

DSC 20 Mid-Quarter Project

Total Points: 100 (10% of the Course Grade)

Submission due (SD time):

- **Checkpoint: Tuesday, February 23th, 11:59pm**
- **Final submission: Wed, March 3rd, 11:59pm**

Starter Files

Download [midqtr_project.zip](#). Inside this archive, you will find the starter files for this project.

Checkpoint Submission

You can earn 5 points extra credit by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Part 1 and the first two methods of Part 2 (*negate*, *grayscale*)** and submit **ONLY** the *midqtr_project.py* file to gradescope.

Checkpoint submission is graded by completion, which means you can get full points if your code can pass some simple sanity check (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you noticed any errors.

Final Submission

Submit **ONLY** the *midqtr_project.py* file to gradescope. You can submit multiple times before the due date, but only the final submission will be graded.

If you are working with a partner, please have **ONLY ONE** person to submit your group's work. You can add your partner to your submission once you submit it. **If you resubmit your work, please make sure to re-add your partner again - Gradescope by default will not link your latest submission to your partner automatically.**

Requirements

1. **Do NOT import any packages** in *midqtr_project.py*.
2. Follow the [style guide](#) on the course website.
3. You are **not required to add new doctests** this time for grading. However, you still need to test your code since we will use our own tests to grade your project.
4. You should still **add docstrings** for modules, classes and functions/methods.

5. You should add assert statements when the question explicitly requires them.
If assertions are not explicitly required, you can assume that the arguments are valid.

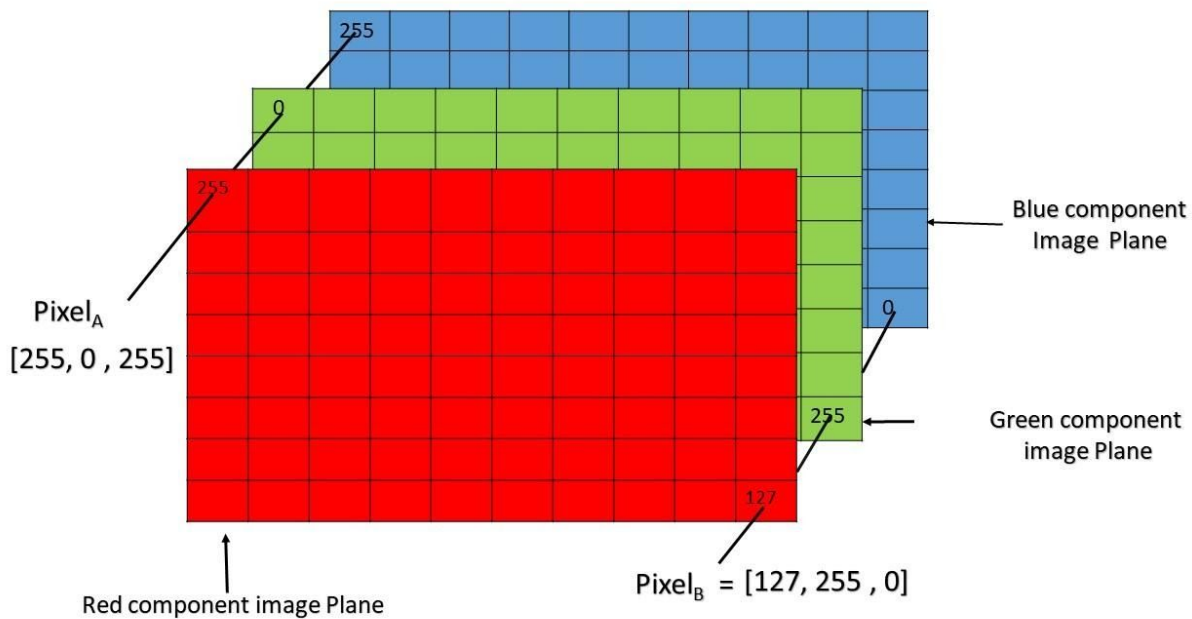
Overview

In this project, we will discover the basics of image processing, a key area in the computer science field. In part 1, we will define an abstraction of images in RGB colorspace with a Python class. In part 2, we will implement multiple image processing methods that you might have seen in your daily life. In part 3, we will implement a K-nearest neighbour classifier to classify and predict the labels of images.

In the digital world, images are defined as 3-dimensional matrices: height (row), width (column), and channel (color). Each (row, col) entry is called a **pixel**. Height and width dimensions are intuitive: they define the size of the image. The channel dimension defines the color of an image.

The most commonly used color model is the RGB color model. In this model, every color can be defined as a mixture of three primary color channels: **Red**, **Green** and **Blue**. Thus, the color of a pixel is digitally defined as a triplet (R, G, B). Each element in this triplet is an integer (called *intensity*) with value between 0 and 255 (both inclusive), where 0 represents no R/G/B is present and 255 means R/G/B is fully present. Thus, (0, 0, 0) represents **black** since no R/G/B are present, and (255, 255, 255) represents white since all these colors are fully present and mixed. To better understand how the RGB color model works, you can play around the RGB value with this [online color wheel](#).

In our project, we will use a 3-dimensional list of integers to structure the pixels. This picture ([source](#)) shows how a *pixels* list is structured.



Pixel of an RGB image are formed from the corresponding pixel of the three component images

The first dimension is the color channel. In other words, $\text{len}(\text{pixels}) = 3$, and each element represents a color channel. Each color channel is a `row` by `column` matrix that represents the intensities (0 - 255) of this color. Therefore, to index a specific intensity value at channel c , row i and column j of the `pixels` list, you should use `pixels[c][i][j]`.

Note that the *width* of an image is the length of the column dimension (number of columns), and the *height* of an image is the length of the row dimension (number of rows). Since in Python we conventionally consider (row, column) as the order of dimensions for 2-dimensional lists, make sure to distinguish these notions clearly. We have also provided an example of `pixels` list in the `midqtr_project_runner.py` to help you understand the structure of `pixels` list.

Testing

For this project, we will not require you to write new doctests for any functions. However, it is still for your benefit to test your implementation thoroughly.

Since this project aims to process images, it makes more sense for you to check the output against real images instead of 3-dimensional lists. Therefore, we have provided some helper functions for you to read and write actual images, so you can manually evaluate if your implementation works correctly. The helper functions (with examples of usage) are provided in the `midqtr_project_runner.py`. This file serves as an entry point to your implementation. Feel free to add more functions and/or test cases in this file. You do not need to submit this runner file to gradescope.

As a refresher, here are the commands to test your code (with the runner):

- **No options:** `>>> python3 midqtr_project_runner.py`
- **Interactive mode:** `>>> python3 -i midqtr_project_runner.py`
- **Doctest:** `>>> python3 -m doctest midqtr_project_runner.py`

(For Windows users, please use `py` or `python` instead of `python3`.)

You can add more tests using a similar format to what we provided. You can add to or create new functions like `image_processing_test_examples()` and run the commands above. You can then check your code by checking the images that are created in the filepath that you specify. For details, please see the examples in [How to correctly test your code with the runner file](#) section.

Install OpenCV and NumPy

You might have noticed that this runner file used two packages: NumPy (np) and OpenCV (cv2). They are the most common packages to use for image processing. Although you cannot use these packages in your own implementation, you can use these packages to help your testing. If you have not installed these packages before, run the following command in your terminal (at any directory):

```
python3 -m pip install numpy opencv-python
```

(For Windows users, please use `py` or `python` instead of `python3`.)

Some students may have trouble installing the OpenCV package. Therefore, there is another runner file that is written with the **Pillow** package as a workaround. To install the Pillow Package, run the following command in your terminal (at any directory):

```
python3 -m pip install Pillow
```

(For Windows users, please use `py` or `python` instead of `python3`.)

If you successfully install the packages and still have issues when trying to run tests, it may be because of a bug in a specific numpy version for Windows users. Try this command if your terminal says there are any issues with Numpy.

```
py -m pip uninstall numpy Pillow
```

```
py -m pip install numpy==1.19.3 Pillow
```

How to correctly test your code with the runner file?

We only provided the example tests as functions, but we did not call these functions in the file. Therefore, if you want to check your implementation against our examples, you need to call these example functions in your own tests. In addition to the examples, you should definitely write more of your own tests in the runner file.

The approach to writing your own tests is up to you. If you prefer:

(1) doctests

Create a dummy function (a function that does nothing) and add your own doctests in the docstring of this function. Then, run it using "python3 -m doctest <filename>"

For example:

```
## END OF THE ORIGINAL RUNNER FILE ##

def my_tests():
    """
    >>> image_processing_test_examples()
    >>> ... (your own test cases)
    """

    return
```

(2) run directly

Append your own test cases to the end of the file and run it using "python3 <filename>". An example method with basic tests is already provided in the runner files. If you have installed the packages and running the file does not work, please try approach (1) above.

For example:

```
## END OF THE ORIGINAL RUNNER FILE ##

image_processing_test_examples()
... (your own test cases)
```

(3) interactive mode

Open the interactive shell with "python3 -i <filename>". Then, call the example functions or run your own tests line by line.

To test the image output, it could be more intuitive to save the image to your local machine and eyeball this new image. After running your tests, you should be able to see the new images generated in the specified path. Note that if you want to use the example we provided, please move the provided output images in the starter to other places so that you won't get confused.

Part 1: RGB Image

In this part, you will implement the **RGBImage** class, a template for image objects in RGB color spaces.

You need to implement the following methods:

__init__ (self, pixels)
<p>A constructor that initializes a RGBImage instance and necessary instance variables.</p> <p>The argument <code>pixels</code> is a 3-dimensional matrix, where <code>pixels[c][i][j]</code> indicates the intensity value in channel <code>c</code> at position <code>(i, j)</code>. You must assign this matrix to <code>self.pixels</code>.</p> <p>You can assume that the index of the channel <code>c</code> is guaranteed to be 0 (red channel), 1 (green channel) or 2 (blue channel). You can also assume that each channel contains a valid (row × col) matrix.</p>
size (self)
<p>A getter method that returns the size of the image, where size is defined as a tuple of (number of rows, number of columns).</p>
get_pixels (self)
<p>A getter method that returns a DEEP COPY of the pixels matrix of the image (as a 3-dimensional list). This matrix of pixels is exactly the same <code>pixels</code> passed to the constructor.</p> <p>Note:</p> <p>Make sure that this returns a deep copy of the pixels matrix, otherwise there will be many issues later on with ImageProcessing modifying additional images, which will impact the tests and your score. You won't be able to import the copy package, so you have to create the deep copy manually.</p> <p>We can create a deep copy of a 1D-list 'x' with a list comprehension (<code>[elem for elem in x]</code>) or with (<code>list(x)</code>). This is a 1D-list, so the implementation will differ for <code>pixels</code> since it is a 3D-list. If you try to just use (<code>list(pixels)</code>) you will not be creating a deep copy of all the dimensions of the list and will run into issues.</p>
copy (self)
<p>A method that returns a COPY of the RGBImage instance. You should create a new RGBImage instance using a deep copy of the <code>pixels</code> matrix (you can utilize the <code>get_pixels</code> function) and return this new instance.</p> <p>Note:</p> <p>Make sure that the new RGBImage instance is initialized with a deep copy of the pixels matrix, otherwise there will be many issues later on with ImageProcessing modifying additional images, which will impact the tests and your score.</p>

get_pixel (self, row, col)

A getter method that returns the color of the pixel at position (row, col). The color should be returned as a 3-element tuple: (red intensity, green intensity, blue intensity) at that position.

Requirement:

Assert that *row* and *col* are valid indices.

set_pixel (self, row, col, new_color)

A setter method that updates the color of the pixel at position (row, col) to the new_color **in place**. The argument new_color is a 3-element tuple (red intensity, green intensity, blue intensity). However, if any color intensity in this tuple is provided as -1, you **should not** update the intensity at the corresponding channel.

Requirement:

Assert that *row* and *col* are valid indices.

Part 2: Image Processing Methods

In this part, you will implement several image processing methods in the **ImageProcessing** class.

Notes:

- (1) All methods in this class have a decorator *@staticmethod*. This decorator indicates that these functions do not belong to any instances, and should be called by the class itself. For example, to use the function *negate()*, you should call it like *ImageProcessing.negate(image)*, instead of initializing an *ImageProcessing* instance first.
- (2) All methods in this class must return a **new RGBImage instance** with the processed pixels matrix. After calling any of these methods, the original image should not be modified.

Hint:

If you find processing 3-dimensional pixels difficult, try to approach these problems in a 2-dimensional perspective (pick any color channel to work with first) and broadcast your solution to all three channels. If you are stuck, try to write down the (row, column) matrix before and after applying the function and derive a pattern.

You need to implement the following methods:



@staticmethod **negate (image)**

A method that returns the negative image of the given *image*. To produce a negative image, all pixel values must be inverted. Specifically, for each pixel with current intensity value *val*, this method should update it with $(255 - val)$.

Requirement:

No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	negate (image)
	



@staticmethod **grayscale (image)**

A method that converts the given *image* to grayscale. For each pixel (R, G, B) in the pixels matrix, calculate the average $(R + G + B) / 3$ and update all channels with this average, i.e. $(R, G, B) \rightarrow ((R + G + B) / 3, (R + G + B) / 3, (R + G + B) / 3)$. Note that since intensity values must be integer, you should use the integer division.

Requirement:

No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	grayscale (image)
	

Note: Checkpoint submission ends here.




@staticmethod **clear_channel (image, channel)**

A method that clears the given *channel* of the *image*. By clearing a *channel*, you need to update every intensity value in the specified *channel* to 0.

Requirement:

No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	clear_channel (image, 0)	clear_channel (image, 2)
		

@staticmethod **crop (image, tl_row, tl_col, target_size)**

A method that crops the *image*.

Arguments *tl_row* and *tl_col* specify the position of the **top-left** corner of the cropped image. In other words, position (*tl_row*, *tl_col*) before cropping becomes position (0, 0) after cropping.

The argument *target_size* specifies the size of the image after cropping. It is a tuple of (number of rows, number of columns). However, when the specified *target_size* is too large, the actual size of the cropped image might be smaller, since the original image has no content in the overflowed rows and columns.




Tip:

When a *target_size* (*n_rows*, *n_cols*) is possible to achieve given the original size of the image, *tl_row*, and *tl_col*, (*tl_row* + *n_rows*) and (*tl_col* + *n_cols*) give you *br_row* and *br_col*, which is the position of the **bottom-right** corner of the cropped image.

Requirement:

No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	crop (image, 50, 75, (75, 50))	crop (image, 100, 50, (100, 150))
 size = (190, 190)	 actual size = (75, 50), <i>target_size</i> does not overflow	 actual size = (90, 140), <i>target_size</i> overflows both row and column, thus the actual size is smaller in both dimensions.

@staticmethod **chroma_key** (**chroma_image**, **background_image**, **color**)

A method that performs the chroma key algorithm on the *chroma_image* by replacing all pixels with the specified *color* in the *chroma_image* to the pixels at the same places in the *background_image*. If the *color* does not present in the *chroma_image*, this function won't replace any pixel, but it will still return a copy.

You can assume that *color* is a valid (R, G, B) tuple.


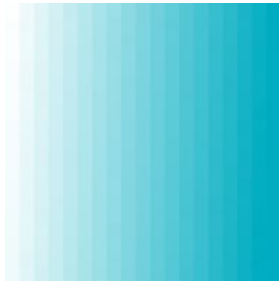


Tip:

When testing this function, you can find pictures with a green or blue screen in the background online, find your favorite pictures as background images, and use this function to replace the background color with the background image you choose. **Make sure you crop them to the same size before applying this function.**

Requirement:

Assert that *chroma_image* and *background_image* are *RGBImage* instances and have the same size.

Example:

chroma image	background image	color = (255, 255, 255)	color = (255, 205, 210)
		 (white background replaced)	 (pink font color replaced)

@staticmethod **rotate_180 (image)**

This method is for **extra credit**. Points are undetermined as of now.

A method that rotates the image for 180 degrees.



Tip:

Try to approach this problem in a matrix perspective: which matrix operations will help to achieve this purpose? Try writing out a test matrix on paper and see which operations need to be in order to rotate it 180 degrees.

Requirement:

No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	rotate_180 (image)
	

Part 3: Image KNN Classifier

Classification is one of the major tasks in machine learning, which aims to predict the label of a piece of unknown data by learning a pattern from a visible collection of data. To train a classifier, you need to fit the classifier with training data, which is a collection of (data, label) pairs. The classifier will apply the training data to its own algorithm. After training, you can provide a piece of known or unknown data, and the classifier will try to predict a label. By training a classification algorithm, we can extract essential information from pieces of complicated data.

In this part, you will implement a [K-nearest Neighbors](#) (KNN) classifier for the RGB images. Given an image, this algorithm will predict the label by finding the most popular labels in a collection (with size k) of **nearest** training data.

But how could we evaluate how near two images are? We need a definition of **distance** between images. With this definition, we can find the nearest training data by finding the shortest distances. Here, we use the [Euclidean distance](#) as our definition of distance. Since images are represented as 3-dimensional matrices, we can first flatten them to 1-dimensional vectors, then apply the Euclidean distance formula. For image a and b , we define their Euclidean distance as:

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \dots + (a_n - b_n)^2}$$

Where a_i and b_i ($1 \leq i \leq n$ based on the above equation) are the intensity values at the same position of two image matrices, and n is the count of individual intensity values in the image matrices, which equals to (number of channels (which is 3 for RGB) \times number of rows \times number of columns). Note that **to calculate the distance, two images must have the same size**. Can you figure out why?

Once we have a notion of distance, we can start implementing a KNN classifier. In the fitting (training) stage, all you need to do is to store the training data. Then, in the prediction stage, the algorithm will find the distance between the provided image and all images in the training data, find k nearest training data, and predict the label by the most popular label among the k -nearest training data.

Our blueprint of the classifier will be defined in the ImageKNNClassifier class. In this class, you need to implement the following methods:

__init__ (self, n_neighbors)

A constructor that initializes a `ImageKNNClassifier` instance and necessary instance variables.

The argument `n_neighbors` defines the size of the nearest neighborhood (i.e. how many neighbors your model will find to make the prediction). When predicting the labels, this classifier will look for the majority between the `n_neighbors` closest images.

fit (self, data)

Fit the classifier by storing all training data in the classifier instance. You can assume `data` is a list of `(image, label)` tuples, where `image` is a `RGBImage` instance and `label` is a string.

Requirements:

- (1) **Assert** that the length of `data` is greater than `self.n_neighbors`.
- (2) **Assert** that this classifier instance does not already have training data stored.
- (3) Make sure that the name of the instance attribute that you store the training data in is `data`

@staticmethod distance (image1, image2)

A method to calculate the Euclidean distance between RGB image `image1` and `image2`.

To calculate the Euclidean distance, for the value at each position (channel, row, column) in the `pixels` of both images, calculate the **squared difference** between two values. Then, add all these values of squared difference together. The Euclidean distance between two images is the **square root** of this sum. You can refer to $d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \dots + (a_n - b_n)^2}$ if you prefer a more formal definition.

Requirements:

- (1) **No explicit for/while loops.** You can use list comprehensions or `map()` instead.
- (2) **Assert** that both arguments are `RGBImage` instances with the same size.

@staticmethod vote (candidates)

Find the most popular label from a list of candidates (nearest neighbors) labels. If there is a tie when determining the majority label, you can return **any** of them.

predict (self, image)

Predict the label of the given *image* using the KNN classification algorithm. You should use the `vote()` method to make the prediction from the nearest neighbors.

Requirements:

- (1) **No explicit for/while loops.** You can use list comprehensions or `map()` instead.
- (2) **Assert** that the training data is present in the classifier instance. In other words, assert that `fit()` method has been called before calling this method.

After implementing this classifier, try to run the example test in the runner, and find more images online (again, crop them to the same size in order to calculate distance) to make your own tests. While testing your implementation, think about the following questions. You don't need to submit your answers to these questions to gradescope.

- (1) How long does it take to run a single prediction? If it runs longer than you expect, why? Think about how many arithmetic calculations this algorithm needs to predict an image.
- (2) Will this algorithm work for all kinds of image data? Why does the example test in the runner works as expected, while your own tests with online images and labels might not work very well?
- (3) Are there any other ways to define this distance for the nearest neighbor? Think about it, but don't change the distance algorithm in your submission.
- (4) What are the advantages and disadvantages of using Euclidean distance to find the nearest neighbors?