# ECE 113DB Final Project: Underwater Image Enhancement

Jonathan Nguyen, Bruno Bombassaro
*University of California, Los Angeles 90095*
*Department of Electrical and Computer Engineering*
{nguyen.j, brunobombassaro}@ucla.edu

***Abstract*—In this paper we explore an implementation of the underwater image enhancement algorithm suggested by Ancuti et. al [1]. Pictures taken underwater suffer from a multitude of issues, not limited to the attenuation of red and blue light as well as haziness from particles in the water which our project aims to correct. We have implemented the vast majority of the algorithm in C, intended for implementation on a portable embedded system, with only image reading and displaying done in MATLAB. From the results it is unambiguous that the algorithm explored in this project is very useful in enhancing the quality of underwater pictures.**

***Index Terms*—color correction, grey world, haze removal, image fusion, image processing, underwater image, white balancing**

## I. Introduction

### A. History

The advent of digital imaging has allowed for the processing of images after they have been taken. By sampling a camera sensor, a discrete and adjustable representation of an image can be obtained and manipulated in the future. Using this digital representation of an image, many algorithms have been devised to enhance or utilize them including haze removal [2] [3], image segmentation [4], image recognition [5], and underwater image enhancement [6] [1] [7].

Previously, underwater enhancement algorithms focused mostly on dehazing, utilizing algorithms designed for images taken above water. While successful in removing haze due to particulate matter such as algae and dirt, these algorithms did not remedy underwater specific issues such as refraction and the attenuation of the red and blue color channel channel. As such, early underwater enhancement algorithms utilized dehazing algorithms cascaded into specific color / refraction algorithms. It was not until later on such as in papers including [1] that a specific algorithm designed for underwater images arose. An underwater specific algorithm has the benefit of removing any redundant steps. For instance, if the dehazing algorithm and refraction algorithms both needed to calculate an edge map, applying both algorithms separately would calculate it twice. Additionally, tailoring an algorithm for underwater images allows it to target specific areas of interest such as refraction or algae.

### B. Global Constraints

The main constraint we faced in this project was memory usage. Since our project utilizes RGB images and performs multiple operations on it, we need to keep multiple copies of the image in memory. In fact, if we were to fully optimize our system, we would still require at least 4 copies of the image. As such, the memory use quickly scales with image size. Take for example our test image (1024 x 768 x 3 pixels). Assuming we keep four copies of it in memory and store it as a 32-bit floating point number, we would need approximately 37 megabytes of memory. This is clearly much larger than the 500 kilobytes of RAM on the H7. Therefore, we switched our focus to implementing this on our laptops which have significantly more RAM. One workaround that we may consider is to write the intermediate results to non-volatile memory such as a disk or flash drive. The trade-off would be a significant increase in run-time but perhaps it would be worth the cost if our code is more accessible to more devices.

## II. Motivation

Existing systems for underwater image enhancement usually fall in one of two categories: 1) specialized physical equipment, such as polarization filters and specialized hardware, or 2) traditional techniques, such as white balancing and gamma correction. The system originally proposed in [1], which we implemented, develops on the latter kind. It utilizes a myriad of image processing algorithms which are then combined into a single, final image. This means working on this specific topic has allowed us to better understand a wide variety of algorithms used in image processing. Among these we may cite color spaces, grey world, histogram normalization, different types of filters such as Laplacian and Gaussian filters.

## III. Approach

In this section we will explain our team organization and development plan, as well as our theoretical and practical approach to the problem at hand. Throughout the course of this project, we utilized several tools to collaborate. This was especially important as we no longer have the opportunity to work and discuss the project in person due to the current state of the world. To this end, we utilized Notion[1] to allocate tasks and keep a neat flowchart of in-progress and upcoming work. Notion allowed us to assign each other tasks as well as document our progress as we performed it. It was especially helpful that Notion supported embedded Latex which allowed

[1]The invite link to our Notion page is: https://www.notion.so/invite/0a1ba2f61ce70ff2c4a073a8e95824ed2622d080

us to copy down key math equations for quick reference. Additionally, in order to synchronize code, we extensively utilized Git through Github[2]. Git allowed us to create a main repository for our code and create independent branches for changes. For example, if Jonathan changed the Grey World Approximation code on his branch, Bruno's branch would remain unaffected unless the changes were specifically merged.

### A. Team Organization

The work breakdown is as given in Table I.

TABLE I
TASK BREAKDOWN BY PERSON

| ***Task/Person Responsible*** | Bruno | Jonathan |
|---|---|---|
| Reading Image | | X |
| Writing Image | | X |
| Color Balancing | X | |
| Grey World | X | X |
| Gamma Correction | X | |
| Edge Sharpening | | X |
| Histogram Equalization | | X |
| Laplacian Weight | X | X |
| Saliency Weight | X | |
| Saturation Weight | X | |
| Weight Normalization | | X |
| Image Fusion | | X |
| RGB-XYZ Conversion | X | |
| XYZ-LAB Conversion | X | |
| RGB-HSV Conversion | | X |

### B. Plan

Table II shows the planned completion week of each task as well as the actual time of completion.

An important note should be made regarding reading and writing the image. Because of the intense memory use of the algorithm, the development board utilized in the system was no longer sufficient. At that point the system utilized to run the project was changed, and required further work on the reading and writing algorithms.

### C. Standard

The only de jure standard we utilized was the Adobe D65 reference white in [8]. The remaining standards, including gamma correction, color spaces (XYZ, RGB, LAB, and LMS), Grey World, Laplacian filtering, and Gaussian blurring are standard practices that are not formally stated but have become industry standard through their widespread adoption.

### D. Theory

Throughout this project, we utilized several image processing algorithms in order to recover the quality of underwater images.

[2]The GitHub repository can be found at: https://github.com/puresoda/Underwater_Image_Processing

TABLE II
TASK EXPECTED VERSUS ACTUAL COMPLETION TIME

| ***Task/Completion Date*** | Goal | Reality |
|---|---|---|
| Reading Image | W1 | W6* |
| Writing Image | W1 | W6* |
| Color Balancing | W1 | W1 |
| Grey World | W7 | W8 |
| Gamma Correction | W2 | W1 |
| Edge Sharpening | W2 | W2 |
| Histogram Equalization | W3 | W5 |
| Laplacian Weight | W5 | W4 |
| Saliency Weight | W4 | W3 |
| Saturation Weight | W4 | W2 |
| Weight Normalization | W6 | W4 |
| Image Fusion | W8 | W8 |
| RGB-XYZ Conversion | W5 | W4 |
| XYZ-LAB Conversion | W5 | W5 |
| RGB-HSV Conversion | W3 | W4 |

*1) Color Compensation:* The first such algorithm is color balancing based on the green channel of our image. This step is crucial because the red and blue color channels are attenuated the deeper we travel underwater due to their longer wavelengths. We compensate for their loss as follows:

$$I_{rc} = I_r + (\overline{I_g} - \overline{I_r})(1 - I_r) \cdot I_g \tag{1}$$

$$I_{bc} = I_b + (\overline{I_g} - \overline{I_b})(1 - I_b) \cdot I_g \tag{2}$$

$I_{rc}$ and $I_{bc}$ represent the adjusted red and blue color channels respectively. $I_g$ and $\overline{I_g}$ represent the green channel and the average value over it. One can extrapolate to understand the meanings for $I_r$ and $I_b$. The compensation is performed pixel by pixel and assumes that every pixel has been normalized in the range of $[0, 1]$.

Once the color has been compensated, we utilize the Grey World algorithm introduced in [9] to obtain a properly color-balanced picture. The Grey World algorithm works on the assumption that the average color of a normal image is gray. As such, the algorithm attempts to adjust an image to match this assumption.

*2) RGB Pixel Linearization:* The first step towards achieving this is by linearizing the RGB values in order to undo any gamma correction that may have been present. Assuming that our image is in the sRGB color space, we can use the formula given in [10]:

$$f(u) = \begin{cases} -f(-u) & u < 0 \\ c \cdot u & 0 \leq u < d \\ (a \cdot u + b)^{\gamma} & \text{otherwise} \end{cases} \tag{3}$$

Where:

$$a = 1/1.055 \tag{4}$$
$$b = 0.055/1.055 \tag{5}$$
$$c = 1/12.92 \tag{6}$$
$$d = 0.04045 \tag{7}$$
$$\gamma = 2.4 \tag{8}$$

*3) Illuminant Calculation:* Then, the linearized RGB values are converted to the XYZ color space as follows:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{9}$$

The next step is to calculate the illuminant of the source image $(X_{WS}, Y_{WS}, Z_{WS})$ [11]. This is the reference white of the image and is used to correct the white balance. At a high level, estimating the illuminant is performed on each color channel (RGB) and converted to XYZ using the matrix in equation (9). On each color channel, we quantize pixels values using a histogram with $2^{10}$ bins. We then take the average of the pixels within the $20^{th}$ and $80^{th}$ percentiles and return that as the illuminant.

Afterwards, we apply matrix transformation to shift the source image to a new reference white $(X_{WD}, Y_{WD}, Z_{WD})$. For this project, we selected the standard Adobe D65 illuminant: $(95.047, 100, 108.883)$ referenced in [8].

*4) Cone Representation:* The cone representation of an image, also known as the long-medium-short (LMS) color space is utilized to represent the relative stimulation to each cone cell involved in human vision. As its name implies, the the long cone is more sensitive to longer wavelength colors such as red. The medium cone is associated with green while the short cone is associated with blue.

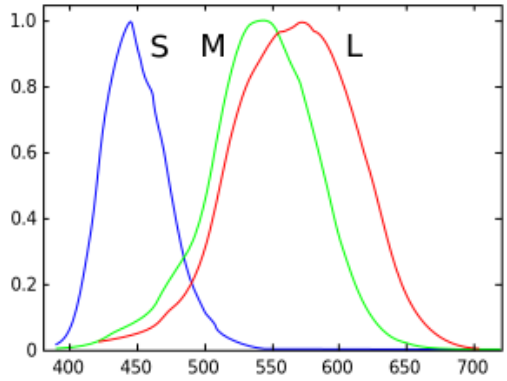


Fig. 1. Human Vision Cone Sensitivities
(By Vanessaezekowitz at en.wikipedia, CC BY-SA 3.0
https://commons.wikimedia.org/w/index.php?curid=10514373)

$M_A$ and its inverse $M_A^{-1}$ given below are the transformation matrices convert to and from the XYZ and LMS color spaces. There are many such matrices, but we chose the Bradford matrix for better perceived performance [12]. As a side note, Adobe Photoshop also utilizes the Bradford matrix.

$$\begin{bmatrix} \rho_s \\ \gamma_s \\ \beta_s \end{bmatrix} = M_A \cdot \begin{bmatrix} X_{WS} \\ Y_{WS} \\ Z_{WS} \end{bmatrix} \tag{10}$$

and

$$\begin{bmatrix} \rho_d \\ \gamma_d \\ \beta_d \end{bmatrix} = M_A \cdot \begin{bmatrix} X_{WD} \\ Y_{WD} \\ Z_{WD} \end{bmatrix} \tag{11}$$

$$M_A = \begin{bmatrix} 0.8951000 & 0.266400 & -0.161400 \\ -0.750200 & 1.7135000 & 0.0367000 \\ 0.0389000 & -0.0685000 & 1.0296000 \end{bmatrix} \tag{12}$$

$${M_A}^{-1} = \begin{bmatrix} 0.986993 & -0.147054 & 0.159963 \\ 0.432305 & 0.518360 & 0.049291 \\ -0.008529 & 0.040043 & 0.968487 \end{bmatrix} \tag{13}$$

The full transformation matrix is as follows:

$$M = {M_A}^{-1} \cdot \begin{bmatrix} \frac{\rho_D}{\rho_S} & 0 & 0 \\ 0 & \frac{\gamma_D}{\gamma_S} & 0 \\ 0 & 0 & \frac{\beta_D}{\beta_S} \end{bmatrix} \cdot M_A \tag{14}$$

To apply the full transformation to the original XYZ image:

$$\begin{bmatrix} X_D \\ Y_D \\ Z_D \end{bmatrix} = M \cdot \begin{bmatrix} X_S \\ Y_S \\ Z_S \end{bmatrix} \tag{15}$$

Finally, we convert our corrected XYZ image back to RGB, marking the end to color balancing. Note that since the conversion from RGB to XYZ (Eq. 9) was achieved by a full-rank matrix transformation, the conversion from XYZ to RGB is simply its inverse:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.041369 & -0.56495 & -0.3447 \\ -0.96927 & 1.876011 & 0.04156 \\ 0.013447 & -0.11839 & 1.01541 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{16}$$

*5) Gamma Correction:* The next step in the algorithm is gamma correction. The basic formula for gamma correction is:

$$x_g = (x_s)^\gamma \tag{17}$$

In which $x_s$ signifies the input pixel and $x_g$ represents the gamma corrected version. The variable $\gamma$ is a fixed positive constant that determines the extent of gamma correction as shown below in Figure 2.

Fig. 2. Gamma Correction with $\gamma = 1.5$ and $\gamma = 0.5$ respectively. As seen here, higher values of $\gamma$ are associated with darker shadows while lower values of gamma are associated with lighter shadows.

*6) Edge Sharpening:* The edge sharpening process is run in parallel with gamma correction. That is, each of these functions utilizes a copy of the white balanced image and runs independently of one another. Edge sharpening is given by:

$$S = (I + \mathcal{N}(I - G * I))/2 \tag{18}$$

$I$ is the input white-balanced image. $S$ is the output sharpened image. $\mathcal{N}$ denotes the linear normalization operator, also known as histogram equalization. The linear normalization operator redistributes pixel values so these values cover the entire dynamic range. $G$ represents a Gaussian filter which is used to perform blurring. It is obtained by sampling a Normal distribution with a standard deviation of $0.5$ as shown in equation 19 and Figure 3. Higher standard deviations would lead to more blurry images. Additionally, more samples of the Gaussian distribution would result in smoother blurring, but $3 \times 3$ showed good performance in our evaluation.

$$G = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix} \tag{19}$$

Fig. 3. Effect of Gaussian Blur Filtering with standard deviation of 20. A higher standard deviation was chosen since the images are relatively small.

To perform histogram equalization on an RGB image, we must first convert it to grey-scale. This is performed by converting the RGB input image to the Hue-Saturation-Value (HSV) color space. At a high level the HSV color space models how colors appear under light as opposed to the additive approach of RGB. As seen in Figure 4, the "value" dimension controls the relative brightness and darkness of an image.

We perform histogram equalization on the "value" channel to increase the dynamic range of it and then convert it back to RGB. The equations for converting from RGB to HSI are given below and in []. We have left out the conversion from HSI to RGB for the sake of brevity but they can be found in [].

$$max = \max(red, green, blue) \tag{20}$$
$$min = \min(red, green, blue) \tag{21}$$
$$\Delta = max - min \tag{22}$$

$$H = \begin{cases} 60.0 \cdot \mathrm{mod}_6\left(\lfloor \frac{green - blue}{\Delta} \rfloor\right) & \text{if } max = red \\ 60.0 \cdot \left(\frac{blue - red}{\Delta} + 2.0\right) & \text{if } max = green \\ 60.0 \cdot \left(\frac{red - green}{\Delta} + 4.0\right) & \text{if } max = blue \end{cases} \tag{23}$$

$$S = \begin{cases} \frac{\Delta}{max} & max \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

$$I = max \tag{25}$$

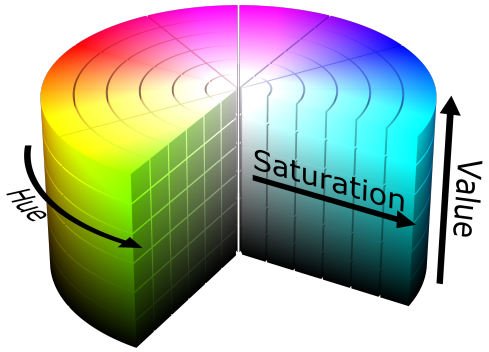


Fig. 4. HSV Color Space Cylinder Representation.
(By SharkD at en.wikipedia, CC BY-SA 3.0
https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png)

To apply histogram normalization, refer to the following pseudo-code in Algorithm 1:

**Algorithm 1** Histogram Normalization

1: Given image $I$ with pixel values $I(i)$ such that $0 \leq i < N$.
2: $I_{norm} = \lfloor 256 \cdot I \rfloor$
3: $histogram(i) \leftarrow \text{count}(I_{norm} = i)$
4: Let $csum(i) \leftarrow \sum_{k=0}^{i} histogram(i)$
5: Let $newgrey(i) \leftarrow csum(i) * 255/N$
6: $I_{norm}(i) \leftarrow newgrey(I_{norm}(i))$

*7) Laplacian Weight:* Both the gamma corrected and edge sharpened images are used to calculate three weight maps each (totalling six weight maps) to better account for specific parts of the picture.

The first weight calculated is the Laplacian contrast weight. Laplacian filters are an approximation the second spatial derivative of an image, meaning it can detect sudden changes in pixel values, especially edges. One can think of a Laplacian filter as an application of a high pass filter. To utilize this filter, one must first calculate the luminance of the image. For any given pixel, one can use either of the following equations. Equation (26) is the quicker to calculate but is less accurate (citation here). Equation (27) is more accurate at the cost of added complexity. For our project, we utilized the first equation for its simplicity.

$$l = 0.299r + 0.587g + 0.114b \tag{26}$$

$$l = \sqrt{0.299r^2 + 0.587g^2 + 0.114b^2} \tag{27}$$

From practical results, the Laplacian filter chosen was:

$$H_{lap} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{28}$$

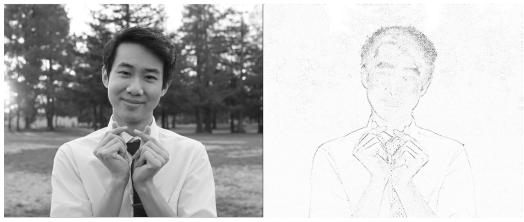

Fig. 5. Laplacian Filtering of an Image (Gaussian Blur performed prior to remove false edges) (Color has been inverted to preserve ink when printing)

This filter is applied to both the gamma correction and sharpened image using a convolution operation to obtain two Laplacian weight maps.

$$W_{lap} = |I * H_{lap}| \tag{29}$$

*8) Saliency Weight:* The next weight is the Saliency weight, introduced in [13]. For a given pixel, its saliency weight is be calculated by:

$$W_{sal} = \sqrt{(L - \overline{L})^2 + (A - \overline{A})^2 + (B - \overline{B})^2} \tag{30}$$

It must be noted from Eq. (30) that the color space used to calculate the saliency weight is not the standard RGB, but rather LAB. The RGB image must first be converted into XYZ, which is done as shown in Eq. (9).

Given the following reference white values (Note that this is the un-normalized XYZ pair used in the Grey World Algorithm):

$$\begin{cases} X_N &= 76.04 \\ Y_N &= 80 \\ Z_N &= 87.12 \end{cases} \tag{31}$$

the LAB image can be obtained as follows [14]:

$$L = \begin{cases} 116 \cdot (Y/Y_N)^{1/3} - 16 & \text{if } Y/Y_N > 0.008856 \\ 903.3 \cdot (Y/Y_N) & \text{else} \end{cases} \tag{32}$$

$$A = 500(f(X/X_N) - f(Y/Y_N)) \tag{33}$$

$$B = 200(f(Y/Y_N) - f(Z/Z_N)) \tag{34}$$

Where

$$f(t) = \begin{cases} t^{1/3} & \text{if } t > 0.008856 \\ 7.787 \cdot t + 16/116 & \text{else} \end{cases} \tag{35}$$

*9) Saturation Weight:* The final weight is the saturation weight. This weight retains information on the saturation of a pixel, which is a measure of the intensity of color of that pixel. This means that highly saturated regions will be represented by higher values. This is useful to collect information on the image colors - as opposed to the two other weights which are mainly focused on edge detection. For any given pixel, the saturation weight is given by the following. Recall that $l$ is the luminance of the pixel.

$$W_{sat} = \sqrt{1/3 \cdot [(r - l)^2 + (g - l)^2 + (b - l)^2]} \tag{36}$$

*10) Image Fusion:* Finally, we simply add each of the trio of weights to form a pair of final weight maps ($W_k = W_{lap} + W_{sal} + W_{sat}$). The final step of the algorithm is to combine the two normalized weight maps. The weight map normalization is given by:

$$\overline{W_k} = \frac{W_k + \delta}{\sum_{k=1}^{2} W_k + 2\delta} \tag{37}$$

Where $\delta$ is a small regularization term used in order to ensure that both weight maps have at least some representation. For our project, we selected $\delta = 0.1$. The normalized weight map is finally applied to the white balanced image where $\odot$ is the Hadamard product, also known as element-wise multiplication.

$$I_{corrected} = I_{white} \odot \overline{W_k} \tag{38}$$

*11) Notes:* Although not mentioned in [1], the results of our implementation of the image fusion algorithm produced slightly darker images. As a result, we added an extra step of gamma correction with $\gamma = 0.7$ to brighten the final result.

### E. Software / Hardware

We first attempted to use the NUCLEO-H743ZI2 development board to perform the majority of our algorithm. However, because of aforementioned memory limitations of the system, we were forced to shift our implementation to our personal computers.

Furthermore, we utilized MATLAB to transform our input images into readable text files to be fed into our C system, and the output of this system was also handled in MATLAB to be displayed. This was done because image reading is not natively supported in C. All popular image file formats utilize some form of compression and undoing the compression in C was beyond our expertise. As a result, we use MATLAB to perform this decompression and read in raw pixel values.

The below was utilized to transform an image file into raw RGB values to be inputted to our C code:

```
function rgb_image = im2bitmap(filename)
%IM2BITMAP Given an image file, write each
    RGB value to a text file
%  The output will be a flattend version of
    the image in the following
%  format [R1 R2 R3 ... G1 G2 G3 ... B1 B2
    B3] where all pixel values are
%  written from left to right and top to
    bottom

% Strip the file extension
base_file = filename(1:end-4);

% Read the file
rgb_image = imread(filename);
y_dim = size(rgb_image, 1);
x_dim = size(rgb_image, 2);

% Loop through image and write each RGB value
fileID = fopen([base_file,
    '_bitmap.txt'],'w');
fprintf(fileID, '%d\n', y_dim);
fprintf(fileID, '%d\n', x_dim);

for k=1:3
   for i=1:y_dim
      for j=1:x_dim
         fprintf(fileID, '%d\n',
             rgb_image(i,j,k));
      end
   end
end

fclose(fileID);
end
```

The following C code is utilized for histogram equalization:

```
void histogramEqualization(float* intensity,
    const int num_pixels)
{
  // Create the historgram of RGB values
  int histogram[256] = { 0 };
  int new_grey[256] = { 0 };
  int cum_sum = 0;

  // Loop thorugh all pixel values, note that
      we multiply by 255 to get an integer
      representaton
  for (int i = 0; i < num_pixels; i++)
  {
    histogram[(int)(intensity[i] * 255.0f) %
        256]++;
  }

  // Calculate the new grey values to assign
  // New grey value the cumulitive
      probability at that point normalized to
      [0,255]
  for (int i = 0; i < 256; i++)
  {
    cum_sum += histogram[i];
    new_grey[i] = (int) (((float)cum_sum) *
        255.0f / num_pixels);
  }

  // Assign the new pixels
  for (int i = 0; i < num_pixels; i++)
    intensity[i] = (float)
        new_grey[(int)(intensity[i] * 255) %
        256] / 255.0f;

  return;
}
```

The next code snippet is part of the larger code which is used to calculate the illuminant of a color channel:

```
cum_sum_forward[0] = histogram[0];
cum_sum_backward[0] = histogram[(NUM_BINS) -
    1];

if (cum_sum_forward[0] > low_threshold)
   idx_low = 0;

if (cum_sum_backward[0] > high_threshold)
   idx_high = 0;

// Loop through the histogram
// Get the cumulative sum going forward and
    backwards
for (int i = 1; i < (NUM_BINS); i++)
{
   // Calculate the cumulative sums and
       update the indexes
   cum_sum_forward[i] = cum_sum_forward[i -
       1] + histogram[i];
   cum_sum_backward[i] = cum_sum_backward[i -
       1] + histogram[(NUM_BINS) - i];

   if (idx_low == -1 && cum_sum_forward[i] >
       low_threshold)
      idx_low = i;

   if (idx_high == -1 && cum_sum_backward[i]
       > high_threshold)
      idx_high = ((NUM_BINS)-i);
}

// Get the L1 norm of the pixels within our
    new range
```

```
float eps = 1e-3;
float sum = 0;
int count = 0;

for (int i = 0; i < num_pixels; i++)
{
   if (image[i] <= (3.0/2*step)*idx_high +
       eps && image[i] >= (3.0 / 2 * step)*
       idx_low - eps)
   {
      sum += ABS(image[i]);
      count++;
   }
}
```

The following snippet refers to how we finally fuse our weights together to form a single, enhanced image.

```
float* applyFusion(float* white_image, float*
    gamma_weight, float* sharp_weight, const
    int num_row, const int num_col)
{
   // Calcuate constants
   const int num_pixel = num_row * num_col;
   const int num_rgb = num_pixel *
       NUM_CHANNELS;

   // Apply regularization / normalization to
       the weights
   normalizeFusionWeights(gamma_weight,
       sharp_weight, num_row, num_col);

   float* reconstructed =
       malloc(sizeof(float) * num_rgb);

   // Apply Naive Fusion which is simply the
       Haddamard product between the image
       and combined weight
   for (int i = 0; i < num_rgb; i++)
      reconstructed[i] = white_image[i] *
          (gamma_weight[i % num_pixel] +
          sharp_weight[i % num_pixel]);

   return reconstructed;
}
```

### F. System Build / Operation

The first step to build the system is to get the image into a readable format. In this system, the choice was made the images should be made into plain text file with a list of pixel values. Then the next step was to correct the color balance, starting with white balance and then utilizing the grey world algorithm; these steps are easy to check, as the difference between input and output is very clear. Following that step, the next portion of the system concerns gamma correction, which is fairly straightforward, and sharpening - a more complicated process. That is followed by weights calculation, which follow the equations given in the previous sections. Finally, image fusion was developed to finish the enhancement process.

The utilization of the system is straightforward, consisting of three steps. An image needs to be fed into the MATLAB function to be converted into a format readable by the C code. Then, simply running that C code in a system robust enough to handle it will result in an output .txt file. This file serves as input into our other MATLAB function, which in turn outputs the final, enhanced image.

## IV. Results

This section will cover the descriptions and discussion of the results obtained from the project.

### A. Description of Results

The algorithm was tested with several pictures. Below is the resulting average run-time for our C code, broken down by task. It is important to note the MATLAB code is natively well-optimized, so the run-time for that specific portion is relatively short, and has little effect on the overall project.

TABLE III
AVERAGE RUN TIME OF OUR C CODE

| Task | Average Runtime (sec) |
|---|---|
| Reading Image | 14.883 |
| White Balance | 0.327 |
| Gamma Correction and Weights | 0.568 |
| Image Unmask Sharpening and Weights | 0.706 |
| Image Fusion | 0.033 |
| Writing Image | 4.789 |
| **Total** | **21.305** |

### B. Discussion of Results

Table shows close to seventy percent of our run-time is tied to either reading or writing our image text file. This is unfortunate, but expected: each image is composed of hundreds of thousands of pixels - for example, a 1024x768 image has close to eight-hundred thousand pixels. Furthermore, each pixel has three color channels (Red, Green and Blue). This means a single 1024x768 image is composed of almost two million and three-hundred thousand values. This is not to say the algorithm cannot be improved. An improvement opportunity is paralleling the reading and writing of each RGB channel, or even further segmenting the reading and writing functions.

The top pictures of Figure 6 present our original underwater images. These original images are very hazy, and their colors are very clearly unbalanced, leaning heavily towards green tones, most evident in the second image from the left.

The bottom pictures of Figure 6 are the resulting enhanced counterparts of the original pictures. It is visible that the enhanced pictures present better color balance. Furthermore, the objects in the image are more discernible from their surroundings, and that much of the haze is removed. The color palette in the middle image is especially useful in qualitatively determining the quality of the algorithm; whereas in the original image it is very hard to determine the colors in the palette, it is very clear in the enhanced picture that the image fusion algorithm implemented is capable of drastically improving underwater images.

Furthermore, we experimented with utilizing an image segmentation algorithm. The purpose of image segmentation, which has been explored in [15] and [4], is to highlight objects

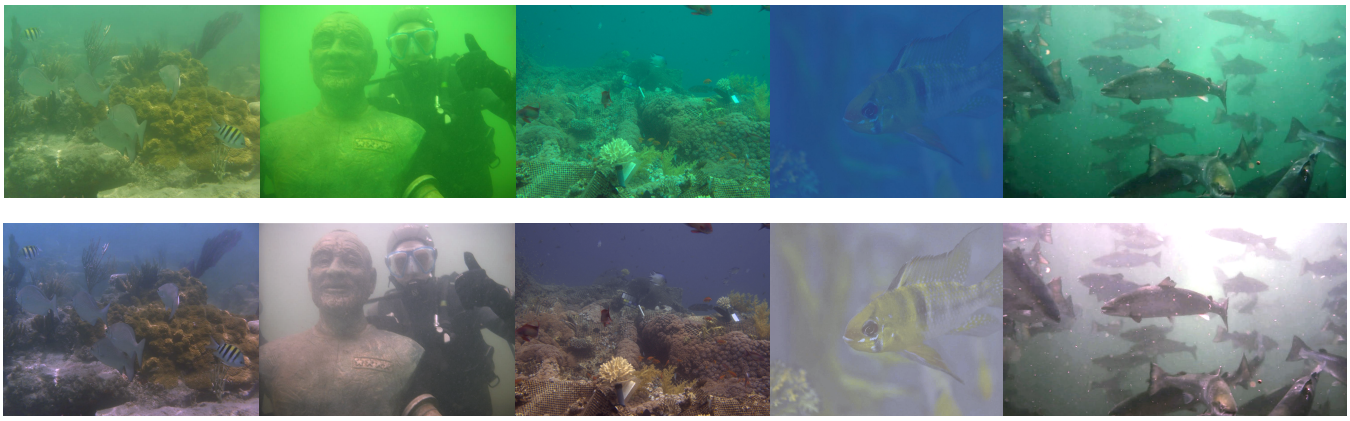

Fig. 6. Top: Original Image, Bottom: Result of the Image using our Image Fusion Implementation (Images from [1])

in the foreground. For example, in the first image in Figure 6, the fish are objects of focus and should be theoretically be highlighted by the image segmentation algorithm. From Figures 7 and 8, we can clearly see that our enhanced image gives better performance using image segmentation. It should noted that neither image correctly highlights the second fish in the bottom center. This is either a limitation of the algorithm (since it only works on the greyscale image) or indicates further room for improvement in our algorithm. Regardless, our algorithm shows at least marginal improvement over the base image in image segmentation applications.

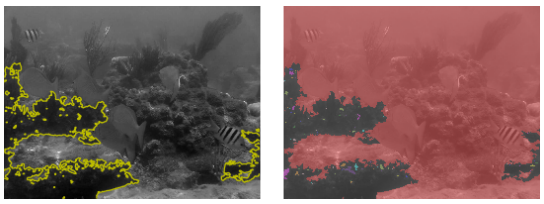

Fig. 7. Image Segmentation Algorithm Performance on the Original Image

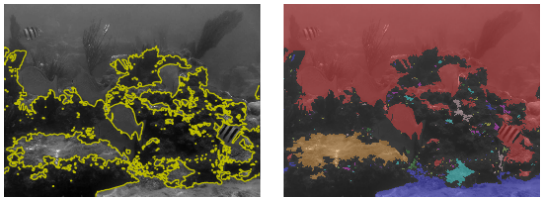

Fig. 8. Image Segmentation Algorithm Performance on the Enhanced Image

## REFERENCES


[1] C. O. Ancuti, C. Ancuti, C. De Vleeschouwer, and P. Bekaert, "Color balance and fusion for underwater image enhancement," *IEEE Transactions on Image Processing*, vol. 27, no. 1, pp. 379–393, 2018.

[2] K. He, J. Sun, and X. Tang, "Single image haze removal using dark channel prior," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 12, pp. 2341–2353, 2011.

[3] W. Kim, H. Bae, and T. Kim, "Fast and efficient haze removal," in *2015 IEEE International Conference on Consumer Electronics (ICCE)*, 2015, pp. 360–361.

[4] Y. Song and H. Yan, "Image segmentation techniques overview," in *2017 Asia Modelling Symposium (AMS)*, 2017, pp. 103–107.

[5] King-Sun Fu and Rosenfeld, "Pattern recognition and image processing," *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1336–1346, 1976.

[6] R. Schettini and S. Corchs, "Underwater image processing: State of the art of restoration and image enhancement methods," *EURASIP J. Adv. Signal Process*, vol. 2010, Jan. 2010. [Online]. Available: https://doi.org/10.1155/2010/746052

[7] D. Kocak, F. Dalgleish, F. Caimi, and Y. Schechner, "A focus on recent developments and trends in underwater imaging," *Marine Technology Society Journal*, vol. 42, pp. 52–67, 03 2008.

[8] E. N. Ohta and A. R. Robertson, "Cie standard colorimetric system," in *Colorimetry*. John Wiley & Sons, Ltd, May 2006, pp. 63–114. [Online]. Available: https://doi.org/10.1002/0470094745.ch3

[9] V. Agarwal, B. Abidi, A. Koschan, and M. Abidi, "An overview of color constancy algorithms," *Journal of Pattern Recognition Research*, vol. 1, pp. 42–54, 06 2006.

[10] J. Cepeda-Negrete and R. E. Sanchez-Yanez, "Combining color constancy and gamma correction for image enhancement," in *2012 IEEE Ninth Electronics, Robotics and Automotive Mechanics Conference*, 2012, pp. 25–30.

[11] P. Hubel, J. Holm, and G. Finlayson, "Illuminant estimation and colour correction," 1998.

[12] B. J. Lindbloom, "Chromatic adaptation," Apr 2017. [Online]. Available: http://brucelindbloom.com/index.html?Eqn_ChromAdapt.html

[13] R. Achanta, S. Hemami, F. Estrada, and S. Susstrunk, "Frequency-tuned salient region detection," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 1597–1604.

[14] *Colorimetry: understanding the CIE system*. CIE/Commission internationale de l'clairage; Wiley-Interscience, 2007.

[15] J. Sklansky, "Image segmentation and feature extraction," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 8, no. 4, pp. 237–247, 1978.