Homework #2: Image Processing in MATLAB

Assigned: Wednesday, September 16
Due: Friday, September 25

For this assignment you will write several small programs in MATLAB as an introduction to both MATLAB and some simple image processing operations.

Implementation tip: it is often a good idea to initially convert integer pixel values after reading in an input image to floating point (using im2double) before performing any image operations on it, and then at the end convert the result back to integer values (using im2uint8) before saving.

A Word On Vectorization

P1 and P2 have each been designed to provide examples of *vectorized code*, in addition to performing an image processing task. Your job in these two problems is just to understand what the code is trying to do and fill in arguments to function calls. However, you should also spend some time to study how the provided code achieves vectorization.

As discussed in class, *vectorized* code does computation using matrices or vectors as the building blocks, as opposed to computation with individual elements as building blocks. Such code tends to not only run more quickly but is generally also more readable. You are highly encouraged to write vectorized code throughout this course, although this is not necessary to get a full grade.

P1. Pencil Sketch Effect

In this problem, you will complete the provided code which creates a pencil sketch interpretation of the input image. Like other effects found in image-editing and photo-sharing apps, this effect uses the *filtering* operation.

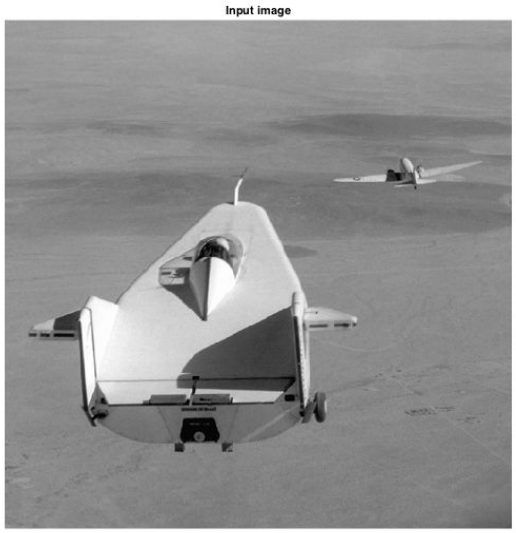
The pencil sketch effect can be achieved by the following algorithm:

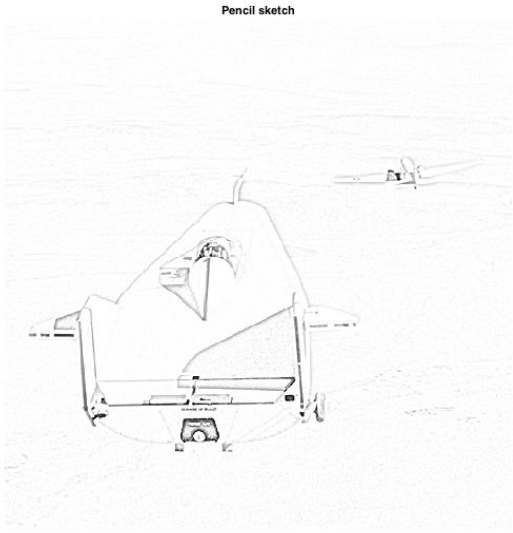
Given: A grayscale image I_{in} of intensity between 0 and 1.
Return: A grayscale image I_{out} of intensity between 0 and 1. I_{out} is a pencil sketch interpretation of I_{in}.

1. $I_{blur} \leftarrow blur(I_{in})$
2. For all x,y coordinates in I_{in}, do: $I_{out}(x,y) \leftarrow I_{in}(x,y) / I_{blur}(x,y)$
3. For all x,y coordinates in I_{out}, set $I_{out}(x,y) = 1$ if $I_{out}(x,y) > 1$.
4. Return I_{out}.

Step 1 can be done with the helper MATLAB function *imgaussfilt* but we use the more generic *imfilter* function in the provided code as it is useful for a wide variety of image processing tasks (see Problem 4).

The following is the display of the completed code processing another input image.





Replace all occurrences of *place_arg_here* in the code provided with the correct function arguments.

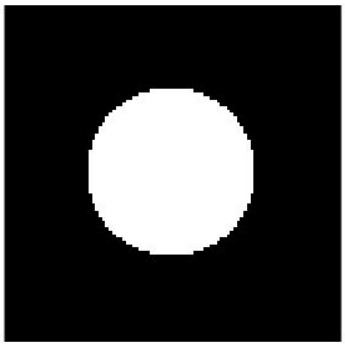
P2. Bug Eyes

In this problem, you will complete the provided code which creates a bug-eyed version of a given portrait image. You will learn about a spatial transformation technique known as *inverse mapping* and related MATLAB functions `meshgrid` and `interp2`. Understanding inverse mapping will come in handy in a future homework assignment.

Meshgrid

`meshgrid` is a commonly used function that defines the x-y coordinates of a rectangular grid. For instance, recall that the equation that defines a circle is given by $(x-x_c)^2+(y-y_c)^2=r^2$ where x and y define the circumference of the circle, x_c-y_c is the center of the circle and r is the radius of the circle. Knowing this, we can easily draw a filled circle using `meshgrid` like so:

```
height = 100;
[x, y] = meshgrid(1:height, 1:height);
im_circle = (x - height/2).^2 + (y - height/2).^2 < (height / 4)^2;
imshow(im_circle);
```



Inverse mapping

To spatially transform an image, we have to somehow express the relationship between the pixel locations of the input and output images. One approach is to answer the question, "For every pixel location in the *input* image, what is its corresponding location in the *output* image?" This approach is known as *forward mapping*. Forward mapping works well for simple transformations like translation, but often leaves "holes" in the output image under more complex transformations. Another approach is to answer the question, "For every pixel location in the *output* image, what is its corresponding location in the *input* image?" This is known as *inverse mapping*, which we'll use to achieve the bug eye effect.

Inverse mapping requires a function that maps a pixel location in the output image to a (not necessarily integer-valued) location in the input image. We will use the function

$$T(r_{out}) = s * r_{out}$$

Where:

- r_{out} is the distance between the output pixel location and the center of the eye
- $T(r_{out})$ a.k.a r_{in} is the distance between the input's pixel location and the center of the eye
- s is some value between 0 and 1 that scales r_{out}. r is short for 'radius'.

Intuitively, we want to enlarge the eye; therefore, we want to map r_{out} to some value **smaller than r_{out}**. This explains why s is smaller than or equal to 1. Furthermore, we want this bulging effect to be limited to the eye area. This requirement suggests that s should be a function of r_{out}, and more specifically, s should be at least close to 1 for large values of r_{out}. With these constraints in mind, we define s as

$$s(r_{out}) = 1 / [1 + exp(b(c - dr_{out}))]$$

where b, c, d are hand-tuned constants, and exp(x) is just a more readable equivalent of e^x. This particular choice of s is somewhat arbitrary but suffice to say, it meets the needs of our problem. Other choices of s would be similarly effective.

Bug eye algorithm

With these functions defined, the bug eye effect can then be done as follows:

Given: An image I_{in} of a person and x-y coordinates x_c-y_c of the center of an eye
Return: An image I_{out} of the person with a bug-like eye. I_{in} and I_{out} are of the same size.

1. Define the x-y pixel coordinates x_{out}-y_{out} of the output image using `meshgrid`
2. Compute the x-y pixel coordinates x_{in}-y_{in} of the input image for each x_{out}-y_{out} using transformation function `T`.
3. Compute the intensities of I_{in} at locations x_{in}-y_{in}, using `interp2`, which returns I_{out}.
4. Return I_{out}.

Replace all occurrences of *place_arg_here* in the code provided with the correct function arguments. You may find it helpful to read the MATLAB documentation for `meshgrid` and `interp2`. In this problem, all these arguments are just variable names, e.g, `x_out`.

*A possible extension: In this problem, we've made just one eye bug-like for the sake of simplicity. It isn't difficult to extend the algorithm to make it work for two bug eyes. Just repeat step 2 for the other eye and average the x_{in} and y_{in} coordinates. You'll get an effect like the following. This extension is **not** required for this assignment.*

Another possible extension: Notice that in the skeleton code we've hardcoded the locations of the eyes and the parameters to the inverse mapping function. Alternatively, we could automate these steps with an algorithm to detect the location and possibly the size of the eyes in the input image.

A note on Cartesian coordinate conventions: It's useful to be aware of two somewhat conflicting conventions when writing image processing scripts in MATLAB. On the one hand, the first and second dimensions of a MATLAB matrix are the vertical and horizontal axes respectively; on the other hand, when we say "x-y coordinates" in image processing parlance, the x and y almost always refer to the coordinates in the horizontal and vertical axes respectively. For example, the intensity at the x-y coordinates (2, 3) of a MATLAB matrix `im` is `im(3, 2)` rather than `im(2, 3)`. As another example, the MATLAB documentation of `meshgrid` refers to output variables `[X, Y]`, where the first and second output variables vary in the second (i.e., horizontal) and first (i.e., vertical) dimensions respectively. Keep these conventions in mind.

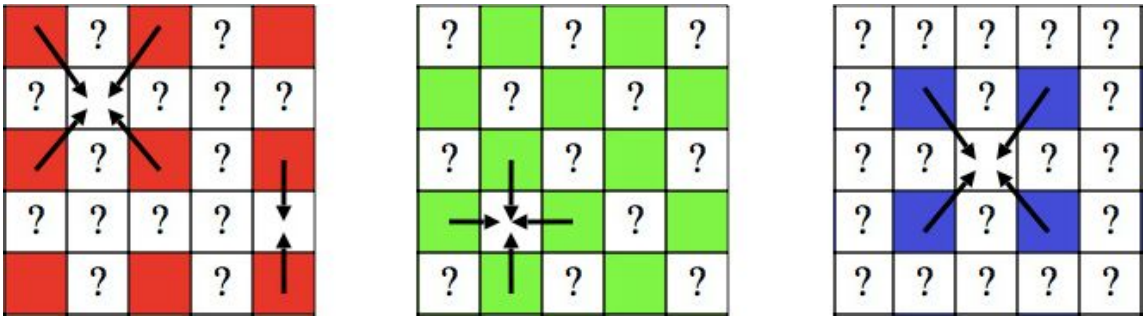
P3. Histogram Equalization

Histogram equalization is a commonly used image operator for enhancing the contrast in an image. To learn about it, read Section 3.1.4 in the Szeliski book and Wikipedia at http://en.wikipedia.org/wiki/Histogram_equalization Next, implement in MATLAB a function that performs histogram equalization by (1) converting an input color image from RGB to HSV color space (using `rgb2hsv`), (2) computing the histogram and cumulative histogram of the *V* (luminance) image only, (3) transform the intensity values in *V* to occupy the full range 0..255 in a new image *W* so that the histogram of *W* is roughly "flat," and (4) combining the original H and S channels with the *W* image to create a new color image, which is then converted to an RGB color output image (using `hsv2rgb`). Information about HSV color space is given in Section 2.3.2 in the Szeliski book. The calling form should be `function J = myhisteq(I)` where the function takes a color image array `I` as input, and outputs a new color image array `J` after equalization.

Your `main_P3.m` script file should read an input image file, call `myhisteq`, and write the output image as a `.jpg` file. So, it should look like: `clear; img = imread('P3-bridge.jpg'); out = myhisteq(img); imwrite(out, 'P3-bridge-out.jpg')` You can hard-code the names of each input image and output image file pair you use. Also create images of the histograms of *V* and *W* (using `imhist` applied to *V* and *W,* not the histogram arrays you generate). Do *not* use `histeq` or `hist`. Turn in `jpg` images for each output image, and its two histograms. Step 3 should be implemented as $W(i,j)$ = max(0, ((256/total number of pixels in *V*) * $c(V(i,j))$ -1) where cumulative histogram *c* is computed by $c(z) = \sum_{j=0..z} h(j)$, and histogram *h* is computed by $h(z)$ = number of pixels in *V* with intensity *z,* where *z* is an integer in the range 0 to 255. Note: To compute *c* and *h, V must* contain discrete, integer values in the range 0… 255; use the function `uint8` to convert *V*. To do step 4, you can simply do: `img(:,:,3) = W; J = hsv2rgb(img)` Submit your output RGB images and histogram images. See hand-in instructions for naming conventions.

P4. Demosaicing

Digital cameras that contain a single image sensor capture a color image by overlaying a color filter array in front of the image sensor's pixels. The color filter array is known as a Bayer pattern that contains red, green and blue filters arranged in 2 x 2 blocks as [R G; G B] starting from the upper-left corner of the image. So, the upper-left corner pixel in an image at coordinates (1,1) is assumed to be a red pixel. The process of converting a raw image into a full color image consisting of three channels, one for red, green and blue, is called **demosaicing**. Read Section 10.3.1 in Szeliski for a brief description. Implement a simple *linear interpolation* method for demosaicing, defined as follows: for each pixel in each color channel, *fill in the missing values* by averaging either the four or the two nearest neighbors' values:



The output image should be the same size as the input image but with three channels instead of one. Implement this as `function J = mydemosaic(I)` where `I` is an input mosaic image (input as a `.bmp` image file) and `J` is an RGB output image. Avoid using loops if possible. Instead use `imfilter`. (You may find it easiest to use several "filters," one, for example, for averaging the two horizontally-adjacent pixels in the G channel image at each missing pixel's coordinates, and another one for averaging the two vertically-adjacent pixels in the G channel image at each missing pixel's coordinates; then combine the two results.) You may *not* use `interp2` or `demosaic`. Your `main_P4.m` file should read an image file, call `mydemosaic`, and write the output file as a `.jpg` image file, preferably without any user interaction by hardcoding the file names into `main_P4.m` For example, do something like: `img1 = imread('P4-union-raw.bmp'); J1 = mydemosaic(img1); imwrite(J1, 'P4-union-demosaic.jpg', 'jpg');`

We will provide each test image in both JPEG and Raw formats. The Raw image is stored as a `.bmp` format image file (containing a single 2D array) so that you can read the original 2D sensor data using `imread`. The provided JPEG image contains the RGB image produced in the camera by the vendor's demosaicing software so that you can compare your results with theirs.

To evaluate each result image, create an "error" image by computing at *each* pixel the squared difference between the provided `.jpg` image and the demosaiced image you produced for *each* color channel separately, and then adding the three numbers together to obtain a value for that pixel in the "error" image (stored as a grayscale, not color image). Display it using `imshow` and scale the output values so that the maximum value is 255 (i.e., white). Most of this image will be black meaning there is little or no error at those pixels. Find a region in *one of your test images* where artifacts of demosaicing are visible, and crop that region out manually into a new "artifact" image. Save that small image and give a brief explanation (in a `README.txt` file) of the likely cause of this artifact. Hand in the two result images, two error images and *one* artifact image (corresponding to a region in *either one* of the two test images). See hand-in instructions for naming convention.

Note: Your demosaicing function should accept images with different pixel value scales. The output image should either be a [0,1] `double` image or a [0,255] `uint8` image.

EXTRA CREDIT: Bill Freeman proposed an improvement to the simple bilinear interpolation approach. Since the G channel is sampled at a higher rate than the R and B channels, one would expect interpolation to work better for G values. Then it would make sense to use the interpolated G channel to modify the interpolated R and B channels. The improved algorithm begins with linear interpolation applied separately to each channel, just as you have already

done above. The estimated G channel is not changed, but R and B channels are modified as follows. First, compute the difference images R-G and B-G between the respective interpolated channels. Mosaicing artifacts tend to show up as small "splotches" in these images. To eliminate the "splotches", apply *median filtering* (use the `medfilt2` command in MATLAB) to the R-G and B-G images. Finally, create the modified R and B channels by adding the G channel to the respective difference images. Implement the demosaicing part of this algorithm using `function J = FreemanDemosaic(I)`, where `I` is the Bayer Pattern image, `J` is an RGB image.

P5. Color Transfer

Color correction is a common image processing operation. One form of this is to modify the colors of one image based on the colors in a second image. This "color transfer" process is described in the paper "Color Transfer between Images" by E. Reinhard *et al*., which is available in the course Readings (focus on the section "Statistics and color correction"). Implement the basic algorithm described there (i.e., just use the mean and standard deviation of all pixels in the image, and don't do gamma correction). Useful MATLAB functions include `mean2` and `std2`. Convert images between RGB and L*a*b* color spaces using the MATLAB functions `rgb2lab` and `lab2rgb`. Write a `function K = mycolortransfer(I, J)` to implement the algorithm, where `I` is the RGB input source image (i.e., the one you want to change), `J` is the palette (target) input image (i.e., the image you want to steal the colors from), and `K` is the output RGB image. `K` should either be a [0,1] `double` image or a [0,255] `uint8` image. Write `main_P5.m` to read two input image files, call `mycolortransfer`, and write the output image as an RGB `.jpg` file. To compute the ratio of standard deviations, use something like: `L_out = (std-dev_L_target / std-dev_L_source)(L_source – mean_L_source) + mean_L_target` where `L_source` is the L channel after converting the source input image to Lab color space, `std-dev_L_source` is the standard deviation of all the values in the 2D matrix `L_source`, and `L_out` is the L channel for the output image. Similarly, compute `A_out` and `B_out`. Then combine these three into a 3D matrix and convert back to RGB. Run your code on the provided test image pair called `P5-source.jpg` and `P5-target.jpg` to create an output image. As a second test, *find your own pair of source and target images* and create their output image. Hand in these three images as well as the output of the test image pair.

Hand-In Instructions

Your submission should have the following directory structure:

- `P1`
  - `main_P1.m`
- `P2`
  - `main_P2.m`
- `P3`
  - `main_P3.m`
  - `myhisteq.m`
  - `P3-bridge-out.jpg`
  - `P3-bridge-Vhist.jpg`
  - `P3-bridge-Whist.jpg`
  - `P3-snow-out.jpg`
  - `P3-snow-Vhist.jpg`
  - `P3-snow-Whist.jpg`

- `P4`
  - `main_P4.m`
  - `mydemosaic.m`
  - `P4-crayons-demosaic.jpg`
  - `P4-union-demosaic.jpg`
  - `P4-crayons-error.jpg`
  - `P4-union-error.jpg`
  - `P4-artifact.jpg`
  - `README.txt`
- `P5`
  - `main_P5.m`
  - `mycolortransfer.m`
  - `P5-out.jpg`
  - `P5-mysource.jpg`
  - `P5-mytarget.jpg`
  - `P5-myout.jpg`
- Any additional scripts or functions that you wrote, placed in the appropriate folder. Files that belong to the extra credit task can be placed in a `P3_extra_credit` folder at the same directory level as the rest of the folders.

Zip all the above into a single zip file called `<your wisc username>-HW2.zip` and submit this *one* file to Moodle.

You do not have to submit `transformRadially.m` or the test images that we provided.

Acknowledgements

pencil sketch from book
bug eye from website