Homework 7

[ 100 points - due by 11:59 pm, Sunday, April 2, 2017 ] *("next" Sunday relative to writing this…)*

Submit these files to the CS submission system at the usual place by 11:59. You may work on your own or with 1-2 partners on the programming portions of this assignment. (The reading/response is individual only.) Groups larger than 3, please split into smaller groups! Remember that partners need to work in the same physical location, share composition time equally (or each compose on their own machines) and be fully equal owners and producers of their work. *Have fun treeing (and foresting)!*  [cs35 homepage](https://www.cs.hmc.edu/~dodds/cs35/)

**Downloads**

There's one (zipped) starter file to download -- grab it at the start of class & follow along:

* [The zip file, hw7.zip, to start this week's problems…](https://drive.google.com/open?id=0BwPWh-3AmiLxbm50MVBweWtEbms)

**Submission**

**Overview** Please submit an archive named **hw7.zip** with the starter-files' filenames:

**hw7pr1.py** [**lab** problem] pixels, OpenCV, and image transforms...

**hw7pr2.py** Steganography: Embedding and extracting secret messages in images

**hw7pr3.py** green-screening! Imagine yourself in a far-of Starbucks… (!!!)

**hw7pr4.py** photoshopping text...meme alert!

As usual, submit your reading response in its own spot at the [submission site](http://cicero.cs.hmc.edu/).

As always, extra-credit is available for posting code and a write-up of any one of these problems to your GitHub repository (be sure to let us know you've done this -- and provide a direct link)

**Problem 0**: What computer vision can (and can't) do… ?! [5 pts]

This homework's problem 0 asks you to read a NYT article entitled [*A lesson of Tesla crashes? Computer vision can't do it all yet.*](https://www.nytimes.com/2016/09/20/science/computer-vision-tesla-driverless-cars.html?_r=0)  Then (or during…) head over to the [imSitu.org demo site](http://imsitu.org/demo/) and try out the image-recognition it provides -- both on a couple of images that it suggests and on at least one image of your own. Finally, share in your reflection (a) how the imSitu.org system did for your image -- what did it get right/wrong? And, (b) reflect on the article's claim that visual understanding requires an underlying model (physics or context) beyond training examples (lots and lots of pixels…!) Do you agree? Disagree? Or have another "view" of the computer vision situation altogether? As with each week's reading, responses should carefully considered, but need not be very long: a 4-5 sentence paragraph is wonderful.

**[Lab problem 1] Problem 1: Pixels, OpenCV, DIY image transforms...**

[30 pts; setup, and trying out OpenCV]

* This problem asks you to run/alter the code in the **hw7pr1.py** file.
* **Installing OpenCV** ! Here are instructions that have worked today, 3/27, for Mac and for Windows:
  + [**Mac instructions**] At the command prompt (not within python), run
  + conda install python=3.5
  + conda install -c menpo opencv3
  + conda update hdf5
  + [**Windows instructions**] At the command prompt (not within python), run
  + conda install python=3.5
  + conda install -c menpo opencv3
* To check if OpenCV is working, run **ipython** and then type **import cv2**
* First, make sure you're in the hw7 folder (so that you have direct access to the images in that folder) and then try a few things at the Python command-line. Perhaps copy-and-paste each one to get a feel for the interactions:
  + %matplotlib
  + import cv2
  + from matplotlib import pyplot as plt
  + raw\_image = cv2.imread('messi5.jpg',cv2.IMREAD\_COLOR)
  + image = cv2.cvtColor(raw\_image, cv2.COLOR\_BGR2RGB)
  + plt.imshow(image)
  + # Yay!
* The examples of image transformations we saw on Monday are often called "filters": functions that take in an image and transform its underlying data (pixels) in some way, returning a new image. The starter file has several such examples, including the ones from class. ***First, try out the functions in the file - and compare their behavior with the code… !***
* **Part A.** This problem asks you to design a filter that takes in a single image, transforms its pixels in an unusual way (something not in Photoshop's menu options, perhaps?), and then runs that filter on at least two of your own images, as well as on the flag and spam images for comparison. The signature of the function would be
  + **def new\_filter( image\_name ):**
* **Part B.** This part extends the previous by asking for a filter that takes in ***two*** input images and outputs a novel image that incorporates information from the pixels of each of the inputs. Ideally, your output will retain noticeable features from each of the inputs, creating a composite image of some sort. Unusual composites/creativity welcome, for sure! Its signature should be
  + **def two\_image\_filter( image1\_name, image2\_name ):**
* Helpful bits for the two-image-filter:
  + You can find the "shape" of an image with **im1.shape** and/or **im2.shape**
    - Here, I'm imagining that **im1** and **im2** are the image objects, not filenames!
  + **min(im1.shape[0], im2.shape[0])** is then the smaller numbers of rows, ...
  + Create a BLANK image **im** with **im = np.zeros((7,42,3), dtype=int)**
  + An image is simply a numpy array! The example **im** above has 7 rows, 42 columns, 3 color channels, and pixels that are integers (change as you'd like!)
* For each of your Part A and Part B filters, be sure to include examples of outputs for on at least *four* images: (1) the flag image, (2) the spam image (both included) and (3)-(4) two new images that you include in your submitted zip archive.
* Also, include in a triple-quoted string at the bottom of your file a comment that briefly explains the motivation and what each of your two filters does (a high-level, big-picture view, since the code itself describes the low-level details :-)

**[Problem 2] Problem 2: Steganography!**

[35 pts; showing the embedding/extracting of hidden messages within images...]

* This problem asks you to run/write code in the file **hw7pr2.py**
* [**Overall goals/tasks**] Steganography is the practice of embedding hidden messages or codes within another message - or image, in our case. [Here's Wikipedia.](https://en.wikipedia.org/wiki/Steganography) This problem asks you to write two functions: a string-hiding function that "deals out" the bits of the string's characters to each pixel in the image and a string-extracting function that takes an image and returns the string (if any) it hides.
* **Part A.** This part asks you to extract a message that has been steganographically embedded into the lowest order of its bits. (This is the opposite of the function of Part B.) Specifically, you'll create
  + **def desteg\_string( image\_name ):**
  + which should take in a steganographized image named **image\_name**. Then, **desteg\_string** should go through each pixel, one-by-one and extract the lowest-order bit from its channels. Use RGB order: first the red channel, then the green channel, then the blue channel.
  + **bin( val )** is Python's built-in function for converting into binary -- try it! You'll see that bin(42) yields "0b101010" Remember that you can strip the "0b" in the front with a slice!
  + When extracted and placed into a single string (or list), all of the bits will (or should) be in eight-bit blocks. The final block -- indicating the end of the message -- is eight 0 bits in a row. (Since this is not a printable ASCII character, there's no source of confusion here.)
    - Continuing our previous example, when you slice bin(42), you get "101010" You'd then want to find the length of that string and subtract it from 8: that will be the number of extra zeros to include in front, so that it's a total of 8 bits!
  + Here's an example: if your 8-bit message was the bits **00101010**  (followed by the end-of-message marker **00000000**), then you would be able to find those bits in this order:
    - The red channel of image[0,0] would end in the bit **0** (the initial bit)
    - The green channel of image[0,0] would end in the bit **0** (the next bit)
    - The blue channel of image[0,0] would end in the bit **1** (the next bit)
    - The red channel of image [0,1] would end in the bit **0** (the next bit)
    - The green channel of image [0,1] would end in the bit **1** (the next bit)
    - The blue channel of image [0,1] would end in the bit **0** (the next bit)
    - The red channel of image [0,2] would end in the bit **1** (the next bit)
    - The green channel of image [0,2] would end in the bit **0** (the next bit)
    - Then, the blue channel of image[0,2] would end in **0** (the first of 8 **0**'s)
    - … and then it would continue in this way for seven more **0** bits, since the message needs to end in 8 zero bits.
  + We will only use color images for this!e previous process and return the message embedded in the image. Remember that the first occurrence of 00000000 (eight zero bits, correctly aligned with the eight-bit boundaries) is the end of the message.
  + Also, to test your desteg\_string, here are links to two small images:
  + these are not steganographized (???)
    - <https://drive.google.com/open?id=0BwPWh-3AmiLxMUtTR3V4QXlubTA>
    - <https://drive.google.com/open?id=0BwPWh-3AmiLxaHNUaTlBUjVQNTA>
  + Each of these has the embedded message "Wow! This worked!" in the style of this problem.
  + The second one does require conversion before decoding. The first one does not require conversion (yes, the names may be backwards!) Here is the code with which to test your desteg\_string function once these two images are in your folder:
    - # test with the first image (no need to convert)  
      im\_rgb2 = cv2.imread('small\_flag\_with\_message\_rgb.png',cv2.IMREAD\_COLOR)  
      desteg\_string( im\_rgb2 )  
      message = desteg\_string( im\_rgb2 )  
      print("message is ", message)
    - # test with the second image  
      im\_bgr = cv2.imread('small\_flag\_with\_message\_bgr.png',cv2.IMREAD\_COLOR)  
      im\_rgb = cv2.cvtColor(im\_bgr, cv2.COLOR\_BGR2RGB)  
      message = desteg\_string( im\_rgb )  
      print("message is ", message)
  + **Hints**: Remember that **chr(n)** returns the character whose ASCII value is **n**. Also, Python has a built-in base converter. Here's the key example:
    - **int( "00101010", 2 )** returns 42
  + Again, in our implementation, this function was helped greatly by building some helper functions (not as many as the next one, however!)
* **Part B.** This part asks you to write
  + **def steganographize( image\_name, message ),** which takes in an image and a message (a string). It should return a copy of the image, but with the *least-significant* bits of some/all of its pixels changed to hold the message, one bit at a time!
  + The output image can be returned (for further processing), but it should be written out to a file with **\_out** appended to the end of the filename (but before the ile extension). So, an input image **spam.png** would lead to the creation of an output image **spam\_out.png** The image will look identical, but the message will be present, decodable with Part A's function.
  + When you're saving your message-encoded images, BE SURE TO SAVE AS PNG format! The reason is that .png files are lossless (your message will be preserved). On the other hand, .jpg files are lossy, and your message will be clobbered! Here's how to write out an image: **cv2.imwrite( "output\_image\_name.png", image\_object\_to\_write )**
  + For **steganographize** there are many details to standardize! Here are those details:
  + We will assume the string message consists of printable ASCII characters
  + Each character in message should be converted to an 8-bit binary representation
    - Note that Python has this built it: run **bin(42)** for an example
    - You'll need to strip the initial "0b" and add initial "0"s (zeros) to ensure 8 bits!
  + The message should end with eight zero bits: 00000000 Since this is not a *printable* ASCII character, there will be no ambiguity here.
  + The bits should be "dealt" in RGB order, one bit per color
  + Thus, there will be 3 bits changed (or possibly changed) per pixel,
  + each in the *least-significant* location (last bit):
  + For instance, if you wanted to include only the bits **00101010**
  + The red channel of image[0,0] would end in the bit **0** (the initial bit)
  + The green channel of image[0,0] would end in the bit **0** (the next bit)
  + The blue channel of image[0,0] would end in the bit **1** (the next bit)
  + The red channel of image [0,1] would end in the bit **0** (the next bit)
  + The green channel of image [0,1] would end in the bit **1** (the next bit)
  + The blue channel of image [0,1] would end in the bit **0** (the next bit)
  + The red channel of image [0,2] would end in the bit **1** (the next bit)
  + The green channel of image [0,2] would end in the bit **0** (the next bit)
  + Then, the blue channel of image[0,2] would end in **0** (the first of 8 **0**'s)
  + … since the message needs to end in 8 zero bits.
  + We will only use color images for this!
* The output should be the transformed image. Be sure to write it to a file with the addition **\_out** placed after the filename and before the file extension.
* **Hint**: (In our implementation, this function required a lot of helper functions - you may design these as you see fit -- but don't try to include too much functionality in a single function!)
* **Images to include (using your encoder/decoder)**: Once these functions are written, you should include
  + **(1)** at least one image with a hidden message for the graders to find. And, the way they should find it is -- using your decoder function.
  + Be sure to have a comment describing and showing which image it is (extra kudos if the image somehow foreshadows the message!)
  + **(2)** Also: ***challenge your grutors*** Find the text of some relatively large work (Romeo and Juliet, Harry Potter #4, etc.), and find a picture that is large enough so that you can embed the entire plain-text version of the work in the picture. **Include in your bottom-of-file comment what picture you chose** - and be sure to include the picture in your zip file.
    - But -- ***don't*** tell the grutors which text it was… . We will ask the grutors to use your decoder to find the text and then see if they recognize it. The objective here is to provide a stark contrast between the amount of data in text and the amount of data in images!

For fun, feel free to swap images with another person/group and check if your decoding works on theirs (report those results, if you have a chance to do this!)

**Extra Credit!** For up to +5 points of extra-credit, instead of embedding and extracting a string message, write an additional two functions that embed and extract at least a portion of another image! (That [Wikipedia page](https://en.wikipedia.org/wiki/Steganography) has a nice example partway down). part asks you to undo the function of Part A. Specifically, create

**def steganographize\_image( orig\_image, image\_message ):**

**def desteg\_image( image\_with\_message ):**

which should hide **image\_message** in **image** and then return it, respectively. There are many details here, which are up to you to decide how to handle:

* Where the image\_message "goes"
* How much of the image\_message gets represented
  + You could make it a b/w or a three-bit grayscale image or even use color...
* Whether, beforehand, you scale image\_message or not

If you try this part, be sure to include in your file comment how the grutors should run it!!

**Problem 3: Green-screening!**

[30 pts; EC for anything above and beyond...]

* This problem asks you to run/write code in the file **hw7pr3.py**
* Chromakeying or green-screening is a commonly-used technique for superimposing a foreground subject atop an artificial background - and is one reason why TV weather personalities so rarely wear lime green… . [Here's Wikipedia's take.](https://en.wikipedia.org/wiki/Chroma_key)
* Here, we ask you to write a general-purpose green-screening function (as well as any helper functions). The main function should be named
  + **green\_screen( orig\_image, new\_bg\_image, corner=(0,0) )**
* This **green\_screen** function should ignore the "green" -- and you will have to create a definition of green, probably factored out into a helper function -- from the **orig\_image** image. For all of the pixels in **orig\_image** that are NOT green, it should place them on top of **new\_bg\_image**.
* Thus, the foreground objects (people, let's say) from **orig\_image** should be superimposed on the background of **new\_bg\_image**.The input named **corner**=(0,0) (with a default value of (0,0)) should indicate where the upper-left corner of **orig\_image** should go within **new\_bg\_image**. Using the default aligns both images' two upper-left-hand corners.
* If you didn't get a green-screen image of yourself, there are, for sure, many you might choose from instead online. Here is one with a green-screened bunny:
  + <https://cdn.shutterstock.com/shutterstock/videos/7835551/thumb/1.jpg>

This function should return and write out to file the new image, named **green\_screen1.png** (or with other trailing numbers/extensions). You will likely need to write one or more helper functions: the design of those is up to you -- but please do include docstrings explaining what each one is!

**Comment and reflection** You should include a short triple-quoted comment at the bottom of hw7pr3.py that reflects on how you implemented your **green\_screen** function. In addition, you should include at least two examples -- using your own images -- of the green-screening in action! Creative backgrounds are especially welcome!

**Extra Credit!** The last piece of this problem (for extra credit of up to +5 points) is to add a steganographic capability to your **green\_screen** function (or, feel free to create a similar function with a slightly different name). The idea is that the **green\_screen** uses steganography to "mark" all of the foreground pixels with a **0** in their least-significant bits and mark all of the background pixels with a **1** in their least-significant bits. Then, write

**def de\_green( green\_screened\_image ):**

which takes in one of these steganographically green-screened images and returns/writes-out either the background (with the foreground pixels removed, e.g., made all one color) or the foreground (with the background pixels removed), or both.

**Problem 4: Rewriting Text!**

[30 pts; EC for anything above and beyond...]

* This problem asks you to run/write code in the file **hw7pr4.py**
* Let's use our pixel processing to rewrite the road sign in share\_road.png.
* Write a function called **rewrite(image)**that takes in share\_road.png and manipulates the text to say something else (a simple, rather amusing example is "SHARE THE TOAD")
* You must rearrange the letters already in the sign to rewrite the text
* In order to replace one letter with another, crop the letter you want and insert it, pixel by pixel, into the position of the letter you want to replace.

This function should return and write out to file the new image, named **share\_road\_rewritten.png**. You are free to write one or more helper functions: the design of those is up to you -- but please do include docstrings explaining what each one is!

**Extra Credit!** Extra credit is available for particularly clever or funny phrases that involve a great deal of manipulation. Remember, all the letters in the sign are at your disposal!

*Good luck with pixel processing, everyone!*

And, as always, the ...

**Eternal Extra-credit: Showing off your results…**

[up to +5 pts extra-credit...]

* As with each week, you're invited to include both your source code and a short write-up of one of the week's problems within your GitHub repo(s). Images and other visuals, of course, are welcome. If you do this, let us know (and provide a direct link :-)