COMPUTER VISION ASSIGNMENT 1

JUSTIN DE WITT* (21663904)

4 August 2022

CONTENTS

LIST OF FIGURES

1 USING A HYPERBOLIC TANGENT FUNCTION TO APPLY CONTRAST STRETCHING

In this section we would like to design a point transformation based on the hyperbolic tangent function:

$$T(s) = c_1 \tanh(c_2 s - c_3) + c_4$$

which applies *contrast stretching* to an input image. Contrast stretching involves deterministically brightening and darkening certain pixels.

1.1 Introduction to the Hyperbolic Tangent Function

The hyperbolic tangent function is defined for all input values of **x**, producing an output value on the interval (-1 1). Consider the plot of $\tanh(x)$, we would like to transform this plot such that the shape of $\tanh(x)$ is maintained, but rather maps pixel intensities on the interval [0-255] to an output intensity defined on the same interval.

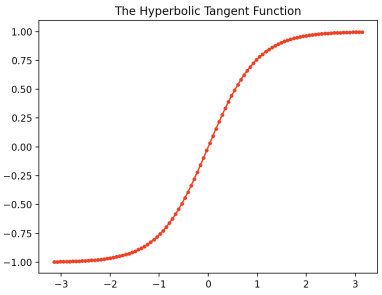


Figure 1: Hyperbolic Tangent

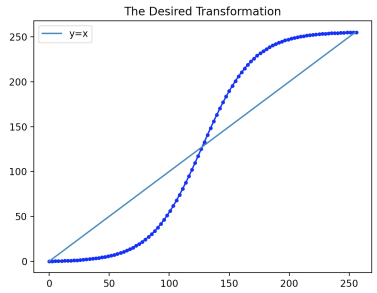


Figure 2: Desired Transformation

1.2 Properties of Using Hyperbolic Tangent in the Context of Contrast Stretching

To increase the contrast of an input image, we would like to darken some pixels in the input image, and brighten others. In figure 2, the inflection point is located at approximately $(127, 127)$, the point where the line $y = x$ intersects the function. Input pixels (**x**-axis) which lie to the left of the inflection point will be darkened in the output image, and pixels to the right will subsequently be brightened in the output image. Contrast stretching will be centered around the inflection point, the most noticeable results will be obtained when the "average colour" of the input image is roughly 127. For the purpose of this report, I have gamma-transformed (see code) my input image until It displays this property.

1.3 Manipulating the Shape of the Transformation

To obtain the desired transformation, the effect that changes to each constant $(c_1,\ c_2,\ c_3,\ c_4)$ impose on the shape of the transformation function should be determined.

- c_1: Changes to the constant c_1 affect the height of the function. If we would like to vertically stretch the hyperbolic tangent function, we can increase the value of c_1.
- c_2: Affects the "steepness" of the hyperbolic tangent function. If we increase the value of c_2 the slope around the inflection point increases. Decreasing the value of c_2 has the opposite effect.

- c_3: If we would like to translate the plot horizontally, we will change the value of c_3. If we increase c_3, the plot will move rightwards, if we decrease c_3, the plot moves to the left.
- c_4: Similarly, c_4 translates the plot in a vertical direction. Increasing c_4 moves the plot vertically, and subsequently, decreasing c_4 moves the plot downwards.

These results are graphically presented in the following figures.

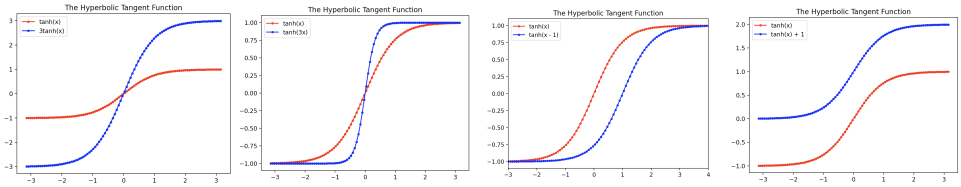


Figure 3: Visualization of the effects of (c_1, c_2, c_3, c_4)

1.4 Deriving a Family of Transformations

In my derivation, I would like the inflection point to always be centered at $(127, 127)$. Furthermore, I would like the curve to intersect the points $(0, 0)$ and $(255, 255)$. To accomplish this task, I derived the following tunings:

- $c_1 = \frac{-c_4}{\tanh(-c_3)}$
- $c_2 = \frac{c_3}{c_4}$
- c_3 is treated as a free variable.
- $c_4 = 127$

The use of the above constants identify a family of hyperbolic tangent curves suitable for contrast stretching. If we increase the value of the free variable c_3, we obtain a curve with a more higher slope around the inflection point, which implies more extreme contrast stretching around the inflection point. The family of curves is plotted below, for various values c_3.

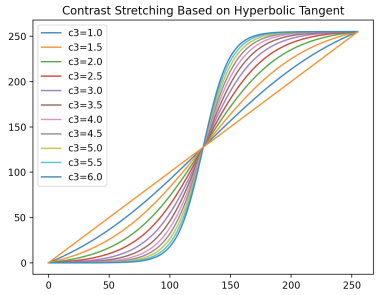


Figure 4: A Family of Contrast Stretching Curves

c_1	c_2	c_3	c_4
167.41	0.0078	1	127.5
140.86	0.0117	1.5	127.5
132.25	0.0156	2	127.5
129.23	0.0196	2.5	127.5
128.13	0.0235	3	127.5
127.73	0.0274	3.5	127.5
127.58	0.0313	4	127.5
127.53	0.0353	4.5	127.5
127.51	0.0392	5	127.5
127.50	0.0431	5.5	127.5
127.50	0.0470	6	127.5

1.5 Applying the Function to perform Contrast Stretching

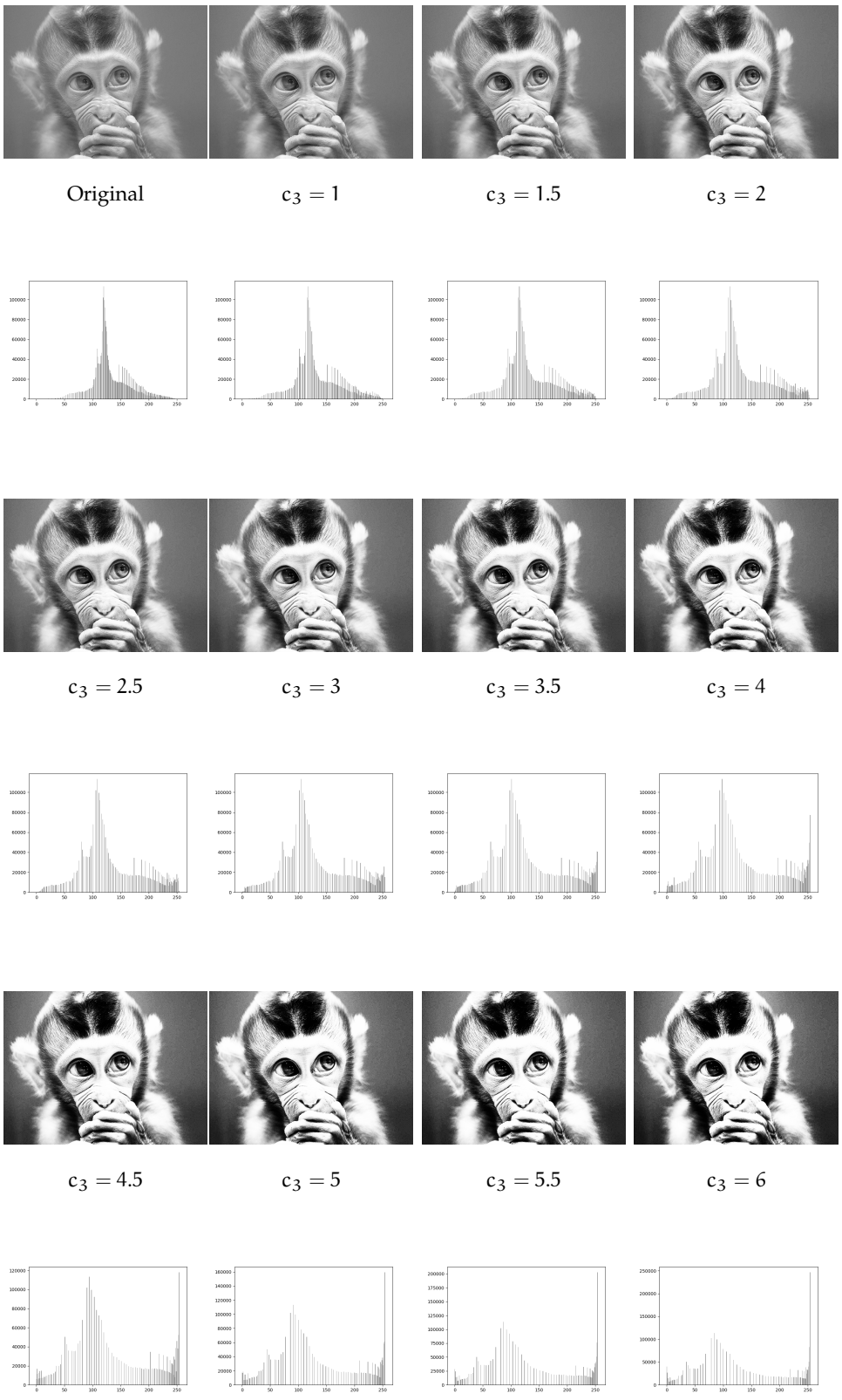


Figure 5: Results after the contrast stretching application

As suggested previously, we expect the intensity of brightening and darkening is directly related to the magnitude of the gradient to the left and right of the

transformation curve's inflection point. Using figure 4, we expect the intensity of contrast stretching to be directly related to the constant c_3. The results shown in the figure above illustrate this finding. Notice the increase in relative frequency of bright and dark pixels in the image histograms associated with large values of c_3. This change in relative frequency is a result of the contrast stretching, because we have darkened the pixels to the left of the inflection point of the function, and brightened pixels to the right.

2 IMAGE DENOISING WITH MEDIAN FILTERS

In this section, we would like to corrupt a coloured input image with "salt and pepper" noise, then attempt to restore the corruption using convolution with a median filter.

2.1 Image Corruption

We begin the corruption process by splitting the input image into its red, green, and blue channels. Once the separate colour channels are obtained, we will iterate over each pixel of the channel. For each pixel in the iteration, we generate a random uniform number on the interval $[0, 1)$. If this number lies below a corruption threshold ($\frac{d}{2}$), the pixel value will be set to black. If the generated number lies above a corruption threshold ($1 - \frac{d}{2}$), the pixel value is set to white. Once each colour channel has been independently corrupted, the channels are merged into a corrupted *RGB* image. This type of corruption is referred to as "salt and pepper" noise. Consider the figures below.

Original d=0.2 d=0.4 d=0.6

Figure 6: Original Image, along corrupted copies

The slurry of bright pixels added during the corruption process appears as a result of the addition or removal of entire colour channel components to each pixel of the original image. Each pixel is effectively split into the three separate colour components, where each component value has a non-zero probability of translating to either 255, or 0. After the (potential) translation, the components are merged into a single *RGB* pixel. The corruption of each pixel is therefore associated with the addition or removal of red, green, or blue light, which alters the appearance of the original pixel. It is evident by the figures that the corruption intensity is directly related to the corruption threshold d. In the following subsections we will attempt to restore the image by using convolution with a median filter.

2.2 Image Restoration Procedure

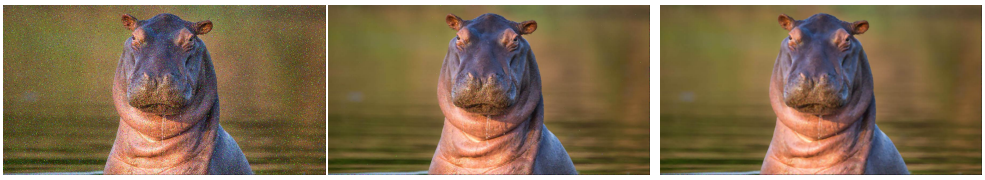
To remove corrupt pixels, we will implement a procedure that attempts to remove outlying values. The technique involves iteratively sliding a window of dimension $\mathbf{k}$x$\mathbf{k}$ centered at each pixel of the corrupted colour channel. During each iteration, the pixels contained within the window are ordered in increasing order of intensity, and the median intensity is determined. If an index of the window is out of bounds of the image, a "zero" is substituted in this position. The identified

median value is written to the corresponding position of the output image. The idea is that the median function is not effected by the magnitude of outlying values. The three obtained output channels are then merged into a single *RGB* image. This image is a restored version of the corrupted image.

2.3 Restoration Results

In this subsection results are displayed in three categories: restoration of minor damage, restoration of moderate damage, and the restoration of severe damage.

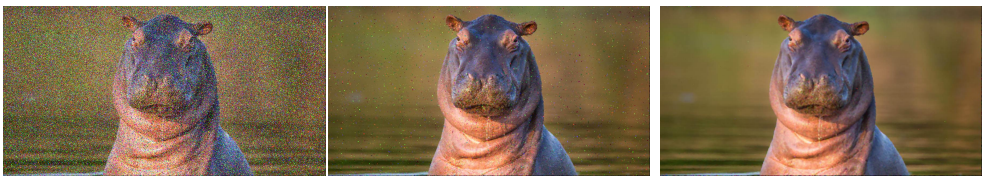
2.3.1 Correcting Minor Corruption (d=0.2)



3x3 Median Filter 5x5 Median Filter 7x7 Median Filter

Figure 7: Restoration of Minor "Salt and Pepper" Noise

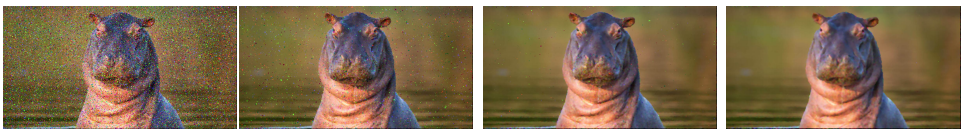
2.3.2 Correcting Moderate Corruption (d=0.4)



3x3 Median Filter 5x5 Median Filter 7x7 Median Filter

Figure 8: Restoration of Moderate "Salt and Pepper" Noise

2.3.3 Correcting Severe Corruption (d=0.6)



5x5 Median Filter 7x7 Median Filter 9x9 Median Filter 11x11 Median Filter

Figure 9: Restoration of Moderate "Salt and Pepper" Noise

2.4 The Median-Filtration Trade-off

As evident by the results provided in the previous subsection, convolution with a larger median filter removes more noise, but also has the added side-effect of blurring the image. For minor and moderate corruption, I think the 5x5 median filters were most effective in removing noise, while maintaining image resolution. There is still residual noise in the output image, but not enough to distract the viewer. For the severe corruption, I think the 9x9 filter provided the best results.

3 IMAGE SHARPENING THROUGH UNSHARP MASKING

In this section we will sharpen a gray scale input image by performing unsharp masking. Unsharp masking is a technique in which we use convolution to blur the input image, and then use this blurred image to obtain an edge image. The original image is subsequently sharpened by repeatedly adding the edge image.

3.1 Blurring The Image with Convolution

To blur the input image, we will apply convolution similar to that described in the previous section, however instead of computing the median, we will compute the mean. This is equivalent to simply applying traditional convolution with the following mask.

$$M = \begin{bmatrix} 1/n & 1/n & 1/n & 1/n & ... & 1/n \\ 1/n & 1/n & 1/n & 1/n & ... & 1/n \\ 1/n & 1/n & 1/n & 1/n & ... & 1/n \\ 1/n & 1/n & 1/n & 1/n & ... & 1/n \\ .. & .. & .. & .. & .. & .. \\ 1/n & 1/n & 1/n & 1/n & ... & 1/n \end{bmatrix}$$ Where n represents the sum of all matrix entries.

The normalization of the mask (division by n) is required to ensure that the image is not brightened or darkened during the convolution.

3.2 Applying the Procedure

We begin the procedure by blurring the input image with the averaging mask. For the purpose of this assignment I am using a 5x5 averaging mask. The width of the blurring mask if directly proportional to the blurring intensity. After experimenting with 3x3, 7x7, and 9x9 averaging masks, I determined that the 5x5 averaging mask yielded the best results. To obtain the edge image, simply subtract the blurred image from the original image. This will result in an image containing the difference between the blurry and original image (the edges). Consider the figures below.

Original Image | Blurred Image | 3 * Edge Image

Figure 10: Obtaining the Edge Image

Once the edge image has been obtained, it can be repeatedly added to the original image. Adding the edge image to the original image increases the visibility of the fine details lost during the blurring process. Naturally, repeatedly increasing pixel values may lead to pixels exceeding the [0, 255] threshold. In this scenario, pixel values are capped to the walls of this threshold. This is superior to an interpolation based technique because we do not want to change the colour of the entire

image, which is a side effect of mapping the obtained intensities back on the interval [0, 255]. Furthermore, an interpolation based technique will reduce the colour granularity of the output image, because a wider domain of pixel intensities are mapped to a smaller (integer) domain. Floating point numbers will therefore map to the nearest (and frequently the same) integer. Clipping the values on the interval [0, 255] mitigates these issues.

The results of the unsharp masking procedure for various values K are shown in the following figures. The constant K corresponds to the amount of times the edge image was added to the original.

K=2 K=3 K=4 K=5

Figure 11: The Results of Unsharp Masking for Various Values K

4 BILINEAR INTERPOLATION AND NEAREST-NEIGHBOR TECHNIQUES OF IMAGE RESIZING

In this section we would like to resize an input image, by a specified factor S. The resize procedure involves initializing an empty image of the desired size, then mapping each pixel in the empty image, to a corresponding pixel in the input image. There are two techniques to determine the value written to the output image, *Bilinear Interpolation* and *Nearest-Neighbor Interpolation*.

4.1 Applying the Inverse Mapping

To determine the mapping from output space, to input space, we can simply multiply the coordinates (treated as a vector) of each pixel in the output image, as follows:

$$\mathbf{z}' = \begin{bmatrix} \frac{1}{s} & 0 \\ 0 & \frac{1}{s} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

which yields the floating point coordinates $\mathbf{z}'$ of a pixel in the original image. If we work with a "forward" mapping, some coordinates in the output image will not be mapped to, which leads to a black mesh covering our image. By working "backwards" we negate this issue.

4.2 Nearest-Neighbor Interpolation

For Nearest-Neighbor Interpolation, we can simply round $\mathbf{z}'$ to the nearest integer indices, and map the value in the original image at the specified coordinates to the output image at coordinates (x, y). This approach does resize the image, but it yields lower resolution granularity, because duplicate mappings (as a result of floating point rounding) lead to successive runs of the same pixel colour (essentially increasing pixel size). To combat this problem, *Bilinear Interpolation* computes the

output pixel value as a function of the pixels in the neighborhood of the original pixel.

4.3 Bilinear Interpolation

In Bilinear Interpolation, the value written to the output image is a weighted average of the four corner pixels surrounding the point that was mapped to by the inverse transformation. To determine the value written to the output image, we use the following equation:

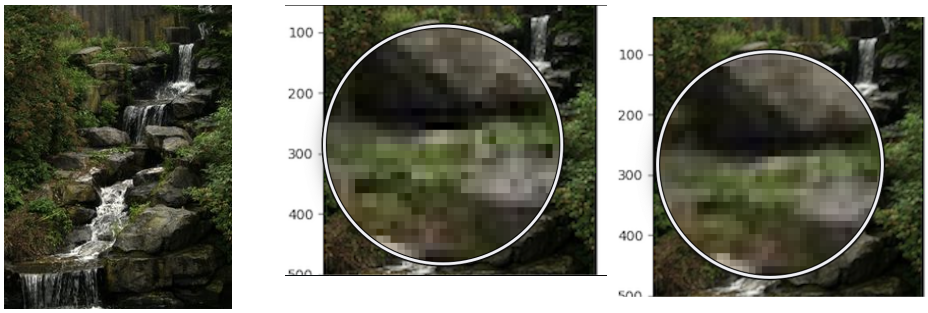
$$\begin{aligned} B(x_{new}, y_{new}) = y_c - y_{old}[(x_c - x_{old})A(x_f, y_f)] + (x_{old} - x_f)A(x_c, y_f)] \\ + (y_{old} - y_f)[(x_c - x_{old})A(x_f, y_c) + (x_{old} - x_f)A(x_c, y_c)] \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{Where } x_f = \lfloor x_{old} \rfloor,\ y_f = \lfloor y_{old} \rfloor \\ &x_c = x_f + 1,\ y_c = y_f + 1 \end{aligned}$$

To map pixel values from an input image (A) to an output image (B).

4.4 Results of Image Resizing

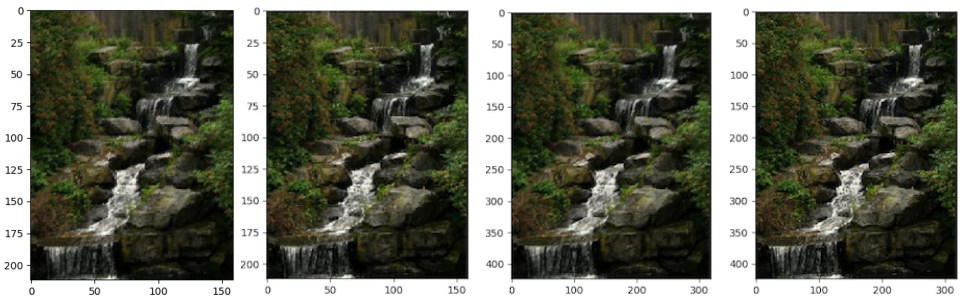
Notice that the two resizing techniques result in output that appear different in resolution. In Nearest-Neighbor Interpolation (NN) we essentially increase pixel size, which leads to a grainy image. In Bilinear Interpolation (BI), we restore some of the lost granularity by mapping each pixel value to some aggregation of the neighboring values. The difference is clear. In the following figure a circular magnification lens is used to highlight the differences between the techniques.



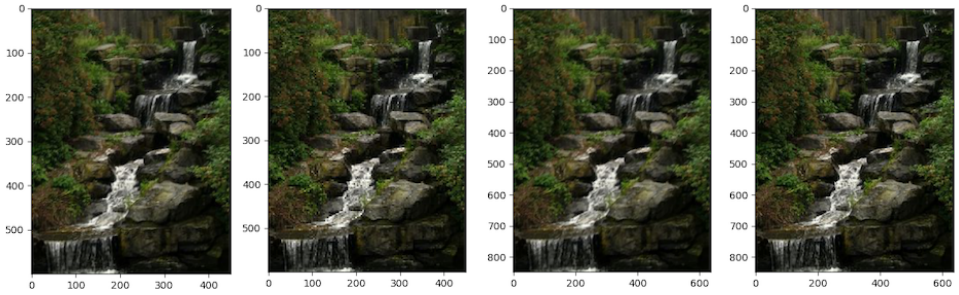
Original Image S=4 with NN s=4 with BI

Figure 12: Obtaining the Edge Image

More results are shown in the following figures. Notice the change in the magnitude of each axis. In my program I have decided to interpret the magnification scale (S) as a pixel count multiplier. This implies that $S = 2$ suggests that the output image should contain twice the pixel count of the input image. We obtain this result by resizing the dimensions of the image by a factor of $\sqrt{S}$. This rooted factor is subsequently used when working out the inverse matrix of the previous section.



S=0.5 with BI S=0.5 with NN S=2 with BI S=2 with NN



S=4 with BI S=4 with NN S=8 with BI S=8 with NN

Figure 13: Resizing the Original Image by Various Size Factors

5 IMAGE FEATURE DETECTION AND MATCHING

In this section, we are presented two images of the same environment, taken from slightly different perspectives. Our task is to match features found in both images, and display the mapping in an intuitive manner.

5.1 Identifying Key Features

There are multiple libraries available which implement a range of feature detection algorithms. One of the most famous feature detection algorithms is the *Scale Invariant Feature Transform* (SIFT). SIFT furnishes identified key points with quantitative information (descriptors) which can be used for object recognition between photos [1]. To identify the key points in an image I have used python's *OpenCV* [2] computer vision library, which contains an implementation of the SIFT algorithm. OpenCV makes it simple to obtain a list of feature coordinates using the *detectAndCompute()* function on a SIFT object. Identified feature coordinates can then be plotted on the original image, for visualization purposes. The output of this algorithm is shown in the following figures.

Image 1 Image 2 Keypoints of Image 1 Keypoints of Image 2

Figure 14: Resizing the Original Image by Various Size Factors

5.2 Matching Descriptors Between Images

Now that a collection of key points has been identified for each image, we must identify the key points in the first image that correspond to key points in the second image. The intended output is a list of key point pairs, where each pair represents a key point mapping between the images. To accomplish this task we will again use a function found in the OpenCV library, the *flannBasedMatcher* and *knnMatch()* methods.

5.2.1 Flann Based Matcher

Traditionally, SIFT feature descriptors are compared and matched using techniques based on Euclidean distance. For each key point in the first image:

1. Find the Euclidean distances between the reference descriptor and all key point descriptors of the other image.
2. Arrange them in ascending order.
3. Set some threshold T (mostly in the range of 0.3 to 0.7, I used 0.6).
4. Take the ratio of the first nearest Euclidean distance to the second nearest distance, if it is below the threshold T, then it is a match.

Arandjelovic et al [3] proposed a square root (Hellinger) kernel instead of the standard Euclidean distance to measure the similarity between SIFT descriptors, which leads to a dramatic performance boost. The *flannBasedMatcher* uses techniques similar to that described above, but rather uses the proposed Hellinger distance instead of standard Euclidean distance. To do the matching, the *knnMatch()* function from OpenCV was used. This function finds the k best matches for each descriptor in the first image. In this implementation, I have chosen to use $k = 2$. If I found that a match was incorrect, more often than not, the second match was correct.

5.3 Displaying The Results

Once the matches have been obtained, the returned object contains coordinate information, which can be used to identify the location of each feature index of the matched pair. To display the detected feature matches, we would like to draw a circle (on image 1) at the location of the feature in image 1. Next, a line emanating towards the location of the feature in the second image is drawn. To draw on images, we can use the *ImageDraw* library from python-Pillow [4]. Consider the results in the following figure.

Image 1 Image 2 Match Mapping

Figure 15: Mapping Identified Matches Between Image 1 and Image 2

Notice that the difference between Image 1 and Image 2 is a rightward change of perspective. This is especially evident if you notice the building at the top left corner, barely visible in the first image, but clearly visible in the second. This is change in perspective is illustrated when viewing the same building in the feature matched image.

6 FEATURE MATCHING ACCURACY AGAINST SCALE CHANGE

In this section we would like to test the robustness of the SIFT algorithm for variation in image scale. Naturally, since SIFT is a *Scale Invariant Feature Transform* we would expect the algorithm to perform consistently, but in practice this might not be the case.

6.1 Analytical Procedure

To test the robustness of SIFT, we need to produce multiple resized versions of an input image. The Bilinear Interpolation method (see section 4) is used during the image resizing. This procedure outputs a series of image-pairs, where the first entry in the pair is the original image, and the second entry is the resized version of the original image. If we would like to visualize the identified matches, it makes sense to draw a circle on the point corresponding to the larger image, and a line emanating towards the point associated with the smaller image. One such image pair is shown in the following figure.

Figure 16: Original Image

Figure 17: Resized by S=1.5

For the purpose of accuracy testing, we do not want incorrect matches to skew the results, so I have tightened the matching threshold discussed in the previous section, to 0.4 instead of 0.6. To determine how resistant the SIFT algorithm is to changes in image scale, we can inverse map each identified feature in the output image, to the true corresponding pixel in the input image using the techniques discussed in section 4. We will then compare the obtained "true" location with the matched position in the original image. If the Euclidean distance between these points is less than 1 (due to floating point casting), we consider the match accurate. The procedure will be repeated for each value $S \in [0.1, 3]$ with step size 0.1.

6.2 Results

The results were almost exactly as expected. The SIFT algorithm appears to accurately identify key point matches regardless of the scale of the second image. The following figure presents plots which track the degree to which the SIFT algorithm was successful. Accuracy is simply measured as the number of successful matches as a proportion of the total identified matches. The two plots are of the same data, however, the **y**-axis scale is simply adjusted between plots to illustrate the degree to which the algorithm's performance was consistent.

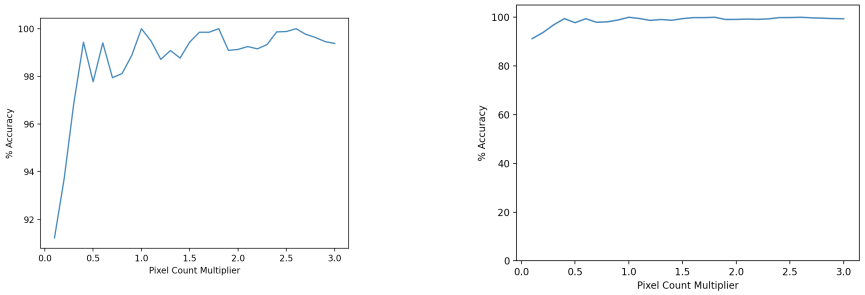


Figure 18: SIFT Algorithm Robustness to Scale Variation

7 REFERENCES

1. Weitz, Edmund. "SIFT - Scale-Invariant Feature Transform." SIFT - Scale-Invariant Feature Transform.

2. Bradski, G. (2000). The OpenCV Library. Dr. Dobb; Journal of Software Tools.

3. Relja Arandjelovic. Three things everyone should know to improve object retrieval. In Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), CVPR '12, pages 2911–2918, Washington, DC, USA.

4. Clark, A. (2015). Pillow (PIL Fork) Documentation. readthedocs. Retrieved from https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf