CS484—MATLAB Tutorial

1 Installation

Please install MATLAB 2017a from the KAIST software library: https://kftp.kaist.ac.kr
Note that we use 2017a version in this course due to compatibility with different OS, even though 2017a is not the latest release.

When it comes to package selection, please install:

1. MATLAB (For 2017a, this is 'MATLAB 9.2')
2. Image Processing Toolbox
3. Computer Vision Toolbox
4. Statistics and Machine Learning Toolbox

On departmental Linux machines, launch MATLAB by calling 'matlab &' from the command line (the & runs MATLAB as its own process to leave the terminal free). On personal laptops, say on Ubuntu, Windows, or Mac, MATLAB should integrate with your GUI app launcher, or you can use your favourite CLI.

2 Official MATLAB Introduction

If you are new to MATLAB, we recommend starting with Mathworks ≈one hour introduction to MATLAB programming, presented as an interactive Website. Unfortumately this tutorial is available only for 2019a version, however it may be still helpful:
https://matlabacademy.mathworks.com/R2019a/portal.html?course=gettingstarted

In general, MATLAB's Help is excellent, so use it liberally. It includes a generous 'Getting Started With MATLAB':
https://www.mathworks.com/help/matlab/getting-started-with-matlab.html

The rest of this document contains concepts we will assume you know. Please become familiar with them, try them out, and let us know if you have any questions.

3 Syntax

3.1 Semicolons

You usually wish to include a semicolon after each line of code in a MATLAB script or function. If a line of code does not have a semicolon, then the result of the variable assignment will be printed to the command window. For example, if a variable x is set to 5 so that the line of code $x = 5$ is included without a semicolon:

```
x = 5
```

Then "$x = 5$" will be printed to the command window.

As expected, this can add clutter to your output; however, you may wish to omit semicolons for simple debugging purposes, as it may be useful to print the values of variables to the command window. Please see Section 7 for MATLAB's actual debugger.

4 Arrays

4.1 Indexing

MATLAB arrays are 1-indexed rather than 0-indexed.

To access or modify the first element of an array, the element at position 1 in the array is called. To see how this works, try the following exercise.

In a MATLAB command window, create a 5-element array A, with integers 1 through 5 as follows:

```
A=1:5;
```

A should now be equal to the array $[1, 2, 3, 4, 5]$.

Now, type $A(1)$ into the command window and press enter. Note that 1 rather than 2 is the return value; 1 is the first element in the array. Also, note that because of 1-indexing, the command $A(0)$ will lead to an error.

Now try modifying the first element of the array by setting its value to 5; type $A(1) = 5$ into the command window and press enter. The value of the array should now be $[5, 2, 3, 4, 5]$.

5 Reading and Displaying Images

The *imread* function is used to load images in the MATLAB environment.

The *imshow* function takes in a matrix representation of an image as a parameter and displays the image.

5.1 Types

There is no type declaration in MATLAB; the program handles this automatically. However, MATLAB variables *do* have types, and it is critical to undertand how this works in order to use various image processing functions.

Try downloading an image from the internet and reading it into the MATLAB environment like so:

```
image = imread('yourimage.jpg');
```

If you look at the 'Workspace' section to the left side of the console, you should notice that the 'image' variable is a *uint8* array, which means that each value in the array is an 8-bit unsigned integer. Thus, if you examine the array, you should notice that it consists of integers from 0 to 255.

At times, you will want to alter the image in ways such that some of the entries become non-integer values, and thus you will want to convert the image to floating point format by using the MATLAB functions *single* or *double*. If this conversion is not done, an entry is automatically rounded to an integer if it is set to a non-integer value.

Note that converting the image array to floating point format can lead to issues when using built-in MATLAB image processing functions. When using *imshow*, for example, MATLAB assumes that the image pixel values are between 0 and 255 if the image is in integer format; if the image is in floating-point format, it is assumed that the pixel values are in between 0 and 1. Thus as a general rule, when you convert an image to floating-point format, you should normalize the image so that all its entries are between 0 and 1; this helps prevent potential bugs. This is easiest done by using the MATLAB functions: *im2single* or *im2double*; these functions convert an array to floating-point format and then normalize the array so that all of its values are between 0 and 1.

To illustrate these concepts, try using these functions yourself: First, try converting the image that you read from a file to floating-point format and displaying it as follows:

```
floatImage = single(image);
imshow(floatImage);
```

The displayed image should look very strange; it should hardly resemble an image. This is because it has not been normalized.

Now, try converting the image to floating-point format and normalize it from 0 to 1; then display the image:

```
floatImage = im2single(image);
imshow(floatImage);
```

The displayed image should now look normal. Also, note that if you examine any of the entries in the floatImage array, you should find that all entries have values between 0 and 1.

5.2 Multidimensional Arrays / Matrices

Multidimensional arrays in MATLAB are an extension of the two-dimensional matrix. One of the most common usages of multidimensional arrays in computer vision is to represent images with multiple channels. For instance, an RGB image has three channels, and can be represented as a 3-D array. Each of these channels can be accessed independently.

Let us create an RGB image. To begin, let us create a 300x400x3 array and initialize it to zeros. This can be done as follows:

```
image = zeros(300, 400, 3);
```

Now, we assign a mid red to the first hundred columns and a bright red to the following hundred columns:

```
image(:,1:100,1) = 0.5; % 'half' red
image(:,101:200,1) = 1; % 'full' red
```

The colon ':' indexes all elements in a particular dimension.

Finally, we can assign green randomly to the first 100 rows:

```
image(1:100,:,2) = rand(100,400);  %Green
```

To view the image, type the following into the command window:

```
imshow(image);
```

5.3 Color images vs. Grayscale

Color images are often built of several stacked color channels, each of them representing value levels of the given channel. For example, RGB images are composed of three independent channels for red, green and blue primary color components. In contrast, a grayscale image (aka black and white image) is one in which the value of each pixel is a single sample, that is, it carries only intensity information.

In MATLAB, it is easy to convert an RGB image to grayscale. This can be achieved using the *rgbtogray* function.

You can also access individual color channels of a color image. This is illustrated in the code snippet below.

```
% Read in original RGB image.
rgbImage = imread('yourimage.jpg');
[m,n,o] = size(rgbImage);

% Extract color channels.
redChannel = rgbImage(:,:,1); % Red channel
greenChannel = rgbImage(:,:,2); % Green channel
blueChannel = rgbImage(:,:,3); % Blue channel

% Create an all black channel.
allBlack = zeros(m, n, 'uint8');

% Create color versions of the individual color channels.
justRed = cat(3, redChannel, allBlack, allBlack);
justGreen = cat(3, allBlack, greenChannel, allBlack);
justBlue = cat(3, allBlack, allBlack, blueChannel);

% Recombine the individual color channels to create the
    original RGB image again.
recombinedRGBImage = cat(3, redChannel, greenChannel,
    blueChannel);
```

Try to view the various results using *imshow*.

6 Performance Improvements

In MATLAB, it is best to avoid using for loops whenever possible; one can attain significant performance improvements through vectorization and logical indexing.

6.1 Pre-allocation

MATLAB supports dynamic array allocation; it is not necessary to allocate space for an array before making assignments. For example, suppose you want to create a 10 element array such that every element is the integer 5. This could be done as follows:

```
for i=1:10
    A(i)=5;
end
```

While convenient, this is slow: performance significantly improves if you pre-allocate as follows:

```
A=zeros(1,10);
for i=1:10
    A(i)=5;
end
```

This initially sets A to a 10-element array of zeros; without pre-allocation, elements need to be re-copied every time the size of the array is increased. For a small list like this, the impact is not noticeable, but for long lists it will quickly become a bottleneck. Thus, you should always pre-allocate if possible.

6.2 Vectors as function parameters

Most MATLAB functions support passing vectors or matrices as parameters. This prevents you having to apply the function to individual elements as a way of improving performance. It is best illustrated with a few examples:

Suppose you have a 10-element array A. You want to take the sine of each element and store the results in another array B. A naive method would use a for loop as follows:

```
B=zeros(1,10);
for i=1:10
    B(i)=sin(A(i));
end
```

The same operation can be accomplished as follows:

```
B=sin(A);
```

Similar operations can be completed if one wishes to raise every element in A to a certain power. For example, suppose we want to square every element in A and store the result in B. This can be done as follows:

```
B=A.^2;
```

There are many examples of vectorization in the MathWorks help here.

6.3 Logical Indexing

Suppose we have an $m \times n$ 2D array, and we want to set every element in the array that has a value greater than 100 to 255. This can be done as follows with a for loop:

```
m = 400;
n = 400;
A = randi( 255, m, n, 'uint8' );
for i=1:m
    for j=1:n
        if A(i,j) > 100
            A(i,j) = 255;
        end
    end
end
imshow( A );
```

A more efficient method uses logical indexing:

```
B = A > 100;
A(B) = 255;
```

B is now a binary logical array, where for all i, j, $B(i, j) = 1$ if and only if $A(i, j) > 100$; otherwise, $B(i, j) = 0$. Then we do the following: $A(B) = 255$. An element-wise assignment is then performed; the result of A the same as it would be using the for loop method. A appears brighter, as more pixels are set to their maximum value.

6.4 Evaluating Performance

We can evaluate time performance using the the built-in MATLAB functions tic and toc. It is used as follows:

```
tic;
% Perform some operation
toc;
```

The elapsed time between tic and toc is then printed to the command window. You should try doing several of the examples above, and note the performance differences between using for loops and using the more efficient methods.

7 Debugging

Like other IDEs, MATLAB has a dedicated debugger. Please see the following Mathworks page for information on how to enter the debugger, and how to set and navigate breakpoints: MATLAB Debugger

8 Homework 1 questions

Made it this far? Good. Now, please attempt the Homework 1 questions.