IMAGE ANALYSIS

There are many different contexts where automated image analysis is important to engineers. Two different scenarios are discussed here, but there are many more.

One scenario is in the manufacture of machined parts. If hundreds or thousands of parts are made per minute, it is difficult for a person to analyze each one. The analysis would include measuring certain dimensions with a caliper. This would take roughly 10 seconds to complete, so one person could keep up with 6 parts per minute. Then there is the whole "I Love Lucy" scenario with the chocolates on the conveyor belt, i.e. the 6 parts per minute assumes a person could do that continuously for hours on end. Strategies for analyzing 1 in hundred or 1 in a thousand or fewer actual parts to determine failure rates are typically performed. These methods are based in statistics and assume that the treatment on a handful of samples can be extrapolated to all pieces and that some failure is acceptable. If these parts are being used in an artificial heart, failure may not be an option. Analysis of each and every part may be required. A strategy to analyze each part involves image analysis. A camera located above the conveyor would take a picture of each and every part that is manufactured. The camera would send the picture to a computer for analysis. The computer would run an algorithm based in something like MATLAB to calculate the dimensions of the part. Since the computer can carry out the calculation a bit faster than the most skilled person can use a caliper, every part can be analyzed.

The second scenario involves analyzing many things in the same picture. A good example is in catalyst manufacture. Often catalysts are made up of metals deposited on an inert support. The metal is usually deposited as little islands of metal on the support. It is definitely not a single layer of metal covering the support. These samples are then analyzed, possibly with electron microscopy. A microgram is seen below for rhodium deposited on aluminum oxide. The small lighter particles are rhodium and the darker gray is alumina. Hundreds or thousands of rhodium particles could reside in this picture on the aluminum oxide. In terms of its efficiency to catalyze a reaction, the size of the particles and the distribution of sizes are important. Rather than measuring all of them or only a few for a statistical treatment, an image analysis algorithm could be used to analyze the entire picture.

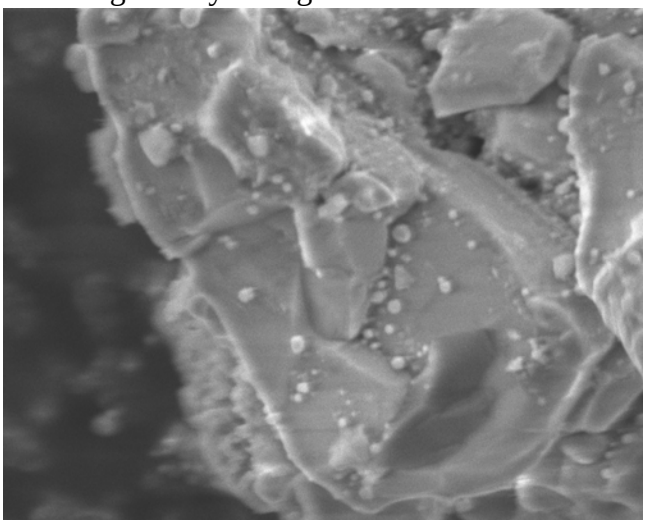

Image analysis will be discussed in the context of the second scenario. Fabricated electron micrograms will be used initially to make things easier. These fabricated micrograms will have black particles on white backgrounds. The goal of this tutorial will be to count the number of particles and measure their area.

In order to do those things, four distinct tasks will need to be completed. Images will have to be loaded into MATLAB. MATLAB will convert the image into a matrix of values for easier computation. Then a method to find each particle must be implemented. Once a particle is found, the diameter must be measured. Finally, once a particle is measured, it will need to be erased for reasons to be discussed.

Loading Images

MATLAB has a built in function called *imread* which takes the name of an image file as the input. The command would look like:

```
>> x = imread('picfile.jpg');
```

This command will take the picture in *picfile.jpg* and assign it to the variable named *x*. MATLAB uses the red-blue-green color system. That means that *x* will consist of three matrices that are *m* by *n*, where *m* and *n* are the number of pixels in the image. The first matrix will be the red hue, the second matrix will be the blue hue, and the third matrix will be the green hue. In this tutorial, only gray scale pictures will be considered. For gray-scale, the intensity of each hue is always the same. For this tutorial, since gray scale will be used, only one hue is needed. To omit the other two, we can redefine *x* as only the first matrix:

```
>> x = x(:,:,1);
```

This command will reset *x* to all of the same rows and columns of the first matrix of the original *x* variable. The colons in the parentheses indicate that all elements of the rows and columns are being translated to the new variable. Only one of the three matrices in the third dimension is being kept.

Finally, when imread translates the intensity of the hue, it translates it into an unsigned 8-bit integer (u8int), i.e. a number between 0 and 255. MATLAB does not carry out addition, subtraction, etc. on u8int numbers. If it were to add 200 to 100, the result would be 300, which makes no sense in u8int terms. To change the type of data to something that can perform calculations the following command can be used:

```
>> x = double(x);
```

This will reassign the data type to a double precision number. Other data types will work, but this method is acceptable for this course.

Finding the Particle

In the previous step, a picture was translated into a matrix that has a number from 0 to 255 in each spot of the matrix. Higher numbers indicate whiter pixels and lower numbers

indicate darker pixels. Now, particles must be found. This is done by looking at each value in the array starting at the top left, scanning column by column, and then row by row until a very low number is found.

Without giving the code to do this, one can guess that two loops will be needed to scan the matrix. The outer loop could scan down the rows and the inner loop could scan across the columns. When this was done with a vector, the length command was used to find the size of the vector. For matrices, it is a little bit different. Instead the size command is used:

```
[m,n]=size(x);
```

This command will return the number of rows of *x* as *m* and the number of columns of *x* as *n*. These two variables can be used for the two loops.

At each element of the matrix, a decision will need to be made: is this point the top of a circle? If it is, something will need to be done. If it is not, then keep moving on to the next one. This decision is often made by comparing the value to some threshold. If the value is below the threshold, the pixel is very dark so it must be the top of the particle. If not, then keep moving on. Obviously an if statement will be required, but how does one determine the threshold? Often this threshold value is case specific, i.e. there is no one answer. However, for this course and tutorial, the average value of the matrix will be used. (Code for the average finder will be given at the end of the tutorial.)

Finding the Diameter

Depending on the resolution of the picture, the first dark point should be the top dead center of the first particle. To find the diameter, one could simply scan downwards to find the bottom. This can be done with a loop that compares the value to the threshold value. This takes about 4 lines of code to complete.

Erasing the Particle

Now that the particle has been found and the diameter has been calculated, the program will move on and find the next one. However, if the particle is not erased, it will simply find the next dark pixel in the particle and assume that point is the top dead center of a new particle.

The easiest way to erase a particle is to reassign all of the values in a square around the circle to a high number. Something like 255 will work. For the sake of this tutorial, say that the coordinates of the top dead center are (*r*,*c*) and the diameter is *d*, the columns that need to be erased will run from *c-d/2* to *c+d/2*. The rows that will be erased will run from *r* to *r+d*. This will require two loops again to reassign values. Instead of starting at one, the loops will start at the lower bound for rows and the lower bound for columns.

This looks fine, but in practice it does not work so well. Two things must be dealt with. First of all, if the diameter is an odd number, then the lower bound for columns will not be an integer. This will cause major problems if the value is an index. To get around this, rounding functions can be used. The *ceil* function is what will be used here. The other problem is that a sphere with poor resolution may have multiple dark pixels in the top row. This means that the *r* and *c* found are not top dead center. If half the diameter is added to *c*, the entire sphere may not be enclosed in the square. To get around this problem, the tutorial suggests that 5 extra pixels in either direction are erased. The new bounds for the columns will be *ceil(c-d/2-5)* to *ceil(c+d/2+5)* and the new bounds for the rows will be *r* to *r+d+5*. The extra pixels are not needed on the top.

Obviously, if the particles are too close together, some particles could be erased accidentally. This will not be a problem in this course, since it has been assured that it will not happen. If the particles are extremely close together a different erasing method will be needed.

Now that the particle has been erased, the part of the algorithm that searches for the top of each particle can only find a new particle and not a fragment of the erased particle.

Code for the Average Finding Function

```
function [avg]=matavg(f)
tot=0;
[m,n]=size(f);
for ii=1:m
        for jj=1:n
                tot=tot+f(ii,jj);
        end
end
num=m*n;
avg=tot/num;
```