INM460 / IN3060 Computer Vision

In this lab, we will use Matlab to perform image processing. First, we will learn how to make scripts and functions in Matlab; these provide convenient (and reusable) ways to code. Then, we'll explore some simple image processing methods, and implement an Instagram-like filter on an image. Finally, we'll apply a projective warping of an image to augment content into an image.

Requirements:

- Matlab
- Image provided on Moodle (Lab materials 02/AlbumReview.bmp)
- Images of your own, and/or those found on the internet

Task 1: Script files

Last week you were introduced to Matlab, which is a very popular environment for scientific computing, and in particular computer vision. You learned about Matlab features, including some (of the many) built-in functions it includes, like sin, cos, and plot. You entered commands at the Matlab prompt in the Command Window to perform numerical operations.

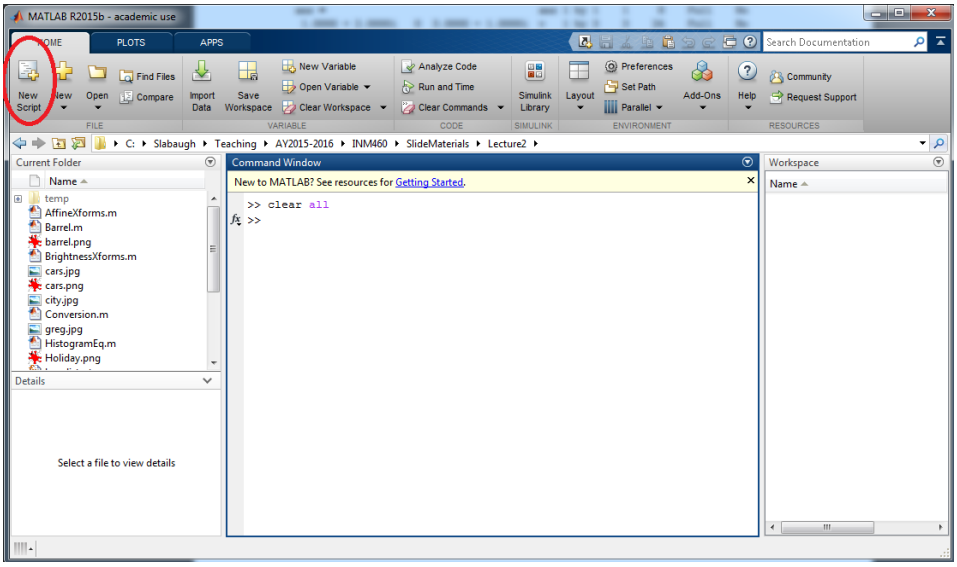
Using Matlab this way is very powerful. However, you may want to run a larger set of commands, and possibly rerun them multiple times. Rather than typing in the commands manually each time, it will be more convenient to use a Matlab *script*. Script files are simply ordinary text files that contain Matlab commands, and in Matlab, they have an extension .m. For this reason, they're commonly called *m-files*.

Matlab files will run in the current directory (or directories on the Matlab path). You can set the current directory at the Command Window using the command cd (which stands for change directory). For example,

```
cd d:\
```

will change the current directory to the D: drive on your computer.

In this directory, you can create a script using the editor provided in Matlab. Click on the New Script button on the Matlab interface, circled in red below.



The Matlab editor will pop up a new, blank script for you. In the script, you can enter some Matlab commands. For example, in last week's lab you wrote Matlab code to evaluate the function for x = 1 and y = 2:

$$f(x,y) = \frac{x^2 + 2xy - \cos(xy)}{\sqrt{x^2 + y^2}}$$

Let's implement this as a Matlab script. Copy and paste the following lines to make the function as a script:

```
x = 1;
y = 2;
f = (x^2+2*x*y-cos(x*y)) / (sqrt(x^2+y^2))
```

and click on the green Run button. Matlab will want to save this script as an .m file in the current directory. Give the file an appropriate name, like script1.m and save it to your D: drive. Then, when you run it, the commands will execute one after another. After the script runs, you should see the result, in the Command Window:

```
>> script1
f =
   2.4222
```

Note: you can also run the script from the Command Window by typing

```
script1
```

Task 2: User-defined functions

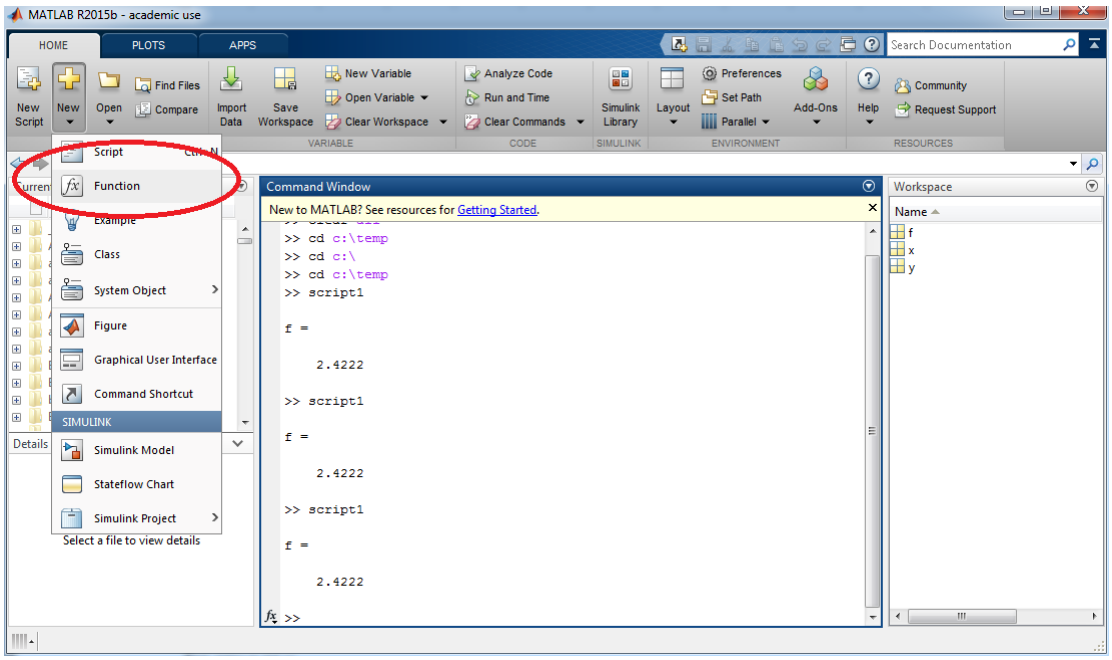
You can take writing scripts that use Matlab's built-in functions very far. However, a great feature of Matlab is that *you can write your own functions* to extend Matlab's capabilities. Putting your code into a function can be a good choice, as functions can be reused between different scripts.

Matlab functions also have an .m extension. The primary difference between a function and script is that *a function receives inputs and returns outputs*. As an example, let's create a Matlab function that can evaluate

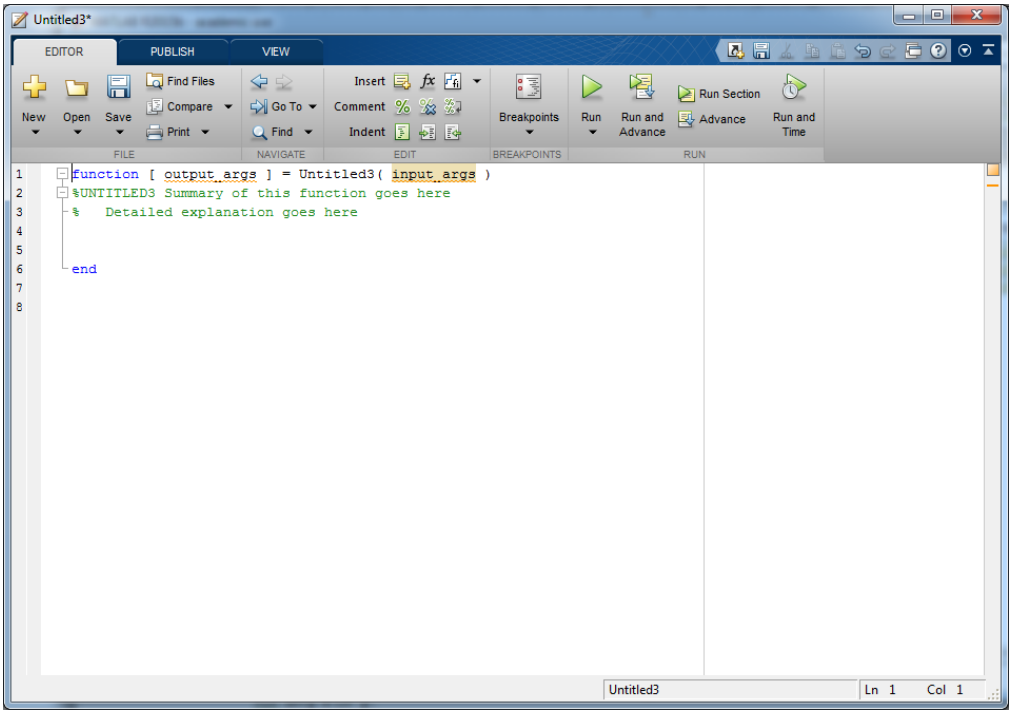
$$f(x,y) = \frac{x^2 + 2xy - \cos(xy)}{\sqrt{x^2 + y^2}}$$

for *any* x or y.

In Matlab, click on New->Function as shown in the figure below.



This will launch the Matlab editor again. The difference this time, is that Matlab has created an interface for the function, with input arguments and output arguments.



Our function will have two input arguments, x and y. It will return a single output argument f. Copy and paste the code into this function:

```
function [ f ] = function1( x, y )

%function1 evaluates my function for any x and y
f = (x^2+2*x*y-cos(x*y)) / (sqrt(x^2+y^2));

end
```

You'll see we can pass in values for x and y, and the function will compute f, and return it as an output argument. Save this function with a filename "function1.m" in the current directory. In the Command Window, now type

```
function1(1, 2)
```

You'll see that the function gets executed and returns a value of 2.4222 as expected.

See if you can write a function called "circleFeatures.m" that returns the area and circumference of a circle of radius r. The interface should look like this:

```
function [ A, C ] = circleFeatures( r )
```

Test your function with r = 1 by calling `[ A, C ] = circleFeatures( 1 )` at the Command Window prompt. If coded correctly, the area should be 3.1416 and the

circumference should be 6.2832. Recall Matlab recognises pi as a keyword representing the mathematical constant.

Task 3: Negative three ways, and colour swapping

Let's take the negative of an image. In fact, we'll do this three different ways so you can see how pixels can be processed in Matlab.

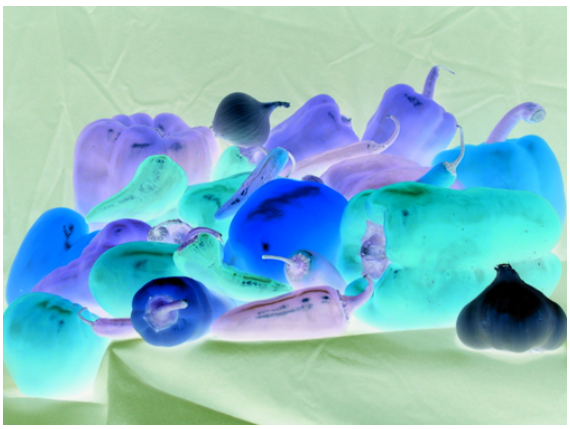
Method 1

First, let's load and display an image. Any image will do; let's use the peppers image that comes with Matlab.

```
I = imread('peppers.png');
imshow(I);
```

As you learned in lecture, a negative of an image can be easily achieved by simply as

```
J = 255 - I;
figure; imshow(J);
```



Here, the value of each R, G, and B component at every pixel in `I` is subtracted from 255.

Method 2

An image is a collection of pixels (for a colour image, each having a red, green, and blue component). The pixels are in the form of a 2D grid. If you wish, you can loop over all these pixels, using a *doubly-nested loop* (loop within a loop), i.e.,

```
I = imread('peppers.png');
[height, width, channels] = size(I);
for x = 1:width
```

```
    for y = 1:height
        J(y, x, 1) = 255 - I(y, x, 1); % invert red
        J(y, x, 2) = 255 - I(y, x, 2); % invert green
        J(y, x, 3) = 255 - I(y, x, 3); % invert blue
    end
end
figure; imshow(J);
```

Note that because the matrix indices are given as rows, then columns, the y comes before the x when indexing pixels from the image.

Method 3

In this example, we'll still use a doubly-nested loop to loop over all the pixels in the image. Since inverting an image applies the same operation to the red, green, and blue colours, we can use the colon operator to access the third dimension in the image (the colour at this pixel) and process the colour with a single line of code.

```
I = imread('peppers.png');
[height, width, channels] = size(I);
for x = 1:width
    for y = 1:height
        J(y, x, :) = 255 - I(y, x, :); % invert red, green, and blue
    end
end
figure; imshow(J);
```

Instead of inverting colours, try to write a script that *swaps* colours between different colour channels, for example, swapping red for blue. You may find it's easier to complete this task using the Method 2 code above. Can you reproduce the image below? Suddenly those peppers look less tasty!

Task 4: Instagram

Instagram is popular online mobile photo sharing service that allows users to apply artistic effects (a.k.a *filters*) to images to change their appearance. The company grew rapidly and was purchased by Facebook for US $1B in 2012. In this task, we will implement some simple Instagram-like filters to images.

Find an image you would like to work with, and in a Matlab script, write some Matlab code to load and display the image. Note that some images are very large; if yours is too big for the screen Matlab will display a warning when you try to display it with `imshow`. If you get such a warning, try to decrease the image size using the `imresize` function. Here is my image:

In lecture today, an example was provided where the saturation of an image was decreased, by scaling the S component by 0.5. Try *increasing* the saturation in your image by scaling the saturation by 1.5. Performing this on my image transforms it to:

Notice how saturated the colours are (blues are really blue, nearby trees are really green)!

Now, let's add a vignette, which is a reduction of the image brightness from the image centre. This helps draw a person's attention to the centre of the image, and gives the image a certain low fidelity look. To do this, we first need to know where the centre of the image is. This can be achieved based on making a call to the `size` function, shown earlier. Based on this, compute the centre $[cx, cy]^T$ of your image, and to check that your computation is correct, display the centre of your image using:

```
hold on;
plot(cx, cy, 'g+');
```

If everything is correct, you should see a green plus in the centre of your image, like this:

Now, we'd like to compute a function that will modify the brightness from the centre of the image. In particular, at each pixel, we will scale the brightness with a value f that is equal to one in the centre of the image (so it has no darkening effect) and falls off to a small value approaching zero away from the centre. This will darken pixels away from the centre of the image. Mathematically, we will use the function

```
f = exp(-r/height);
```

where `height` is the image height, and `r` is the radius, or distance, from the pixel at $[x, y]^T$ to the centre of the image $[cx, cy]^T$. Think about this function for a second. At the centre of the image, r = 0, and exp(0) = 1. However, as we move away from the centre, the argument to exp is a negative number, producing an f between 0 and 1. When we move far away from the centre, the argument to the exp function is a larger negative number, which produces an f close to zero.

With this in mind, form a *doubly-nested loop* to loop over every pixel in the image; that is

```
for x = 1:width
    for y = 1:height
        % Darken image pixels here
    end
end
```

and in the inner part of the loop, compute the radius r as the distance between $[x,y]^T$ and $[cx,cy]^T$, and the value of f given the function above. Use f to scale the image brightness by multiplying the image's R, G, and B components (if your image is in RGB colourspace) or the V component (if your image is in the HSV colourspace) to apply the vignette. Here is my result:

Let's compare the before and after:

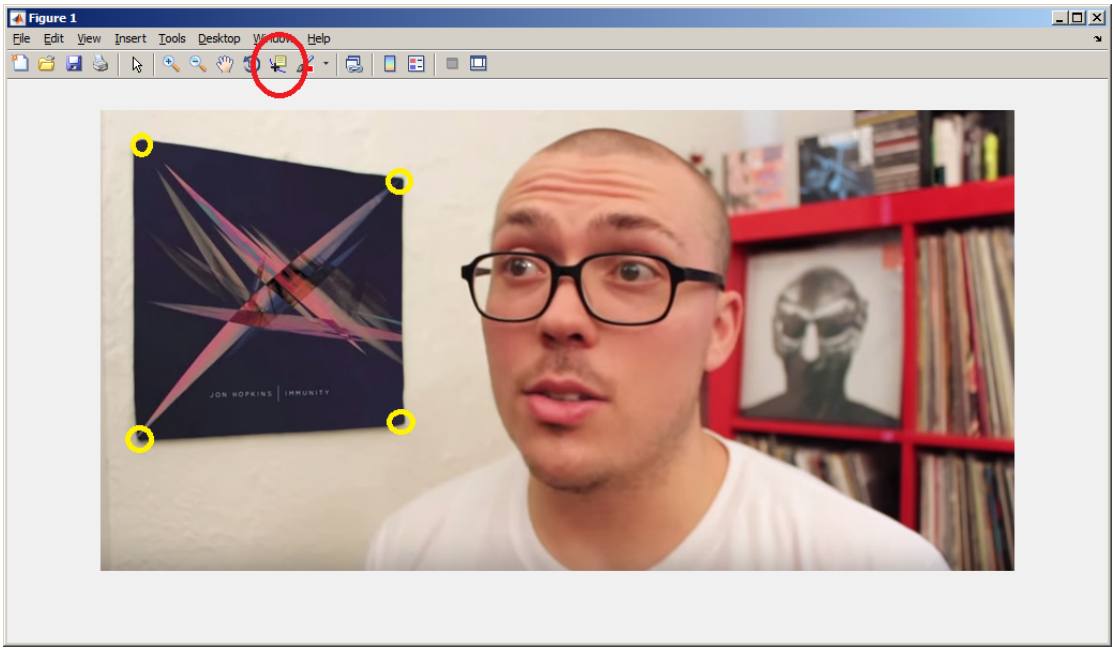
	
Before	**After**

Awesome! You can probably think of many other ways to create Instagram-like filters; this webpage provides some discussion and ideas.

Task 5: Augmented reality with a projective transformation

Anthony Fantano is self-described as the "internet's busiest music nerd". Through his YouTube channel *theneedledrop* he does video blogs related to music, providing album reviews. In many of his videos, an album cover is virtually displayed in the background, as if it is mounted, on an angle, to his wall. In this task, we will reproduce this trick using a projective transformation.

Download the photo AlbumReview.bmp from the Lab materials 02 folder on Moodle, and save it to a directory on your hard drive. We'll call this image I. In the same folder as the image, write a Matlab script to load and display the image on the screen. It should look like this (excluding the red and yellow circles):



Next, using the Data Cursor on the Matlab figure (circled in red above), find the four 2D coordinates (yellow circles) of the album cover in the background, on the left side of the image. Save these coordinates in your Matlab script. Referring to today's lecture notes, these are points denoted with "**q**".

Next, find an album cover from your favourite musical artist. Download this image and save it in the same directory. Load and display the image in Matlab. We'll call this image J. For example, I'm a big fan of Com Truise; the cover for his album Galactic Melt looks like this.

Now find four corners on the album cover to establish *correspondences* between the two images. Save these points in your Matlab script as well. Referring to today's lecture notes, these are points "**p**".

Based on the four 2D correspondences, you can estimate the projective transformation that will align the album cover to the background of the AlbumReview.bmp image. In fact, you have learned two to do this; either

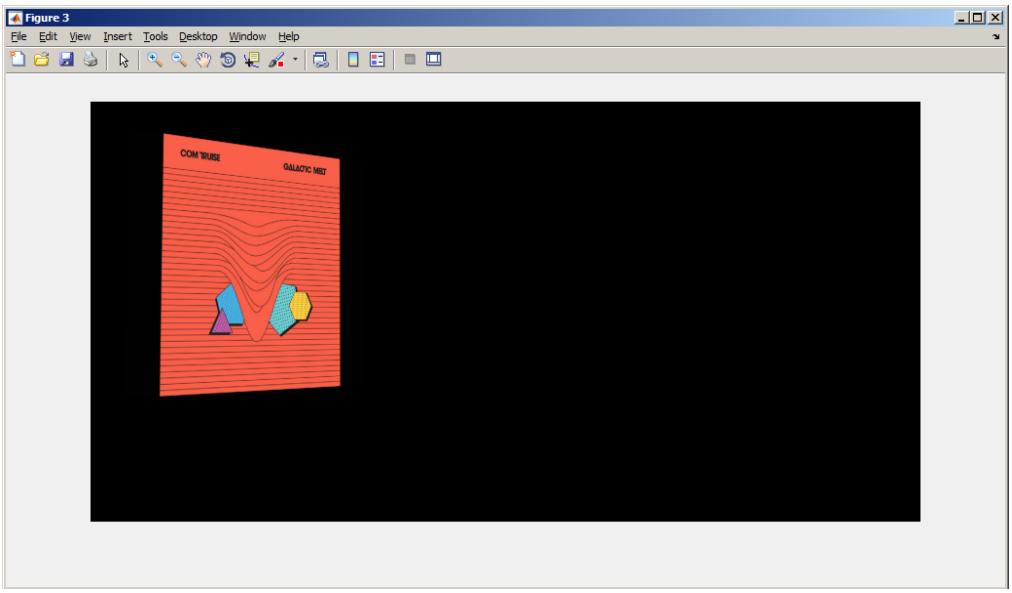
1. By setting up your own matrix of linear equations, then forming a projective transform in Matlab by calling the function projective2d.

2. By calling the Matlab function `estimateGeometricTransform` with 'projective' as the third argument.

If you're unsure how to do this, please consult today's lecture notes. Once you have the transform estimated, warp the album cover image J using this transformation, using the coordinate system of I as reference. This can be achieved as

```
RI = imref2d(size(I))
J = imwarp(J, tform, 'OutputView', RI);
```

Display the warped image. Mine looks like this:

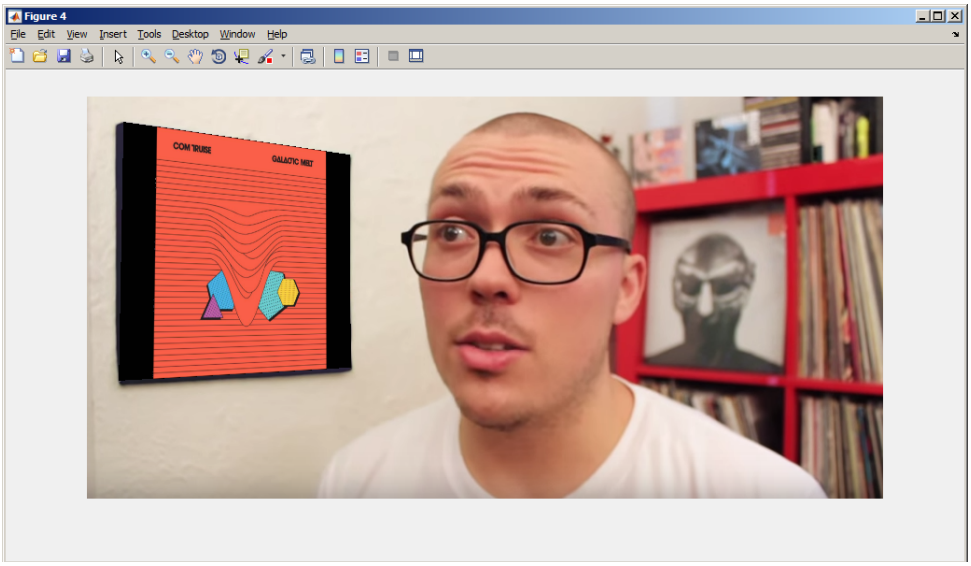


Finally, composite the warped album cover image with the original image. One way to do this is to loop over all the pixels in the warped image, and overwrite any pixels that are not perfectly black ([R, G, B] = [0, 0, 0]). Note: this assumes your album cover doesn't have perfectly black pixels in it. This can be achieved like this:

```
C = I;
[height, width, channels] = size(J);
for y=1:height
    for x =1:width
        if (J(y,x,1) ~= 0 && J(y,x,2) ~= 0 && J(y,x,3) ~= 0 )
            C(y,x,:)=J(y,x,:);
```

```
        end
    end
end
figure; imshow(C);
```

After doing this, your album cover should be augmented into the image! Mine looks like this:



Whilst my augmented image looks pretty good, there is a pixelated line at the top of the album cover, making the image look somewhat synthetically generated. How might this be improved? If instead of augmenting an album cover into an image, you augmented an album cover into a video, what challenges might you encounter?

Optional Task 6: Funhouse effect

In Task 3, the vignette computed a radius from the centre of the image. A polar representation can represent not only the radius, but an angle theta from the centre of the image. In Matlab, you can convert from Cartesian to polar coordinates using `cart2pol`, and convert from polar to Cartesian using `pol2cart`.

The code below transforms pixels into polar coordinates with a centre at the centre of the image, and then transforms them back. This has no effect on the image.

```
[height, width, channels] = size(I);
cx = width/2;
cy = height/2;
for x = 1:width
    for y = 1:height
        [theta, r] = cart2pol(x-cx, y-cy);
        [xx, yy] = pol2cart(theta, r);
```

```
            xx = round(xx) + cx;
            yy = round(yy) + cy;
            xx = max(min(xx, width), 1);
            yy = max(min(yy, height), 1);
            J(y,x,:) = I(yy,xx,:);
        end
    end
    figure;imshow(J);
```

How would you modify the code above (it only requires one line to be changed) so that the angle theta increases as a function of radius to achieve an effect like the funhouse effect below? *Hint: you may need to scale the radius by a factor like 1/100*.