Analog Gauge Reading

EENG 510 - Final Paper

Gilchrist Hoshiaki Day
Introduction
Analog gauges are widely used in today’s industry. Even with the emersion of digital meters, some systems have not conformed due to cost, time, or other reasons. The biggest distinctive feature between the two gauges is that the digital gauges provide a straight forward reading. For analog gauges, the technician has to decipher the value from the surrounding values if it isn’t on one of the labeled displayed values, and then perform some type of in-head linear interpolation estimate to obtain a final reading. Basically, analog gauges are prone to errors, but what if we could just snap a picture of an analog gauge and run it through a program determine the value? This would limit the chances of an error occurring from either the technician’s quick interpolation or lack of attention to detail. This paper will discuss how to implement such a system, as well as previous work done on analog gauge readers and previous methods or solutions, the approach taken to the problem, the other encountered issues along the way, and the results.

Previous Work
Previous work read on this topic have shown the importance of detecting the analog pointer on the gauge. One paper, “Automatic Reading of Analog Display Instruments”, mentions an approach to design a system for general use and allow some anomalies to occur [1]. Also, the motion of the pointer is important to the detection and reading of the value. Because most analog displays move in a circular fashion, there is a relation between angle and displayed value. This paper mentions square analog readers, but that was irrelevant to this project as circular gauges were used.

Another method used to detect the pointer was found in a paper called “A Computer Vision System to Read Meter Displays”, which used the Canny edge detector along with the circular Hough transformation [2]. First, a frontal image of the gauge is taken as a baseline. Then, images can be subtracted from the baseline image to obtain a new image, the difference between the image and baseline. Next, the Canny edge detector is applied to the image difference along with the circular Hough transformation to obtain the center of the pointer.

Similar to this, the use of morphological operations, such as thinning, along with the circular Hough transformation could help locate the pointer. This method was used by Xiaofeng Ye, Dailiang Xie, and Shan Tao in their paper “Automatic Value Identification of Pointer-Type Pressure Gauge Based on Machine Vision” [3].

Experiments and Results
The approach taken on this problem can be broken down into three categories: detect gauge circle, determine where the pointer is, and output the corresponding gauge value. It should also be noted that this is the general approach; the end actual route taken will be discussed with the stated categories above as milestones.
The first milestone is the most important aspect to this project. Detection of circles was a lot more difficult than anticipated. Initially, the detection of the outer rim of the analog gauge was attempted; however, even with morphological operations coupled with edge detection, it proved to be rather difficult. Next, the Circle Hough Transformation in MATLAB was used, which searches an image looking for circles, dark or light circles, within a radius range specified. The output of this function would be the X-Y coordinates of the midpoint of the circle detected and the circle’s radius. This method proved to yield decent results, but was having a tough time detecting the outer rim of all the analog gauges. Lighting in the picture provided also played an important role in the detection. Because the outer rim of the analog gauge was silver, the reflected light off the rim caused the circle to be discontinuous at certain points on the gray-scaled image. The reflecting of light in the outer rim can be seen below in Figure 1.

After several attempts, a different approach to this problem was considered. The black pointer on the white background of the analog gauge was a distinguishing feature noticed in all images. Additionally, the center of the pointer had a dark circle which could be used to determine the center of the pointer using the same method previously used. Using the ‘imtool’ function in MATLAB, the radius of each dark circle was noted and a radius range was created. Once again, the Circle Hough Transformation was applied. The minimum radius (Rmin) used was 12 pixels and the maximum radius (Rmax) used was 35 pixels. The results were successful detection of all the pointer circles along with some other circle detections. The false detections did not hinder the progress as long as all centers of the gauges were detected. This was the most problematic part of the project – successful detection of all the gauges. Results of the detection are shown below in Figure 1 and Figure 2. This was also successful for Images 2, 3, 4, 6, and 9.

Figure 1: Image 1 - Circle Hough Transformation
Issues encountered during the detection process were that all images were not taken at the same angle and/or scale. For instance, image 1 (shown in Figure 1) did not encompass all the gauges fully. Mainly, the first gauge is cropped off. Likewise, there was images, such as Image 7 (shown below in Figure 3), that the picture was taken at a distance further back. This posed a problem where the minimum radius (Rmin), which had to be reduced just for these distant images. This also caused the increase in more false detections on the other images when trying to generalize this process for all images. This issue was noticed on images 5, 7, 8, and 10.
The next issue encountered was the angle at which the picture was taken. If the picture was taken at a slanted view, such as image 8 (shown below in Figure 4), there could be a bigger range in radius (Rmin and Rmax), thus some gauge centers could be missed. Additionally, the angle of the picture could alter the appearance of the gauge pointer center such that it could appear oval-like, thus getting missed by the circular detection.

![Image 8 – Circular Hough Transformation](image)

Unfortunately, there was no way to access these gauges and take pictures myself; therefore, some assumptions had to be made before continuing. The first assumption is that all pictures would frontal pictures of the gauge. This would help remedy the issue with change in range of Rmin and Rmax; thus, a generalized range could be determined. The next assumption would be that the picture taker, or technician, would be ‘trained’ such that all gauges would be encompassed within the image. Basically, the left-most gauge would be at the edge of the left side of the image and the right-most gauge edge would be on the right side of the image. This is similar to Image 8 shown in Figure 4, but without the angular distortion of the gauge layout.

To implement these assumptions, the best that could be done is to reorient, or rectify, these images so that they all ‘appear’ to be a frontal view with minimal angle. Using MATLAB’s ‘cpselect’ function, all the pictures were rectified to ‘appear’ like image 2 (shown on next page in Figure 5). The ‘cpselect’ function in MATLAB allows the user to take two images and create reference points between the two images. The function then creates a new image that uses the reference points determined by the user to orient the second image to the first. In this case, all centers of the gauges were marked and corresponded to the second image. Also, the
corners of the black label strips shown above and below the gauges were marked. This process also shrunk the size of the images that were rectified from the original 1456x2592. The solution to this was to resize the images back to the original 1456x2593 by using the MATLAB command ‘imresize’. The code for rectifying the image can be seen in Appendix B at the end of this report.

This fixed two of the 3 problematic images. For Image 10, it was determined that the quality of the image wasn’t good to begin with and it was taken at a greater distance; therefore, the image will be discarded. The rectified images for 8 and 5 yielded a successful detection with the previous Rmin and Rmax value so these images were kept. The results of successful detection on Image 8 can be seen below in Figure 6.
Before moving onto the pointer detection, some of the false detections had to be discarded. It was noticed that all five gauges correlated somewhat around a point the $y$-axis of the image. The easiest way to eliminate the outliers outside this area was to take the mean of the $y$-coordinates and then find out if they fall within a variance around the mean. If they do, the coordinates of the midpoint of the circle is kept, otherwise discard. A correlation code was considered to find out which 5 correlated the best; however, this could be problematic for images that had less than 5 detections and for images with false detections near or around the pointer as seen in Figure 3. The more general approach was taken in this case, and it was found that this method worked for all images. Additionally, it provided a quick way to get rid of the outliers in the false detections. For example, the circle detected in the upper-middle in Figure 6 was quickly eliminated using this method. After the outliers were eliminated, the image would then be concentrated around the mean area which can be seen below in Figure 7.
The next milestone was to figure out the pointer and the direction. Also, this process would also have to determine if the circle point detected had a pointer. The approach taken for pointer detection was to go through each detected circle coordinates and extract pixels at a determined radius (Rs) around the circle’s center. Logically, it should be easy to determine if it was a pointer if the circle had a head and a tail when examining the pixel extractions. Additionally, the head of the pointer would be shorter and thin, whereas the tail end would be wider when plotted. Figure 8 illustrates the pixel extraction radius (Rs) in yellow with a radius 40 pixel, and Figure 9 shows two plots of the extracted pixels from image 6.
From the pixel extraction method, a grayscale threshold value could be determined. The general threshold value used was 80, which was determined as a good value from the other images this method was applied to. By thresholding these plots, it allowed a way of edge detection for the head and tail of the pointer. A visual representation is shown in Figure 10.
From *Figure 10*, it is easy to distinguish which is a pointer and which is most likely a false detection. Now that the edges of the pointer head and tail have been determined, both the widths are recorded. Next, a second filtering method was implemented in order to eliminate the remaining false detections. Symmetry of the pointer’s head and tail played a key role in this part. From the pixel logic plots, the pointer would check each of the first 180 points, that is the first 180 degree’s of the pixel extraction, and compare it with the value on the opposite side, or 180 degree offset. A true pointer will have an overlap of matching high values for the duration of the pointer head, or some overlap if it was the tail compared to the head. This can be seen on the next page in *Figure 11*. 
This method of eliminating the false detections worked for all the images but image 4 and image 9. Image 4 appears to be in bright light; therefore, the threshold value is different and makes it harder to distinguish between darker and lighter pixels. This can be seen in Figure 12 on the next page. For image 9, the distance the picture was taken at does not allow the pointer head to appear in the image well. This is appears as a 1-2 pixel width in the pixel extraction and threshold plot.
From the pixel overlap graph (shown in Figure 11), the center of the pointer can be determined as the mid-point of each rise. The code looks for the middle of each rise and logs the nearest index, which corresponds to the angle in degrees. The matrix, labeled “L” in the code, stores this information and is formatted such that the first rise width is in the first column, the second rise width is the second column, and angle of the midpoint of the first rise is stored in the third column, and lastly, the angle of the midpoint of the second rise is stored in the fourth column. The minimum value between the first two columns will be the pointer head, and if it is in the first column, it will take the index in the third column. Otherwise, if the smaller value is in the second column, it will take the value in the fourth column.

The last milestone is determining what the value of the pointer. Taking a closer look at the gauges, it was noticed that there are two different gauges used in these images (shown in Figure 13). The left-most gauge in all the images was different from the other four displayed. Because determining the type of gauge was out of the scope of this project, a user input was made which told the program which look-up table to use. The approach taken was to first determine what angles on the gauge correspond to what values listed on the gauge. If the value was between two listed values, linear interpolation would be needed to accurately determine a value.
Next, a pseudo pointer radius (Rp) was created for calibration in order to create the look up table. This pseudo pointer went from the detected centers out 120 pixels. By trial and error, each of the angles for the displayed value points, which are shown circled in green in Figure 13, were obtained. Because the look-up table was generalized in accordance to the assumptions, there is some slight error depending on the angle the picture is taken.

![Figure 13: Two gauges are shown being calibrated for Look-up Table](image)

After the two look-up tables were created, the user would predetermine which gauge’s look-up table are to be used. The user input would be how the gauges are shown in the image from left to right. When running the script, the user will be asked first “How many gauges are there?”, followed by “Input the gauge type, 1 or 2?”. Gauge 1, or input 1 in the array, will corresponds to the first ‘odd’ gauge and Gauge 2, or input 2 in the array, will be the look-up table for the other type of gauge.

The script then takes the value of theta found for each pointer head before finding out which two angles on the look-up table does lie between. The two angles that theta lies between corresponds to the two values of the gauge. Next, it will take the corresponding look-up table with the gauge values and perform a linear interpolation. The linear interpolation output will be the reading of the gauge, which is the last milestone. This final value is also rounded such that an integer is displayed rather than decimal values.

Lastly, to compile all the milestones into one output, the code displays the remaining gauge circles detected in blue, the pixel extraction circle in yellow, the projection of the pointer in red,
and lastly, the interpolated value of the gauge, on the original image. The results were successful detections on images 2, 3, 5, 6, and 8 shown in Figures 14-18.

Figure 14: Image 2 - Final Results

Figure 15: Image 3 - Final Results
Figure 16: Image 5 - Final Results

Figure 17: Image 6 - Final Results
The right-most gauge reading in Figures 14-18 is noticed to fluctuate slightly. It was noted that it should be outputting the value 0, and although the values are close, it does calculate values ranging from 1 to 3. This is due to the slight angle of each picture and the assumption that the picture would be a frontal image. Additionally, the actual value for the gauge displayed value 1 is calculated to be is 0.53, which when rounded, adds to the error. Even with this error, the error margin is minimal when compared to ‘eyeballing’ it.

**Suggestions and Improvements**

The results were positive overall; however, improvements could be made on this project if time wasn’t an issue. Some suggestions could be done by either the customer or by improving the system stated in this paper.

One suggestion was that to improve speed and reduce processing power of the computer. Dark circles could be made around the gauges such that the image could be resized to a smaller resolution. Next, the circle for the outer rim of the gauge could be detected before performing the similar process stated in this paper. This is a suggestion that the customer can do on all the gauges before the image is taken. A trial of this suggestion is shown in Figure 19 by using the program Paint to draw black circles around the gauges. The image was then reduced to 0.15 of the original size before the detection of these circles were performed. The results of successful detection of the gauge rims are displayed in Figure 20.
Additionally, a reference point could be made so that the axis of the image could be found and corrected before performing the detection and calculation. This is similar to the ‘cpselect’ command used in MATLAB, and could be an addition to the current method proposed which will ensure a minimum amount of error due to the angle of the image.

Another suggestion would be to write a code that would find the big blocks with the values within each gauge (as shown in Figure 13). Using the pixel extraction and mid-detection methods, the angle look-up table could be self-made for each individual gauge; thus, making the interpolation more accurate for each gauge.
Conclusion
In conclusion, the project was successful because the program was able to take an image, detect the midpoint of the gauges, find the pointer, and output the value. Assumptions were made to assist in successful detection of each gauge midpoint and also to provide assistance of using this method on future images. The biggest problem that occurred in this project was that the images were provided by some randomize picture taking process instead of being obtained through some standardization picture-taking process. Additionally, there were some inconsistencies within the images given such as blur, distance, and angle. This made it hard to detect circles, distinguish between dark pixels and lighter pixels in order to create a threshold, and interpolate in order to obtain a more accurate gauge value. A higher success rate could be achieved if a standard method of taking the image could be established. With some minor tweaking, this project can be applied to any analog reading image. Suggestions and improvements are also mentioned if someone were to continue this project. Overall, five of the original ten images originally used were 100% successful and the elimination of the images that did not yield good results is stated in this paper.
References


Attachments
Images.zip - Zip file containing all images used in this project.

List of images in the zip folder:
- Image (1).jpg
- Image (2).jpg
- Image (3).jpg
- Image (4).jpg
- Image (5).jpg
- Image (6).jpg
- Image (7).jpg
- Image (8).jpg
- Image (9).jpg
MATLAB Code for Project

```matlab
%%
% Final Project for Digital Image
%
close all
clear all
tic
%Load image
I = imread('C:\Users\Hoshiaki\Desktop\EENG510 - Image\Final Project\Detect\Image (9).jpg');
I = imresize(I, [1456 2592]);
Igray = rgb2gray(I);
%imtool(Igray, [])

%Find Dark Circles
Rmin = 12;  %Minimum Radius
Rmax = 35;  %Maximum Radius
[centersDark, radiiDark] = imfindcircles(Igray,[Rmin Rmax],'ObjectPolarity','dark');

%This part just plots the detected circles for troubleshooting
figure, imshow(I,[])
hold on
viscircles(centersDark, radiiDark,'EdgeColor','r');
hold off

%Find mean of y since all circles will be 'roughly' around the same point
y_mean = mean(centersDark(:,2));

%discard any centersDark values above and below +/-200 mean
ymin = y_mean-200;
ymax = y_mean+200;
[r,c] = size(centersDark);
count = 1;
matrix = [];
radius = [];
for i = 1:r
    x = centersDark(i,2);
    if x > ymin && x < ymax
        matrix(count,:) = centersDark(i,:);
        radius(count) = radiiDark(i);
        count = count + 1;
    end
end

%Eliminate images above and below the mean R_pixels
R_pixels = 250;
[x y] = size(Igray);
y_mean = mean(matrix(:,2));
for i = 1:x
    for j = 1:y
```

```
if i > y_mean-R_pixels & i < y_mean+R_pixels
    Igray2(i,j) = Igray(i,j);
else
    Igray2(i,j) = 0;
end
end
end

%*******************************************************************************
% %This part just plots the detected circles for troubleshooting
% red = 40*ones(1,length(radius));
% figure, imshow(Igray2,[])
% hold on
% viscircles(matrix, radius,'EdgeColor','b');
% viscircles(matrix, red,'EdgeColor','y');
% %Label points according to their rows in matrix
% for i = 1:length(matrix)
%    str = ['Point ', num2str(i), '
  \rightarrow'];
%    text(matrix(i,1),matrix(i,2),str,'HorizontalAlignment','right')
% end
% hold off
%*******************************************************************************

%Calculate the circular coordinates for a radius of Rs
Rs = 40;
for theta = 1:360
    Ixr(theta) = Rs*cosd(theta);
    Iyr(theta) = Rs*sind(theta);
end

%Extract pixel values around each circle center at radius Rs from the
%center of the 'detected' circles.
for j = 1:length(matrix)
    for i = 1:length(Ixr)
        Xl = matrix(j,1);
        Yl = matrix(j,2);
        Ix(j,i) = round(Xl + Ixr(i));
        Iy(j,i) = round(Yl + Iyr(i));
        Pixel(j,i) = Igray(Iy(j,i),Ix(j,i));
    end
end

Pixel_Logic = Pixel<80;

%*******************************************************************************
% %Show Image with Pixel Extraction Circle and Pointer Projection Line
% figure, imshow(I,[])
% hold on
% red = Rs*ones(1,length(radius));
% viscircles(matrix, radius,'EdgeColor','b');
% viscircles(matrix, red,'EdgeColor','y');
%*******************************************************************************

%Identify which if the tail and which is the head
count = 0;
k = 1;
for i = 1:length(matrix)
    for j = 1:359
        check = Pixel_Logic(i,j);
        %look for first logic high
        if check == 1
            count = count + 1;
            check1 = Pixel_Logic(i,j+1);
            if check1 == 0
                L(i,k) = count;
                count = 0;
                k = k+1;
            end
        end
    end
end
k = 1;
count = 0;
end
L = L(:,1:2);

%Check for logic high's 180 degree's apart.
for i = 1:length(matrix)
    for j = 1:180
        %look for first true value
        True = Pixel_Logic(i,j);
        if True == 1
            %Check if the 180 is also 1 (Pointer/Tail combo)
            True1 = Pixel_Logic(i,j+180);
            if True ~= True1
                Pixel_Logic(i,j) = 0;
                Pixel_Logic(i,j+180) = 0;
            end
        elseif True == 0
            Pixel_Logic(i,j+180) = 0;
        end
    end
end

% Matlab cosmetics. This part rearranges the values such that it will
% order the values from the left of the image to the right of the image.
count = 1;
for i = 1:length(matrix)
    z(i) = sum(Pixel_Logic(i,:));
    if z(i) > 5 && z(i) < 30
        matrix_new(count,:) = matrix(i,:);
        Pixel_Logic_new(count,:) = Pixel_Logic(i,:);
        radius_new(count) = radius(i);
        L_new(count,:) = L(i,:);
        count = count + 1;
    else
        count = count;
    end
end
[rm cm] = size(matrix_new);
[rpl cpl] = size(Pixel_Logic_new);
[rr rc] = size(radius_new);
[rl cl] = size(L_new);

matrix = matrix_new;
Pixel_Logic = Pixel_Logic_new;
radius = radius_new;
L = L_new;

Arrange = [matrix Pixel_Logic radius' L];
Arrange = sortrows(Arrange,1);

matrix = Arrange(:,1:cm);
Pixel_Logic = Arrange(:,(cm+1):(cm+cpl));
radius = Arrange(:,(cpl+cm+1))';
L = Arrange(:,cpl+cm+2:length(Arrange));

%**********************************************************************
%Check the results after filtering
% figure, imshow(I,[
% hold on
% red = Rs*ones(1,length(radius));
% viscircles(matrix, radius,'EdgeColor','b');
% viscircles(matrix, red,'EdgeColor','y');
%**********************************************************************

%Calculate the pointer width in pixels
for i = 1:length(matrix)
    Point(i) = min(L(i,1),L(i,2));
end

%Now Look for the index of the first logic high
k = 3;
for i = 1:length(matrix)
    for j = 1:359
        check = Pixel_Logic(i,j);
        check1 = Pixel_Logic(i,j+1);
        %look for first logic low, then high
        if check == 0 && check1 == 1
            L(i,k) = round(j + 1 + Point(i)/2);
            k = k+1;
        end
    end
    k = 3;
end

for i = 1:length(matrix)
    if L(i,1) < L(i,2)
        Theta(i,1) = L(i,3);
    elseif L(i,2) < L(i,1)
        Theta(i,1) = L(i,4);
    end
end
% Use to find out the degree of one side and other side
% Pointer 1 is at 132 [-30 psi] to 42 [30 psi]
% All other pointers are 140[0 psi] to 50 [60 psi]
% Theta(:,1) = 360;

Rp = 120;
for i = 1:length(Theta)
    Ixf(i) = Rp*cosd(Theta(i));
    Iyf(i) = Rp*sind(Theta(i));

    X1 = matrix(i,1);
    Y1 = matrix(i,2);
    x_final(i) = round(X1 + Ixf(i));
    y_final(i) = round(Y1 + Iyf(i));
end

% Assuming we have a full frontal view
% Computer Vision course would be the modification of this code part
% Predetermine which mater types are being used
% prompt = 'How many gauges are there? ';
% pt = input(prompt);
% Meter_Type = zeros(1,pt);
% for i = 1:pt
%     ask = 'Input gauge type, 1 or 2? ';
%     Meter_Type(1,i) = input(ask);
%     disp('Next gauge')
% end
Meter_Type = [1 2 2 2 2];

% Now interpolate the angle with the range
Values_R1 = [-30 -20 -10 0 10 20 30 -30];
Values_R2 = [0 10 20 30 40 50 60 0];
Angles1 = [132 164 193 223 283 343 42 132];
Angles2 = [140 178 224.5 271 316.5 360 50 140];

% Determine which meter and then determine where theta is at and interpolate
for i = 1:length(Theta)
    M = Meter_Type(i);  % Load Type of Meter to use
    if M == 1
        T = Theta(i);  % Load Theta and determine where to interpolate
        for j = 1:(length(Angles1)-1)
            T_min = Angles1(j);
            T_max = Angles1(j+1);
            if T > T_min && T < T_max
                Index(i) = j;
            end
        end
    elseif M == 2
        T = Theta(i);  % Load Theta and determine where to interpolate

for j = 1:(length(Angles2)-1)
    T_min = Angles2(j);
    T_max = Angles2(j+1);
    if T > T_min && T < T_max
        Index(i) = j;
    else
    end
end
end

for i = 1:length(Index)
    if Meter_Type(i) == 1
        Tmin = Angles1(Index(i));
        Tmax = Angles1(Index(i)+1);
        Xmin = Values_R1(Index(i));
        Xmax = Values_R1(Index(i)+1);
        T = Theta(i);
        Values(i) = round((T-Tmin)/(Tmax-Tmin) * (Xmax - Xmin) + Xmin);
    elseif Meter_Type(i) == 2
        Tmin = Angles2(Index(i));
        Tmax = Angles2(Index(i)+1);
        Xmin = Values_R2(Index(i));
        Xmax = Values_R2(Index(i)+1);
        T = Theta(i);
        Values(i) = round((T-Tmin)/(Tmax-Tmin) * (Xmax - Xmin) + Xmin);
    end
end

% Show Image with Pixel Extraction Circle and Pointer Projection Line
figure, imshow(I,[],)
hold on
red = Rs*ones(1,length(radius));
viscircles(matrix, radius,'EdgeColor','b');
viscircles(matrix, red,'EdgeColor','y');

% Label points according to their rows in matrix
for i = 1:length(matrix)
    str = ['Point ', num2str(i), ' → '];
    str = ['Value = ', num2str(Values(i)), ' → '];
    text(matrix(i,1),matrix(i,2),str,'HorizontalAlignment','right')
end

% Draw a line from the center of the analog reader in the direction of the % 'pointer' we found.
for i = 1:length(Theta)
    line([x_final(i) matrix(i,1)], [y_final(i) matrix(i,2)], 'LineWidth', 3, 'Color', 'r');
end
hold off
%**********************************************************************
toc
Appendix B:
MATLAB Code for rectifying an image

% Image Rectification Code
I1 = imread('C:\Users\Hoshiaki\Desktop\EENG421 - Semiconductors\Project\green2.jpg');
Igray1 = rgb2gray(I1);

I2 = imread('C:\Users\Hoshiaki\Desktop\EENG421 - Semiconductors\Project\green.jpg');
Igray2 = rgb2gray(I2);

[pts1, pts2] = cpselect(I1, I2, 'Wait', true);

t21 = fitgeotrans(pts2, pts1, 'projective');

I2warp = imwarp(I2, t21, 'OutputView', imref2d(size(I1), [1 size(I1,2)], [1 size(I1,1)]));
figure, imshow(I2warp,[]);