Assignment 02

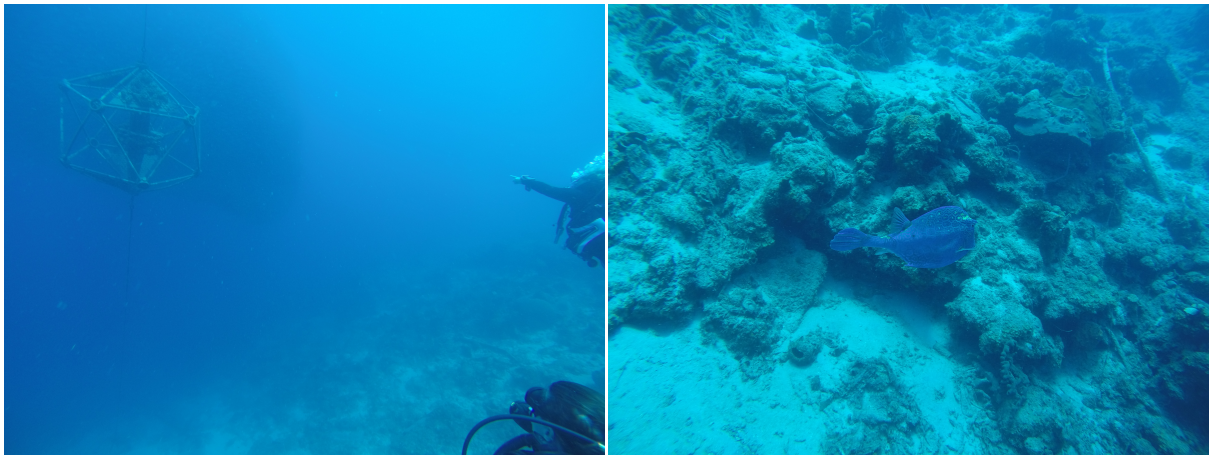
Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- color spaces, color channels, and boosting contrast with histogram equalization
- transparency and alpha blending
- projective geometry, homography, and panoramas

Part 1: Underwater images

Taking good-looking pictures underwater is hard. Within the visible spectrum, long wavelengths of light (redder) are absorbed more in water than short wavelengths (bluer). This gives photos taken underwater a strong bluish tint and low overall contrast. This assignment includes three photos taken underwater - `underwater01.jpg`, `underwater02.jpg`, and `underwater03.jpg`. Your task is to perform some color adjustments using numpy and OpenCV to make them look better. Here are two examples of images that are captured underwater:



And here are the same images after some adjustments (which you are about to implement):

I hope you agree that these are an improvement! (If you look online for state-of-the-art handling of underwater photos, you'll find much fancier methods. We're going to build a simple one here to exercise some of the concepts we've learned in class.)

The algorithm we're going to implement in the `enhance()` function is as follows:

1. Increase the *gain* of the red channel to account for the fact that red light is absorbed more underwater and/or reduce the *gain* of the blue and green channels. (This step might also be called the "white balancing" step.) This will be handled by a call to the `adjust_gain_bgr()` function.
2. Increase the *contrast* of the overall image by converting to LAB color space and applying partial histogram equalization to the L channel. The main step here is that you need to construct your own lookup table in the `create_lookup_table()` function, then apply it to the L channel using the `cv.LUT()` function.
3. Convert back to BGR color space and return the result.

The `enhance()` function in `src/underwater.py` is where you will implement this algorithm – the `enhance()` function should include calls to `adjust_gain_bgr()` and `create_lookup_table()`.

Recall from class that the 'identity' lookup table is `lut[i] = i`, and that the 'full' histogram equalization lookup table is `lut[i] = cdf[i]`, where `cdf` is the cumulative sum of the histogram normalized to the range 0-255. The partial histogram equalization lookup table is a linear interpolation between these two lookup tables. The `create_lookup_table` function takes an `alpha` argument which controls how aggressively to apply histogram equalization. An `alpha` of 0 means to use the identity lookup table, and an `alpha` of 1 means to use the full histogram equalization lookup table.

You may find this OpenCV tutorial on histogram equalization helpful, though it covers "full" rather than "partial" histogram equalization. You may also need to get creative with debugging, for instance by

plotting histograms of the L channel before and after applying your lookup table to make sure it is doing what you expect.

Part 1 of your actual assignment is:

1. Implement missing code in `src/underwater.py` following the docstrings. When you upload to gradescope, we will run separate tests on the `adjust_gain_bgr()`, `create_lookup_table()`, and `enhance()` functions.
2. There are some parameters for you to choose – how much to boost or reduce each color channel, and how aggressively to apply histogram equalization (the `alpha` argument to `create_lookup_table()`). Pick sensible values for these parameters in the `enhance()` function. Your output does not need to exactly match the images above - you are encouraged to "tune" the parameters to your liking and make the images look as good as you can. **Make these changes to the `_DEFAULT_GAIN_BGR` and `_DEFAULT_ALPHA_LUT` variables in `src/underwater.py` so that we can see what values you chose.**

Part 2: Projective geometry, homography, and compositing images

This is a "debugging" style assignment. You will be given a file that is supposed to do something, but it doesn't work. Your job is to fix it. Use this as an opportunity to learn/practice your IDE's debugging tools - they will make your life much, much easier! Here is some information on how to use the PyCharm debugger, and here is some information on how to use the VSCode debugger.

Partial credit is possible. If you are unable to fix all bugs, you may still receive partial credit by detailing your thought process and debugging strategy in the `debug-process.txt` file. For any failing tests, we will look in the `debug-process.txt` file to see your thought process and strategy for that function. We may award partial points based on the quality of your debugging process, even if you were unable to fix the bug. If you write your own test code and include it in the `src` directory, make a note of it in the `debug-process.txt` file, and we will take that into account when grading.

OpenCV provides tools for stitching together multiple images into a panorama using the `cv.Stitcher` class. You can read about that interface here if you're curious. In this part of the assignment, you have been given buggy code that is intended to stitch together two images into a simple two-image panorama using homography and alpha blending. Your task is to fix the bugs and make it work as described here.

Some more information about homography and compositing can be found in Chapter 8.3 of the CVAA book. If you're wondering how we got from chapter 2 to chapter 8, the short answer is that chapter 8 gets into computing homographies *automatically* from sets of corresponding points, but here we're going to have you, the user, click on points in the images manually to solve the (hard) correspondence problem. The main skills you're practicing here are

- seeing homography “in action”
- understanding how homography relates to projective geometry
- seeing how to composite together two images by taking averages of overlapping pixels
- reading code with an expected-behavior in mind and debugging it to make it work

A function called `get_clicked_points_from_user()` is already provided for you in `src/panorama.py`. This function works (not part of the debugging). If you run the file, the first thing it will do is display two images and allow you to click on corresponding points in the left image and right image (as demonstrated in class). Recall from class that you need to select at least 4 points in each image in order to compute a homography (because each point has x and y coordinates, and a homography has 8 degrees of freedom). The interface looks like this:

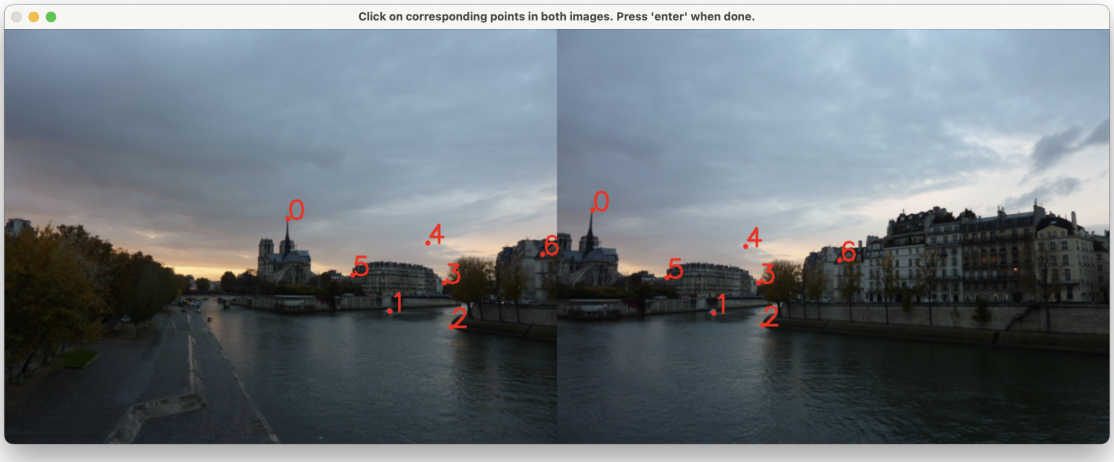


Figure 1: Panorama corresponding points interface

You’ve been provided with three images to play with: `paris_a.jpg`, `paris_b.jpg`, and `paris_c.jpg`. These came from a public github repository which was in fact a project by a student at another university who implemented all of these operations without using any help from OpenCV! You are welcome to use these images to test your code, but you are also further encouraged to go take a few photos of your own around campus and try stitching them together. (If you do this, you might even share your photos on Slack for others to try!)

Here’s what the file is **supposed** to do:

1. Load the images and get corresponding points from the user (done for you, not part of the debugging).
2. Compute the homography matrix that maps homogenous coordinates from the second image to homogenous coordinates in the first image.

3. Figure out how big the output image needs to be to contain both `image1` and the warped `image2`. This essentially introduces a third coordinate system `image3` which contains both `image1` and `image2`.
4. Warp *both* images to a common coordinate system that will fit inside the output image.
5. Blend the two warped images together by taking a weighted average of overlapping pixels. Pixels near the center of each image should be weighted more heavily than pixels near the edges in order to avoid the hard-edge artifacts that we saw in the example in class with pictures of a mountain.

Your task is to identify and fix the bugs in `src/panorama.py`. To focus your efforts, we will tell you that

- You do not need to make any changes to `get_clicked_points_from_user()`
- You do not need to make any changes to `main()`
- You do not need to make any changes to the `if __name__ == '__main__':` block at the bottom of the file
- You do not need to make any changes to the helper functions in the `utils` file
- All the comments and docstrings tell you what is *supposed* to happen. Sometimes the code will do what the comments say, and sometimes it won't. Your job is to make it do what the comments say that it's trying to do.

The file given to you *runs* but the output is wrong in several ways. Use whatever debugging tools you have at your disposal. Both PyCharm and VSCode have built-in debuggers that will save you a lot of time and sanity to try out. You will also get good mileage out of installing the "OpenCV Image Viewer" add-on in PyCharm, which will let you visualize intermediate images while paused in the debugger.

Discussing with your classmates about your debugging strategy and tools is encouraged. Do not share information about the actual bugs you find or the solutions you come up with.

Further details so that you can visualize what it is *supposed* to do once you fix all the bugs

To illustrate the basic idea, here's what it looks like when we apply `cv.warpPerspective()` to `paris_a.jpg` and `paris_b.jpg`:

Figure 2: paris_b warped to the coordinate system of the output

Now, one of the challenges is that the warped image is larger than the original image, so we also need to "make room" in the original image. The approach is to also warp the original image to a coordinate system that is large enough to contain both the original image and the warped image. This looks like this:

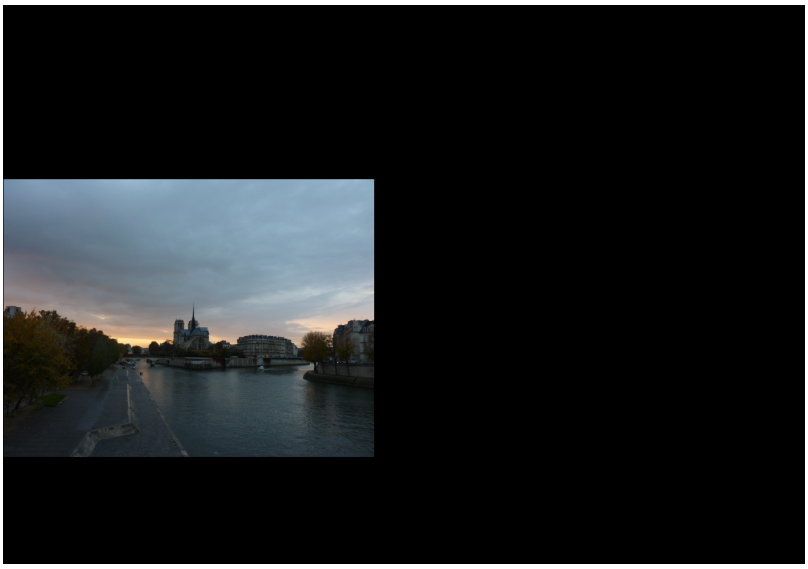


Figure 3: paris_a warped to the coordinate system of the output

So far, this shows that homography can warp images into a common reference frame. But, if we just take an average of all the pixels in the above two images, we get a hard edge where they meet:

Figure 4: paris_a and paris_b blended with no weighting

To address this, ther is a `add_weights_channel()` function that adds a fourth channel to the images that is a "weight" channel. The weight channel is a grayscale image that is black near the edges of the image and white near the center. The weights for one of the images look like this:

Figure 5: weights channel by itself

Fun fact: such 4-channel images are common, and the 4th channel is typically called the "alpha" channel. Where "alpha" is 1, the image is opaque, and where "alpha" is 0, the image is transparent.

OpenCV uses this convention, so we can in fact visualize what the “4-channel” image looks like:

Figure 6: 4-channel image where weights are ‘alpha’

Note that where the weights are 0, the image is treated by your computer as transparent, and where the weights are 1, the image is treated by your computer as opaque. The other nice thing about adding weights as a fourth channel is that we can ‘warp’ the weights along with the images in one fell swoop. Here’s what the weights look like after warping:

Figure 7: warped weights (a)

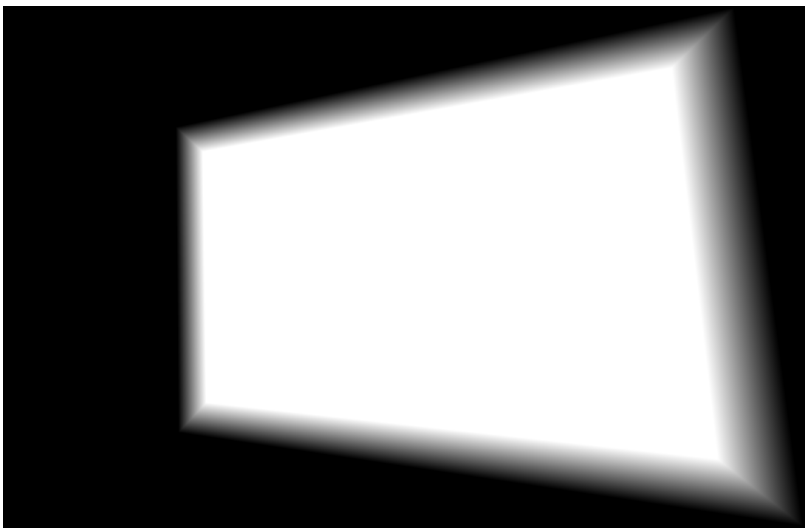


Figure 8: warped weights (b)

Figure 9: warped 4-channel image with weights (a)

Figure 10: warped 4-channel image with weights (b)

Now, the final task is to 'blend' the two images together by taking a weighted average of the corresponding pixels. The weights will come from the 4th (alpha) channel of the warped images. Since the weights are 0 near the edges of the image, the average at those edge points will be dominated by the pixels near the center of the other image. This is exactly what we want to avoid the hard edge visual artifact. Here's what the final blended image looks like:

Figure 11: paris_a and paris_b blended with weighting (this is the 'correct' output that you should get once you fix all the bugs, but the exact result will depend on your choice of images and points.)

Let's make this precise. If pixel `img1[i,j]` has weight `w1[i,j]` and pixel `img2[i,j]` has weight `w2[i,j]`, then the blended pixel `img_blend[i,j]` is computed as

$$`img_blend[i,j] = (w1[i,j] * img1[i,j] + w2[i,j] * img2[i,j])/(w1[i,j] + w2[i,j])`$$

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.