Day 1: Introduction to MATLAB and Colorizing Images

CURIE Academy 2015: Computational Photography

Sign-Off Sheet

NAME: _____

NAME: _____

	Sign-Off Milestone	**TA Initials**
Part 1.1	Introductory Math Problems	
Part 1.2	Function Calls	
Part 1.3	Hailstone Sequence	
Part 2.1	turtleForwardDashed	
Part 2.2	turtleNgon	
Part 3.1	Negative	
Part 3.2	Horizontal Flip	

1 MATLAB basics

We'll be programming in MATLAB. To open up MATLAB, go to **Start**, then type **MATLAB**. Select **MATLAB R2015a**.

MATLAB has a default "working directory" on your computer. We want you to be able to load and save work from your USB drive instead. To do this, plug in the USB, then click the triangle next to the **C:** at the top of the screen and select your USB drive instead.

To download the files for the day, go to the official CURIE webpage at

`http://www.cs.cornell.edu/workshop/curie2015`

then click **handouts**, and select the **starter files** link next to Lab 1. Download the ZIP file and save it to your USB, then unzip the file. You should be able to see these files in the left panel of your MATLAB window.

1.1 Introductory math problems

Refer to the MATLAB reference sheet for help figuring out how to solve the problems in the MATLAB command window. Make sure to switch the driver and navigator after every question. At the end of each mini-section, get a TA to check off that you've done the assignment before moving on to the next part.

Exercise 1 *Set* `foo` *to be 2.37. What is* $10 * \texttt{foo} + \texttt{foo}^{\texttt{foo - 1}}$*? Write the answer below.*

Exercise 2 *Set a variable* x *with value 2. Set* y *to be* x *raised to the fourth power. Set* z *to be* y *raised to the fourth power. Write the value of* z *below.*

Exercise 3 *Write a script that uses a "for" loop to find the product of all even numbers between 2 and 20. Write the product below.*

Checkpoint: get a TA to sign off for your answers in this section.

1.2 Function calls

Again, use the MATLAB reference sheet to help you in solving the following problems. Remember that you can find information on how to use any function by typing `help functionname`.

Exercise 4 *Write a script in **ex4.m** that finds the sum of the square roots of 1 through 10. What is that sum?*

Consider the following code:

```
if x < 0
    y = 0;
else
    y = x;
end
```

Exercise 5 *Write a single line of code that accomplishes the same thing as the `if/else` statement above. Hint: remember the `min` and `max` functions.*

Checkpoint: get a TA to sign off for your answers in this section.

1.3 Hailstone Sequence

The Hailstone sequence is a way of generating a list of numbers following a strange pattern. To generate a Hailstone sequence, you start with a number, x. The next number in the sequence depends on the current one:

1. If x is even, the next number is $x/2$.
2. If x is odd, the next number is $3x + 1$.

So if we start with x = 10, the hailstone sequence is looks like this:

$$10, 5, 16, 8, 4, 2, 1, 4, 2, 1 \ldots$$

We'll call each of the numbers in the sequence an *element*; the *index* of each element is just its numerical position in the sequence - the first element has index 1, the tenth element has index 10, and so on.

A useful function for computing elements of the Hailstone sequence is the `mod()` function, short for the modulus or remainder operator. So, for instance, `mod(10, 3)` is 1, as 10 divided by 3 is 3 with remainder 1. More generally, `(mod(a, b)` finds remainder when `a` is divided by `b`, or the difference between the `a` and the biggest multiple of `b` that is less than or equal to `a`. Here, it's useful to notice that when divided by 2, an even number has remainder 0 whereas an odd number has remainder 1.

Exercise 6 *In **ex6.m**, write a script to print the first 10 numbers in the Hailstone sequence starting with 14. You can leave off the semicolon when computing the number, or you can use the* `disp` *function to print out each number a little more cleanly. Verify on paper (or in your head) that your code is printing the first few elements correctly and write the 10th element below.*

Exercise 7 *Find the first 200 numbers in the Hailstone sequence for 54. What is the index of the first 1 to appear in the sequence? Write your answer below.*

Checkpoint: get a TA to sign off for your answers in this section.

A "conjecture" (something observed but not proven to be true in all possible cases) is that the Hailstone sequence for any positive integer will eventually reach 1. This is often called the Collatz conjecture. We're not going to try to prove anything about this, but we can write a program to test an individual case.

Exercise 8 ***(Bonus Challenge)*** *Write a script in **ex8.m** that will print the index of the first 1 in the Hailstone sequence for some variable* `start`*. See if you can get your script to stop after you find the first 1. Hint: look up the* `help` *for the* `break` *statement. Try out different starting numbers; what's the largest number you can get your program to output?*

2 Turtle Graphics and Functions

2.1 Drawing with Turtles

The next exercises simulate you drawing out the path of a turtle walking around a blank canvas with a pen attached to its belly. (No actual turtles will be harmed in this exercise). We can give this turtle programmatic instructions to have it draw things!

To make a new turtle, run `t = Turtle()`. This will create a new Turtle object that we store as the variable `t`, that starts in the center of the canvas at 0, 0, facing up.

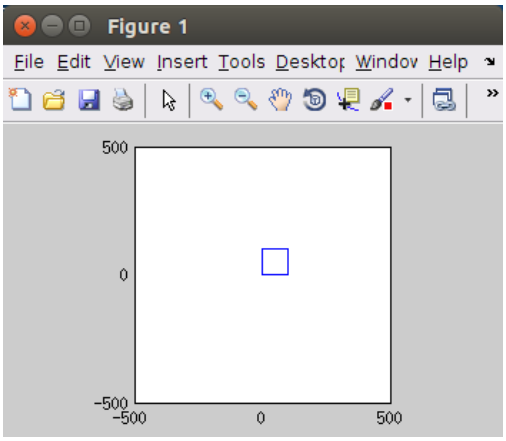
We have given you the following commands for controlling Turtles, each of which returns a modified Turtle.

- `t = turtleForward(t, x)`: Move the Turtle x pixels in whatever direction it's facing.
- `t = turtleTurn(t, r)`: Turn the direction of the Turtle r degrees clockwise.
- `t = turtleDown(t)`: Start having the Turtle draw its path.
- `t = turtleUp(t)`: Stop having the Turtle draw its path.

Here's the code we saw in the demo for a function that takes in a Turtle object and a sidelength and uses the Turtle to draw a square with the given side length and the first corner where the Turtle begins.

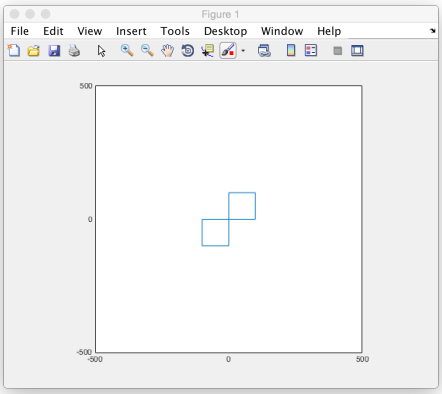
```
function t = turtleSquare(t, sidelength)
    % For each of the four sides
    for i = 1:4
        % We draw the side, then turn to prepare for the next one
        t = turtleForward(t, sidelength);
        t = turtleTurn(t, 90);
    end
end
```

If we make a new Turtle `t = Turtle()` and then run `t = turtleSquare(t, 100)` after writing this code, we see a square of side length 100 on our canvas.



Using these instructions and the MATLAB reference sheet again, try out the following Turtle exercises. *Remember that you always need to pass the turtle into the function, and assign the returned turtle back to your turtle variable, or the turtle won't remember where it went!*

Exercise 9 *Use the provided turtle functions to draw an image that looks like the figure below in a script in* ***ex9.m****.*



Exercise 10 *Fill in the* `turtleNgon(t, n, sidelength)` *function in* ***turtleNgon.m****, a function that takes in Turtle* `t`*, the number of sides for a polygon* `n`*, and the length of each side* `sidelength`*, and draws a regular polygon with that many sides. For instance,* `turtleNgon(t, 5, 100)` *should draw a regular pentagon with sides of length 100 pixels.*

Exercise 11 *Fill in the* `turtleForwardDashed(t, dist)` *function in* ***turtleForwardDashed.m****, a function that moves the Turtle* `t` *forward* `dist` *pixels (just like* `turtleForward` *but makes it draw a dashed line as it goes. Use* `turtleUp` *and* `turtleDown` *to draw dashes. We start with code that goes a fixed 100 pixels and draws one dash; modify it to move* `dist` *pixels forward with five dashes. Hint: you might find it easiest to write this with a for loop.*

Checkpoint: get a TA to sign off for your answers in this section.

Exercise 12 *Make a copy of* ***turtleNgon.m****, and save it as* ***turtleNgonDashed.m****. Change this file to implement a function* `turtleNgonDashed(t, n, sidelength)` *that does the same thing but draws the n-gon with a dashed line. Remember that you can re-use functions you've already written!*

Exercise 13 ***(Extra Bonus)*** *Make a new Turtle function in* ***ex13.m*** *that draws something cool! Feel free to use any of the functions we've given you as well as any of the ones you've written yourself.*

2.2 (Bonus Challenge) State and Properties

For each of these instructions, it's important to save the new *state* of the Turtle, or the information needed to specify where the Turtle is right now and whether or not it's drawing. So, rather than just running `turtleForward(t, x)`, we need to run `t = turtleForward(t, x)` so that t can keep track of where the Turtle is and what direction it's facing after each action. We can access specific parts of the state of a Turtle, or its *properties*, using the format `t.property` to mean "the value of property *property* for the Turtle t". Turtles have several properties:

- `t.x`: The x coordinate of the Turtle,
- `t.y`: The y coordinate of the Turtle,
- `t.angle`: The current angle of the Turtle (where 0 means directly up),
- `t.draw`: Whether the Turtle will draw when moved (pen is down) or not (pen is up). `draw` is 0 if the pen is up and 1 if the pen is down.
- `t.fig`: Which figure window the Turtle is drawing in (1 for Figure 1, etc.).

You can see these values when you create a new Turtle, or anytime you enter your Turtle variable without a semicolon at the Matlab prompt:

```
>> t = Turtle()
t =
         x: 0
         y: 0
     angle: 0
      draw: 1
       fig: 1
```

Exercise 14 ***(Super Extra Bonus Challenge!)*** *Fill in `turtleTeleport(t, x, y)` function in* ***turtleTeleport.m*** *that takes in an x and y coordinate and moves Turtle t to those coordinates. If the Turtle had the pen down before teleporting, it should have the pen down after; otherwise, the pen should be up. It should also be facing the same direction as it was before teleporting. You'll need to access properties of the Turtle to do this.*

3 Image Transformations

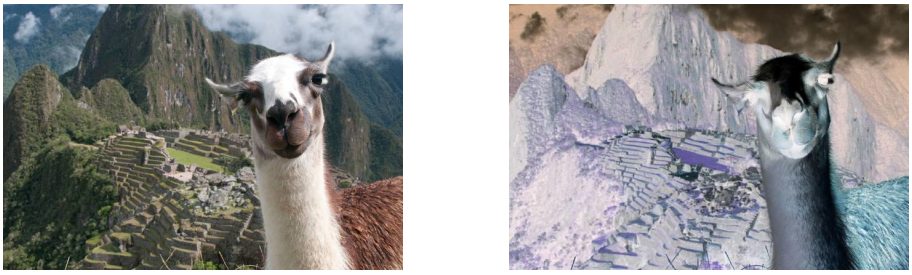
Use the MATLAB reference to load one or more of your favorite pictures from your phone or the internet (or any of the provided images) into MATLAB with the `imread` function. For instance, if you want to load the llama we use in our tests, you can type `machu = imread('images/machu.jpg');`.

Let's try transforming some images!

3.1 Negatives

We know that each pixel is a mix of **red** , **green** , and **blue** [RGB] with each have a value from 0 to 255 [$2^8 - 1$, the biggest number you can represent with 8 bits]. We use these because **red** `[255, 0, 0]`, **green** `[0, 255, 0]`, and **blue** `[0, 0, 255]` are the primary colors for light, the same way that **red** , **yellow** and **blue** are the primary colors for ink. There are also secondary colors: **red** + **green** is **yellow** `[255, 255, 0]`, **red** + **blue** is **magenta** `[255, 0, 255]`, and **green** + **blue** is **cyan** `[0, 255, 255]`.

The *negative* of an image is the inversion of every single pixel's color by subtracting each color value from 255. For instance, the inverse of the color **blue** is `[255 - 0, 255 - 0, 255 - 255]`, or **yellow** .



(a) Original image (b) Negative image

Figure 1: Output of the `negative` function on a test image.

Exercise 15 *Fill in the function `negative(im)` in **negative.m** so it returns the inverted version of the input image. Hint: there's a way to do this in one line!*

Checkpoint: get a TA to sign off for your answers in this section.

3.2 Flip

To flip an image horizontally, we simply want to swap the colors of pixels across a line running down the center of the image. If we have an image of width 100, we know that the pixel in row 25 and column 15 in the new image should be the same color as the pixel in row 25 and column (100 - 15) = 85 from the old image.

Exercise 16 *Fill in the function `flipHorizontal(im)` in **flipHorizontal.m** so it returns the horizontally flipped version of the input image. Remember: `im(i, j)` is the pixel in the ith row and jth column of the image.*

Checkpoint: get a TA to sign off for your answers in this section.

(a) Original image

(b) Flipped image

Figure 2: Output of the `flipHorizontal` function on a test image.

3.3 (Extra Bonus Challenge) #Colorful

Image transformations can be a fun way to play with images. In particular, we're going to tint images, or make images appear with a single color. Pretty much every single method for doing this starts from turning an image to grayscale, which we can do for each pixel by averaging out the intensities of the three colors for each pixel, and then use that value for each color channel in that pixel. We give you the starting code for doing this in **tint.m**.

(a) Greyscale

(b) Level 1

(c) Level 2

(d) Level 3

Figure 3: Different outputs of the `tint` function.

3.3.1 Level 1

The easiest way to tint a black-and-white image is just to scale the three numbers specifying your tint color by the brightness, or average of the three colors, of each pixel in the original image. A white pixel (one with brightness 255) will be turned to the color you choose - for instance, if you pick **red** the resulting pixel will be `(255 / 255) * [255, 0, 0]` = `[255, 0, 0]`. In effect, we're turning everything white in the greyscale image to red in our new image. On the other hand, a **black** pixel (one with brightness 0) will be transformed to be `(0 / 255) * [255, 0, 0]` = `[0, 0, 0]`; it will just be black in the resulting image. Every shade in between will have the same proportions between RGB values as the color you picked, but scaled down to be darker according to the brightness of that pixel.

Exercise 17 ***(Extra Bonus Challenge)*** *Rewrite the function* `tint(im)` *to return a black-and-red image (or whatever color you prefer).*

3.3.2 Level 2

A somewhat harder version of this is, instead of making a black-and-your-color image, to make a your-color-and-white image. Instead of having our brightest pixels be red, we might want them to be white, and our darkest pixels to be red, giving a cool faded-image look. This takes a bit more math: we want black to

map to our starting color (say `[255, 0, 0]`) and white to map to, well, white. In effect, instead of keeping the ratios between the three color channels the same, we want to keep the ratio of the difference between 255 and each of our color channels the same. So, a pixel halfway up our grey spectrum (brightness 127) would be translated to `[255, 0, 0] + (127 / 255) * (255 - [255, 0, 0])` = `[255, 127, 127]`.

Exercise 18 ***(Super Extra Bonus Challenge)*** *Modify the function* `tint(im)` *to return a red-and-white image (or whatever color you prefer). Feel free to comment out your old code with %s or start a new function file so you can use the Level 1 function later.*

3.3.3 Level 3

Both of these images end up being a little bit harder to see clearly than the original black-and-white image. The reason is that each of the two methods before is only using half of the set of colors that have the same tint: Level 1 is using from black to red, and Level 2 is using from red to white. What would be really cool is if we could find a way to make darker pixels (say, those with brightness less than the brightness of your color) use the strategy from Level 1, and make lighter pixels use the strategy from Level 2. That way, we can get black and white to show up clearly and see our color.

Exercise 19 ***(Super Ultra Extra Bonus Challenge)*** *Modify the function* `tint(im)` *to return a black-and-red-and-white image (or whatever color you prefer). Remember that whatever brightness level you switch from the method in Level 1 to the one in Level 2 should map to your color (e.g. red), so you might have to tweak the brightness ratio part of the formulas from Level 1 to make sure that you get the full range of different shades of your color.*