram forward, potentially removing the negative bias.

Appendix

A.1 - Flow chart segmentation



A.2 - Flow chart rod processing



A.3 - Attempt at line detection and shape identification using the OIP library

An attempt was initially made at using the OIP library's hough_lines() function for line detection. This would also be used to identify clusters of rods or triangles. The code would be run only if less than three circles were found in the image before that, meaning that it would likely not be a picture with clusters of circles.

```
Nth = (np.floor_divide(M,2)).astype(np.uint8) # number of THETA values in the accumulator array
Nr = (np.floor_divide(N,2)).astype(np.uint8)  # number of R values in the accumulator array
K = 30
```

First, the number of theta and r-values for the accumulator array would be set, along with the K number of strongest lines. The K would be set to a value much larger than the expected number of actual lines in the image. This was done to increase the likelihood of identifying all lines, as the hough_lines() function would commonly identify multiple strong lines over just a few of the actual lines.

```
Acc, MaxIDX, MaxTH, MaxR = hough_lines(img_edge, Nth, Nr, K)
```

The hough_lines() function would then be run using the image, theta, r and k-values, returning the accumulator array, the indices of the maximum values and most importantly the theta and r-values of the strongest lines.

```
avg_angles = []
for line in range(K):
    #plot_line_rth(M, N, MaxR[line], MaxTH[line], output_axs[count-1])
    avg_angles.append(np.average(np.abs(MaxTH - MaxTH[line])))

avg_angle = np.average(avg_angles)
```

A new list is created and a for-loop in range K is started. Here, the lines can optionally be plotted. The average of the difference between each line and all the other lines is added to the list. Following the loop the average of all these values is taken. Clusters of rods would fairly consistently give an average difference in angles of about 1 radian, while clusters of triangles would return more variable, but still somewhat consistent values of around 0.7, with some major spikes. This would be enough to identify both overall rod and triangle images.

A.4 - Line filtering

The filter lines function was an attempt at filtering out duplicate lines from a large number of detected lines and getting the correct number of particles in a rod cluster.

```
MaxTH, MaxR = filter_lines(MaxTH, MaxR, 1, 10)
```

This function takes the theta and r-values and a minimum offset of theta and r. It is meant to remove closely overlapping lines which have very similar theta and r-values, thus removing duplicate detected lines.

```
add_line = True
THout = []
Rout = []
THout.append(MaxTH[0])
Rout.append(MaxR[0])
```

In the filter_lines() function, a boolean and two lists are created. The first values are added.

```
for i in range(1, len(MaxTH)):
    for j in range(len(THout)):
        if (MaxTH[i] < (THout[j] + th_range) and MaxTH[i] > (THout[j] - th_range)
            and MaxR[i] < (Rout[j] + r_range) and MaxR[i] > (Rout[j] - r_range)):
            add_line = False
    if add_line:
        THout.append(MaxTH[i])
        Rout.append(MaxR[i])
    add_line = True
```

A for-loop and a nested for-loop are started. The outer loop iterates a variable i from 1 to the number of input theta (and r) values. The inner loop iterates the variable j from 0 to the number of values in the output theta list. An if-statement checks if the current theta and r-values from the input array are within the range of the any of theta and r-values in the output list. If both are true, the add_line boolean is set to false. After the latest values are compared to all values in the output list, an if-statement checks the value of add_line. If True, the values are added to the output lists. The boolean is reset to True and next iteration of the outer loop then begins, or the loop ends. Lastly, the output lists are returned.

While this function did show some success, the many falsely detected lines from the hough_lines() function would still remain. Thus, using this along with hough_lines() was abandoned as a way to find the number of rods.