Task 1

Background

Task 1 required using an algorithm to align three images with one another. These three images correspond to the red, green and blue channels making up an equivalent colour image. It can be observed that these channels are each slightly different in their greyscale tones as well as they are used to represent different information.

Template matching[1] is the suggested algorithm for identifying the alignment mismatch. The algorithm works by having a pair of images, a "search image" and a "template image", where the "search image" is the image to be searched within the "template image". This search is done by sweeping the "search image" along an (x,y) offset relative to the "template image" (in units of pixels) until the correct alignment offset is found. As per the referenced paper, to implement this sweeping process, the "template image" should be padded with an external border of up to half the length or width (respectively) of the "search image" to allow the centre of the "search image" to correspond to the a corner of the "template image".

To determine that the current alignment offset is correct, an error metric needs to be used as a comparison value for the current offset. By minimising or maximising the comparison value, the most optimal offset can be found. Suggested error metrics include **SSD** (sum of squared differences) and **NCC** (normalised cross-correlation), where the metric is computed using the pixels in the "search image" and "padded template image" that overlap. As referenced from the paper, a possible mathematical implementation for each is outlined below:

Algorithm	Equation	Comparison Requirement
SSD	$SSD[u,v] = \sum_{i=0}^{M}\sum_{j=0}^{N}(f[i,j] - g[i+u, j+v])^2$	Minimise the SSD value. Perfect alignment implies 0.
NCC	$NCC[u,v] = \frac{\sum_{i,j\ \epsilon\ A}\sum(f[i,j]\cdot g[u+i, v+j])}{\sqrt{\sum_{i,j\ \epsilon\ A}\sum(g[u+i, v+j])^2}}$	Maximise the NCC value. Perfect correlation implies 1. $NCC[u,v]\ \epsilon\ [-1,1]$

For the parameters:

- *(u,v)*: corresponds to the alignment offset
- *(i,j)*: corresponds to each pixel in the overlap of the "search image" and the "padded template image" (i.e. including its padding)
- *(M,N)*: corresponds to the size of the overlap. Since the padding is also included, this should correspond to the size of the "search image".
- *(f,g)*: these functions refer to the search and template images.

Implementation

The first step of my code was to implement `getImageComponents()` to read the provided image and divided it into three partitions, to present the blue, green, and red channels respectively (from top to bottom).

Next, in Version 1 of my alignment implementation, the template matching algorithm was implemented exactly as described in the Background. I decided to write the implementation from scratch as I was unable to get `cv2.matchTemplate()` to produce a perfect offset. However, upon finishing this write-up, I found that my implementation produced identical flaws to the `cv2` library code, wherein a perfect offset could not be obtained.

To solve this, in Version 2 of my alignment implementation, I first apply **histogram matching** `histMatch()` (drawing inspiration from Lab 1) to each of the red and blue channels (with reference to the green channel) as a form of **pre-processing**. This ensures all three channels look more alike one another in their local areas of greyscale tonality and intensities. After this, my implementation of template matching is called, and the computed offset alignments are improved greatly to consistent and accurate alignment across all the provided images. On applying the actual offset

[1] Universiti Malaysia Perlis, 2015. Template Matching using Sum of Squared Difference and Normalised Cross Correlation. https://www.researchgate.net/publication/301443589_Template_Matching_using_Sum_of_Squared_Difference_and_Normalized_Cross_Correlation

alignment, the original red and blue channels are used instead of their histogram matched equivalents, to not change the expected colour of the output image.

My implementation of template matching relies on the following interfaces, in this order:

- `alignComponents()`: this applies a dynamically sized border for the "template image" (green channel). Internally, `_matchTemplate()` is then called on the red and blue channels to perform the template matching algorithm and compute the offset.
- `applyAlignment()`: list slicing is done to move the red and blue channels to the correct alignment, on a new canvas that is the same size as the padded green channel (template).
- `getCropAlignment()`: list slicing is done to remove the padding, using the green channel as the reference for the cropped image.

Results

Image 1 runtime parameters: `TEST_TASK_1 = True; TASK_1_IMAGE = 's3.jpg'`

Figure 1 shows the histogram matched pre-processing of the red and blue channels. It is shown that their histograms now match that of the green channel, and that, for instance, the dark cabinet in the original blue channel is now closer in intensity to that of the green channel.

Figure 2 compares the output of the of the template matched aligned image to just simply calling `cv2.merge()` on the raw output of `getImageComponents()`, which just returns the red, green and blue channels.

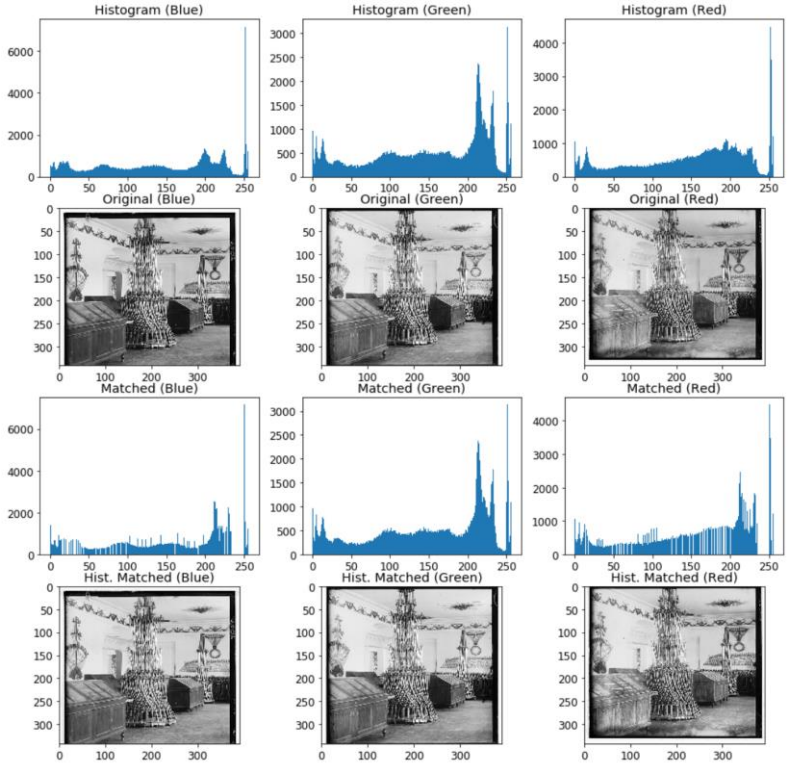

Figure 1. Histogram matched pre-processing to match green channel.

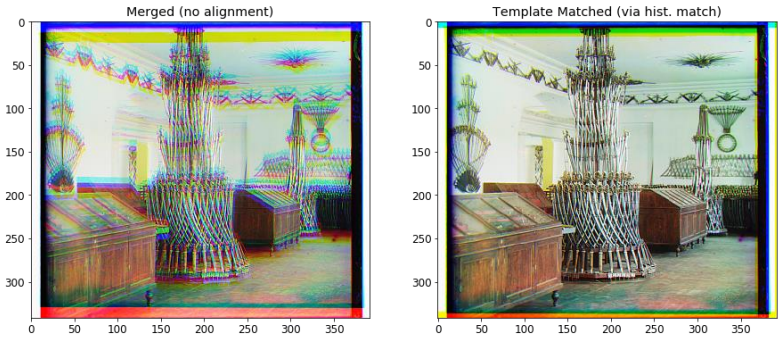

Figure 2. Results of image alignment

Image 2 runtime parameters: `TEST_TASK_1 = True; TASK_1_IMAGE = 's4.jpg'`

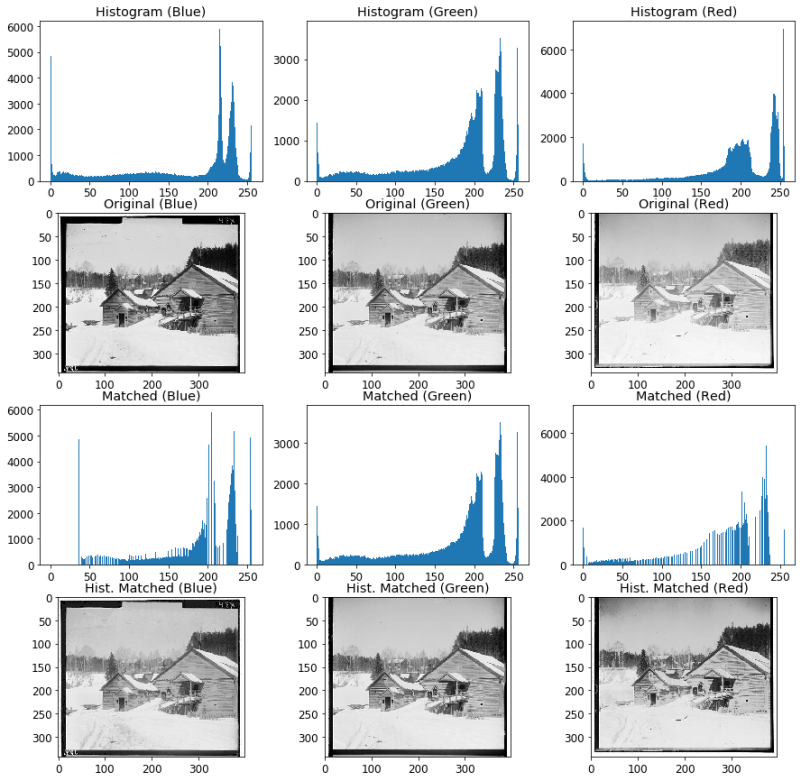

Figure 3. Histogram matched pre-processing. This example is even more pronounced, with all three images looking more similar after matching.

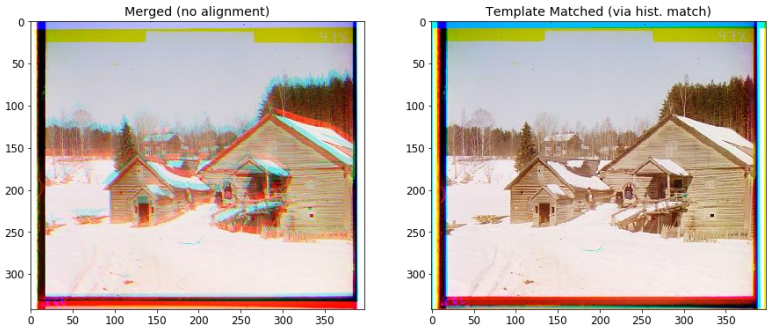

Figure 4. Results of image alignment

Selection of Metric

Empirically, I found that both my implementation of SSD and NCC worked well under my Version 2 implementation across all provided images (s1-s5), but that my NCC was unacceptably slow (due to the increased computation requirements). As such, I decided to continue using SSD. However, ignoring computation requirements, it is said that NCC is a better metric as it is more robust to intensity differences between a pair of images.

Task 2

Background

An image pyramid allows for high resolution images to first be downscaled to significantly lower resolutions (often by factors of 2, down to a minimum size of a 1x1 pixel) before template matching is run on it, for computational efficiency. The estimated alignment offset will be coarse at a low-resolution level, so template matching should then be run on each subsequent level of resolution upwards (back to the original resolution) to fine-tune the offset estimate. In addition, the result of low-resolution levels should also be used to fine-tune and reduce the region of interest needing to be scanned at the next higher-resolution level. This reduces the computational penalty of having to analyse a higher-resolution image.

Implementation

My implementation uses a user-defined parameter TASK_2_ATTEMPTS to determine how many layers to downscale by as I believed that down-scaling to a 1x1 pixel would not provide any meaningful estimation result. For the purposes of my testing, it is set as: `TASK_2_ATTEMPTS = 10`.

For each attempt (i.e. pyramid iteration), a downscale factor, and a dynamic region of interest window size for the template matching (controlled by parameter TASK_2_BORDER_WINDOW_SHRINK_RATIO), is computed and then applied in `pyramidAlignComponents()`. Here, `cv2.resize()` is called to downscale all three channels. The underlying `alignComponents()` function is then called to accept these new channels and the additional Task 2 parameters. These parameters eventually make their way to `_matchTemplate()` where the window size for the region of interest is centred to the previous alignment offset and progressively reduced in size, depending on what the current iteration is.

Results

Image 1 runtime parameters: `TASK_2_IMAGE = '00549u.jpg'; TASK_2_ATTEMPTS = 10;`
`TASK_2_RUNTIME_SECONDS_THRESHOLD = 2; TASK_2_REDUCE_ROI = True; TASK_2_BORDER_WINDOW_SHRINK_RATIO = 0.5`

Figure 3 shows the results for each successive layer of the image pyramid (from 2^9 to 2^0). The area of the window size may change in piecewise manner to accommodate different padding requirements for small images (this can be seen in the debugging print output if the code is run). Unfortunately, if the "shrink ratio" parameter is increased further, my implementation does not perform optimally with regards to window sizing readjustment, and alignment errors reappear at higher-resolution levels. However, this issue does not occur if the window size is strongly restricted (or if restrictions are entirely disabled, which can be done by setting: `TASK_2_REDUCE_ROI = False`). This appears to suggest that my image pyramid provides limited computational benefits if it is to be automated and utilised across multiple pyramid levels (as opposed to just applying the result of the lowest resolution level that works). Consequently, I cannot run the image pyramid algorithm all the way to 2^0.

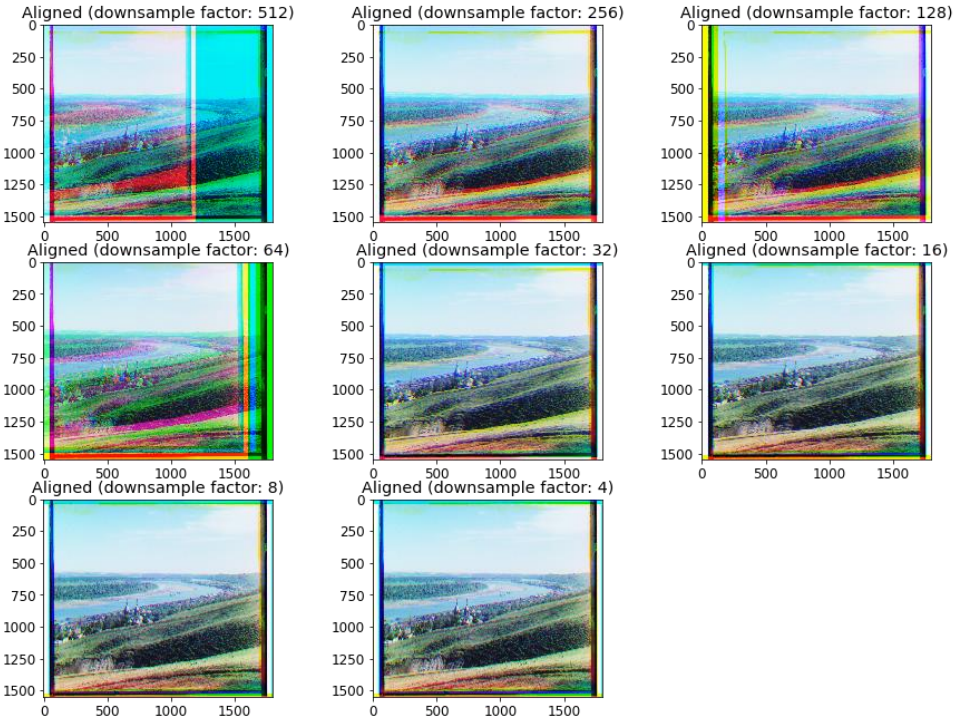

Figure 3. Layers of the image pyramid. Each higher-resolution layer relates to the previous lower-resolution layer by the shrinkage of the region of interest window size, used in template matching.

Image 2 runtime parameters: `TASK_2_IMAGE = '00911u.jpg'; TASK_2_ATTEMPTS = 10;`
`TASK_2_RUNTIME_SECONDS_THRESHOLD = 2; TASK_2_REDUCE_ROI = True; TASK_2_BORDER_WINDOW_SHRINK_RATIO = 0.125`

Figure 4 shows the results for Image 2. Unfortunately for my implementation, the results of 2^5 and 2^4 do not appear to assist in convergence of the image pyramid result. Consequently, I have had to decrease the window shrink ratio (i.e. decrease the rate at which the window shrinks) to allow the higher-resolution layers to correct for this image.

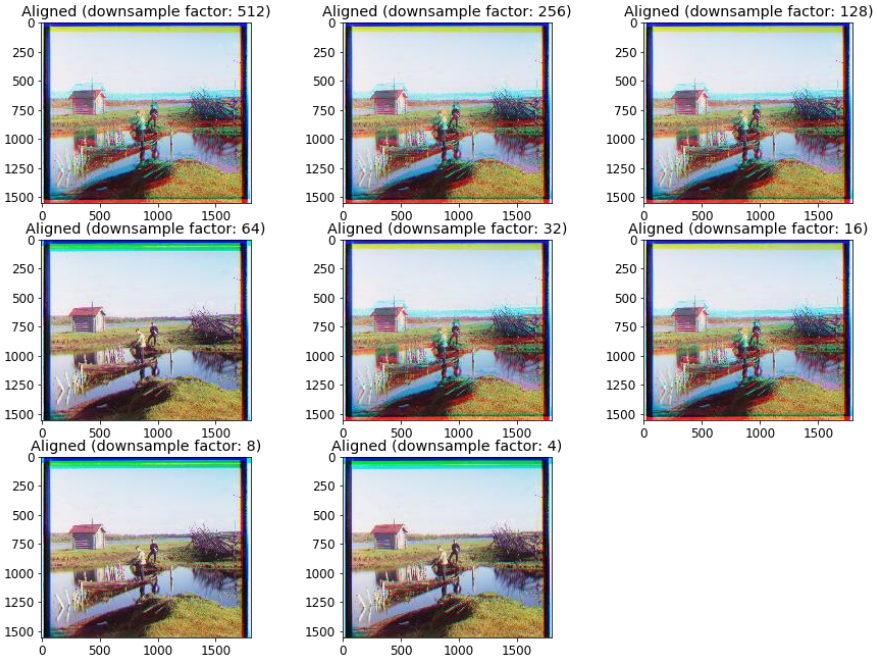

Figure 4. Layers of the image pyramid. The desired alignment result appears at a down-sampling factor of 64 but severe alignment errors then reappear at factors of 32 and 16.

Task 3

Background & Enhancement Techniques

I wanted to improve the detail of the images as I felt that they looked quite dim. After some online searching and experimentation, I came across an algorithm called CLACHE (Contrast Limited Adaptive Histogram Equalization) which **improved the details and contrast in the image** substantially (as compared to simpler or more general-purpose techniques covered in our course like power law transform).

CLACHE[2] is a type of histogram equalisation, that to my understanding, takes only a portion of the histogram (of the original image) with the most significant range of intensity values and spreads that across the x-axis (intensity levels). This causes clipping of the other values (and this can occur for multiple iterations), where some redistribution process is applied to more effectively utilise the information in those pixels. CLACHE can be applied in OpenCV by converting the image from BGR to LAB format and applying `cv2.createCLACHE().apply()` on the L-channel.

Results

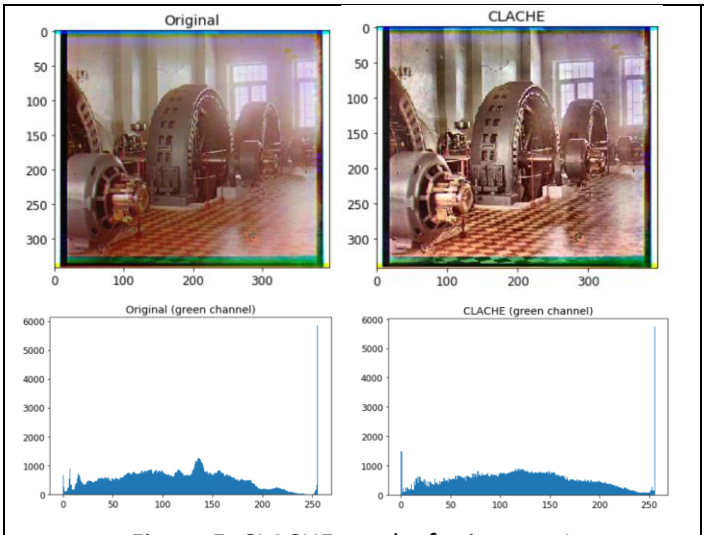

Figure 5. CLACHE results for image s1

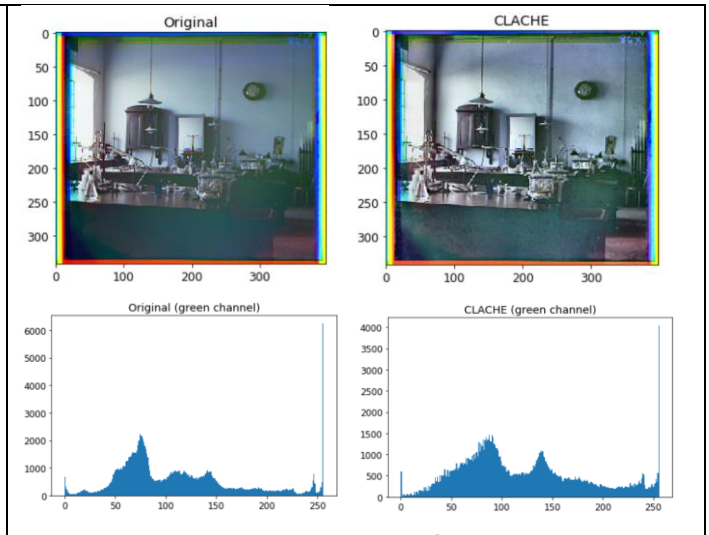

Figure 6. CLACHE results for image s2

[2] S. Philip. CS6640 – Project 2 (CLACHE). http://www.cs.utah.edu/~sujin/courses/reports/cs6640/project2/clahe.html