Homework 2 - NumPy for Image Manipulation

(Total: 30 points)

In this exercise, you will practice your NumPy skills, and along the way you will also learn Matplotlib – which is Python's plotting library. This exercise will therefore also serve as a short intro to Matplotlib.

There are several uses for high-dimensional arrays in data analysis. They can be used to store:

- matrices, solve systems of linear equations, find eigenvalues/vectors, find matrix decompositions, and solve other problems familiar from linear algebra
- multi-dimensional measurement data. For example, an element an 2D array a[i,j] might store the temperature t_{ij}, where i might be location and j is date
- images and videos can be represented as NumPy arrays:
    - a gray-scale image can be represented as a 2D array
    - a color-image can be represented as a 3D image, the third dimension contains the color components red, green, and blue
    - a color video can be represented as a 4D array

In this exercise we will work with color image data, and use NumPy arrays and functions to produce manipulated image data.

Why would we want to do this?

There are many purposes for applying image analysis techniques to solve real-world scientific and business problems, including extracting information from images, identifying the location and type of objects – known as segmentation, and classifying objects in an image, to give some example.

One Machine Learning (ML) application of image manipulation is for **data augmentation –** a pre-processing technique. It is a technique of altering the existing data to create some more data for the model training process. In other words, it is the process of artificially expanding the available dataset for training a deep learning model in order to improve the performance and ability of the model to generalize.

However, it is not always possible for us to obtain more training data (it could be very time-consuming or very expensive), and yet we know complex ML models will perform better with more data. Training with small data can lead to overfitting and subsequently poor results on test data, and therefore low generalizability. *[For instance, if you, as a human, only see three images of people who are lumberjacks, and three, images of people who are sailors, and among them only one lumberjack wears a cap, you might start thinking that wearing a cap is a sign of being a lumberjack as opposed to a sailor. You would then make a pretty lousy lumberjack/sailor classifier].*

Being able to make the most out of very little data is a key skill of a competent data scientist. The data augmentation technique has is aplied as a first step in pre-processing to produce augmented images from the original dataset. Of course, to produce good external generalization of our machine or deep learning task, we need to apply other techniques as well during training. But for now, we will only focus on data augmentation.

Caveat: Of course, nothing beats having more data to train with (in supervised classification problems, such labeled training data is also referred to as the ground truth, gold standard etc), but augmentation can be a second-best solution. If your dataset is really small you should first contemplate on the type of model you are going to use. It is meaningless to apply classical heavy neural network to the few samples as its optimizer was not designed for low data setup. In such cases, it is recommended to try simpler ML models or use neural network approaches designed specifically to solve low-data problems (zero-shot, one-shot, few-shot learning and meta learning models).

Let's carry on to see how we can perform data augmentation using NumPy and Matplotlib.

A. What is an image made of?

Images that we see are stored in the computer as numeric abstractions. Before working with images, let's first see how these abstractions work.

Take a 2-D image. While there are many formats, I will only discuss a jpeg/jpg or png file below. Each component in an image – i.e. a pixel - can be represented by 3 dimensions of data:

- the position on the row axis,
- the position on the column axis and
- the colors (described using 3 channels – RGB).

The color is described by a combination of 3 channel values –levels of red, green, blue, or RGB. A value of 0 is absence of color, whereas a maximum color value of (1.0 or 255) represents the highest level of intensity. If all 3 RGB are 0.0, we get black; if all 3 RGB values are 1.0, we get white. Sometimes images contain a fourth piece of data on the 3rd channel – the opacity or depth of color. The fourth piece of information, if present, is known as an *alpha channel* and enables the use of partial transparency by specifying a level of opacity. We will refer to an RGB image with an alpha channel as an RGBA image. An alpha value of 0 is fully transparent, and the maximum value for the pixel depth (1.0 or 255) is completely opaque.

PNG images use these 4 pieces of data - RGBA, whereas JPG format uses the first 3- RGB - to represent the 3rd dimension or channel.

Each level of color is either a float (0.0 to 1.0) for PNG images, or a 1-byte unsigned integer for JPG images. The total number of values that can be represented in 1 byte is 2^8 - 1 or 255 (since we start counting from 0).

Now let's think about how we can represent an image in a programming language such as Python. In NumPy, this is just a 3-D ndarray made up of 3 dimensions: height, width and channel. The third dimension provides the color (and opacity) of each pixel as a (3,) vector for JPG or (4,) vector for PNG. Manipulating an image now is now a matter of using NumPy functions to manipulate the ndarray.

Today many image packages are available that are built on top of NumPy (uses NumPy arrays to store data), including Scikit-image (skimage.io), OpenCV, ImageIO among others. Another one - PIL/Pillow uses its own proprietary image format and doesn't use Numpy arrays. While these packages provide built-in functions to manipulate and process images, in this exercise we will only use NumPy to write the code from scratch to help us practice our NumPy (array manipulation) skills. So do not use the built-in functions (unless indicated).

An example to understand some indexing basics

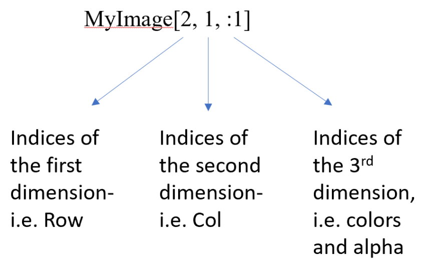
Let's use an example of a 3X3 PNG image called ***MyImage***– each pixel is a cell in the table below.

MyImage	Col=0	Col=1	Col =2
Row=0	[R1, G1, B1, A1]	[R2, G2, B2, A2]	[R3, G3, B3, A3]
Row=1	[R4, G4, B4, A4]	[R5, G5, B5, A5]	[R6, G6, B6, A6]
Row=2	[R7, G7, B7, A7]	[R8, G8, B8, A8]	[R9, G9, B9, A9]

row is the 1st axis, col is the 2nd axis and colors/alpha form the 3rd axis. Dimensions of this array are (3,3,4). Each pixel is accessed as [row, col], which will give us 4 values [R, G, B, A] for that pixel.

To access specific color values within each pixel's third axis, we can use slicing or fancy indexing.

To access the red value in [R8, G8, B8, A8] we can type: `MyImage[2,1,:1]` or `MyImage[2,1,0]` or `MyImage[2,1,[0]]`


MyImage[2, 1, :1]
Indices of the first dimension- i.e. Row
Indices of the second dimension- i.e. Col
Indices of the 3rd dimension, i.e. colors and alpha

We want just one value (row) in the first dimension, so we write 2. We want just one value (column) in the second dimension, so we write 1. We want just one value (red color) in the third. The red-value is in the position of index 0 in the third dimension. We can refer to that as `0:1:1` or `0:1` or `:1` or just `0` or `[0]`. The first three are using slicing and the last is using fancy indexing.

To access both the green and blue values in [R1, G1, B1, A1] we can type `MyImage[0,0,1:3]` or `MyImage[0,0,[1,2]]`

Okay, now on to our homework tasks.

B. Image Manipulation Tasks

In this section, you will write code corresponding to each task in a Python Notebook named as **Hw2_yourname.ipynb**.

Task 0. Housekeeping (0.5 point)

Make sure to enter your name at the top of the notebook using Markdown as follows.

Completed by: <your name>

Task 1. Import two libraries

Matplotlib is a popular library in Python for plotting and graphing purposes. Within the Matplotlib module we will specifically use a sub-module called pyplot and import it as plt – that's what we will call it within our program. It is custom to import and name it this way. The other library we need is NumPy.

```
import matplotlib.pyplot as plt
import numpy as np
```

To display the plots, matplotlib uses a backend. There are several choices, but we can either use *inline* (non-interactive but embedded in the notebook) or *notebook* (interactive but not embedded in the notebook) . This is set using a magic function.

```
%matplotlib inline
```

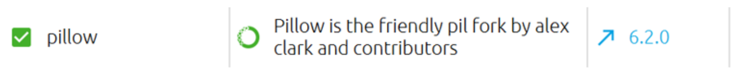
Task 2a. Read in an image as both PNG and JPG

You can use any colored image that you want, and save it in both formats. It is important to do two things so that we will be able to run your code and see outputs when we grade.

a) Please name the images as `img1.png` and `img1.jpg`. I have provided two sample images for you to download and use if you like.
b) Please store this image file ***in the same folder as your ipynb notebook***. This is what we will assume when we grade.

Aside: Matplotlib can only read PNGs natively. Further image formats are supported via the optional dependency on Pillow. Python's Imaging Library in its newer version (with Python 3 support) is called Pillow and older version (with Python 2) was called PIL.

You can first check Anaconda Navigator to make sure Pillow is installed (it should be already be installed for you with Anaconda); if not, you can install it there itself by selecting the library and choose Apply.



✓ pillow | Pillow is the friendly pil fork by alex clark and contributors | 6.2.0

Or you can install using Anaconda command line.

```
conda install -c anaconda pillow
```

Read in the image using `plt.imread()` function.

```
img_png = plt.imread("img1.png")
img_jpg = plt.imread("img1.jpg")
```

Task 2b. Display images in the Notebook (2 points)

Create a function called `display_image()` so we can reuse the steps for both images without having to write them again.

Matplotlib uses a drawing area called a Figure to place its image objects in. The Figure has several properties that we can set. We can set the labels on x-axis and y-axis as follows.

```
plt.xlabel('columns')
plt.ylabel('rows')
```

To draw the image, we create a new Figure and draw the image on it. This can be done by calling `plt.imshow()` function, and give it the variable that stores the image as an argument.

```
plt.imshow(img)
```

Let us add a colorbar to the Figure. Write the following command in the same cell as the `plt.imshow()` command.

```
plt.colorbar()
```

Finally, after we have plotted all the things we want in the Figure, we can call `plt.show()` to display the image in whatever GUI backend we want to use (we use the notebook itself as the backend). This is like `print()` but for images. Recall without `print()` we will not see the output outside of a notebook, same thing applies here.

```
plt.show()
```

Call the funtion twice, once with img_png as argument, and then with img_jpg. Notice the difference in colorbar scale for the colors across the two image formats.

Task 2c. Resizing the Figure (0.5 point)

The default size of a Matplotlib Figure is [8.0, 6.0] in inches. You may wish to resize the display of your image. `plt.figure()` is used to create a new Figure and set its properties/ parameters. If this is not explicitly called (we didn't call it earlier), then it creates and uses a Figure with default settings. We only need to explicitly call it when we want to set its properties. It has an optional parameter called `figsize` that you can specify as `(newrowsize, newcolsize)`. For e.g.,

```
plt.figure(figsize=(18, 12))
```

You have to write the above line of code at the *top of the cell* before you call the function to plot the image. Then, plot the PNG image again.

NOTE: This did not change the height/width of the image itself just the size of the Figure drawing area. Actual resizing of the image itself – is much more complex and requires the algorithm to deal with downsampling or upsampling, interpolation, and smoothing/averaging. So, we will leave that to functions in the specialized image processing libraries.

TASK 3. What kind of object is the image? (1 point)

Use the type function to obtain the class type of the PNG image object.

NOTE: Know the difference between *type* (what class does the collection of values belong to) and *datatype* (what type of value is each data element in the collection)

TASK 4. Print the shape of both the PNG and the JPG images (1 point)

You can get this using the `ndarray.shape` attribute which returns a 3-tuple

TASK 5. Print the number of pixels the image contains, for each image (2 points)

This calculation requires you to multiply the length (number of rows) by the width (number of columns) of the image. Use indexing on the tuple returned by the `ndarray.shape` attribute to obtain the length and width, and multiply them to produce the number of pixels.

Recall that values in a tuple can be indexed using []. The printed output should look similar to below (with diff values for the pixels corresponding to your image):

```
The PNG image consists of 3666673 pixels
The JPG image consists of 3662820 pixels
```

TASK 6. Print the values of the 3rd axis (i.e. 3 colors and opacity) of the pixel at position [55,80] for both images. Obtain and print the datatype of the values in the 3rd axis for both images. (2 points)

Verify that your PNG image produces a (4,) array while the JPG image produces a (3,) array. Note that the datatype *(not type)* of colors is also different across PNG and JPG.

Your output should look like:

```
The value at [__,__] of the png is : [__,___,___,___]
The datatype of the 3rd axis values of the png image is : __________
The value at [__,__] of the jpg is : [__,__,___]
The datatype of the 3rd axis values of the jpg image is : _________
```

From here on forth, we will only use the PNG image.

To display an image in the following tasks, ***please call the function we created in Task 2b***, except in task 12, where we will write different code.

TASK 7. Obtain the mean values of just the 3 colors – RGB – in your PNG image (but not the alpha value). (3 points)

Use a NumPy function to calculate the mean. Your output should contain a vector of 3 values, one for each color. For each color we want to take the mean over the rows and columns. So, we need to specify this by setting the `axis` parameter appropriately. Axis values 0, 1, 2 indicate each dimension.

When we have a 2D array, and we want means over either rows OR columns, then we specify axis as a scalar (0 for rows or 1 for columns). We have a 3D array. Since we want the mean level of each of the 3 colors (axis=2) for the whole image, we want to take means over both rows and columns. i.e. avg red over all pixels, avg green over all pixels, avg blue over all pixels.

You can set `axis =(rowaxis, columnaxis)` – which will obtain means over all rows and columns for each of the values in the 3rd axis. This will give you 4 mean values for RGBA for the PNG image. Use appropriate indexing to ***only print*** the means of the RGB (but not A).

Next, we will perform data augmentation tasks. To get full points here, please use NumPy array commands and not built-in functions from any image-processing libraries (unless asked to).

TASK 8a. Reverse the PNG picture on the second-axis (flip horizontally) using array indexing (2 points)

Since we are flipping horizontally, the rows and the third-axis are not affected. Only the column values have to be flipped (reversed). To do this, you can use indexing with slicing on the original array, and use step size -1 for the second-axis (col). Since we want the whole image, all rows, cols and all values in the 3rd dimension, be sure to indicate that in the start and stop values in those dimensions.

Tip: Recall that a slice has 3 parts – start, stop, step. If you don't provide a start or end value, the default is assumed (so you should not type in hardcoded numbers for start and stop), and all values are used. To use all values in a dimension, you can use an : (empty single colon with nothing before or after) for that dimension.

Step = -1 will read the values in that dimension from last to first, in reverse order.

Call the function created in step 2b to display the image with axis labels and colorbar.

TASK 8b. Reverse the PNG image on the first axis (flip vertically) using a NumPy function (1 point)

Let's use a built-in NumPy function for comparison. Perform a similar task as in 8a, but flip the original image vertically, using `np.flip()` or `np.flipud()`.

The first function has an `axis` parameter that you have to set to get the correct output. Please look up its documentation inline to determine what the value of axis should be.

Call the function created in step 2b to display the image with axis labels and colorbar.

TASK 9. Render the PNG image with the lower half of the image set at transparency level = 0.4. The rest of the image (the top half) should be at its original level (4 points)

Make a copy of the image. Usually when we manipulate an image by setting its values using the = symbol, it is good to make a copy if you do not wish to overwrite the original image.

This can be done by using the `ndarray.copy()` method of NumPy's ndarray (replace ndarray with your original image array name), and save this copy of the PNG image as a new ndarray.

```
img_png_transp = img_png.copy()
```

Next, you will need to use indexing (such as slicing) to select only the lower half of the rows, all the columns and then set the fourth value (opacity) in the 3rd axis = 0.4. Write the code to do this by replacing the three values inside [] below with appropriate values.

```
img_png_transp[row, col, value in 3rd-axis to change] = 0.4
```

Hints

a) determine the rows we want to change. Use the shape attribute of img_png to obtain just its rows. Then divide that by 2 to get half the rows. This will now be the starting index of the row slice at which you want to start changing the transparency.
   One issue here is that dividing by 2 may produce a fraction. Fractions cannot be used as indices (they have to be ints). So how can we convert the fraction to an int? Recall Python's `int()` function – used to cast a float (decimal) to an int. Convert the starting row index to an int. Or you can use floor divide operator (//) to get an integer quotient when you divide by 2 – which is what we want as the starting row index).
b) select all the columns.
c) select only the opacity (alpha) value from the 3rd axis.
d) Now you can set the opacity value for only the pixels in the lower half of the image (selected by the indexing described above) to the required value.
e) Call the function created in step 2b to display the image with axis labels and colorbar.

*NOTE: To get full points, you should **not hardcode** the row values i.e. do not obtain the actual size of your photo and divide that by 2 to get a number to use as a startindex for row. You should use a formula for startindex. In other words, if we use an image of a different size, your code should still work correctly.*

TASK 10. Obtain a negative of the PNG image (3 points max with bonus, else 2)

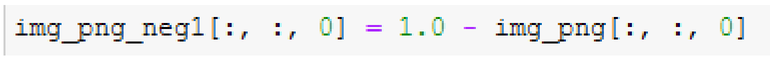
Reverse the colors for each channel - RGB. Since the max value is 1.0 for PNG, we want to subtract the value of each color from 1.0 to obtain its negative on that color channel. We will keep opacity the same.

First create a copy of the original image. Then select the values that you want to change in this image copy and set those values = newval.

Next, we want to select each color one by one and subtract it from 1.0, for all pixels. Remember the colors are in the 3rd axis, so we need not select anything for rows and columns as we want to select all pixels. Broadcasting will be applied to the 1.0 value to change it to the same size (row x column) as the image.

You can perform the subtraction for all three colors in one go using one line of code (using broadcasting along 2 dimensions), or you can write 3 lines of code to subtract each color one by one (using broadcasting along 1 dimension).

Here's how to obtain the negative of only the red color.



```
img_png_neg1[:, :, 0] = 1.0 - img_png[:, :, 0]
```

Complete the rest. Call the function created in step 2b to display the image with axis labels and colorbar.

BONUS (1 point):

Bonus point here for writing one line of code instead of 3 to perform the color subtractions from 1 (*you have to* ***also write the non-bonus code*** *first as well to get full 3 points*).

TASK 11. Display a cropped PNG image (2 points)

Cropping is akin to selecting contiguous slices along both the row axis and the column axis, and keeping all the values on the 3rd dimension. Focus on any interesting aspect of your picture and obtain only that area of the image. Make sure its not too small.

For this task only: it is necessary to hardcode the row and column slices - in other words, you have to put in actual numbers for start and end indices, without using a formula to calculate them.

TASK 12. Display the PNG image as grayscale (2 points)

A grayscale image only uses a single-color channel – from black to white with a single luminosity value to indicate the level of gray. A grayscale image is therefore a 2D array (row, col) and no longer 3D.

There are several ways to obtain this value – one approach is to use the weighted average of the three colors- RGB. What we want is to multiply the (3,) array of RGB colors with another (3,) array that contains desired weights for each color channel (opacity is ignored).

To perform the weighting, we can perform array multiplication using `np.dot()` (for two vectors) or `np.matmul()` (for two matrix or matrix and vector) function. Let us use the weights 0.21, 0.72 and 0.07 - with green weighted the highest because humans can see more contrast in green.

> Tip: To perform this task, we will first obtain the RGB values from the 3rd axis (for all rows and columns) of the image and matrix multiply that array with another (1,3) array of weights, `[redweight, greenweight, blueweight]`

Print the resulting image's shape to verify it is a 2D image. That is, the 3rd axis has been removed, because there is no longer a need for 3rd axis – it has been reduced to a single value instead of 4 values earlier.

Now plot this image in the Figure. To view in grayscale, we have to set an optional parameter `cmap` of `plt.imshow()` to `'gray'`. So, don't call the function we created in task 2b. Rather write your own display command here.

TASK 13. Apply Gamma correction to the PNG image (2 points)

Gamma correction is applied to make the picture have more light or less light. This is achieved by raising the color values of each pixel to a power (gamma value).

Mathematically,

```
outputImage_colorsarray = inputImage_colorsarray ^ gamma
```

Recall, however that the power operator in Python is **.

Each of the three colors is a float value (in the range 0.0 - 1.0). A gamma value > 1 makes the image darker (more shadows), whereas a gamma value of < 1, makes the image lighter and a gamma = 1 keeps the image the same.

Create a variable called `gamma = 1/5`.

Obtain an image array of ***only the 3 color values*** for all pixels in the image, and raise it to the power gamma (a scalar; no need to make it a vector because broadcasting will take care of it). This operation will change all the three colors for each pixel in the image to the same new level of brightness (by applying a gamma correction).

Do not change the alpha value, leave it the same.

Display the image with axis labels and colors by calling display_image() function. Can you see the difference as compared to the original?

Then, change the value of gamma to 5. Repeat above steps in a different cell.

Show the two images with more light (gamma <1) and less light (gamma >1). Save these images in new variables.

TRY OUT THIS OPTIONAL CODE

Sometimes, we wish to plot multiple images in one figure. Here we will plot all 3 in one figure with 3 subplots. The three images in order are –gamma=1/5, gamma = 1 (the original image), gamma = 5

You can do this using **Matplotlib subplots** to print the three images horizontally within one plot. Let us give the title of the plot as 'Horizontally stacked subplots'. Each sub-image should have its own title (gamma = value), it should not show the yticks (row), but show the xticks (column).

Below is the code.

```
fig, ax = plt.subplots(1, 3)
fig.suptitle('Horizontally stacked subplots', size =16)

ax[0].set_title('gamma = 1/5')
ax[1].set_title('gamma = 1')
ax[2].set_title('gamma = 5')

ax[0].set_yticks([])
ax[1].set_yticks([])
ax[2].set_yticks([])

ax[0].imshow(img_png_light)
ax[1].imshow(img_png)
ax[2].imshow(img_png_dark)
```

Explanation of the code

`plt.subplots()` allows us to make a plot with sub-plots. Each subplot has a plotting area, known as **AxesSubplot**. The dimensions of the subplots are specified as a 2-tuple. (1,3) means that plots are arranged using 1 row and 3 columns – plots arranged horizontally. (3,1) would mean 3 rows and 1 column of plot – plots arranged vertically)

The `subplots()` function returns a tuple of two objects – (Figure, Axes), and we separately save each output in a different variable - fig and ax, respectively.

```
fig, ax = plt.subplots(1,3)
```

`fig, ax` is a tuple (`fig, ax`). This is called ***unpacking a tuple*** (separating its values out) and saving each in a separate variable in one line of code.

Figure is the entire plotting area including (sub)plots, titles, legend, ticks, labels etc, and the **Axes** (the plot/ subplots). `fig.suptitle()` accepts the title of the main plot. It has a `size` keyword parameter that you can set using an int.

`ax` is an ndarray of shape (3,). That is, a 1D vector with 3 elements or 3 sub-plots. Each subplot is accessed as `ax[0]`, `ax[1]` and `ax[2]`, respectively. We can then set individual titles using the `set_title()` method of each AxesSubplot. By default, x and y ticks will show, or you can use `set_xticks` and `set_yticks` method of AxesSubplot (set it to empty [] to not show ticks).

We set the sub-title of each plots by specifying which object in the Axes we want to set, i.e. `ax[0]` is the first subplot, `ax[1]` is the second subplot etc.

> If instead we had specified we want subplots of dimensions (2,3) – i.e. two rows and 3 columns, we have to use indices to refer to each subplot. For example, `ax[0][1]` is the second plot in the first row.

Finally, print the images one by one using `plt.imshow()` for each subplot.

TASK 14. Obtain and print one image formed by horizontally stacking 3 images – redscale, greenscale and bluescale of the original PNG image (3 points).

To do this, we obtain 3 different images from the original image – one in each color scale, and then use a NumPy function to concatenate or horizontally stack them.

For each color, first create a copy of the original image using `ndarray.copy()` and save in a new variable, say img_png_Red. The redscale image is where the colors green and blue are set to 0; the greenscale is where the colors red and blue are set to 0, and the bluescale image is where the colors red and green are set to 0.

For the redscale image, change the values in the green and blue channel to zero. You can use:

```
img_png_Red[:,:,[1,2]] = 0
```

Do this for the other two colors as well, and you will then have 3 new images.

Next, use a NumPy function to join the three images together horizontally (left to right with the red scale first, followed by green scale and then blue scale).

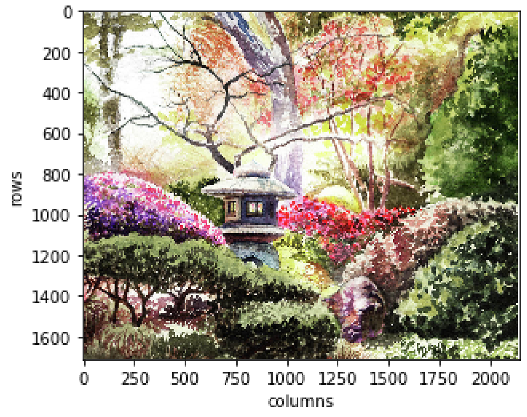
> HINT: use a function to concatenate matrices. There are a few different ones (we learned one in class) that you can use. You may need to specify the axis along which you want to concatenate.

Adjust the size of the plot to enlarge the images. Finally since you have created a single image you can once again call the display_image() function to display the horizontally stacked image.

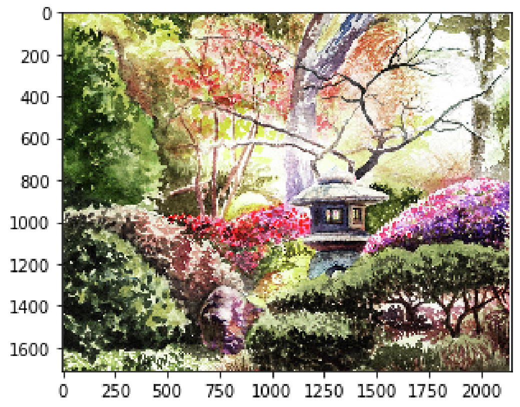
We've seen a few different manipulations of the data (you created a photo editor along the way!).There are others you can do – rotate, distort, shift etc. Hope you are feeling good about your NumPy skills now.

C. Sample outputs of tasks that produce an image

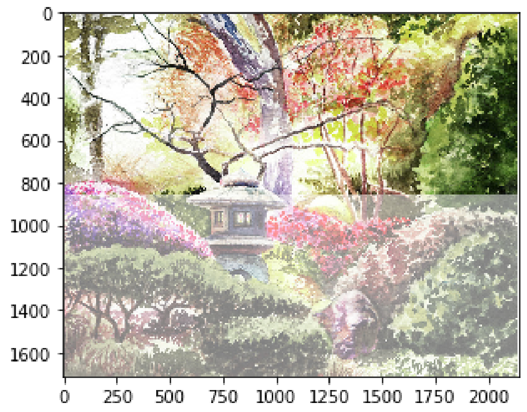
Task 2



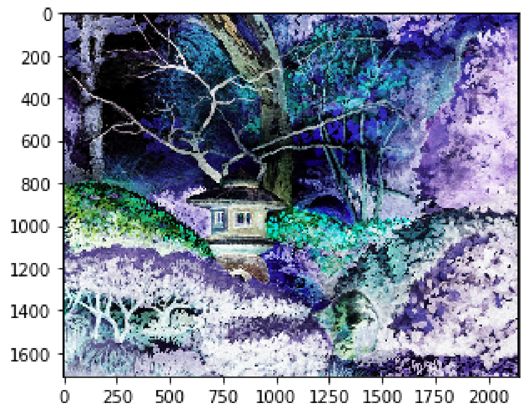
Task 8



Task 9



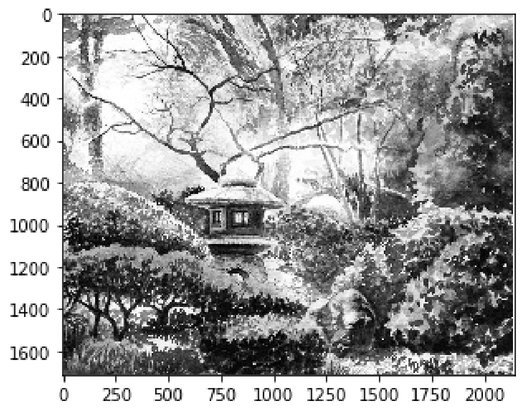
Task 10



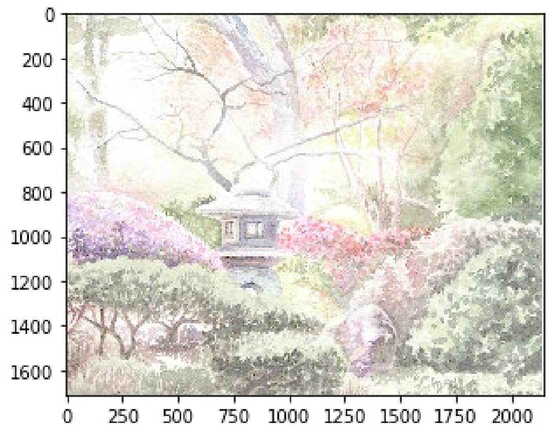
Task 11

Show any interesting part of the image (zoomed in)

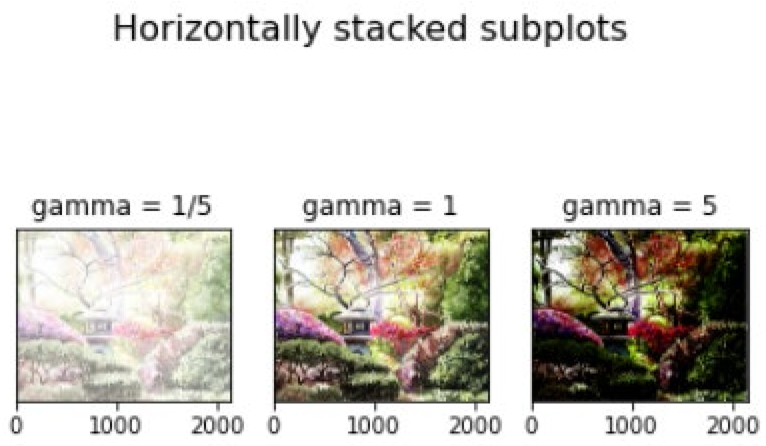
Task 12



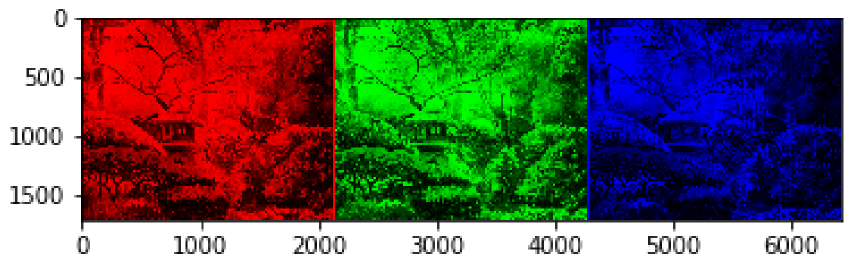
Task 13



Optional output



Task 14


0
500
1000
1500
0
1000
2000
3000
4000
5000
6000