Image Analysis with MATLAB

David Van Valen
APh 162
01/05/2010

To turn in

- A commented m-file containing all of your written MATLAB code
- A contrast adjusted image of a graticule
- A 3-color image of a prepared slide
- A plot of the photobleaching measurement

Introduction

In this set of notes, we will explore the image analysis capabilities of MATLAB, with an emphasis on analyzing images taken from biological experiments. Before continuing, lets ask a simple question: *why image analysis?* One can list a number of objections to this approach. The initial investment in hardware and software can be costly, and one can argue that the time spent writing and refining scripts is time that might be better spent looking at images by hand. Some features might be readily identifiable by eye, but difficult to identify by machine. On the other hand, there are very tangible benefits to image analysis, such as removing any bias present when it is done by hand and the gaining greater quantify experimental outcomes. I would argue that the biggest advantage is *time*. Automating image analysis can greatly reduce the amount of time required to process data. To see this, lets look at a quick case study. Below is a movie of a DNA strand being ejected by lambda phage, a virus that infects *E. coli* cells. In these images, we (myself and my colleague David Wu) observed the ejection by stained the DNA with the organic dye SYBR gold and imaging with fluorescence microscopy.

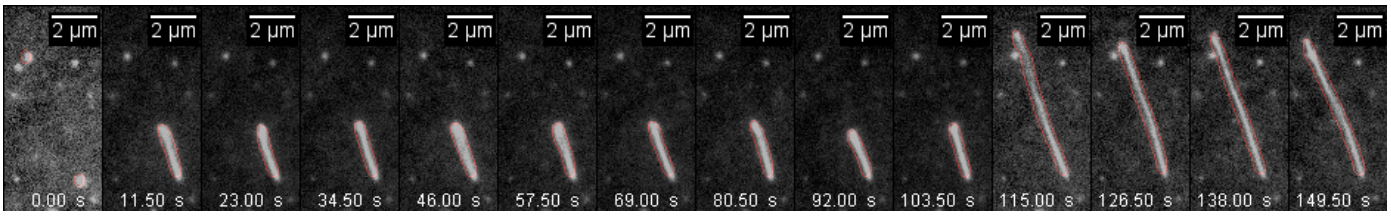


Figure 1: Montage of a DNA strand.

In our case, we were interested in finding the length of the DNA strand as a function of time. There are two ways we can extract this - either measuring the length in each frame by hand, or write a MATLAB script that will identify which pixels of the image belong to the DNA strand tell us the length. Lets estimate how long it would take to analyze this data by hand. It takes roughly 5 seconds to extract the length from a frame by hand, and there are about 500 frames in a typical movie. Lets say that we want to look at ejections in 5 different buffer conditions and for each condition, we want about 20 ejection events. Looking at this data by hand would take us roughly

$$5x500x5x20 = 250{,}000 \text{ seconds} = 70 \text{ hours}$$

That's 70 hours! If one (namely me) were to do nothing but select two points on a frame for 10 hours a day, it would take a whole week to get anything useful out of this data. Further, there is a cost to running additional experiments if we learn something interesting. The marginal cost of analyzing another condition is 14 hours, and thats on top of the time it takes to actually run the experiment. This estimate gets much worse it we want to look at a different metric, like the total intensity of the stained DNA. Alternatively, a set of MATLAB scripts was able to analyze this data set in roughly 15-30 minutes per condition, a savings of an order of magnitude or two in time.

In this instance, it was preferable to automate your image analysis. MATLAB is a powerful tool for image analysis, as it has a number of implemented algorithms that makes writing analysis code much simpler. The purpose of this set of notes is to serve as a simple guide to the basics of image analysis in MATLAB. In the next section, we will cover how to open images and cover some basic considerations for memory management. Then we will go over tools for viewing and manipulating images. Finally, we will give a brief exercise in segmentation where you will measure the amount of fluorescence in individual *E. coli* cells.

Opening image files

Lets start by learning how to open and view image files. In this tutorial, we will use some images that were acquired during the microscopy session on Monday. A copy of the images taken by your TAs are also available on Snowdome folder http://snowdome.caltech.edu/aph162/Dave/bootcamp2009.zip. First, lets direct MATLAB to the directory with your image files. In my case, the files are stored in F:\bootcamp2009\Bootcamp; this will likely be different for you. At the MATLAB prompt, we can store the location of this directory in a variable by typing

```
>> direc = 'F:\bootcamp2009\Bootcamp'
```

at the MATLAB prompt. Alternatively, you can type in the directory where you have stored your files from the microscopy session. Here the ' ' symbols denote that I am saving a string to the variable "direc." We can then move to this directory by typing

```
>> cd(direc)
```

Alternatively, we can skip storing the directory name as a variable and type

```
>> cd F:\bootcamp2009\Bootcamp
```

to view the directory contents in the MATLAB window. A third way to change the current directory is by typing F:\bootcamp2009\Bootcamp into the current directory window at the top of the screen. Once this is the active directory, we can view its contents by typing

```
>> dir
```

or

```
>> ls
```

In my case, the output is

```
>> ls
.
..
Graticule_WARD_94W_9910_1micron_10X_0
bovine_pulmonary_artery_cells_0
photobleaching_250ms_0
timelapse_EcoligrowthonLB_0
```

Notice that because the images were taken with Multi-D acquisition in micro-manager, each has its own

folder.

Lets open the image of the 1 micron graticules. The function that we will use to open the images is imread. To learn more about imread, type

```
>> help imread
```

into the MATLAB prompt. We can do this with any MATLAB function to gain information on how it can be used. We can also access help information by selecting "Product Help" under the help menu. The upshot of the help file is that we need to send a string with the file name to imread - micromanager only saves images with the tif format, so we don't need to worry about the second input. There are a couple ways we can send the image name to imread. The simplest is to simply move to the directory by typing

```
>> cd  Graticule_WARD_94W_9910_1micron_10X_0
>> im = imread('img_000000000_Brightfield_000.tif');
```

The semicolon is placed at the end of the last line to suppress output - otherwise we would see the value of every pixel displayed onscreen. The downside of this approach is that we have to move directories, and we need to know the name of the files ahead of time. This is especially inconvenient if we want to automate our analysis later. Another approach is to use the "dir" command to extract the name of the files. By typing

```
>> cd F:\bootcamp2009\Bootcamp
>> direc_contents = dir
direc_contents =
6x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

we can obtain the contents of the directory in a structure we name "direc_contents". We see that there are 6 elements in this structure, each one having 5 fields. To look at the names of the directory contents, type

```
>> direc_contents(1).name
ans =
.
>> direc_contents(2).name
ans =
..
>> direc_contents(3).name
ans =
Graticule_WARD_94W_9910_1micron_10X_0
```

An important observation is that the first two elements are simply "." and ".."; this is often the case for the "dir" command. Now we can make use of the "dir" command to determine the name of the image graticule image by typing

```
>> direc_name = direc_contents(3).name;
>> direc_contents_2  =  dir(strcat(direc_name,'/*.tif'));
```

```
>> file_name = direc_contents_2.name;
```

The first command saved the name of the directory in the variable "direc_name". The second command told MATLAB to look in that directory for all files ending in ".tif". Note that in this line, we made use of strcat - this function simply joins two strings together. In this case, its output is the string `'Graticule_WARD_94W_9910_1micron_10X_0/*.tif'`, exactly what we need to input into "dir" so that it only searches for tif files. The third function saved the name of that file in the variable "file_name". With these variables, we can assemble the complete file name with the command

```
>> graticule_name = strcat(direc,'\',direc_name,'\',file_name)
```

We can also use the function fullfile

```
>> graticule_name = fullfile(direc,direc_name,file_name)
```

Now lets load the image of the graticule by typing

```
>> im = imread(graticule_name);
```

In MATLAB, images are stored in 2 dimensional matrices - to get some information about this image in MATLAB, type

```
>> whos im
Name          Size                Bytes  Class     Attributes

  im        1040x1392             2895360          uint16
```

We see that the image is stored is in a 1040x1392 matrix, takes up about 3 MB in memory, and is of type "uint16". What does this mean?

Before continuing, lets take a moment to look at some data classes we will run into when analyzing images with MATLAB. There are 4 data types that are common

- uint16: This stands for unsigned 16 bit integers. They range from [0 65535] and are generally used in images acquired by a CCD camera. Each pixel that contains a uint16 number takes up 16 bits, or 2 bytes, in memory. Converting to uint16 is accomplished by using the "im2uint16" function.
- uint8: This stands for unsigned 8 bit integers. They range from [0 255] and are generally used in images acquired by a standard digital camera. Each pixel that contains a uint8 number takes up 8 bits, or 1 byte. Converting to uint8 is accomplished by using the "im2uint8" function.
- double: This is MATLAB's floating point number. Its range is [10^-308 10^308], and usually arises when you convert a raw image that is uint16 or uint8 into a matrix with floating point numbers. Any operation that uses a decimal point will only accept double numbers as inputs. Each pixel that contains a double number will take up 8 bytes. Converting to double is accomplished by using the "double" function. One can also use "mat2gray", which converts images to type double and also normalizes their value range to [0 1].
- logical: These are binary matrices - each element has a value of only 1 or 0. These usually appear during segmentation when we want to isolate a particular piece of an image. Each pixel that contains a logical number will take up 1 bit, or .125 bytes. Converting to logical is accomplished by performing any logical operation (such as im < 20) or by using the "logical" function.

These numbers are important to keep in mind, because memory management is a large part of constructing an efficient image analysis routine. Consider the example of the time lapse of *E. coli* we took during the

microscopy session. Each image, if stored as a uint16 array, takes up 3 MB. If we wanted to do floating point operations, we would need to store each image as a double, so each image would now require 12 MB. If we were taking images every 5 minutes for 4 hours, this would give us 48 images - loading them all at once would take 576 MB - a significant portion of our available memory. More complicated exercises might involve a longer movie, multiple channels, or intermediate images that we create during processing. As you can see, it can become quite easy to exhaust the available memory. A good rule that helps us avoid memory limitations is to *keep as few images in memory as possible*. Two ways to accomplish this are to design programs that only analyze one image at a time and to save images to disk rather than keep them in memory.

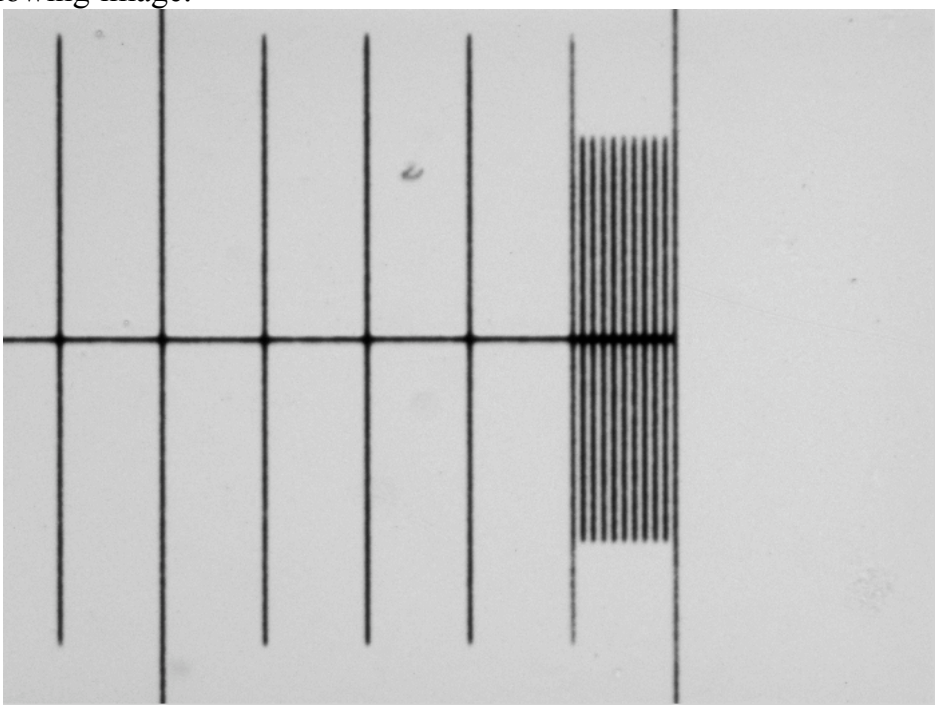
Viewing images
Lets view the image of the graticule with imshow.

```
>> imshow(im)
```

What we see is not what we want - the image is completely black! We get this output because "imshow" has 0 set to black and 1 set to white by default. We can change the display range to fit the range of our image by typing

```
>> imshow(im, [])
```

This gives us the following image.



By inputting "[ ]" into "imshow", we've told the function that we want the minimum pixel value to represent black and the maximum pixel value to represent white. We can set the black and white levels manually by typing

```
>>imshow(im, [low high])
```

where low and high are the pixel values representing black and white. An alternative method to change the black and white levels is to scale the picture. By typing

```
>> im_gray = mat2gray(im);
```

we can create a new image that has been normalized to the pixel range [0 1]. Viewing "im_gray" with "imshow" won't have the same problem as viewing "im."

Sometimes, to observe features more clearly, we will need to adjust the contrast of an image. Contrast adjustment refers to a mapping from one set of pixel values to another. Examples of this mapping are shown below.

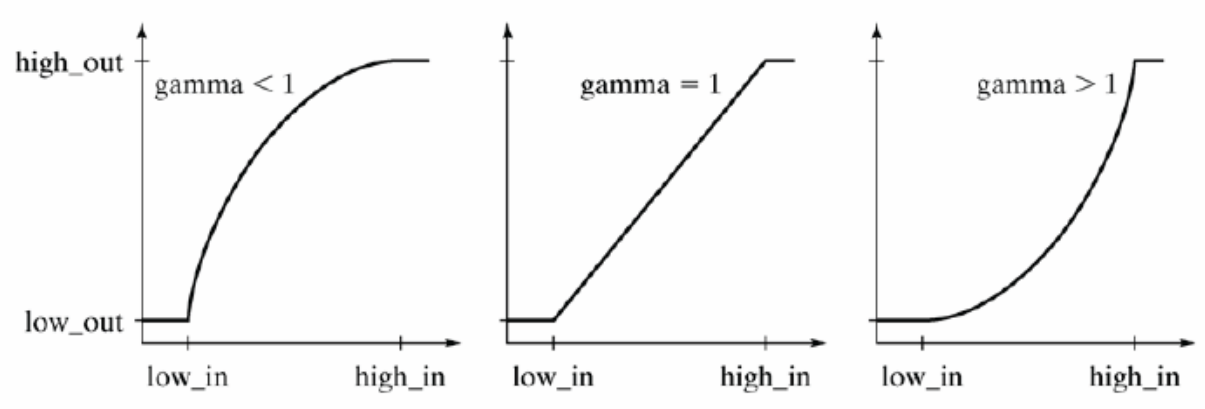


Figure 2: Examples of a transfer functions to adjust contrast

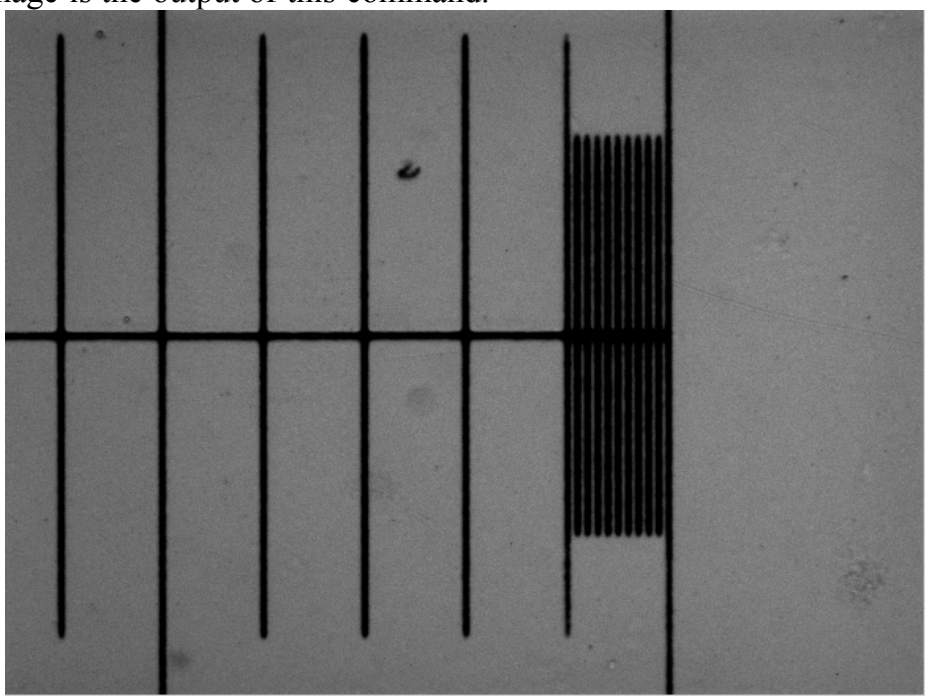
A typical transfer function is the gamma transfer function, which is given by

$$I_{out} = (hi_{out} - low_{out}) * \left(\frac{I_{in} - low_{in}}{high_{in} - low_{in}} \right)^{\gamma} + low_{out}$$

For low powers of gamma, contrast between dim objects in increased. For high powers of gamma, the contrast between bright objects is increased. We can perform contrast adjustment in MATLAB on our image of the graticule by typing

```
>> im_adj = imadjust(im_gray,[0 1],[0 1],0.5);
```

Here, we are setting the input and output lows and highs to be 0 and 1 respectively - gamma has been set to 0.5. The following image is the output of this command.



Try toying around with different values of gamma to see the effect on the image. MATLAB can also perform contrast adjustment automatically. By typing

```
>> im_adj = imadjust(im_gray);
```

MATLAB selects a value of gamma so that 1% of the pixels have the minimum value and 1% of the pixels have the maximum value.

With contrast adjustment under our belt, we can make use of it to pretty up some of the images we took earlier. In this next part, we will look at fluorescent images of bovine pulmonary artery cells. The difference with this image set is that now we have 4 different channels - brightfield, DAPI, FITC, and TRITC. As an exercise, lets load these images, perform contrast adjustment in each channel, and then assemble it into a color image.

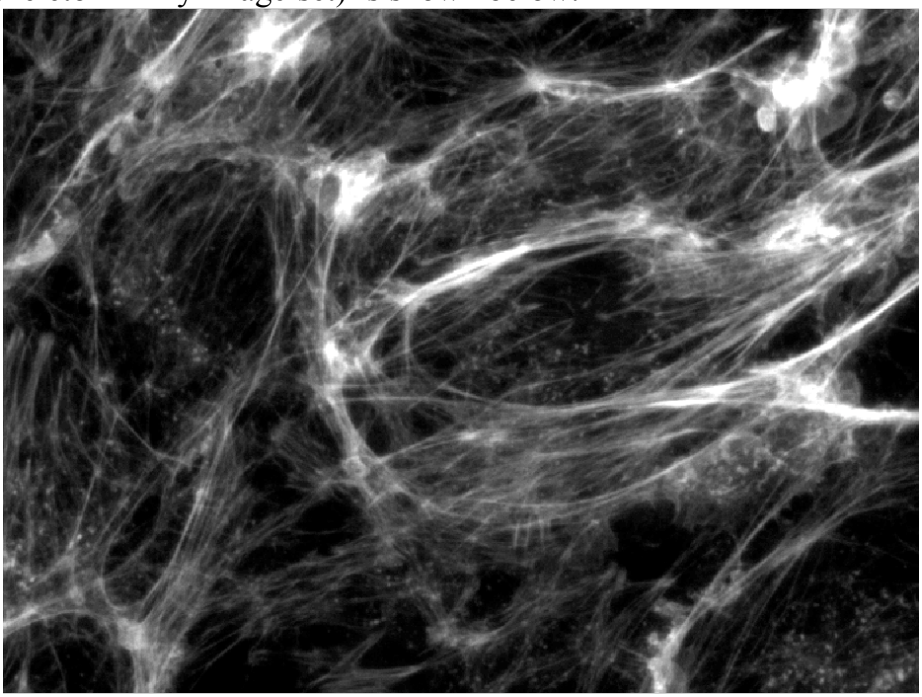
To load the images, we first go to the directory, identify the image files, and load them with imread. The following MATLAB commands accomplish this

```
>> direc = 'F:\bootcamp2009\Bootcamp
\bovine_pulmonary_artery_cells_0';
>> bright_content = dir(strcat(direc,'/*Bright*'));
>> bright_name = bright_content.name;

>> DAPI_content = dir(strcat(direc,'/*DAPI*'));
>> DAPI_name = DAPI_content.name;

>> FITC_content = dir(strcat(direc,'/*FITC*'));
>> FITC_name = FITC_content.name;

>> TRITC_content = dir(strcat(direc,'/*TRITC*'));
>> TRITC_name = TRITC_content.name;

>> bright_path = fullfile(direc,bright_name);
>> DAPI_path = fullfile(direc,DAPI_name);
>> FITC_path = fullfile(direc,FITC_name);
>> TRITC_path = fullfile(direc,TRITC_name);

>> im_bright = imread(bright_path);
>> im_DAPI = imread(DAPI_path);
>> im_FITC = imread(FITC_path);
>> im_TRITC = imread(TRITC_path);
```

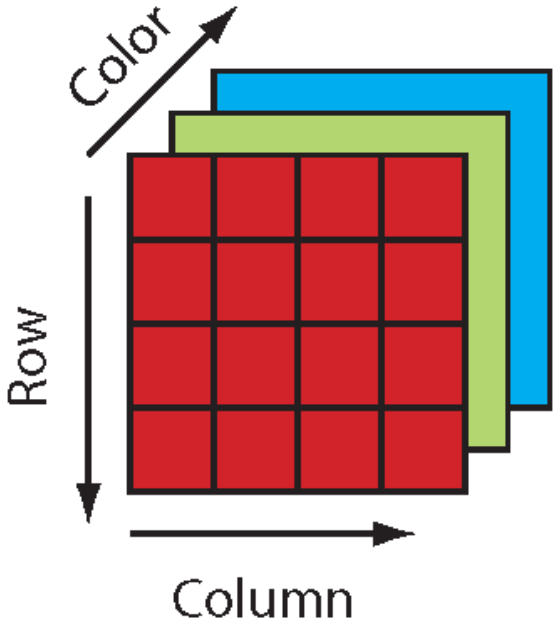
With these commands, all four images have been loaded into memory. Take a moment and look at what each image looks like with "imshow". Next, we can convert these images to grayscale using mat2gray, and automatically adjust the contrast with imadjust.

```
>> im_bright_adj = imadjust(mat2gray(im_bright));
>> im_DAPI_adj = imadjust(mat2gray(im_DAPI));
>> im_FITC_adj = imadjust(mat2gray(im_FITC));
>> im_TRITC_adj = imadjust(mat2gray(im_TRITC));
```

Take and moment and look at some of these pictures using imshow - for instance the FITC channel (which is the stain for the cytoskeleton in my image set) is shown below.



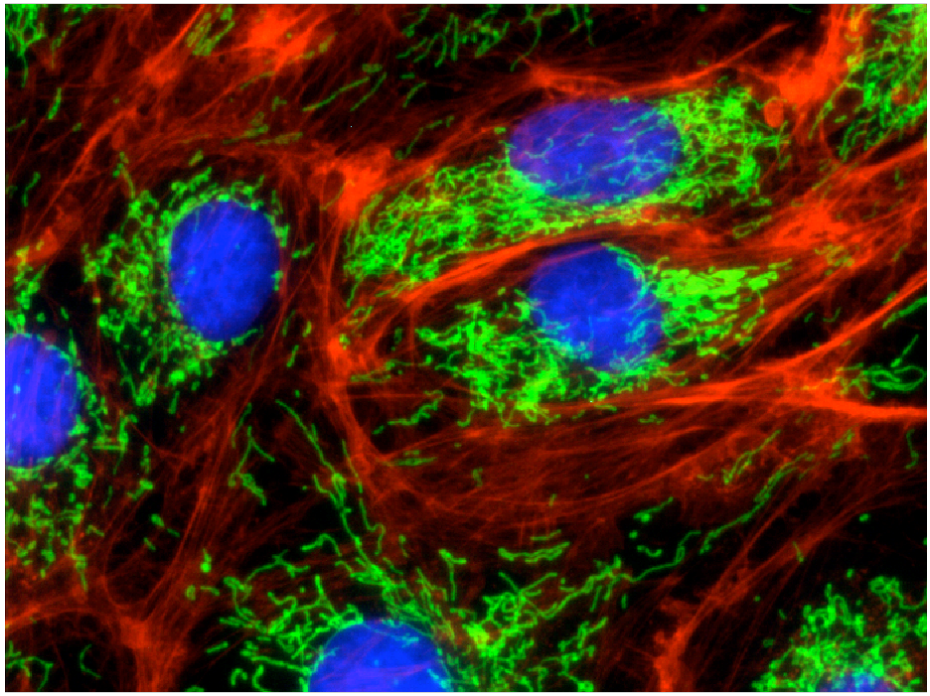
It would be useful if we could look at all three channels at once - we can do this with MATLAB by creating a color image and setting each channel to represent a different color. The way color images are created in MATLAB is schematized below.



Every intensity image in MATLAB is stored as a matrix, with the first dimension being the rows of the matrix and the second dimension being the columns of the matrix. Color is included by using a 3rd dimension. This dimension is 3 layers thick, with each layer used to represent a different color. MATLAB uses the Red/Green/Blue convention for color; the first element stores the red channel, the second element stores the green channel, and the third element stores the blue channel. To generate a color image, we need to create this 3 dimensional matrix - for the images I've taken this matrix will have dimensions of 1040x1392x3. We create this matrix using the "cat" command, which concatenates matrices. By typing

```
>> three_channels = cat(3,im_FITC_adj,im_TRITC_adj,im_DAPI_adj);
```

into the MATLAB prompt, we can do exactly this. Looking at "three_channels" with "imshow" gives the following image.



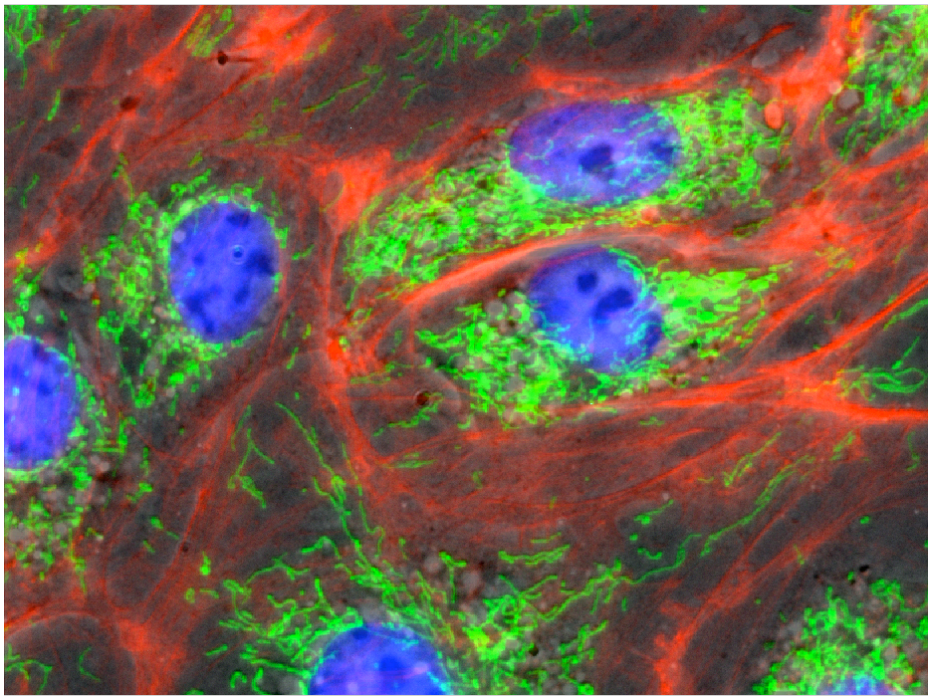
Manipulating images: Arithmetic and digital filters

At this point, we've familiarized ourselves with loading and viewing images with MATLAB. Now, lets try our hand at manipulating images. Because the images are stored as matrices, we can apply a number of mathematical operations - some of these include addition, subtraction, multiplication, division, and various kinds of digital filters. In this next section, we will look at a couple applications of the arithmetic operations.

Recall for the fluorescent image, we actually had four channels - the DAPI/FITC/TRITC that we viewed and bright field. Lets say we wanted to overlay the bright field image in white over the three color image we generated. We can accomplish this by adding the bright field image to all the other channels. When dealing with color images, adding a white color to a pixel means adding the same pixel value to each color channel. The MATLAB command

```
>> three_channels_new = cat(3,im_FITC_adj+im_bright_adj/3,im_TRITC_adj
...
+im_bright_adj/3,im_DAPI_adj+im_bright_adj/3);
```

gives rise to the following image.

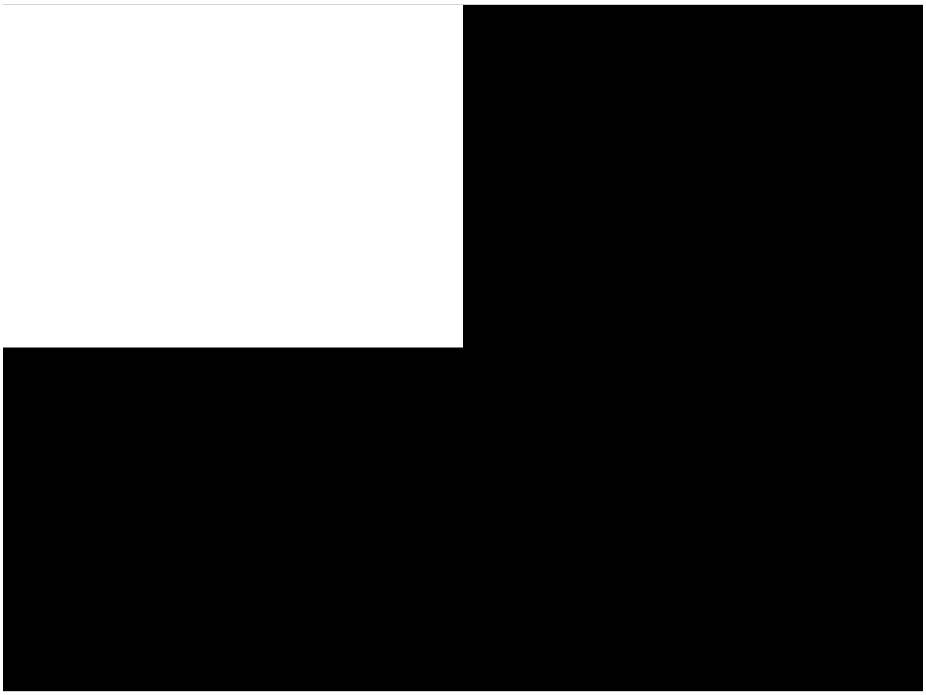


Notice how the bright field image was divided by 3 before it was added to each channel. This was done to keep the bright field channel from flooding out all the other channels.

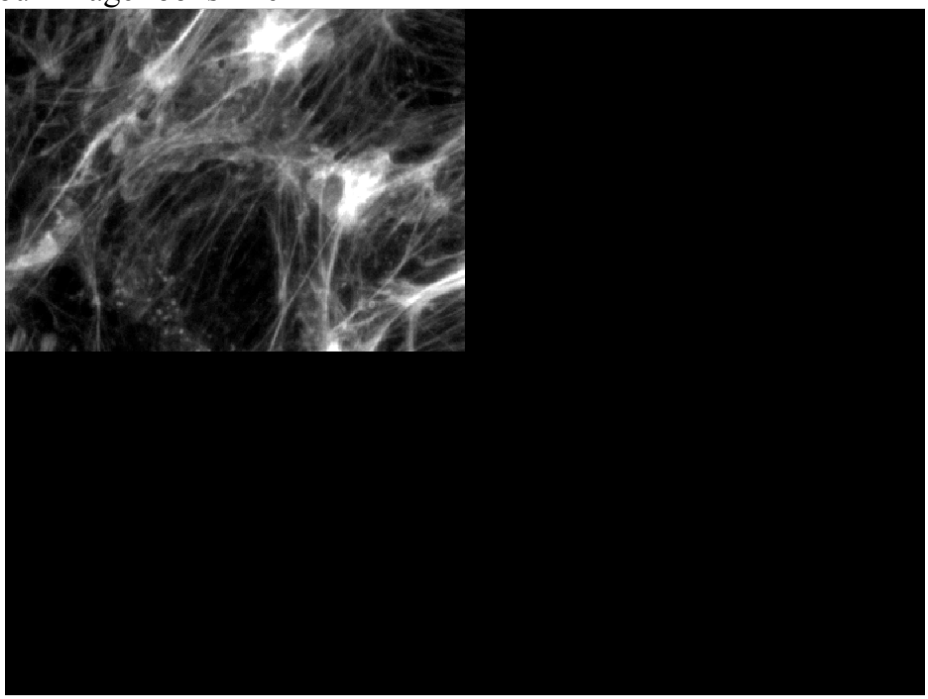
Another useful arithmetic operation is multiplication. Multiplication is frequently used to isolate portions of an image for analysis. This is typically done by creating a matrix called a mask, a matrix that consists of 1's in the pixels of interest and 0's everywhere else. Lets say we wanted to isolate the upper left corner of our our FITC channel. We can do that by creating a matrix of zeros the same size as the image and setting the pixels in the upper left corner to 1. This is accomplished by typing

```
>> mask = zeros(size(im_FITC_adj));
>> mask(1:1040/2,1:1392/2) = 1;
>> im_FITC_ul = im_FITC_adj.*mask;
```

The mask looks like



and its product with our image looks like



Masks are often employed to isolate particular features of images. This kind of problem is typically called a segmentation problem, which is the partitioning of an image into regions to identify objects and features. The identification of DNA strands can be considered to be a segmentation problem. In the following section, we will run into another segmentation problem when we will attempt to identify *E. coli* in a field of view.

Lets look at a simple segmentation problem to get our teeth wet. Lets say that we wanted to isolate the pixels in the bright field channel that belonged to the nuclei of each cell. How would we do this? One method would be to take advantage of the different fluorescent images we have at our disposal. One of the stains used in my images was DAPI, a DNA staining die that stains the nuclei of cells. This is almost exactly what we need - the die will identify which pixels belongs to the nuclei! All we need is a method of converting our fluorescent image into a binary image that has value 1 inside the nuclei and value 0 everywhere else. A simple method of doing this is thresholding. Thresholding converts an intensity image to a binary image by setting all pixels above a certain value to 1 and all images below a certain value to 0. For instance, lets set our threshold to 0.2. We can convert our image to binary by using the command "im2bw". Type

```
>> im_DAPI_bw = im2bw(im_DAPI_adj,0.2);
```

into the MATLAB prompt and use "imshow" to see what the output is. An alternative method to threshold is to use logical operations. A logical operation will assay each pixel and return 1 in that pixel if the logical statement is true and 0 if it is false. For instance, the MATLAB command

```
>> im_DAPI_bw = im_DAPI_adj>0.2;
```

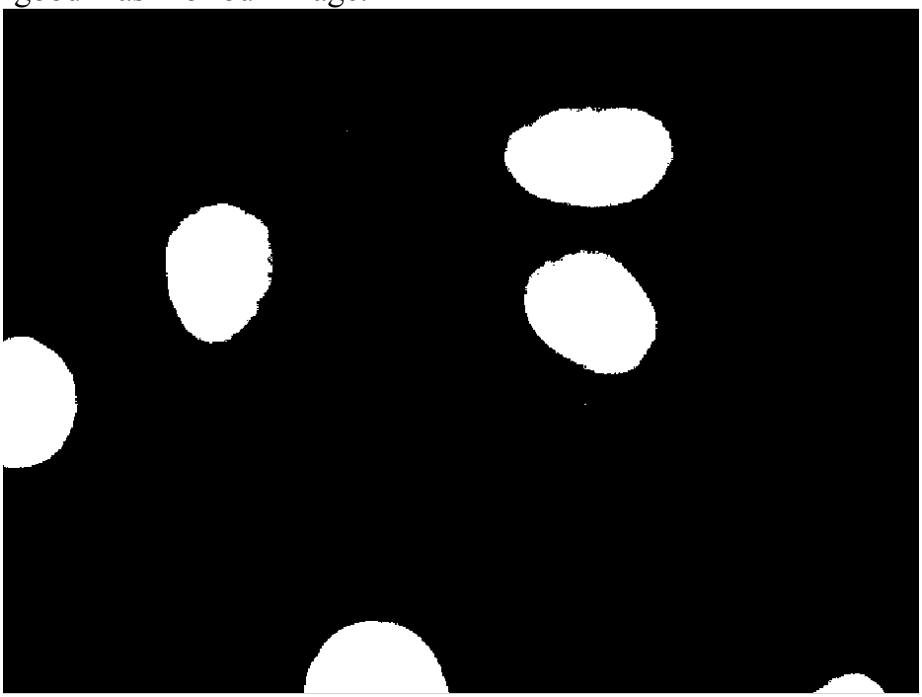
will return the same matrix as the "im2bw" operation. Both commands will output images of type logical.

The key to getting thresholding to work is finding the right threshold value to use. We can pick one by hand until the black and white image suits our liking, or we can have MATLAB pick one automatically. The function "graythresh" will automatically pick a threshold for our image. This function works by separating the pixels into two classes - those above the chosen threshold level and those below. It then selects a

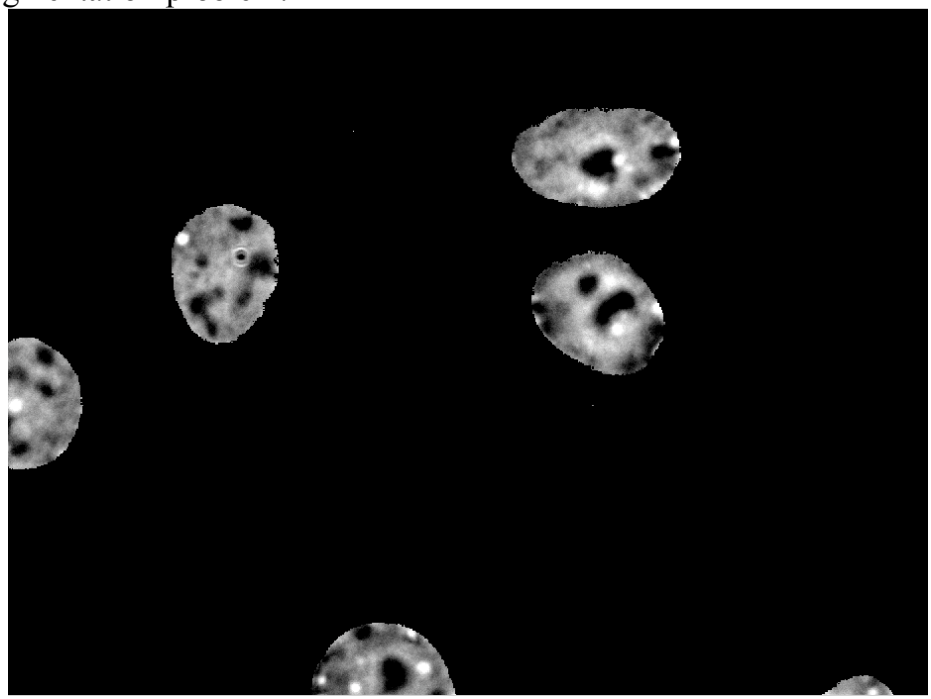
threshold level so that the variance of the pixels values within each class are minimized. We can use this function and then threshold our image by typing

```
>> th = graythresh(im_DAPI_adj);
>> im_DAPI_bw = im2bw(im_DAPI_adj,th);
```

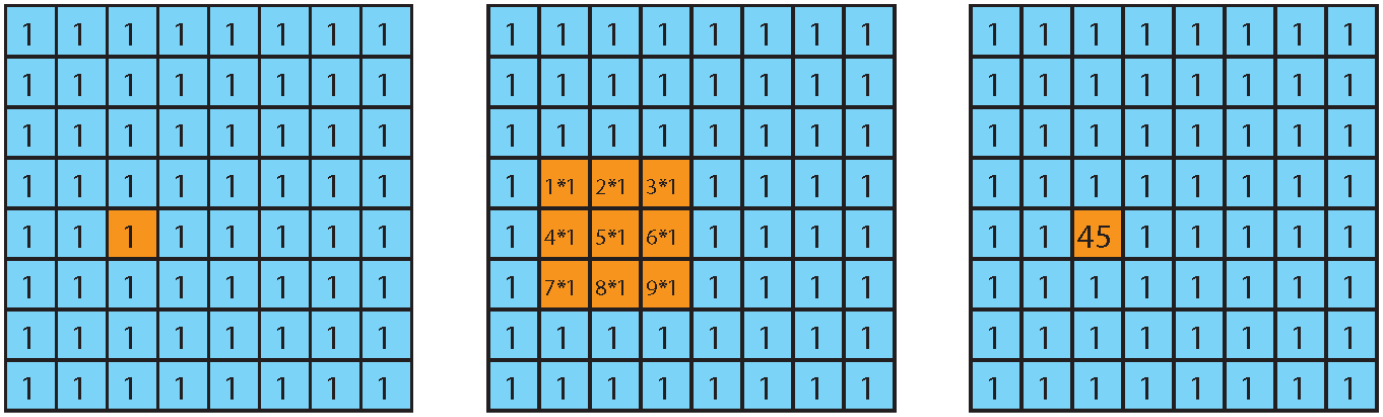
This gives us a pretty good mask for our image.



There are some very small regions that were above the threshold value that are much too small to be nuclei. How would you get rid of them? As the last step, we can multiply our mask with the bright field image to solve or miniature segmentation problem.



The final manipulations we are going to learn about in this tutorial are digital filters. A schematic of how a filter works is shown below.



Filter:

1	2	3
4	5	6
7	8	9

Essentially, a digital filter works by replacing each pixel with a linear combination of pixels in a small neighborhood of that pixel. This neighborhood is typically called a window, and the filter defines the weights for each element of this neighborhood. This process is schematized above. In the example image, every pixel has value 1. When the filter is applied to the highlighted pixel, each weight is multiplied with its corresponding pixel value. The products are summed to produce the output, in this case, 45. This example just shows the action of the filter on one pixel; in practice, the filter is applied to every pixel in the image to generate the output.

One problem with applying the filter to every pixel are the edges. Say for instance we wanted to apply the filter to the bottom left hand pixel in the above example. The weights 1,4,7,8, and 9 would have no pixel to multiply. This is usually solved by specifying the boundary conditions - i.e specifying values for the imaginary pixels outside of the image that the filter has to multiply. A very common boundary condition is zero-padding, where the imaginary pixels are set to 0. Other boundary conditions include mirror boundary conditions (the imaginary pixels are a reflection of the original image) or reflexive (the imaginary pixels take on the value of the closest pixel in the image).

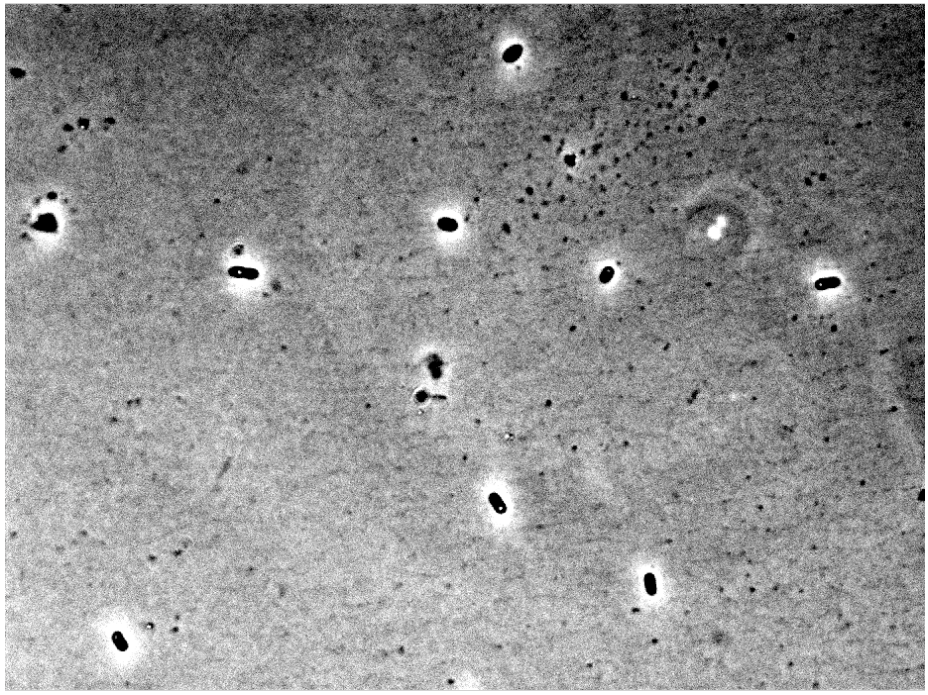
Filters have a variety of uses in image analysis. We can use Gaussian and median filters to remove noise that is present in our images. Gradient filters are useful for edge finding techniques. As an example, lets use filters to clean up the bright field picture of *E. coli* cells we took during the photobleaching measurement. First, we load one of the bright field images in MATLAB.

```
>> direc = 'F:\bootcamp2009\Bootcamp\photobleaching_250ms_0';
>> file_names = dir(strcat(direc,'\*Bright*.tif'));
>> file_path = fullfile(direc,file_names(1).name);
>> ecoli_bf = imread(file_path);
```

Next, we convert the uint16 image into a normalized double image using "mat2gray" and perform a contrast adjustment using "imadjust".

```
>> ecoli_adj = imadjust(mat2gray(ecoli_bf));
```

For my data set, this gives the following image.



We can use filters to remove some of the noise in the image. The first filter we can use to clean up the image is a mean filter. The filter for a 3 X 3 mean filter is

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Note that the sum of all the entries for the mean filter is 1. This is generally true for all smoothing filters. We can apply this filter using the function "imfilter". First, we create the a matrix for the filter, then we apply it to our image. This is accomplished using the following MATLAB commands.

```
>> mean_filt = ones(3)/9;
>> ecoli_mean = imfilter(ecoli_adj,mean_filt);
```

You can view the result using "imshow". Another smoothing filter is the Gaussian filter. The Gaussian filter removes noise by convolving the image with a gaussian of a given shape and standard deviation. We can create a Gaussian filter using the "fspecial" MATLAB command. "fspecial" is a library of filters that you can access for image and signal processing. We can use this function call up and apply the Gaussian filter.

```
>> gauss_filt = fspecial('gaussian');
>> ecoli_gauss = imfilter(ecoli_adj,gauss_filt);
```

Note that by not supplying additional inputs to "fspecial", we are using the default setting of a 3 X 3 filter window and standard deviation for the gaussian of 0.5. The last smoothing filter that is at our command is the median filter. This filter is functions differently than most filters, because the output for an individual pixel is no longer a linear combination of the pixels in some surrounding neighborhood. Rather, this filter replaces each pixel with the median pixel value in a specified neighborhood. We can apply the median filter to our image using the "medfilt2" command.

```
>> imshow(ecoli_med)
```

The median filter is especially good at dealing with spiky data, which can arise if you have dead pixels on your CCD camera.

Segmentation exercise: Measuring fluorescence in bacteria
Now we are ready to use some of the tools we learned about to make quantitative measurements of YFP photobleaching in *E. coli*. The outline of what we want to do is straightforward. We need to

- Open a brightfield image
- Adjust the contrast and remove the noise
- Identify the *E. coli* cells and create a mask
- Open each FITC image and use the mask to measure the fluorescence in each cell.
- Plot the fluorescence as a function of time for each cell

As we go through these 5 steps, one thing to keep in mind is that writing scripts to do image analysis involves a lot of trial and error. We will try a number of different ideas, and hopefully one of them will produce the result that we want. Another thing to keep in mind is that a method developed for one set of images won't necessarily work for another. Getting this segmentation algorithm to work on the data you collected yourself might require some tweaking.

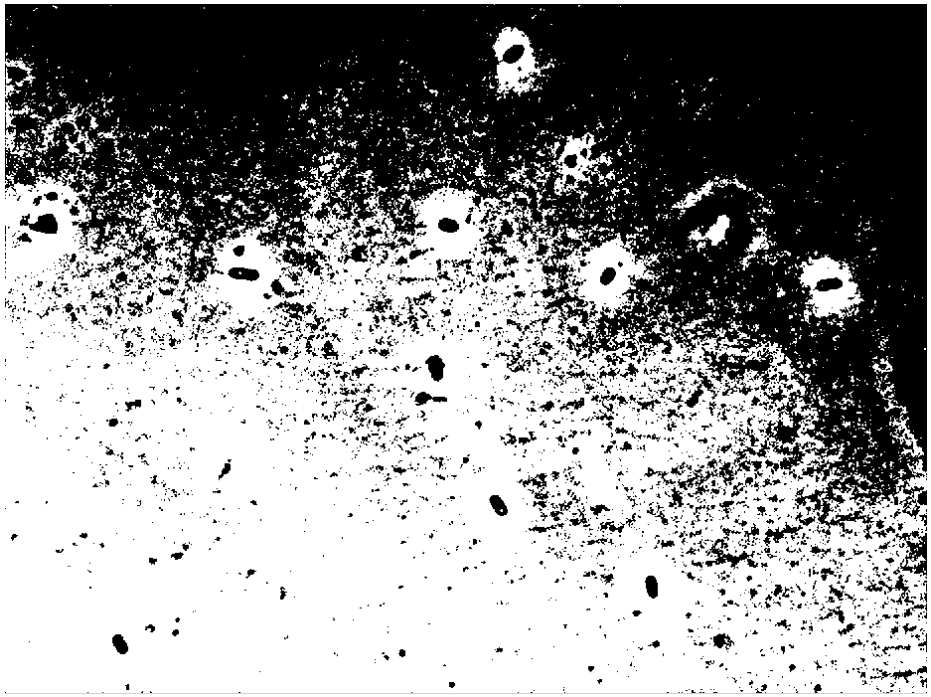
We have covered how to do the first two steps in the earlier sections. This is accomplished with the following MATLAB commands.

```
>> direc = 'F:\bootcamp2009\Bootcamp\photobleaching_250ms_0';
>> bf_names = dir(strcat(direc,'\*Bright*.tif'));
>> bf_path = fullfile(direc,file_names(1).name);
>> ecoli_bf = imread(file_path);
>> ecoli_adj = imadjust(mat2gray(ecoli_bf));
>> ecoli_med = medfilt2(ecoli_adj);
```

Take a look at the image using imshow. At this point, we need to find a way to discern a way to pick out the *E. coli* cells. We learned about thresholding earlier, so lets give that a try. The MATLAB commands

```
>> th = graythresh(ecoli_med);
>> ecoli_bw = im2bw(ecoli_med,th);
```

produce the following image.



It looks like using MATLAB to automatically pick our threshold level is not a good method to generate a mask. It is easy to see the cells in the black and white image, but a good portion of the background has also been labeled as a cell (in this instance, the black pixels, ie the ones with value 0, are the ones that are the cells ... this is the opposite of what we want in a mask, but we can always invert the mask later). We have two choices - either attempt to refine the mask or find a new way to make generate one. Before we choose, lets get a better sense of what the pixel values are for our image. We can use a very useful function called "imtool" to examine our image.

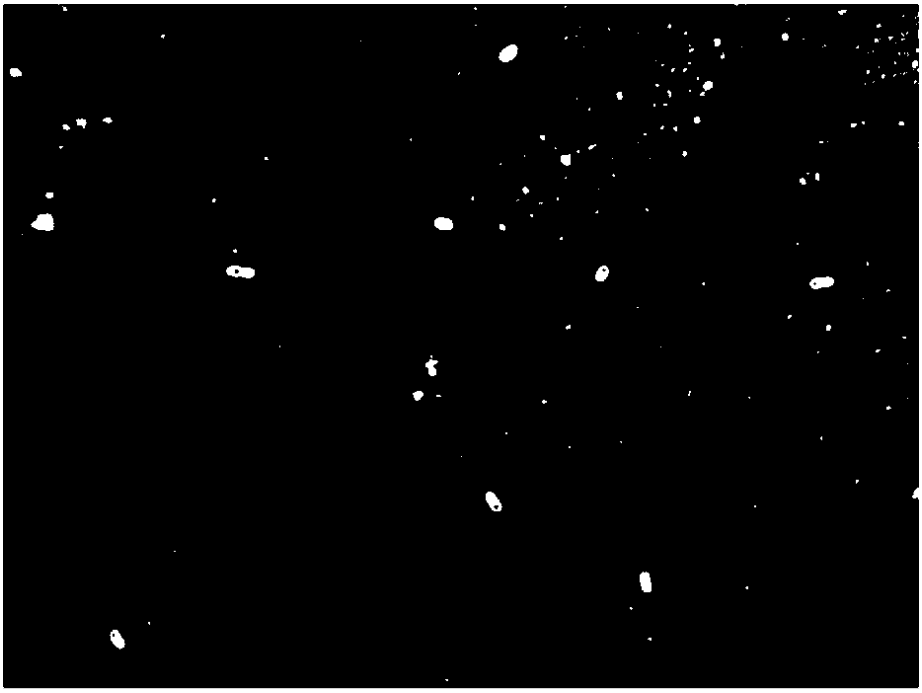
```
>> imtool(ecoli_med)
```

This opens a window showing the image and also gives us additional tools at our disposal. For instance, we can look at a histogram of intensities using the "Adjust contrast" button at the top or measure distances using the "Measure distance" button. All of these tools are for another time. What is of immediate interest is the "Pixel Info" that is located at the bottom left of the screen. With this tool, we can find out the value of a pixel by moving our mouse pointer to that location. A cursory glance gives us a clue why MATLAB didn't succeed at finding the right threshold. For this image, the background is non-uniform. Further, the cells in the image take up a very small part of the screen. These two factors probably threw off MATLAB's algorithm.

By pointing our cursor over the *E. coli* cells we can make another observation. In this image, it appears that the pixels in the cells have a value of 0. This suggests 0 as our threshold - because our image doesn't have negative pixels, we can create our mask by looking for pixels with value 0. This is accomplished by the logical operation

```
>> ecoli_seg_1 = (ecoli_med == 0);
```

This operation produces the following picture.



This is much better, but still not good enough. There are a number of pixels that were assigned as cells because of noise. Also, in the upper right quadrant of the image, there were a number of particles that are too misshapen or small to be *E. coli*. The identity of these particles is unclear - it's possible that it is just dust that was on the agar pad or coverslip. Regardless, we want them removed from our mask. How can we go about this. Image analysis allows plenty of room for creativity. One way to remove them would be to rely on the observation that the actual cells are usually surrounded by a halo, that is a region of pixels with high intensity value. If we could segment out the location of the halos, we could use them as another layer of segmentation. Anything that would go in our final mask would have to be both identified by the initial thresholding and be near a halo. Implementing this would be fun, but is beyond the scope of this tutorial. An alternative method is to threshold based on area. All of the unwanted regions have one thing in common - they are too small. All we have to do is measure the area of each white region in our mask, identify which ones are too small, and then set their pixel values to 0. Because it is easier, we will attempt the latter method.

To measure the areas of the white regions, we will make use of the functions "bwlabel" and "regionprops". "bwlabel" does what its name suggests - it labels the white regions in black and white images. In the output image, the pixel values for each white region have been multiplied by a label number. For example, the pixels in the 14'th region would all have value 14. The output of "bwlabel" is usually called a label matrix.

The label matrix that we generate can then be fed into the function "regionprops". What "regionprops" does is extract pieces of information about each of the labeled regions and return it in a structure. If I wanted to find the area of each region, the perimeter, or perhaps the ellipicity, this would be the function to use. Take a look at the help file to see the different kinds of queries we can make about the labeled regions. In our case, we want two things, 'FilledArea' and 'PixelIdxList'. The former will tell us how big each region is, while the latter will tell us where its located. We can generate the label matrix and extract these properties with the following commands.

```
>> L = bwlabel(ecoli_seg_1);
>> stats = regionprops(L,'FilledArea','PixelIdxList');
```

The tricky part about "regionprops" is that it returns a structural array. Because of this, it is not always easy to get the information we want in the right format. The first thing that we need is an array of all the areas of labeled region. In other words, we want an array where the first element is the area of region 1, the second

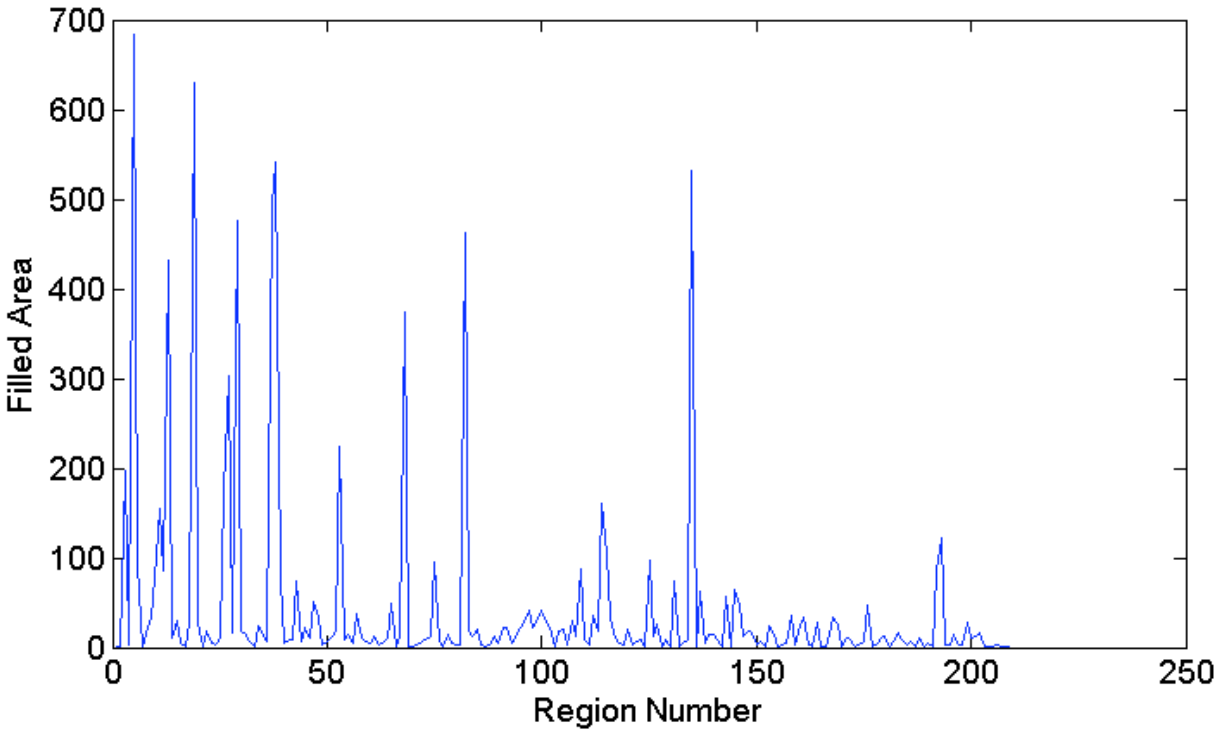
element is the area of region 2, and so on. We can get an array by typing the command

```
>> areas = [stats.FilledArea];
```

The brackets are used to store the output into an array. We can look at the areas using the plot command

```
>> plot(areas)
```

which gives the following plot.



The spikes are likely the regions that correspond to actual cells. We can always create a method to automatically pick a threshold, but for now it looks like an area of 200 pixels is a reasonable threshold to distinguish cells from noise/dust. To perform the areal thresholding, we need to identify which regions are small and need to be discarded. We can identify them with the "find" command

```
>>  too_small = find(areas<200);
```

Next, we need to use our array of indices to extract which pixels belong to these regions and set them to zero. The problem is that the field 'PixelIdxList', which contains a list of all the pixels for each region, is an array. Our earlier trick of saving the elements of a field in an array won't work because array elements must be numbers, not other arrays. We can circumvent this by saving the field elements into a cell array. A cell array is just like a regular array, but without the restriction on the element type. Cell elements can be strings, numbers, and even other arrays. Once we capture everything into a cell, we can use the command "cell2mat" to collapse all of the cell elements into one giant array. This is accomplished with the commands

```
>> too_small_pix = {stats(too_small).PixelIdxList};
>> too_small_pix_mat = cell2mat(too_small_pix');
```

With the pixel locations stored in an array, we can create a mask where these pixels are set to 0.

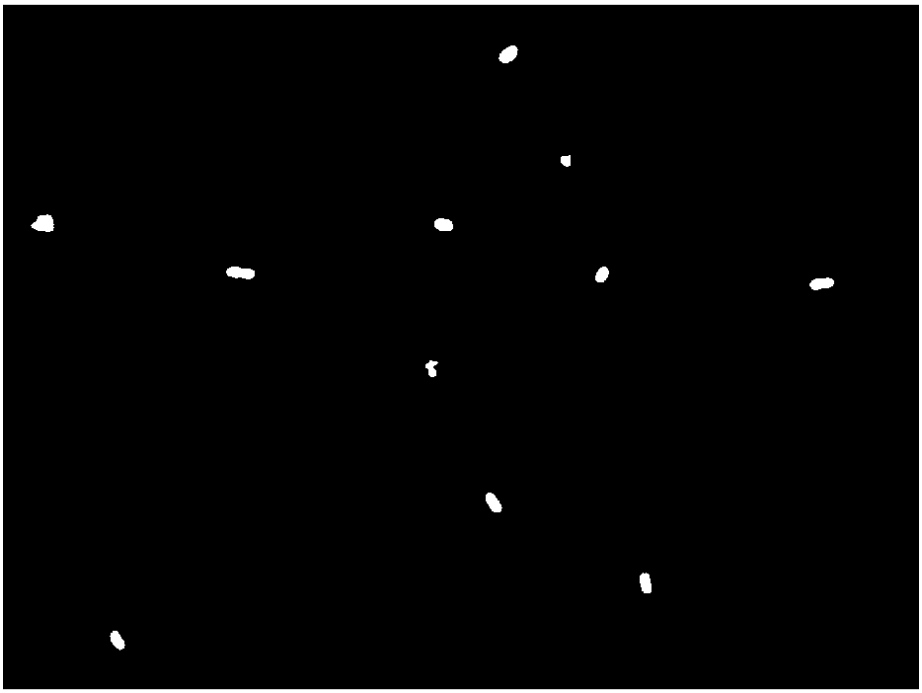
```
>> mask = ecoli_seg_1;
```

```
>> mask(too_small_pix_mat)=0;
```

As a final housekeeping chore, we can remove all the holes in the mask using the "imfill" command.

```
>> mask = imfill(mask,'holes');
```

This produces the final mask.



Comparison with the original image shows we've done a pretty good job identifying the *E. Coli* cells. The final step is to extract the fluorescence for each cell. This is pretty simple - we can get the pixel list for each region in our final mask, open each fluorescence image, extract the fluorescence in those pixels, and take their mean. Next, we find the background fluorescence by taking the mean of the fluorescence pixels outside of the cells. We can then save the difference between the signal and the background in a matrix. This is accomplished with the following script.

```
% Get file names
FITC_names = dir(strcat(direc,'\*FITC*.tif'));

%Identify cell pixels
mask_label = bwlabel(mask);
mask_stats = regionprops(mask_label,'PixelIdxList');

%Identify background pixels
background_mask = 1-mask;
bg_mask_label = bwlabel(background_mask);
bg_stats = regionprops(bg_mask_label,'PixelIdxList');

%Create a matrix to store the mean fluorescence
fluor_means = zeros(numel(FITC_names),numel(mask_stats));
for i = 1:numel(FITC_names)
    %Load FITC channel
    FITC_path = fullfile(direc,FITC_names(i).name);
```

```
    FITC = imread(FITC_path);

    %Get background fluorescence
    pixel_list = bg_stats.PixelIdxList;
    background = mean(FITC(pixel_list));

    %Extract mean fluorescence
    for j = 1:numel(mask_stats)
        pixel_list = mask_stats(j).PixelIdxList;
        mean_fl = mean(FITC(pixel_list));
        fluor_means(i,j) = mean_fl-background;
    end
end
```

In the data I collected, the exposure time was 250 ms. I've selected 5 traces that actually represent photobleaching and placed them on the plot below.