Matlab Tutorial: Basic Matrix Manipulation; Merging Fluorescence Channels

Hernan G. Garcia, Daniel L. Jones

April 4, 2013

1 Matrix Manipulations: A Brief Introduction

1.1 Creating a matrix; Accessing Matrix Elements

Let's create a 5x4 (i.e., 5 rows, 4 columns) matrix consisting of 1's everywhere:

```
a = ones(5,4,'uint8')

a =

    1    1    1    1
    1    1    1    1
    1    1    1    1
    1    1    1    1
    1    1    1    1
```

To refer to the i, jth element of the matrix, use the expression

```
 a(i,j)
```

For instance, let's say we want to change the number in the 2nd row, 3rd column to 4. We would type the following:

```
a(2,3) = 4

a =

    1    1    1    1
    1    1    4    1
```

```
    1     1     1     1
    1     1     1     1
    1     1     1     1
```

If we didn't want Matlab to print the updated version of "a" to the screen, we would have typed

```
a(2,3) = 4;
```

where the semicolon at the end of the line tells Matlab to suppress output to the screen. This becomes important when you're dealing with images that have approximately one million matrix elements! You don't want to sit there waiting while they are all printed to the screen. Let's set the element in row 5, column 2 to 10:

```
a(5,2) = 10;
```

1.2 Slice Notation

Matlab also has a convenient "slice" notation that allows us to select ranges from within a matrix. In general, the expression

```
a(i:j,s:t)
```

selects the part of the matrix ranging from rows i to j and columns s to t. To make this a little more concrete, let's say we wanted to select the first two rows of our matrix a. We would type

```
a(1:2,1:4)

ans =

    1     1     1     1
    1     1     4     1
```

Now let's say we wanted to select the 4th and 5th rows, from the 2nd to the 4th columns:

```
a(4:5,2:4)

ans =

    1     1     1
   10     1     1
```

Finally, let's say we wanted to select the third column of the matrix. Technically, we would type

```
a(1:5,3:3)

ans =

    1
    4
    1
    1
    1
```

But Matlab has a couple notational shortcuts to make this easier. In practice, we would probably type

```
a(:,3)

ans =

    1
    4
    1
    1
    1
```

where the ":" in the rows position tells Matlab that we want ALL rows, and the "3" in the columns position tells Matlab we want column 3 only.

For more one Matlab's slice notation, see (for instance) http://www.math.ufl.edu/help/matlab-tutorial/matlab-tutorial.html#SEC11. It is quite flexible and powerful and frequently allows one to replace what would be nested "for" loops in other languages with a single line of code.

1.3 The image is a matrix of numbers

In this section we will explore the idea that an image is really just a matrix. We will create a matrix and learn how to view it as an image using Matlab's "imshow" command.

Let's first create a 500x500 matrix of zeros (don't forget the semicolon).

```
img = zeros(500,500,'uint8');
```

For this "image", every pixel is equal to zero. So if we look at it, we'd expect it to be black everywhere:

```
imshow(img)
```

which is exactly what we see. To make a more interesting image, let's create a bright region by setting some pixels to 15:

```
img(250:350,250:350) = 15;
```

Now if we look at our "image", we can barely see a region of dark grey pixels:

```
imshow(img).
```

Why are our "bright" pixels so dim? Well, because this is an 8 bit image, our pixels values can range from 0 to 255 (where $255 = 2^8$ - 1). By default, Matlab takes the brightest possible pixel value of 255 to be white, and the darkest possible value of 0 to be black, while pixels from 1 to 254 are progressively brighter shades of gray. 15 is still pretty small compared to 255, and so pixels set to 15 appear quite dark.

The "imshow" command has a convenient syntax to force the brightest pixel in the current image to display as white, and the darkest pixel to display as black:

```
imshow(img,[])
```

Now the contrast is much improved. Let's set the upper left corner of the matrix to an even brighter value:

```
img(:100,:100) = 30;
```

Try viewing the image both with and without the automatic contrast adjustment:

```
imshow(img)
figure; imshow(img,[])
```

2 Scale Bar Calibration

When we take an image using a CCD all distances are given in pixels, the fundamental spatial unit of a digital camera. Of course, we can guess the size of what we're looking at from the supposed magnification of the scope (the magnification of the objective that was chosen). This tutorial will walk you

through calibrating a microscope such that you can include a scale bar on all your images. Unfortunately, even though widespread, this is not common practice in all biology. You'll often find images on text books without any scale bars.

2.1 Taking a good image

You will take an image of your calibration target using Micro-Manager (refer to the Micro-Manager manual and the TAs!). The calibration target is a cross with lines spaced every 10 μm. Notice that the targets we have already come with a coverslip on top of the slide. We will always image through the coverslip. For 100x imaging we'll have to put oil on it. When you're done imaging make sure you wipe the oil off using a cotton swab and isopropanol. You can find both of these cleaning supplies in the drawers underneath the microscopes.

Fig. 2.1 shows two representative images of the same calibration target. Both images are good enough for the purposes of calibration. On the one hand, there's a clear aesthetic difference between the two of them. On the other hand, the one that looks "nicer" is a little bit crooked. The image on the left presents some "bubbles" which could be due to the coverslip being dirty.

2.2 Loading the images in Matlab

As you have seen already in the microscopy tutorial, images can come in different bit depths. Most of the digital cameras you will use throughout the course have a native bit depth of 16 bits. This means that Windows won't be able to display them using the standard viewers. However, ImageJ and Matlab won't have any problems.

We start in Matlab by going to the folder where you have your image. This can be done by clicking on the "..." to the right of the white bar and just browsing for the folder. First, we load the image using the command `imread` and put it in the variable `Im`

```
Im = imread('100xTarget.tif');
```

Now, remember that a 16 bit image has $2^{16} = 65536$ levels of grey, but that your screen has only $2^8 = 256$ levels. If we want to display the image on the screen we'll have to tell Matlab how to scale the image down to the screen bit depth. For example, if you type

```
imshow(Im);
```

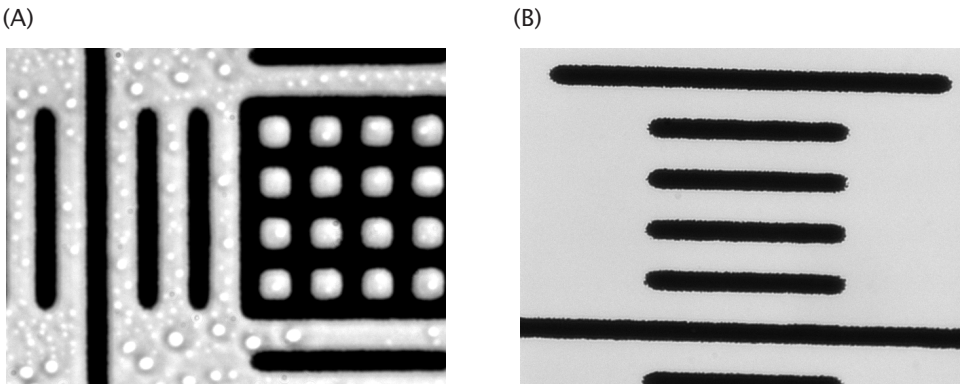

Figure 2.1: Brightfield images of a calibration target at 100x magnification. Both images are acceptable for calibration purposes. (B) just looks a little bit nicer than (A), probably because the coverslip in (A) wasn't cleaned properly. However, (B) is clearly a little bit crooked.

you'll just see a black image. The command `imshow` has a way of inputting the bit depth scaling. One option, for example, is to do

```
imshow(Im,[]);
```

This tells Matlab to grab the brightest pixel in the image and assign it a brightness of 1 and to grab the darkest one and assign it a brightness of 0. Matlab uses the $[0, 1]$ range for displaying images. Alternatively, one can specify the scaling by giving the thresholds between the `[]` in `imshow`. Please, refer to the help in order to learn a little bit more about it.

2.3 Measuring distances and calibrating

2.3.1 Mouse clicks

Now that we have successfully displayed the image we want to measure distances. We could write a really fancy algorithm for automatically finding the position of the bars. However, sometimes it's easier to do things manually. Matlab has a function called `ginput`, which allows you to click on different parts of the image and which returns the coordinates of where the

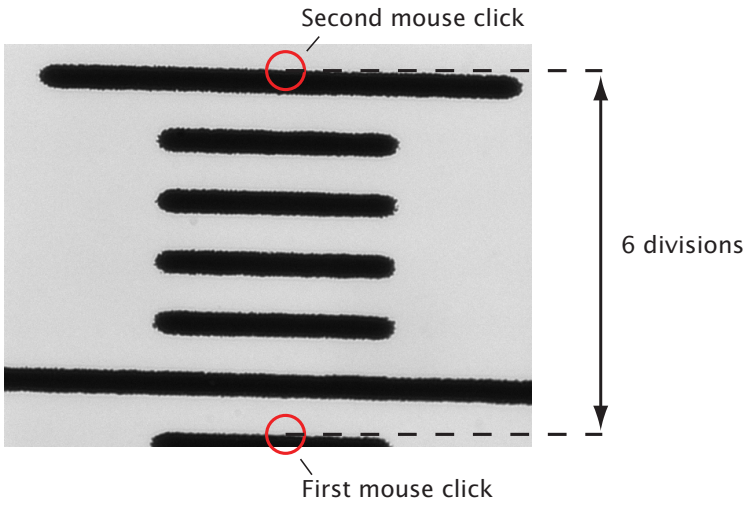

Figure 2.2: Using `ginput` to measure the distance between marks on the calibration slide. `ginput` is used to click on two different marks. The function outputs the coordinates of the clicks allowing us to compare distances in pixels with distances in μm.

clicks where made. We ask for two clicks and put the information in the variable `Pos`

```
Pos = ginput(2);
```

An example of where to click is shown in fig. 2.2. You want to click on the two bars that are the farthest apart from each other. This should decrease the error in finding the position of the edge of each mark.

The variable `Pos` is a matrix. Each row corresponds to a click of the mouse. The first column gives the x-coordinate and the second column gives the y-coordinate. Calculating the distance between the two points is just a matter of geometry

```
sqrt((Pos(1,1)-Pos(2,1))^2+(Pos(1,2)-Pos(2,2))^2);
```

In this particular case we get 386 pixels. This corresponds to five divisions or 50 μm. Therefore, the calibration factor is 0.13 μm/pixel. Now you can go back to your images and do a sanity check.

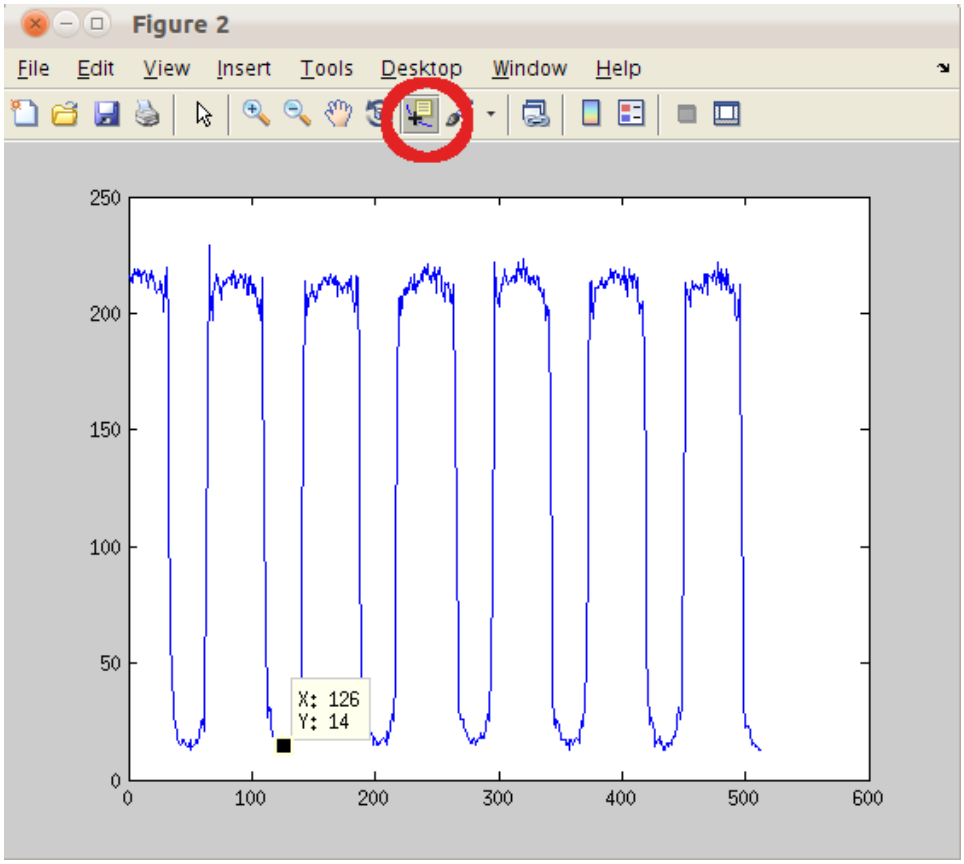

Figure 2.3: Pixel intensity vs row number along column number 336 of our scalebar image.

2.3.2 Plotting a line

Remember that an image is just a matrix. To see how pixel values change as we traverse an image vertically, we can use Matlab's slice notation to look at pixel values in a given column of the matrix. The image is 672 pixels wide, so let's look at the column in the middle of the image:

```
col = Im(:,336);
plot(col)
```

where the ":" tells Matlab that we want to slice across all rows of the matrix. As shown in the figure, we can use Matlab's "Data Cursor" to select the positions of various troughs in the plots. For instance, we find

that there are troughs at 47 pixels and 436 pixels corresponding to marks that are 50 μm apart. We can thus compute the scale as

```
x_1 = 47;
x_2 = 436;
scale2 = 50/(x_2-x_1);
```

This yields a scale of 0.13 μm per pixel, just as we found before.

2.4 Adding a scale bar

Adding a scale bar is relatively easy. One approach is to just set some pixels to 1, which is white. In the case of our image, its size in pixels is 512x672. This can be found out using the command

```
size(Im)
```

This command gives the size of the matrix with the image. The first number corresponds to the number of rows (y-coordinate) and the second one is the number of columns (x-coordinate). Let's make a 10 μm scale bar, which corresponds to $10/0.13 = 77$ pixels. First, we copy the image to a new variable. We also use the command `mat2gray` to convert the image directly to the $[0, 1]$ range

```
ImScale=mat2gray(Im);
```

Notice that if you do

```
imshow(ImScale)
```

you don't need to add the `[]` as an option of `imshow`. This is because `mat2gray` has already rescaled the image for the $[0, 1]$ range.

Now, let's make a bar that is 154 pixels wide and 10 pixels in height and locate it on the lower-right edge of the image

```
ImScale(950:959,1100:1253) = zeros(10,154);
imshow(ImScale)
```

Finally, we can save our newly generated image to a new TIF file

```
imwrite(ImScale,'ImageScalebar.TIF','TIF');
```

ImageJ can also add scale bars to images. This can be done by loading the image and then going to `Analyze` $\rightarrow$ `Tools` $\rightarrow$ `Scale Bar...`. From there everything is pretty self-explanatory.

2.5 Introduction

In this tutorial we will show how to merge the images of a fluorescent sample taken using different channels into one composite color image. When quantifying images it is usually better to work with the independent channels. However, it useful to be able to display all channels at once. Combining different channels can also be useful when looking for colocalization of different floruophores.

2.6 Loading and setting up the images

We start by loading the images corresponding to the three fluorescent channels. You'll be most likely working with a DAPI, FITC, and TRITC snaphots of the same cell. We'll assign DAPI to the blue channel, FITC to the green channel, and TRITC to the red channel. This assignment is related to the actual wavelengths of the emission of the fluorophores.

```
ImR=imread('brain_60x_700ms_TRITC_wheels.tif');
ImG=imread('brain_60x_400ms_FITC_wheels.tif');
ImB=imread('brain_60x_200ms_DAPI_wheels.tif');
```

Remember that these images have a bit depth of 16 bits and that the range of each pixel goes between 0 and 65535. First, we convert this range to the $[0, 1]$ range that Matlab uses by applying the function `mat2gray`

```
ImR=mat2gray(ImR);
ImG=mat2gray(ImG);
ImB=mat2gray(ImB);
```

Note that applying this function rescaled each one of the images based on their minimum and maximum pixel values. After applying this function we've lost the quantitative information stored in the absolute value of each pixel in each channel. We'll assume that we're making an image for displaying purposes (in your notebook, for example) so that this rescaling is not of importance.

2.7 Combining the different images

In order to combine them we'll use the Matlab function `cat`. This creates a structure that can be directly interpreted by `imshow` as a color image.

```
ImRGB=cat(3,ImR,ImG,ImB);
```

Where the "3" in the command is an option to set the dimension of this concatenated structure. Finally, we can show the whole image using `imshow(ImRGB)`.