ANERIS

Operational Sensing Life Technologies for Marine Ecosystems

Deliverable 3.5 – ATIRES Code and Documentation

Lead Beneficiary: University of Haifa (UH)

Author/s: Nir Zagdanski, Derya Akkaynak

15/12/2024

Funded by the European Union

Prepared under contract from the European Commission

Grant agreement No. 101094924

EU Horizon Europe Research and Innovation action

Project acronym:	**ANERIS**
Project full title:	**operAtional seNsing lifE technologies for maRIne ecosystemS**
Start of the project:	January 2023
Duration:	48 months
Project coordinator:	Jaume Piera
Deliverable title:	ATIRES code and documentation
Deliverable n°:	D3.5
Nature of the deliverable:	Report
Dissemination level:	Public
WP responsible:	WP3
Lead beneficiary:	University of Haifa (UH)
Citation:	Zagdanski, N., & Akkaynak, D., (2024). *ATIRES Code and Documentation*. Deliverable D3.5 EU Horizon Europe ANERIS Project, Grant agreement No. 101094924
Due date of deliverable:	Month n°24
Actual submission date:	Month n°24

Deliverable status:

Version	Status	Date	Author(s)
1.0	Draft	13.12.2024	Nir Zagdanski, D. Akkaynak (UH)
1.1	1st Review	15.12.2024	Tali Treibitz (UH)
1.2	2nd Review	17.12.2024	Berta Companys (CSIC)
1.3	3rd Review	17.12.2024	Sebastian Luna-Valero (EGI)
1.4	4th Review	17.12.2024	Sara Montalbán (Quanta Labs)
2.0	Final	19.12.2024	Derya Akkaynak (UH)
2.0	Final Review	20.12.2024	Berta Companys (CSIC)

Table of Contents

Executive Summary

Underwater imaging presents unique challenges due to how water affects light, and consequently, the colors captured in an image. When light travels through water, different wavelengths are attenuated at different distances, leading to color distortions and reduced contrast. In most cases (though not all), red light is attenuated quickly, while blue and green light penetrate further, causing underwater images to often appear bluish-green with poor contrast. The ATIRES (Automatic underwaTer Image REstoration System) image enhancement algorithm addresses these issues through two simple but powerful techniques: red color correction and contrast stretching.

This document integrates usage instructions, technical implementation, and optimization details to provide a comprehensive overview of the underwater image enhancement system. The code is available on a public GitHub repository: https://github.com/VISEAON-Lab/aneris_enhance.

The ATIRES algorithm for image enhancement is planned to be applied on the imagery collected from undersea observatories in real-time. Currently, we have acquired imagery from the OBSEA cabled observatory. The OBSEA is an underwater cabled observatory located 4 km off the coast of Vilanova i la Geltrú, Spain, at 20 meters depth. The observatory is equipped with cameras, built by the EMUAS system partners. Future plans include using imagery from the Smart Bay observatory as well. Enhanced imagery will be the input for AIES-MAC partners, whose main task is species segmentation from images and AI services for recognition, counting, etc.

At the time of this writing, preliminary tests of the ATIRES code have been conducted on several videos provided by the EMUAS partners from the OBSEA observatory, and on citizen science images obtained from the citizen science observatory MINKA (https://minka-sdg.org/). Immediate next steps are to speed up the processing of the algorithms, and to fine-tune algorithm parameters to optimally benefit AIES-MAC algorithms. Depending on the performance of the algorithms presented here on the results obtained by downstream users such as the AIES-MAC team, these algorithms will continue to be developed and improved.

Following implementation and optimization details, we provide examples on datasets analyzed thus far in an off-line fashion. The algorithms already work near real-time; work is ongoing to make them real-time.

List of Abbreviations

AIES-MAC	Automatic Information Extraction System for MACro-organisms
ATIRES	Automatic underwaTer Image REstoration System
AWIMAR	Adaptive Web Interfaces for MARine life reporting, sharing and consulting
EMUAS	Expandable Multi-imaging Underwater Acquisition System
FPS	Frames Per Second
LUT	Look Up Table
OMB	Operational Marine Biology

1. Background and Introduction

Reconstruction of colors in underwater images is a challenging task. In fact, until recently, our understanding of the physics of how light propagates in the water column, hits the sensor of a camera and forms an image was not accurate [1], [2]. Once a more physically accurate model was developed, it became possible to reconstruct colors in underwater images in an objective and repeatable manner [3]. These advances were made by the UH partners prior to their joining of the ANERIS consortium.

The main goal of the ANERIS consortium is to build a network that will enable operational marine biology (OMB) products. Among the main goals of the consortium are the acquisition (EMUAS partners) and enhancement of underwater imagery (ATIRES partners, this document) for the purposes of species identification and monitoring (AIES-MAC partners) (Fig. 1). Additionally, the ATIRES technologies are being applied to citizen scientist images obtained from the MINKA citizen science observatory (https://minka-sdg.org/).

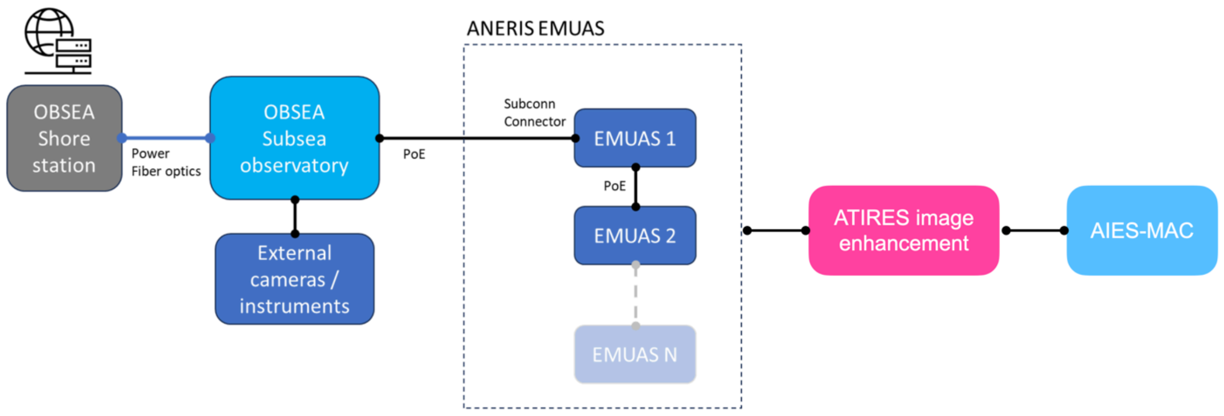

Figure 1 Overview of the image acquisition, enhancement, and interpretation tasks carried out by ANERIS partners. Image modified from [4].

The use cases of the imagery collected within the ANERIS project make the already-difficult task of color construction even more difficult, because the imagery comes from 1) underwater video, and 2) citizen scientists (with a wide range of underwater cameras and participants). The difficulty arises from the following: we currently only understand how to reverse color loss based on the laws of physics. For those laws to be applicable, imagery must be acquired in a way that the pixel intensities remain linearly related to the light in the scene. This is almost always true for RAW images, which are the sensor-level images captured by a given camera. However, while physically accurate and valuable, RAW images do not look visually pleasing to the human eye. Thus, many camera manufacturers have built-in algorithms that add non-linearities to the images and videos. These non-linearities increase brightness and contrast, and make the colors pop—at the expense of irreversibly biasing the colors. RAW images also require additional processing, which the non-specialist user cannot readily do. As camera manufacturers cannot afford to sell cameras which only take RAW images or video (because for the majority of consumers this imagery will look unpleasant), their standard outputs are enhanced, and therefore, non-linear.

An additional limitation that makes the job of the ATIRES algorithms difficult is the lack of information on *scene depth*. Scene depth is simply the distance between the camera and the scene being imaged. This information is critical to do color reconstruction, because colors fade as an exponential function of distance [1]

Over time, UH partners have developed several color reconstruction algorithms that work very well on linear imagery. These include Sea-thru [3], Sea-thru NERF [5], and OSMOSIS [6]. Yet, because these methods require linear images, they are not readily applicable to imagery that is being produced by ANERIS partners. However, there are many lessons learned from their development that helped us converge to the simple but powerful algorithms presented in this document.

For non-linear imagery, there is no standard way to reconstruct colors in a consistent manner, because the color distortions can come from several sources, including, but not limited to, camera sensor, in-camera algorithms, ambient light, and user settings at the time of image capture. Thus, learning-based algorithms are most suitable for color correction in such user cases. But they require training on very large datasets and even if successfully trained, may not be fast to apply. In this project, we have a requirement to provide enhancements as close to real-time as possible.

Thus, given the nature of the imagery collected by ANERIS partners, we have converged on using two simple but powerful algorithms on all data streams. These are two algorithms presented in this deliverable, Red Color Reconstruction and Contrast Stretching. The document also details the regular and optimized implementation of these algorithms. While these two algorithms will not yield the best possible color reconstruction (as that is not possible without linear imagery and distance information), they will provide sufficient enhancement of contrast and visibility, across all possible ocean states, to be able to identify species of interest with confidence.

Below, we describe these two algorithms. Next steps are to fine-tune the parameters of these algorithms to provide optimal input to the AIES-MAC partners’ algorithms.

2. Red Color Reconstruction

1.1. Overview

Red color correction compensates for the rapid absorption of long-wavelength light underwater by balancing the red channel relative to the green channel. In underwater images, though not in all geographic locations, the green channel of an image often serves as a stable reference due to its ability to penetrate deeper. The algorithm identifies discrepancies between the red and green channels and proportionally restores red levels where they are deficient, ensuring natural color reconstruction.

It should be noted that the best (and most consistent) color reconstruction can be done on images that have a linear relationship to the light that was in the scene. Images collected by citizen scientists are almost always non-linear, and so are the majority of video streams due to the need to compress immediately. Thus, the color information that can be recovered is inherently limited by the nature of the data our algorithms need to work on.

Red color correction compensates for the rapid absorption of red light underwater by:

1. Analyzing the average levels of red and green in the image.
2. Using the green channel as a reference to estimate how much red should be restored.
3. Adjusting red values proportionally based on the difference between the red and green means.

1.2. Implementation

The red correction algorithm works as follows:

1. **Convert the Image to Floating-Point Format**: This normalization step (scaling pixel values to the 0-1 range) ensures that arithmetic operations are accurate and consistent.
2. **Calculate Mean Values**: The algorithm computes the mean intensity for the red and green channels. This comparison determines how much correction the red channel needs.
3. **Apply Proportional Adjustment**: The red channel is adjusted proportionally based on its difference from the green channel, ensuring smooth and natural enhancement without oversaturation.
4. **Scale Back to 0-255 Range**: Once the adjustment is complete, the pixel values are rescaled to their original range for rendering.

Below, we provide two implementation versions for demonstration purposes, namely in Python and C++. Please refer to the GitHub repository for full code.

Python Implementation

```python
# Function to correct red channel based on green channel
# Input: BGR image
# Output: Red-corrected BGR image
def red_correction(img):
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    corrected_img = img_rgb.astype(np.float64)/255

    mean_g = np.mean(corrected_img[:,:,1])  # Green channel mean
    mean_r = np.mean(corrected_img[:,:,0])  # Red channel mean

    # Adjust red channel values proportionally
    corrected_img[:,:,0] = corrected_img[:,:,0] + (mean_g -
mean_r)*(1-corrected_img[:,:,0])*corrected_img[:,:,1]

    # Clip values to 0-1 range and convert back to 8-bit
    result = (255*corrected_img).astype(np.uint8)
    return cv2.cvtColor(result, cv2.COLOR_RGB2BGR)
```

C++ Implementation

```
// Corrects red channel in an image based on green channel
// Input: cv::Mat (image in BGR format)
// Output: Processed image with red correction
void redCorrection(cv::Mat& img) {
    img.convertTo(img, CV_32FC3, 1.0 / 255.0); // Normalize to 0-1 range

    std::vector<cv::Mat> channels(3);
    cv::split(img, channels);

    double mean_r = cv::mean(channels[2])[0]; // Red channel mean
    double mean_g = cv::mean(channels[1])[0]; // Green channel mean
    double diff = mean_g - mean_r;

    // Adjust red channel based on green channel
    channels[2] += diff * (1.0 - channels[2]).mul(channels[1]);

    cv::merge(channels, img);
    cv::threshold(img, img, 1.0, 1.0, cv::THRESH_TRUNC); // Cap values at 1.0
    img.convertTo(img, CV_8UC3, 255.0); // Scale back to 0-255 range
}
```

3. Contrast Stretching

3.1. Overview

Contrast stretching enhances the dynamic range of the image by redistributing pixel intensities. Underwater images often suffer from poor contrast, with most pixel values concentrated in a narrow intensity range. This algorithm identifies percentile-based thresholds (to exclude extreme outliers) and ensure they don’t affect the rest of the image, and stretches the pixel values within this range to utilize the full dynamic range, improving visibility and detail clarity.

Contrast stretching improves image visibility and image quality by:

1. Finding the darkest and brightest points in each color channel using percentile-based thresholds.
2. Stretching the color values between these points across the full available range.
3. Applying this enhancement while preserving the relative relationships between colors.

The percentile-based approach (set at 98%) avoids extreme stretching caused by outlier pixels.

3.2. Implementation

For source code and contributions, please visit: GitHub Repository.

The contrast stretching algorithm performs the following steps:

1. **Determine Thresholds**: Using percentile calculations, the darkest and brightest pixels within the acceptable range are identified. This step excludes outlier values like extreme shadows or highlights.
2. **Normalize Pixel Intensities**: Pixel values are linearly scaled between the identified thresholds. This operation ensures that dark areas become darker, bright areas become brighter, and mid-tones are distributed evenly.
3. **Clip and Rescale**: After scaling, pixel values are clipped to the 0-1 range and then rescaled to their original range (0-255) for display.

Python Implementation

```python
# Function to stretch contrast of an image
# Input: BGR image, percentile threshold (default 98)
# Output: Contrast-stretched BGR image
def contrast_stretch(img, prcn=98):
    high = np.percentile(img, prcn, axis=(0, 1), keepdims=True)  # High
threshold
    low = np.percentile(img, 100 - prcn, axis=(0, 1), keepdims=True)  # Low
threshold

    # Stretch pixel values between low and high thresholds
    img_stretched = (img - low) / (high - low)
    img_stretched = np.clip(img_stretched, 0, 1) * 255
    return img_stretched.astype(np.uint8)
```

C++ Implementation

```cpp
// Stretches contrast of an image
// Input: cv::Mat (image), percentile threshold (default 98%)
// Output: Contrast-stretched image
void contrastStretch(cv::Mat& img, double percentile = 98.0) {
    std::vector<cv::Mat> channels(3);
    cv::split(img, channels);

    for (int i = 0; i < 3; ++i) {
        cv::Mat flat;
        channels[i].reshape(1, 1).copyTo(flat);

        // Calculate low and high percentile values
        cv::sort(flat, flat, cv::SORT_ASCENDING);
        int total_pixels = flat.cols;
        int low_idx = static_cast<int>((100.0 - percentile) / 100.0 *
total_pixels);
        int high_idx = static_cast<int>(percentile / 100.0 * total_pixels -
1);

        uchar low_val = flat.at<uchar>(low_idx); // Low threshold
        uchar high_val = flat.at<uchar>(high_idx); // High threshold

        // Stretch pixel values between low and high thresholds
        channels[i].convertTo(channels[i], CV_32F);
        channels[i] = (channels[i] - low_val) / (high_val - low_val) * 255.0;
        channels[i].convertTo(channels[i], CV_8U);
    }

    cv::merge(channels, img);
}
```

3. Optimized Implementation for Real-time Performance

3.1. Overview

The optimized implementation introduces performance enhancements, such as precomputed Lookup Tables (LUTs) and batch processing, while maintaining the same general structure as the standard versions.

1. **Dynamic LUT Updates**: Precomputed LUTs adaptively adjust based on the current scene’s intensity distribution. This ensures efficient and real-time processing with minimal overhead.
2. **Efficient Memory Management**: The use of in-place operations reduces unnecessary data copying, enhancing performance for high-resolution images or real-time video.
3. **Frame-by-Frame Processing**: Consistent with other versions, it processes videos frame-by-frame but leverages LUTs to significantly improve frame rate.

3.2. Implementation

These snippets collectively demonstrate the optimized approach: normalizing values with a float LUT, dynamically updating contrast LUTs, correcting red hues based on green intensity, and finally applying contrast stretching. All operations are designed to efficiently enhance underwater imagery frame-by-frame without altering the original code’s logic.

3.2.1. Lookup Table (LUT) Initialization

This snippet sets up a floating-point LUT to streamline normalization and prepares channel-specific LUTs for contrast stretching.

```
void initializeLUTs() {
    // Initialize float conversion LUT to normalize [0-255] to [0-1]
    float_lut = cv::Mat(1, 256, CV_32FC1);
    float* lutData = float_lut.ptr<float>();
    for (int i = 0; i < 256; i++) {
        lutData[i] = i / 255.0f;
    }

    // Initialize LUTs for contrast stretching (one per channel)
    for(int i = 0; i < 3; i++) {
        stretch_luts[i] = cv::Mat(1, 256, CV_8U);
    }
}
```

3.2.2. Updating Contrast Stretching LUTs

This snippet calculates percentile-based thresholds, identifies appropriate low/high intensity bounds, and updates each channel's LUT to improve contrast according to the current frame's histogram.

```
void updateStretchLUTs(const cv::Mat& frame) {
    const int kHistSize = 256;
    int lowerBound = static_cast<int>((100.0 - percentile) * 0.01 *
frame.rows * frame.cols);

    std::vector<cv::Mat> channels;
    cv::split(frame, channels);

    for(int c = 0; c < 3; ++c) {
        // Calculate histogram for current channel
        int histogram[kHistSize] = {0};
        const uchar* data = channels[c].ptr<uchar>();
        const int totalPixels = channels[c].rows * channels[c].cols;

        for(int i = 0; i < totalPixels; ++i) {
            histogram[data[i]]++;
        }

        // Find low and high intensity values based on percentile
        int count = 0, low = 0, high = 255;
        for(int i = 0; i < kHistSize; ++i) {
            count += histogram[i];
            if(count >= lowerBound) {
                low = i;
                break;
            }
        }

        count = 0;
        for(int i = kHistSize - 1; i >= 0; --i) {
            count += histogram[i];
            if(count >= lowerBound) {
                high = i;
                break;
            }
        }

        // Compute scaling for contrast stretching and update LUT
        uchar* lutData = stretch_luts[c].ptr<uchar>();
        float scale = 255.0f / (high - low);
        for(int i = 0; i < 256; ++i) {
            lutData[i] = cv::saturate_cast<uchar>((i - low) * scale);
        }
    }
}
```

3.2.3. Red Channel Correction

Here, the red channel is adjusted based on the green channel's intensity. Since green penetrates deeper underwater than red, it serves as a reference to restore balanced coloration.

```
cv::Mat redCorrection(const cv::Mat& img) {
    std::vector<cv::Mat> channels;
    cv::split(img, channels);

    // Convert red and green channels to float range [0-1]
    cv::Mat r_float, g_float;
    cv::LUT(channels[2], float_lut, r_float);  // Red channel
    cv::LUT(channels[1], float_lut, g_float);  // Green channel

    // Calculate mean difference between green and red channels
    cv::Scalar mean_r = cv::mean(r_float);
    cv::Scalar mean_g = cv::mean(g_float);
    float diff = mean_g[0] - mean_r[0];

    // Adjust red channel proportionally to green channel
    cv::Mat correction = diff * (1.0f - r_float).mul(g_float);
    r_float += correction;

    // Convert corrected red channel back to 8-bit [0-255]
    r_float *= 255.0f;
    r_float.convertTo(channels[2], CV_8U);

    cv::Mat result;
    cv::merge(channels, img);
    cv::merge(channels, result);
    return result;
}
```

3.2.4. Applying Contrast Stretching

This code applies the updated LUTs to enhance contrast, redistributing intensities for each channel. LUT updates occur at set intervals to adapt as conditions change over time, especially important for video processing.

```
cv::Mat contrastStretch(const cv::Mat& img) {
    static int frame_count = 0;
    // Periodically update LUTs based on update interval
    if (frame_count++ % update_interval == 0) {
        updateStretchLUTs(img);
    }

    std::vector<cv::Mat> channels;
    cv::split(img, channels);

    // Apply per-channel LUTs to stretch contrast
    for(int c = 0; c < 3; ++c) {
        cv::LUT(channels[c], stretch_luts[c], channels[c]);
    }

    cv::Mat result;
    cv::merge(channels, result);
    return result;
}
```

4. System Requirements and Benchmarks

Python requirements for our system are relatively simple and standard: Python 3.x, OpenCV (opencv-python), and NumPy. The C++ requirements are Opencv 4.x and a C++11 compiler.

We tested performance using a video resolution of 2548 × 1440 at 30 FPS. Tested on a system with CPU: Intel i7-11800H @ 4.6GHz (8 cores, 16 threads), RAM: 32GB DDR4 3200MHz and operating system: Ubuntu 22.04.4 LTS x86_64.

Results were as follows: Python implementation: 3.7 FPS, standard C++ implementation: 5.0 FPS, and optimized C++ implementation: 20 FPS. In computer graphics and many areas of rendering, 20 FPS is considered real-time because it gives the user/viewer a fluent experience.

5. Project Structure

```
.
├── python/
│   ├── underwater_enhance.py
│   ├── image_processor.py
├── cpp/
│   ├── underwater_enhance.cpp
│   ├── underwater_enhance_opt.cpp
├── README.md
```

6. Usage Examples

6.1. Python

Here is how our script can be called using Python, on images and video:

```
python underwater_enhance.py input_image.jpg output_image.jpg
python underwater_enhance.py input_video.mp4 output_video.mp4
```

6.2. C++

Here is how our script can be called using C++ for both standard and optimized versions, on images and video:

```
# Standard version
g++ underwater_enhance.cpp -o underwater_enhance `pkg-config --cflags --libs opencv4`
./underwater_enhance input_image.jpg output_image.jpg

# Optimized version
g++ underwater_enhance_opt.cpp -o underwater_enhance_opt `pkg-config --cflags --libs
opencv4`
./underwater_enhance_opt input_video.mp4 output_video.mp4
```

7. Example Results

Below are examples of imagery we have enhanced using the ATIRES algorithm offline (Figures 2 and 3). We are working with the EMUAS group to be able to do the enhancement online, in real-time. Current processing speeds are near real-time, 20 FPS.

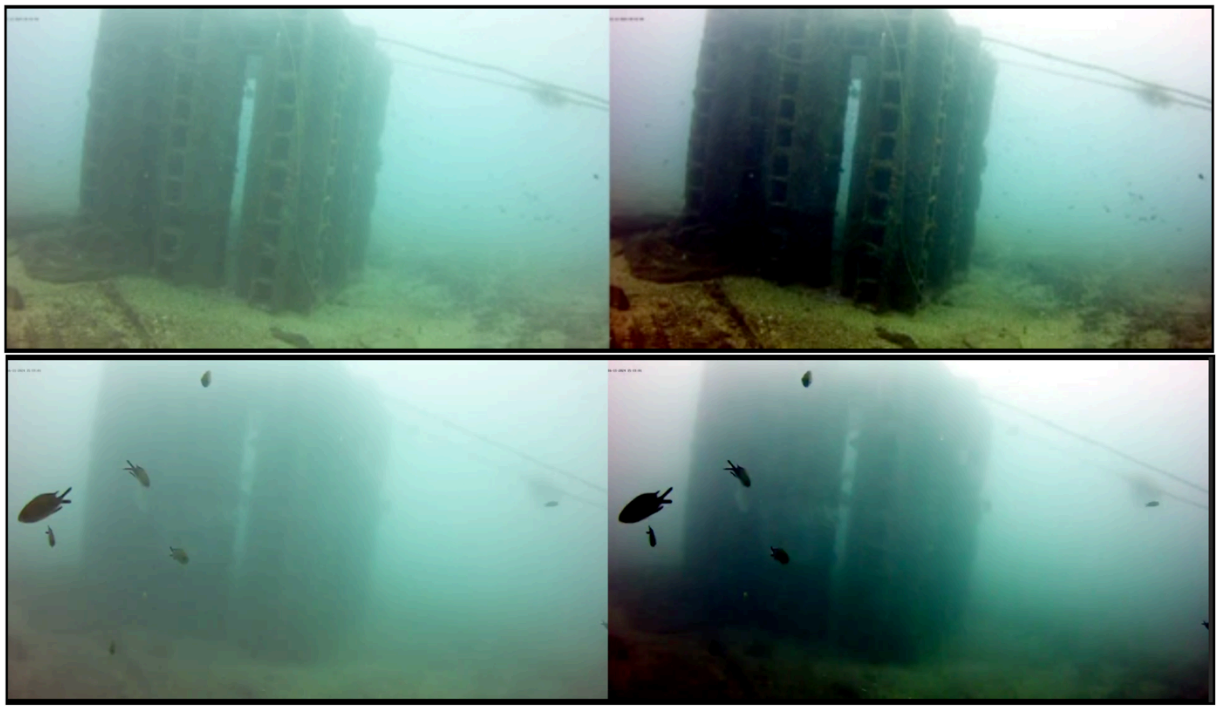

Figure 2 Frames taken from videos acquired by the EMUAS group before (left) and after (right) ATIRES enhancement.

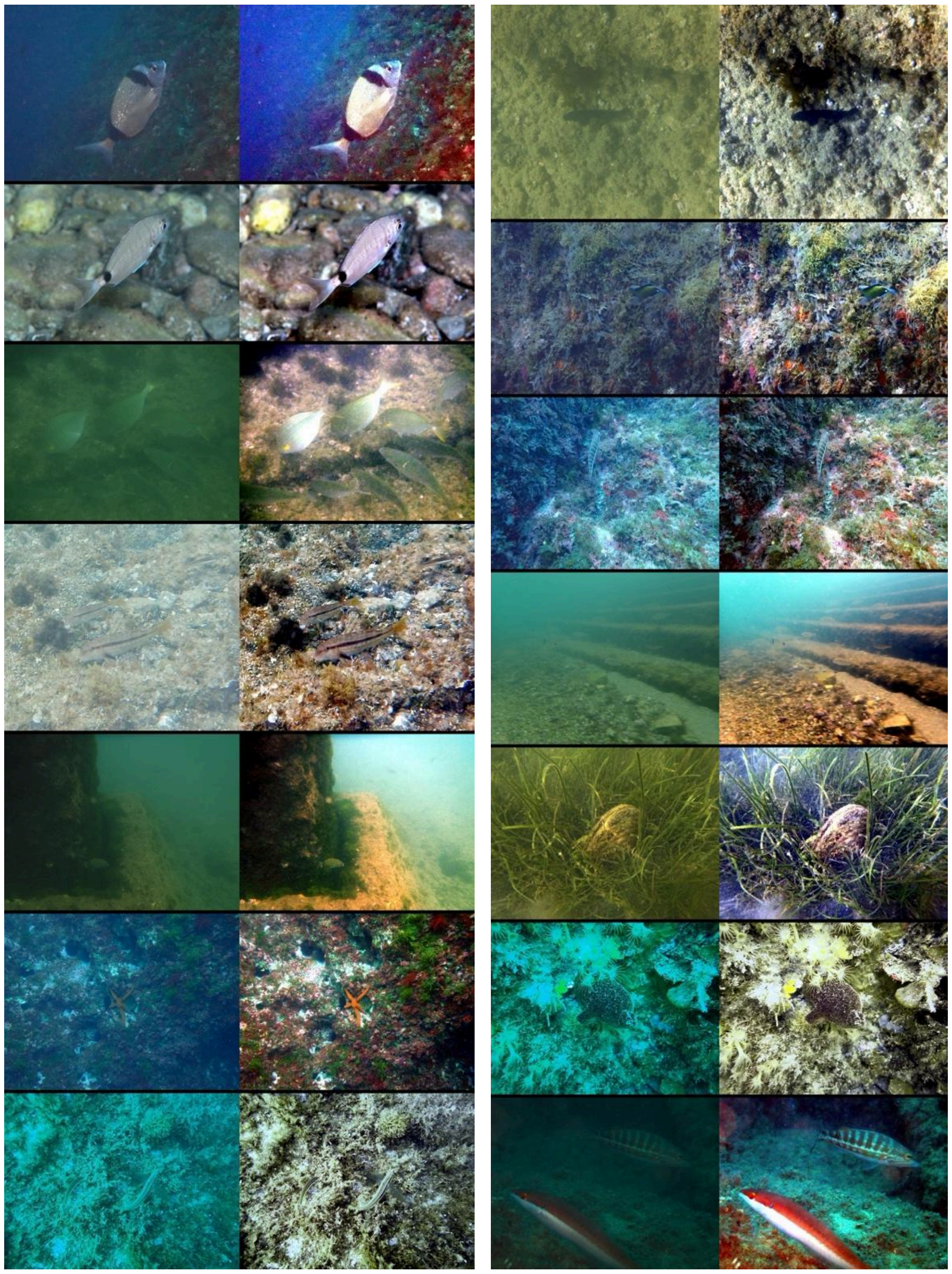

Figure 3 Examples of images captured by citizen scientists, uploaded into the MINKA observatory, enhanced by the ATIRES algorithm. Original images are on the left, and enhanced images are on the right. Note that these enhancements were made on non-linear images (which violate the laws of physics and therefore our best understanding of color distortions), and also without a depth map (which is key for proper color reconstruction).

8. Summary and Outlook

This document summarizes the code and documentation for ATIRES, ANERIS deliverable 3.5. This deliverable is a computer vision algorithm for the enhancement of colors and contrast in underwater imagery. In the scope of the ANERIS project, imagery for which this code is relevant, will come from two sources: cabled underwater observatories, and images collected by citizen scientists. The final requirement for ATIRES is to run on real-time. At this moment, we have achieved 20 FPS with an optimized C++ implementation, which qualifies as real-time.

Future work for the development and integration of these algorithms with other ANERIS partners will prioritize the implementation of this code into the EMUAS image capture pipeline. In parallel, we will be working closely with the AIES-MAC partners to fine-tune the parameters of our algorithm to maximally benefit theirs.

We expect that with access to more and diverse imagery being collected by partners, we will have a better understanding of the spectrum of the dominant color distortions, and will be able to develop even more customized solutions for their enhancement.

References

[1] D. Akkaynak and T. Treibitz, “A Revised Underwater Image Formation Model,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT: IEEE, Jun. 2018, pp. 6723–6732. doi: 10.1109/CVPR.2018.00703.

[2] D. Akkaynak, T. Treibitz, T. Shlesinger, Y. Loya, R. Tamir, and D. Iluz, “What is the Space of Attenuation Coefficients in Underwater Computer Vision?” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI: IEEE, Jul. 2017, pp. 568–577. doi: 10.1109/CVPR.2017.68.

[3] D. Akkaynak and T. Treibitz, “Sea-Thru: A Method for Removing Water From Underwater Images,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, CA, USA: IEEE, Jun. 2019, pp. 1682–1691. doi: 10.1109/CVPR.2019.00178.

[4] Alcocer, Alex *et al.*, “Validated Bio opt imaging solution (EMUAS).” ANERIS Project, Grant agreement No. 101094924, Dec. 12, 2024.

[5] D. Levy *et al.*, “SeaThru-NeRF: Neural Radiance Fields in Scattering Media”.

[6] O. B. Nathan, D. Levy, T. Treibitz, and D. Rosenbaum, “Osmosis: RGBD Diffusion Prior for Underwater Image Restoration,” in *Computer Vision – ECCV 2024*, A. Leonardis, E. Ricci, S. Roth, O. Russakovsky, T. Sattler, and G. Varol, Eds., Cham: Springer Nature Switzerland, 2025, pp. 302–319. doi: 10.1007/978-3-031-73033-7_17.