



COMP9417 COMPUTER VISION

ASSIGNMENT 1

Term 1, 2020

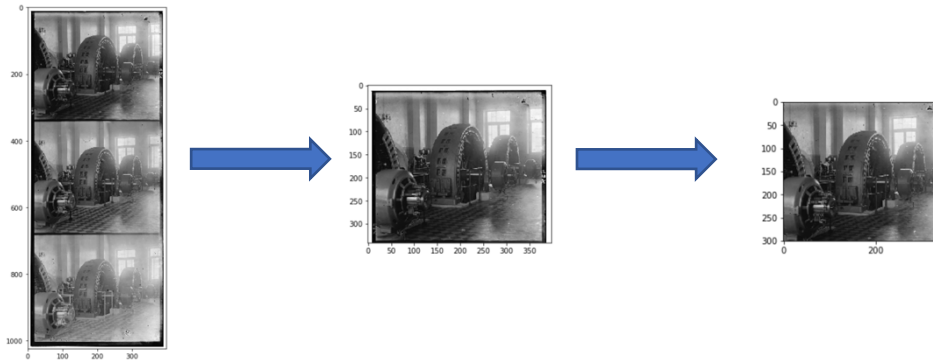
Submitted by Vandhana Visakamurthy

Z5222191

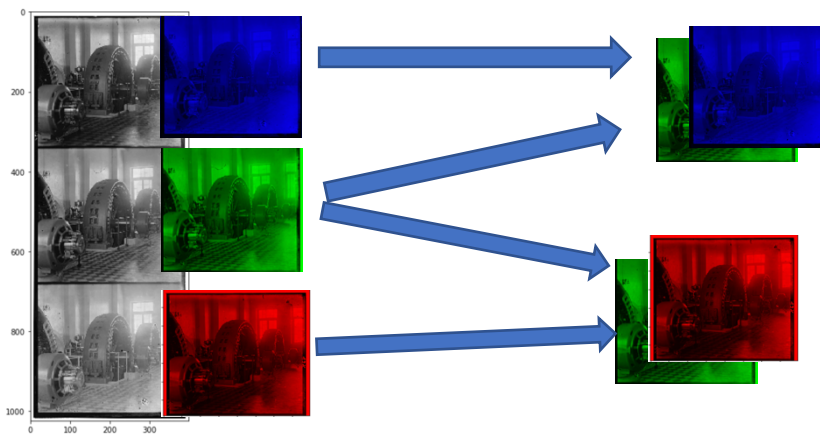
Task 1 :

The objective of Task 1 is to reconstruct the given negatives of the image by splitting the channels from the respective portions of the image and align them to obtain a single multi-channel image that is in the RGB color space.

1. To achieve the desired results, the images were preprocessed using two custom functions – `crop_image()` and `remove_borders()`. The former divides the given image to return three images and the latter removes the black borders by removing twenty pixels on all sides. This gives three clean, borderless images of well-defined shape.



2. After preprocessing the images, a custom function called `channel_splitting()` is called to split a given image into respective channels (B,G,R) and this is done for all the three images. The blue channel of the first image, the green channel of the second image, and the red channel of the third image is stored in new variables as they are relevant, and the remaining channels of the images are discarded. The channels were split (`[:, :, c]`) by accessing the channel dimension of the image instead of using `cv2.split()` to reduce computation.



3. Upon merging these newly obtained color channels, it is evident that the channels are misaligned. To reduce this error in misalignment three functions `ssd(x,y)`, `search_space(base,img)`, and `minimum_error(config)` functions are used to find the optimum offsets for the red and blue channel images with the green channel image as the fixed base. The `search_space` function takes in a base image and a template image and calculates the SSD by calling the custom written SSD function that returns the sum of the differences squared for both images. This is then repeated for every offset that varies by n columns (where n is any value between 0 and 20) in all four directions (Left, Right, Top, Bottom). The offsets are created by adding a padding sequence for calculating the error. These values are stored in a dictionary called `config`.

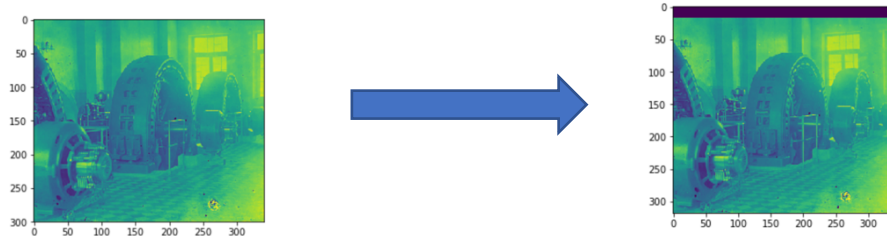
Example : Let's say the error of the two images in the configuration of right direction after padding 4 columns is 146, these are recorded in the config dictionary like this $\{ 'R4' : 146 \}$. This is done for all directions and the configurations are R_i, L_i, A_i, B_i where 'i' stands for the padded value.

A helper function called `minimum_error` was created to obtain the configuration with minimum error in the config dictionary.

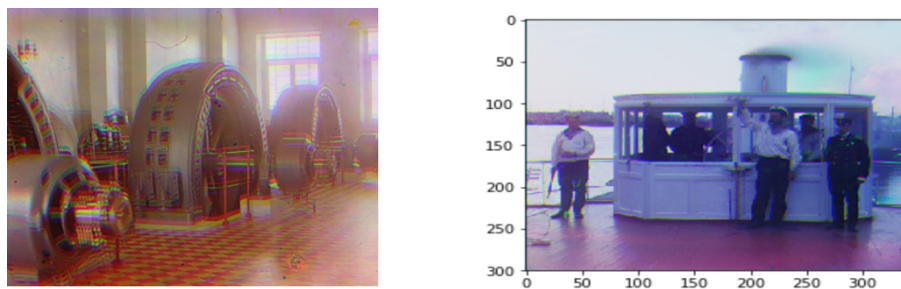
4. This was done twice – once for red and green and once for blue and green. The green channel was made a constant base for red and blue templates while calculating the optimum offset. The `red_config` stored the values for the former and the `blue_config` stored the value for the latter. The minimum offsets were identified and stored as `red_min` and `blue_min` which was later used to reconstruct the optimum offset.

(The minimum offset and the calculation of all offsets and their error can be found in the .ipynb file)

5. A function called `optimum_offset(base_img, min_config)` was created to take in two parameters – the image that had to be realigned and the offset that produced the minimum error. The newly aligned image would be padded accordingly.



6. The newly aligned images are cropped to obtain only the relevant area of the same shape as the base green channel. This is repeated for the blue channel. These images are merged using `cv2.merge()` and converted into RGB color space to write as output as the program uses `matplotlib` which reverses the RGB/BGR color space.
7. The output files that are written as jpeg files and are attached separately. The sample images that are reconstructed look like this.



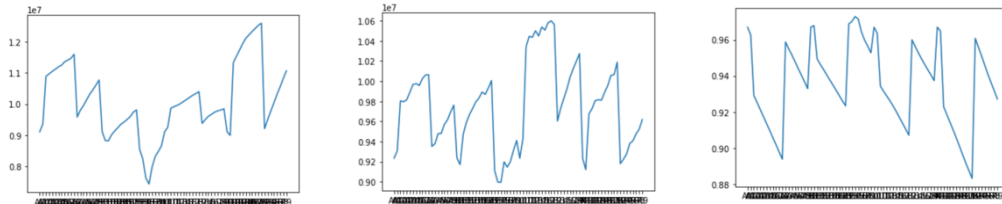
8. So, to determine the optimum offset for each channel, a total of $(20 \times 4 \times (m \times n))$ pixels) had to be searched and the SSD of all those images have to be calculated and stored.

In this case a total computational cost of $(20 \times 4 \times (300 \times 340))$ was spent to find the optimum offset. This can be quite expensive.

9. SSD was chosen as the suitable metric for this assignment because upon estimating the time it takes to perform the search for the offset in all directions SSD was the most efficient in terms of time for larger images and had a more normalized error value. This conclusion was made by calculating the time each metric took to search for 20 offsets in all four directions using `time.time()` and visualizing the error value for all configurations, the range within the error was minimum in SSD than for SAD or NCC. (Kindly refer Jupyter Notebook titled Estimation of Metrics for detailed analysis).

Metric for Searching Offsets	Time Taken for 20 offsets in 4 directions
SAD	0.025965213775634766 seconds
SSD	0.0326077938079834 seconds
NCC	0.09199690818786621 seconds

Error value range of SAD, SSD, and NCC plotted for all configurations are given below



SAD and NCC had an error value range of 0.8 to 1.2 whereas SSD had an error range of 0.9 to 1.2 and therefore could ensure the best alignment.

- SSD was chosen considering the tradeoff between time taken to calculate all offsets and the error value range. SSD was not as computationally expensive as SAD or NCC and was in the optimum time for computation as well.

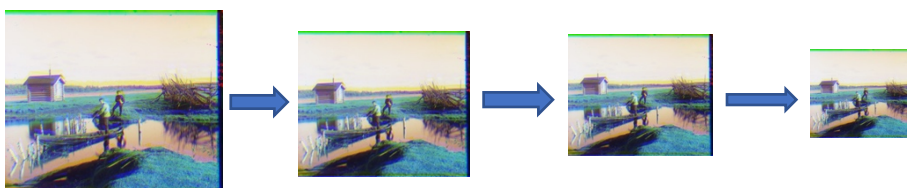
Task 2 :

Upon analyzing the results of Task 1, it is evident that searching through all offsets for an image of average resolution is computationally expensive. Therefore, performing such a task for a fairly large image can be explosive in terms of time and memory. Therefore, the aim of Task 2 is to figure out an efficient way to perform Task 1 on an image of a relatively large resolution and still obtain results of the same precision. To achieve this a series of steps were followed :

- An image of large resolution was initially preprocessed similar to Task 1 using the previously written functions `crop_image()` and `remove_borders()`. Following which the image was split into three channels.
- Step 1 has prepared the image for preprocessing and now the images of various color channels are ready to be built as a pyramid.

An image pyramid (or a so called image pyramid) is an image processing concept that either increases the resolution (`pyrUp`) of an image or reduces the resolution of an image (`pyrDown`) to ensure that the manipulations on the relevant resolutions are efficient than that of the initial resolution.

- A function called `downsize_image(img)` was created to take a given image and downsize the image into smaller resolutions. All these images are stored in a list called `resolutions = []`. This list contains all the resolutions of all the three channels. These images are accessed using list comprehension to search for the best offsets for alignment.




- The image was repeatedly downsized using the `cv2.resize()` function by a factor of 2 as a whole. Therefore, the hyperparameters $fx = 0.5$ and $fy = 0.5$ were used to achieve this. The Resolutions for the given image were the following and the break point set was three. After this the images are too distorted to work with. If the count wasn't equal to the breaking point the image underwent downsizing, else it remained the same. This was done recursively.
Largest Resolution : (1500, 1650)
Medium Resolution : (750, 825)
Smallest Resolution : (375, 412)
- Task 1 was repeated for the smallest resolution and the optimum offsets were stored in `red3_min` and `blue3_min`. The offsets helped narrow down the search space. The computational complexity for the smallest image would be $(20 \times 4 \times (375 \times 412))$.
- The algorithm for Task 1 is modified and a new function called `optimum_search_space()` is created. This function takes in the minimum offset, the base image, and the image to align. Unlike the previous function this algorithm searches only in the direction of the minimum offset. For example, in this case, the best offset for the red channel was 'A18'. So, the function will nudge the algorithm in only the up direction. This was achieved by tweaking the original function using string manipulation and if/else statements.

```

# search space is an algorithm to find error between offsets and store them to find the configuration with lowest error
# images are padded to get offsets from 0 to 20 and calculate error between base and template images
def optimum_search_space(base_img, temp_img):
    config = {}
    height = int(base_img.shape[0])
    width = int(base_img.shape[1])
    ##### RIGHT SEARCH #####
    for i in range(21):
        temp_img = cv2.pad_img(temp_img, (0, i, 0, 0), constant_values=0)
        temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
        #print(temp_img)
        error = ssd(base_img, temp_img)
        setting = 'R' + str(i)
        config[setting] = error
    ##### LEFT SEARCH #####
    for i in range(21):
        temp_img = cv2.pad_img(temp_img, (0, 0, 0, i), constant_values=0)
        temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
        #print(temp_img)
        error = ssd(base_img, temp_img)
        setting = 'L' + str(i)
        config[setting] = error
    ##### BOTTOM SEARCH #####
    for i in range(21):
        temp_img = cv2.pad_img(temp_img, (0, 0, i, 0), constant_values=0)
        temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
        #print(temp_img)
        error = ssd(base_img, temp_img)
        setting = 'B' + str(i)
        config[setting] = error
    ##### TOP SEARCH #####
    for i in range(21):
        temp_img = cv2.pad_img(temp_img, (i, 0, 0, 0), constant_values=0)
        temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
        #print(temp_img)
        error = ssd(base_img, temp_img)
        setting = 'T' + str(i)
        config[setting] = error
    return config

```

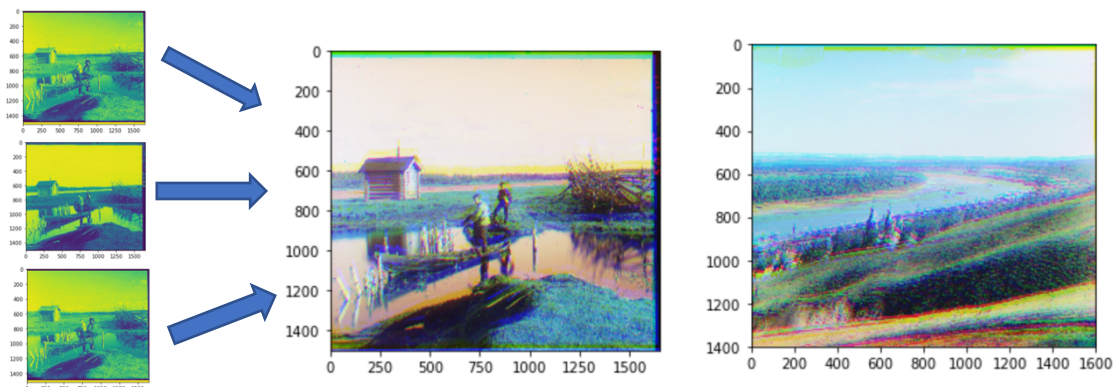


```

#The list resolutions have a copy of all the sizes of all the three channels
#After searching the smallest resolution we search in the next smallest resolution
def optimum_search_space(base_img, temp_img, search_area):
    config = {}
    height = int(base_img.shape[0])
    width = int(base_img.shape[1])
    ##### RIGHT SEARCH #####
    if search_area == 'R':
        for i in range(21):
            temp_img = cv2.pad_img(temp_img, (0, i, 0, 0), constant_values=0)
            temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
            #print(temp_img)
            error = ssd(base_img, temp_img)
            setting = 'R' + str(i)
            config[setting] = error
    ##### LEFT SEARCH #####
    if search_area == 'L':
        for i in range(21):
            temp_img = cv2.pad_img(temp_img, (0, 0, 0, i), constant_values=0)
            temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
            #print(temp_img)
            error = ssd(base_img, temp_img)
            setting = 'L' + str(i)
            config[setting] = error
    ##### BOTTOM SEARCH #####
    if search_area == 'B':
        for i in range(21):
            temp_img = cv2.pad_img(temp_img, (0, 0, i, 0), constant_values=0)
            temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
            #print(temp_img)
            error = ssd(base_img, temp_img)
            setting = 'B' + str(i)
            config[setting] = error
    ##### TOP SEARCH #####
    if search_area == 'T':
        for i in range(21):
            temp_img = cv2.pad_img(temp_img, (i, 0, 0, 0), constant_values=0)
            temp_img = temp_img[height-i:height, 0:width] #image size = 300 x 340 -----> change size
            #print(temp_img)
            error = ssd(base_img, temp_img)
            setting = 'T' + str(i)
            config[setting] = error
    return config

```

- The computation for the medium resolution image would be $(20 \times 1 \times (750 \times 825))$. We have reduced the computation cost of a larger resolution image by $1/4^{\text{th}}$. If we repeat step 6 for the medium resolution image for red and blue channels with green as base then it is evident that the search direction is correct and the offsets are within the same range as their smaller counterparts. This is verified by multiplying the number of offsets to be shifted by 2 as a whole or 0.5 in a single direction. (Factor of downsizing = $0.5 \times \text{offset}$)
- After obtaining the correct offsets for the higher resolution image by repeating step 7, the images are corrected by cropping off the unnecessary values and the single high resolution image is obtained by aligning the channels and merging using `cv2.merge()`.

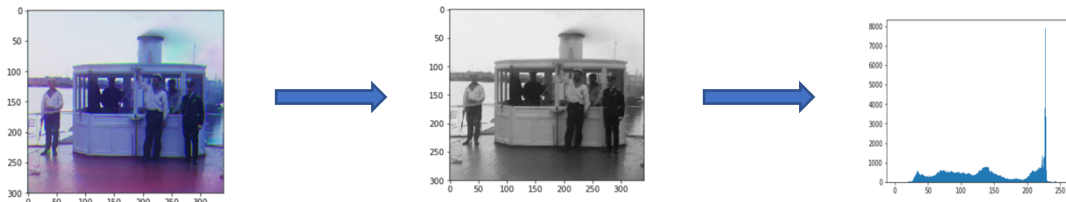


(The sample outputs are attached separately and the code for this task can be found in the .ipynb file)

Task 3 :

The aim of Task 3 is to enhance the color and contrast of the reconstructed images. The reconstructed images are written into the system as Task1Output1 and Task1Output2. The following steps were used to enhance the reconstructed images.

1. The reconstructed image that's read in was converted to a grayscale image and the histogram for the image was plotted to understand how the distribution of pixel intensities looked like.



2. The pixel intensities are not well distributed and hence this was improved using contrast stretching. This enhances the color range as well.

#Formula to achieve contrast stretching

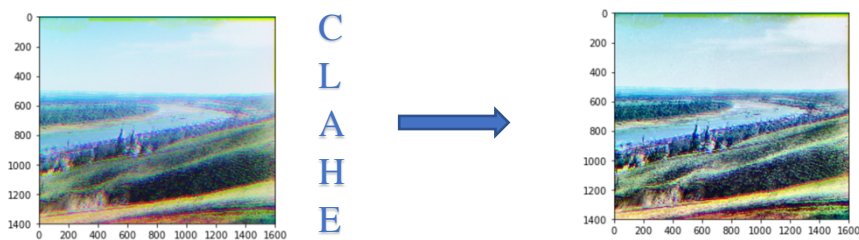
$$Tr = (Or - c) * (255/(d-c))$$

The histogram on the right is stretched and the pixel intensities are distributed evenly when compared to the histogram above.



3. An automatic border cropping algorithm has already been implemented in Task 1 and 2 (crop_image() and remove_borders())

The enhancements to improve higher resolution images is by using CLAHE. Contrast Limited Adaptive Histogram Equalization is a contrast enhancement technique used for high resolution images. It reduces noise amplification and provides a sharper image. The output image has a lot of noise and is of high resolution, hence using CLAHE for this case felt relevant.



4. Gaussian Filtering and Bilateral Filtering also provide similar results as in Step 3. The results can be found in the jupyter notebook titled Task 3.