Automated Microcarrier Detection

USING MATLAB

JAMES BRAY

2016

Contents

Method outline

Initial plans

In order to begin tackle the problem I first had to conduct heavy research into the very basics of image processing as I was a real novice on the subject having never even read into the topic before. From my research I devised a basic outline of a process that would first be able to remove noise from the image then find relevant edges and translate those edges into possible circles.

My initial plans were as follows;

- Threshold the image
- Remove small objects and fill voids in others
- Perform a watershed transform to separate out joined circles
- Calculate roundness of remaining objects. Roundness is calculated by $4 \times \frac{Area}{\pi \times (Major\ Axisx)^2}$

Initial results were positive. The method was extremely fast, taking a fraction of a second to complete. It was also fairly accurate with the first set of test images I was given. It being able to sort through them all with 100% accuracy. However it quickly became apparent after receiving a second set of test images that something much more advanced would be required. The initial thresholding of the image would not work on the actual apparatus as the light levels were inconsistent across the background with many images containing a large “halo” of light in the center. This threw off the succeeding method thus requiring a new one be devised.

I was also being limited by imageJ the software in which I had been working. I was having problems with the limited documentation available both officially via the official user guide and unofficial on forums and in articles. This coupled with the very limited scripting language meant that in order to go further I would need to move into a more capable software package with greater documentation.

Further Development

After looking into alternative software, one package came out as a clear winner. MATLAB has a tried and true scripting language, well documented commands and a lively community releasing functions and other bits of code as well as helping out on both official and unofficial forums. Its toolset is also considerable more advanced than imageJ with more advanced edge detection and additional transforms available to use. With this new toolset I began devising a new method. What I came up with is as follows.

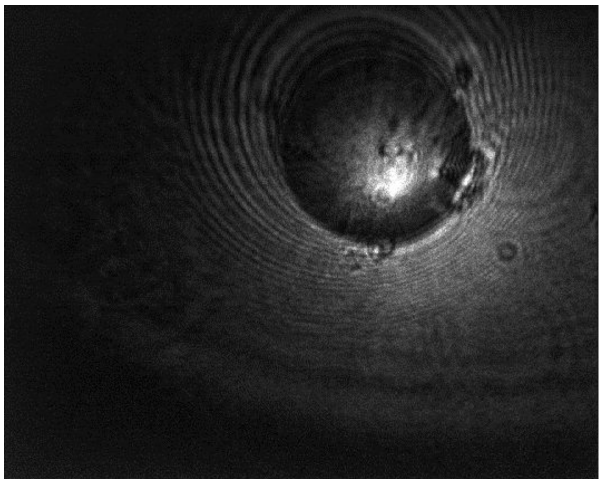
- Remove noise
- Detect edges
- Remove small lines
- Dilate lines in image
- Detect circles

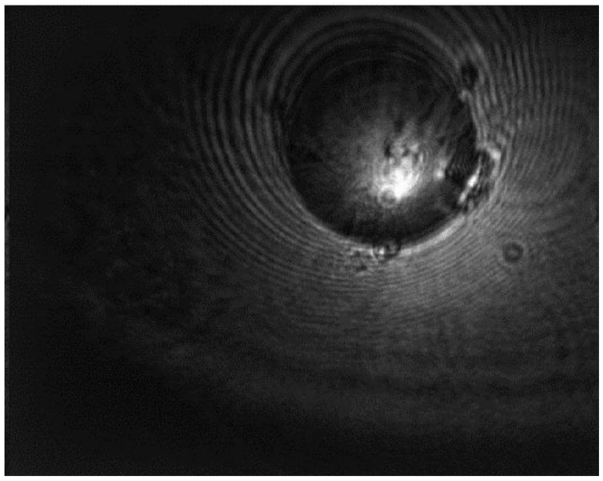
The preceding document sets out the specific of this method and the decisions that were made during its formation

Noise Removal

The images produced by the camera contain shot noise especially in the darker arias of the image. This poses problems for edge detection algorithms with many detecting false edges around arias of noise. Thus to improve the any edge detection algorithm removing this noise was important.

After researching into noise reduction technique and the already completed work I came to the conclusion that the best method for removing this noise from the images was a Gaussian blur with a [45x45] kernel. Below is an example image with the method chosen.

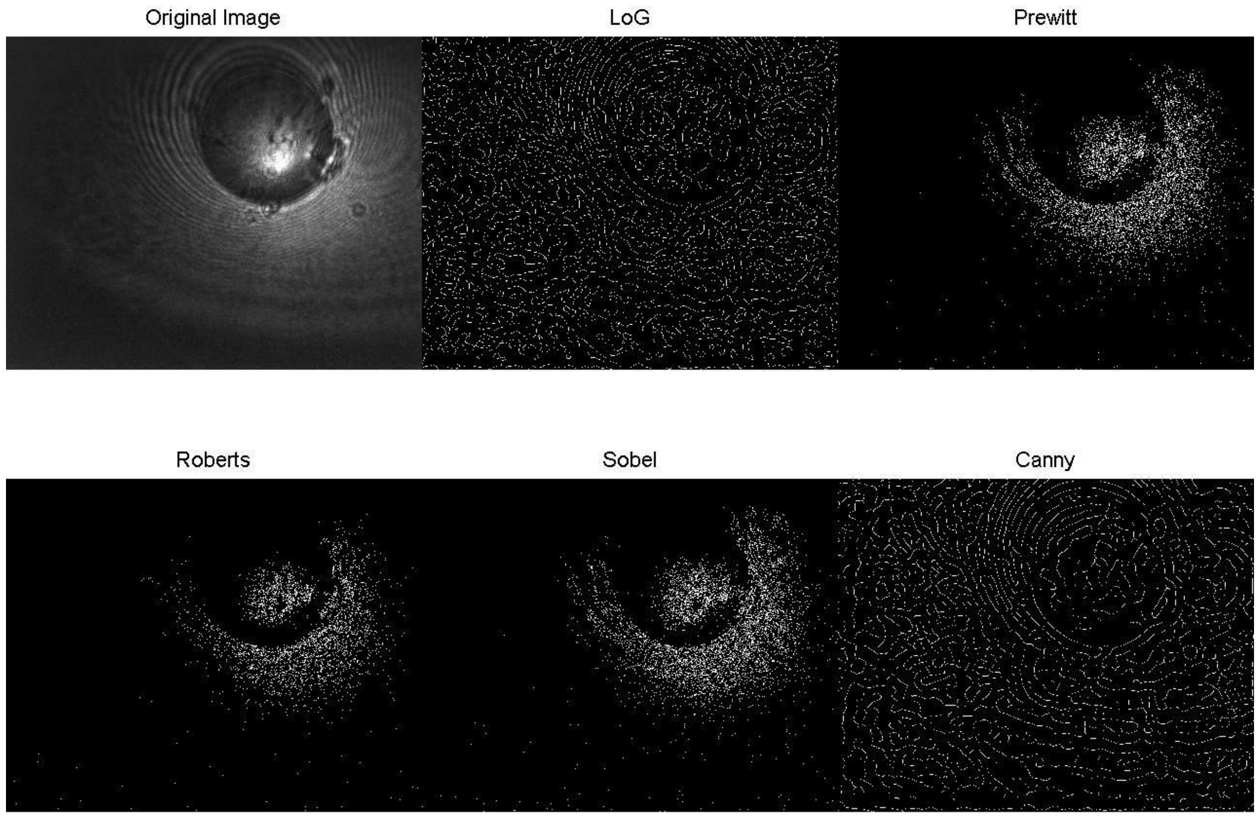

Original Image


Gaussian Blur

The final implementation is on lines 34 and 35

Edge Detection

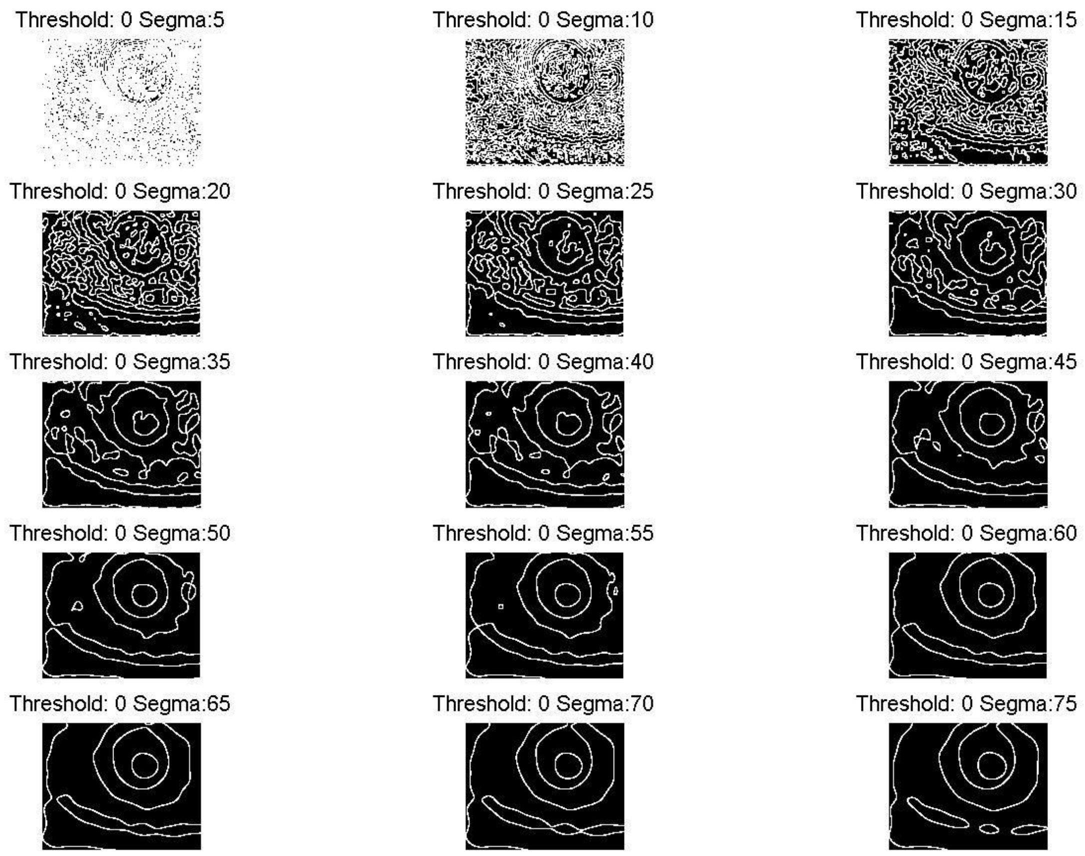
Once the image has gone through noise reduction the edges of any microcarrier present must be detected. There are various method of accomplishing this many of which are implemented in MATLab. Below is a chart comparing different edge detection methods using the image above.



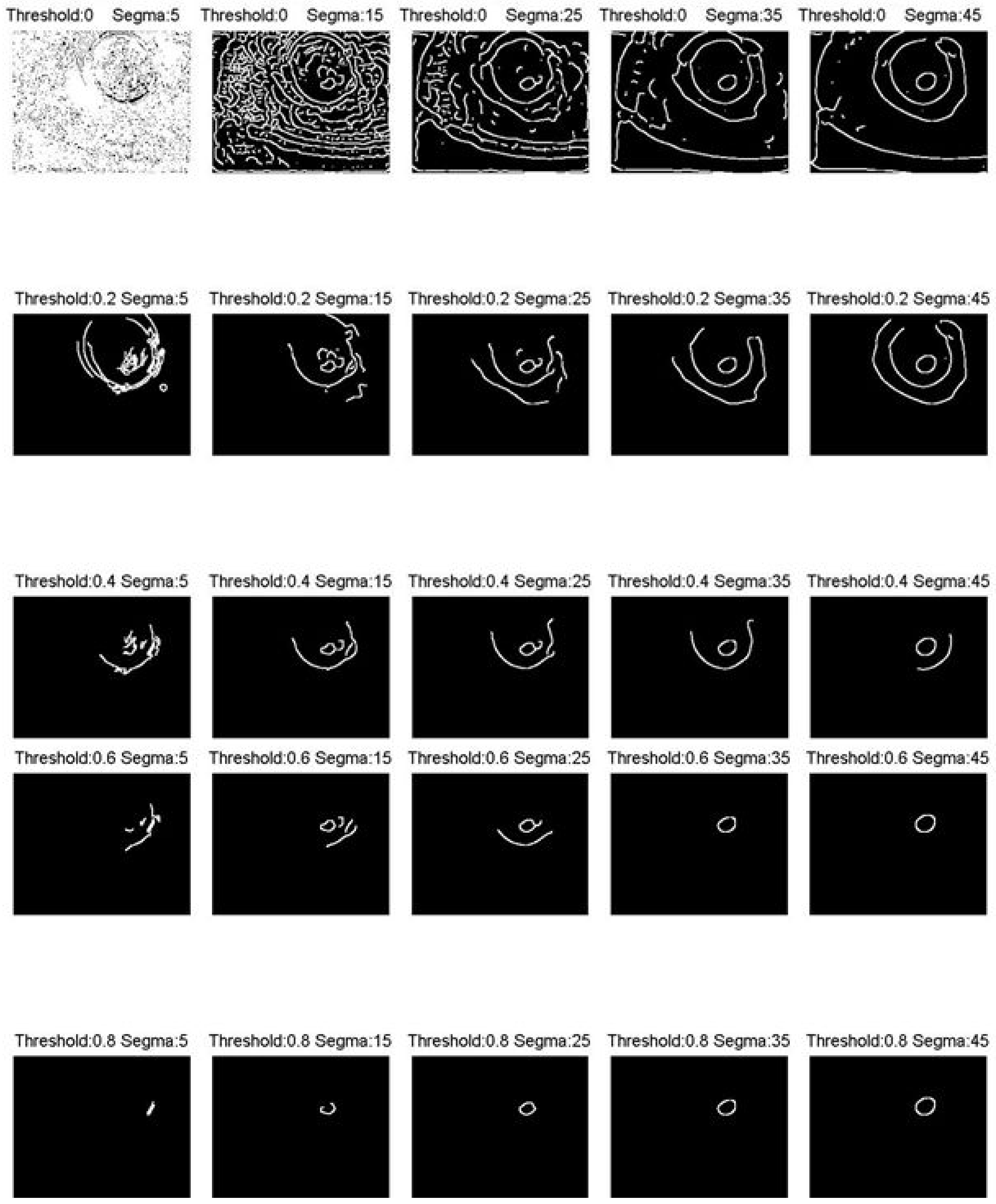
The most desirable algorithm will pick out consistent, connected edges around visible microcarriers and be tunable to remove any remaining noise, these conditions are met by both Laplacian of Gaussian (LoG) Kong, H., Akakin, H., & Sarma, S. (2013). and Canny edge detection methods, Canny, J. (1986).

As previously mentioned both algorithms are configurable. Canny has a threshold and sigma "sensitivity" value whereas Laplacian of Gaussian only makes use of the sigma value. Below is a grid containing the output of the 2 algorithms using the same image as above, the pixels have been dilated for ease of viewing

Laplacian of Gaussian



Canny

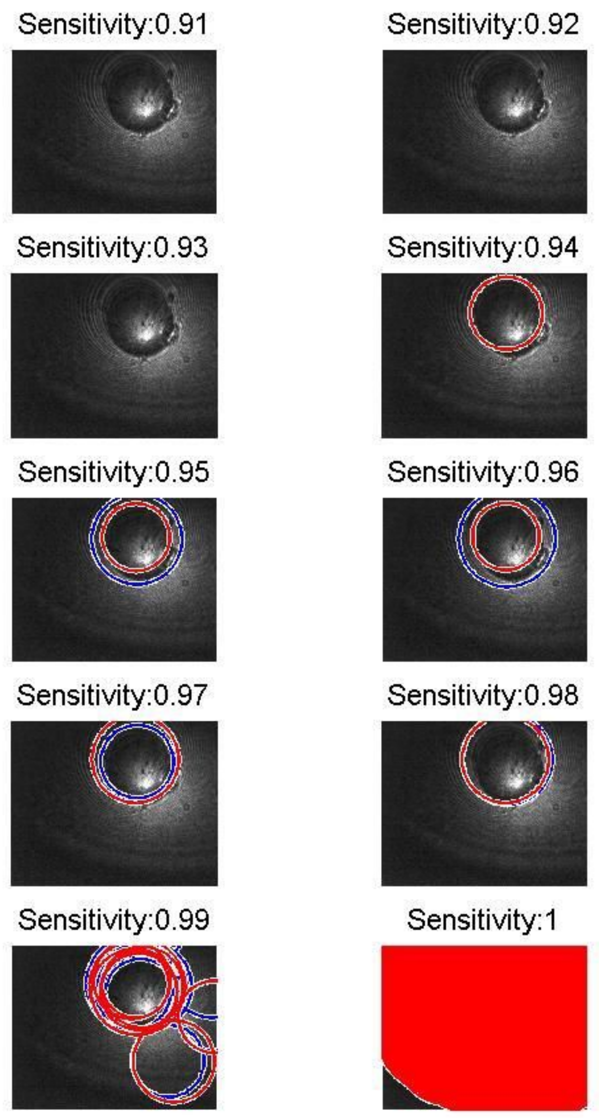


The best result at this stage is one with a clearly defined circle with little to no background information, as such a canny edge detection with a threshold of 0.3 and a sigma of 12 was chosen after conducting more tests on other good and bad images.

The final implementation in on line 38

Circle detection

A Hough transform Duda, R.O. & Hart, P.E. (1972) is by far the most effective of detecting circles within an image. MATLAB has a built in function for just this purpose. This function accepts an image, sensitivity, radius range, object polarity and method. We are using the Hough transform in this case called ‘two stage’. The function returns a matrix of circle radii and a 2 x n matrix of [x,y] coordinates containing the centers of the circles found. This data is then used to discard circles whose edges go off screen. It was however important to get the sensitivity just right. Too ‘picky’ and it would miss circles present, too lenient and it would see circle that weren’t in the original image. Below is the sample image from the figures above. The algorithm is running on the image generated using the line detection parameters chosen in the above section, dark circles are shown in red and bright ones in blue.



After testing the algorithm more on this and other images a value of 0.955 was chosen, this best represented the wide range of circle intensities collected

The final implementation is on lines 89 to 95.

Parallel processing

The speed as well as the accuracy of any algorithm developed was a crucial factor for its eventual usefulness. After running the algorithm over 1000 images each one on average took an average 2.084 seconds to process running on a single core of a 2.40GHz CPU. This is not a desirable time. Although the algorithm does not have to be real time it does be to processes images at a reasonable speed. Thus a way to increase the speed of the algorithm was required, the obvious path to take in this case was parallelization.

After researching possible methods of parallelization I discovered there were two different approaches;

- Process one image but break the computation up and run each part simultaneously on different GPU cores
- Process multiple images simultaneously on either different CPU cores or GPU cores

However I soon ran into problems when trying to implement GUP functionality in MATLab. After reading the documentation for the MATLab edge function I discovered that canny edge detection was not supported on a GPU. This ruled out running the algorithm on a GPU but the benefits we would gain were mainly in the field of latency, something that is not particularly for the application being implemented.

That left the only option being to run the algorithm on multiple CPU cores. To do this in MATLab a parafor loop is used to loop though the function loading images in parallel into all 6 test cores. After running my test rig again on the same image set the time per image came down to an average of 0.60576 seconds per image a much more reasonable time and one that would make it possible to batch the backlog of images effectively.

This testRigPool script implements this parallel processing scanning a directory for any changes, processing all the files it finds in parallel moving good ones into a Good folder and rejects into a Bad folder. It then deletes all files from the scan directory before going back to scanning for directory changes.

Camera interface

In order to fully automate the process of image processing in MATLab the camera capturing images from the microscope would have to be interfaced with the system. MATLab has an inbuilt image acquisition toolkit designed to interface with scientific cameras. However the Thorlabs camera used in the microscope is not supported. Thorlabs lists MATLab compatibility using “mex” files on their website. However after extensive research all I found was an outdated script on the MATLab forum that would not compile and other people having the same problems that I was encountering, thus I decided to put that idea on hold and explore other avenues.

After exploring the Thorlabs software further I discovered a far the easier way to interface the camera with MATLab and ultimately this is the implementation I used. The camera's built in software is able to place images into a directory. MATLab then scans that directory for any new files. If files are found the algorithm in parallel moves them into good and bad directories depending on if any microcarriers are detected. It deletes processed files from the original directory as it goes along. This simple solutions allows for automatic image processing whilst running the software in parallel.

Conclusion

When starting the project I had a list of requirements in mind that would ensure that the project was a success, these are discuss below:

Speed

In order for the algorithm to be considered a success it be able to process images quickly, ideally at 5 Hz (2 tenths of a second) although this speed was not essential if the algorithm could process the image backlog eventually.

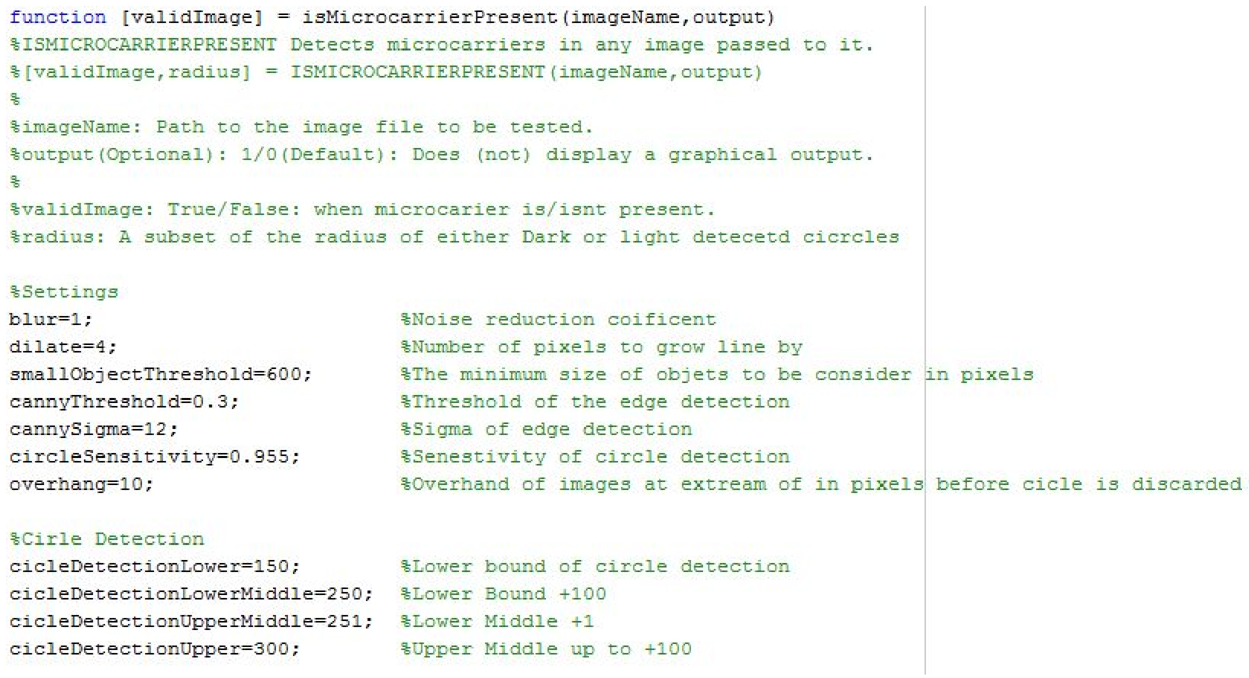
After conducting extensive tests on 1000 images from actual apparatus in actual time of 2.084 seconds per image was actually achieved, over 10 times too slow However running the algorithm in parallel reduces this time to 0.60576 seconds per image, a much more reasonable number and although not the 0.2 seconds required to achieve 5 HZ it should be sufficient to allow the algorithm to run posthumously and in reasonable time.

Accuracy

An algorithm is no use if it is ineffective at detecting microcarriers or gives false positives. However this proved difficult not due to the complexity of the task but rather lack of available high quality data. All images I received contain a lot of interference patterns that would not be visible in the final equipment, making it impossible to tune the algorithm to make it more accurate as these seriously messed with edge and circle detection.

Configurability

As it is not yet possible to tune the algorithm yet a new requirement was added. In order for the code to be tuned at a later date it was essential that any code was easy to configure. To that end at the top of the main file is this set of parameters, allowing for full control of every aspect of the algorithm with descriptions of their function, making tuning easy.

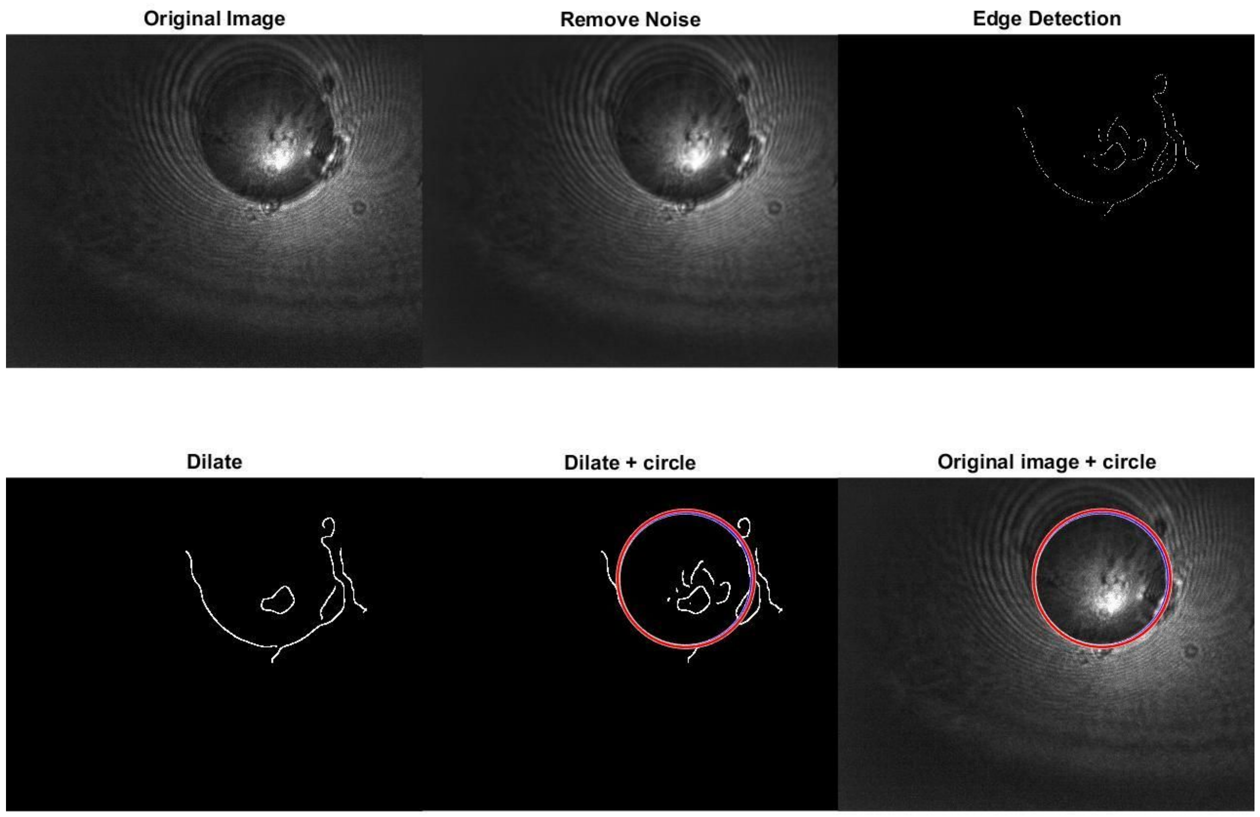


```
function [validImage] = isMicrocarrierPresent(imageName,output)
%ISMICROCARRIERPRESENT Detects microcarriers in any image passed to it.
%[validImage,radius] = ISMICROCARRIERPRESENT(imageName,output)
%
%imageName: Path to the image file to be tested.
%output(Optional): 1/0(Default): Does (not) display a graphical output.
%
%validImage: True/False: when microcarier is/isnt present.
%radius: A subset of the radius of either Dark or light detecetd cicrcles

%Settings
blur=1;                         %Noise reduction coificent
dilate=4;                       %Number of pixels to grow line by
smallObjectThreshold=600;       %The minimum size of objets to be consider in pixels
cannyThreshold=0.3;             %Threshold of the edge detection
cannySigma=12;                  %Sigma of edge detection
circleSensitivity=0.955;        %Senestivity of circle detection
overhang=10;                    %Overhand of images at extream of in pixels before cicle is discarded

%Cirle Detection
cicleDetectionLower=150;        %Lower bound of circle detection
cicleDetectionLowerMiddle=250;  %Lower Bound +100
cicleDetectionUpperMiddle=251;  %Lower Middle +1
cicleDetectionUpper=300;        %Upper Middle up to +100
```

Interfacing with existing code

All the existing project is written in MATLab with the existing scripts reading images from a directory, this could easily be automated with my system either by having it read directly from the “Good” directory or by having my code call the existing code on detection of a good image passing the image variable across.

Below is an output of the microcarrier detection program for a sample image, this can be achieved by setting the output flag to 1 in script, isMicrocarrierPresent


Original Image
Remove Noise
Edge Detection
Dilate
Dilate + circle
Original image + circle

Reverences

Duda, R.O. & Hart, P.E. (1972). Use of the Hough Transformation to Detect Lines and Curves in Pictures. Communications of the ACM, 15 (1), 11-15. Retrieved from https://www.cse.unr.edu/~bebis/CS474/Handouts/HoughTransformPaper.pdf.

Canny, J. (1986). A Computational Approach to Edge Detection. TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, PAMI-8 (6), 679-698. Retrieved from http://cmp.felk.cvut.cz/~cernyad2/TextCaptchaPdf/A%20Computational%20Approach%20to%20Edge%20Detection.pdf.

Kong, H., Akakin, H., & Sarma, S. (2013). A Generalized Laplacian of Gaussian Filter for Blob Detection and Its Applications. TRANSACTIONS ON CYBERNETICS, 43 (6), 1719-1733. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6408211.

MathWorks. Find edges in image intensity. Retrieved from http://uk.mathworks.com/help/images/ref/edge.html.