Project 4 Report

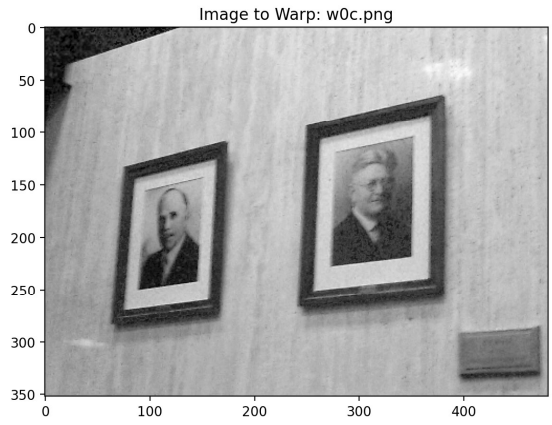
Nick Elliott

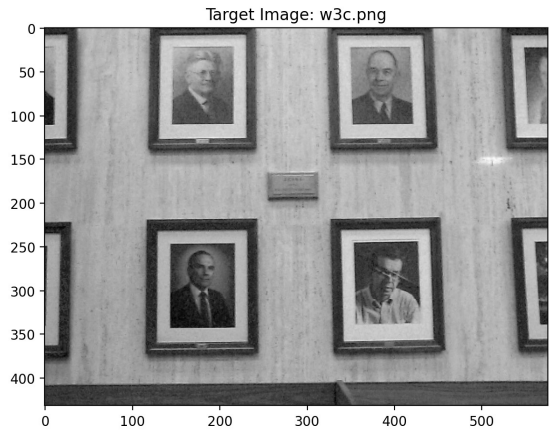
Developing the forward and backward transformations:

Using the provided transformation techniques, a warping transformation can be constructed using matrices in the form $Ax = b$ to solve linear systems. A is constructed as follows:

$$\begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' \\ & & & \vdots & & & & \\ -x_N & -y_N & -1 & 0 & 0 & 0 & x_Nx_N' & y_Nx_2' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' \\ & & & \vdots & & & & \\ 0 & 0 & 0 & -x_N & -y_N & -1 & x_Ny_N' & y_Ny_N' \end{pmatrix} \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{21} \\ p_{23} \\ p_{23} \\ p_{31} \\ p_{32} \end{pmatrix} = \begin{pmatrix} -x_1' \\ -x_2' \\ \vdots \\ -x_N' \\ -y_1' \\ -y_2' \\ \vdots \\ -y_N' \end{pmatrix}$$

Matrix multiplication of $Ax = b$ creates a system of equations. The desired result is a mapping of correspondence x_i -> x_i' $\forall\, i \in N$. Depending on N, the linear system results in an under-constrained case, sufficiently constrained case or over-constrained case. Note that our linear system will solve for 8 P values. For this project, the over-constrained case is our typical use case as there are generally sufficient correspondences to achieve the over-constrained case. Additionally, Python has libraries with an algorithm, Single Value Decomposition, to solve the sufficiently and over-constrained case to get each value of P. Here is an example transformation from the provided dataset:



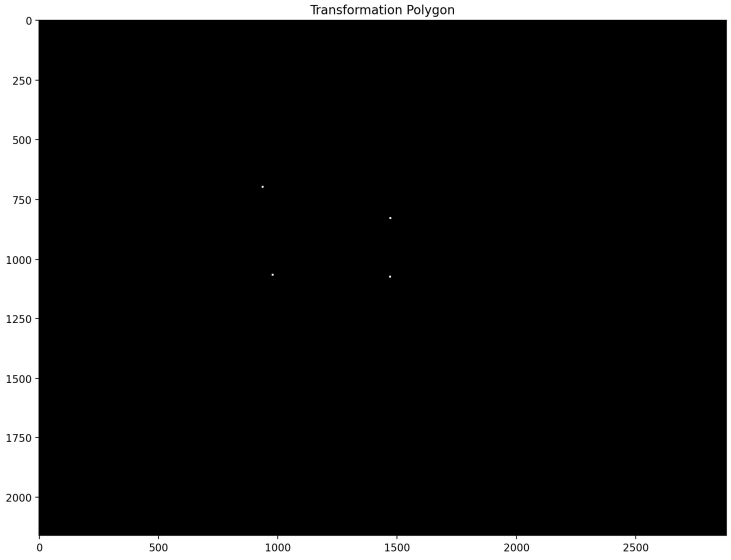


Choosing two images from a set of images, one of the two is chosen as a “target” the other as an image to be warped “to_warp”. Additionally, note that one image should be chosen as an image which all other images are warped to. There are more sophisticated warping schemes where you may have multiple images which the remaining images warp to, but for this project’s purposes, a single image will be the target of all warped images. Using SVD and an implemented function **get_P**, the “A” matrix and “b” matrices are passed to the SVD algorithm with the over-constrained case parameter “full_matrices =

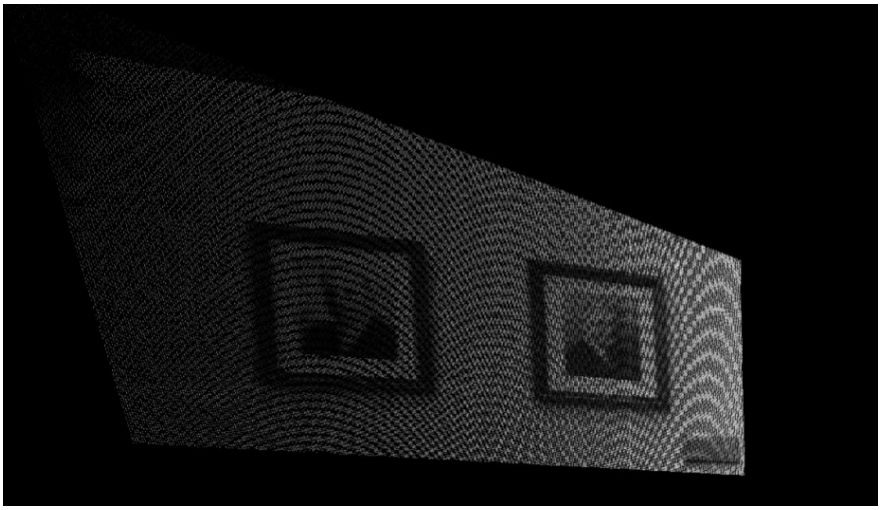
false", this function returns a matrix P. Next, a mapping function is implemented according the formula below:

$$x' = \frac{p_{11}x + p_{12}y + p_{13}}{p_{31}x + p_{32}y + 1}$$
$$y' = \frac{p_{21}x + p_{22}y + p_{23}}{p_{31}x + p_{32}y + 1}$$

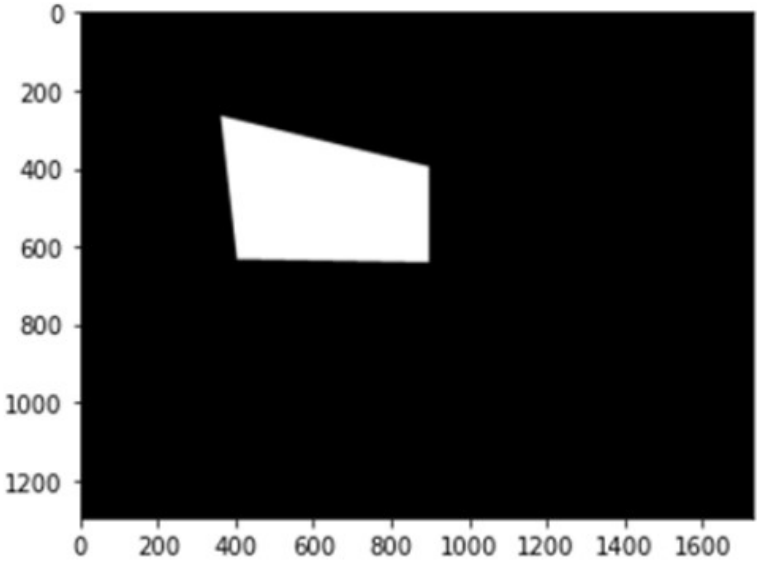
Finally, a polygon is generated by transforming each corner of the "to_warp" image. Here is an example of the polygon placed onto a canvas.



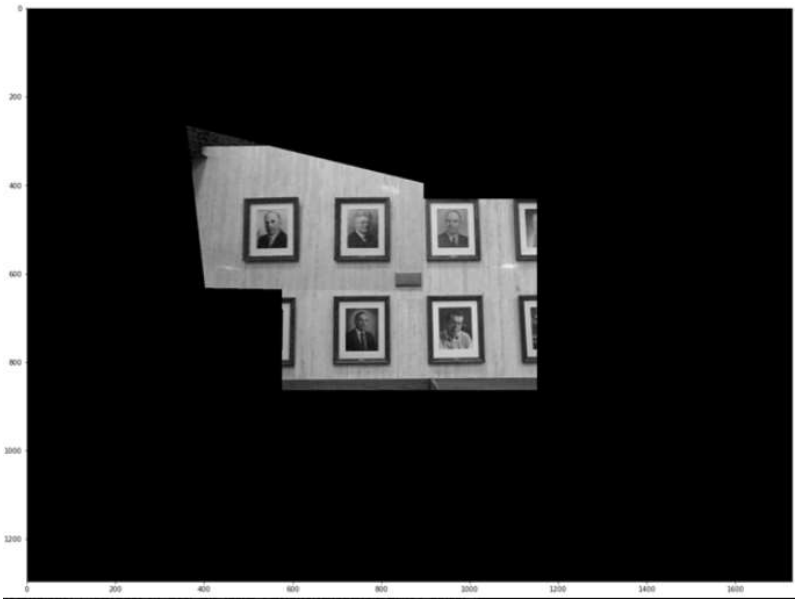
In addition to the forward transformation matrix P mapping x_i -> x_i', a "backward" transform matrix P_back is also generated mapping x_i'->$x_i \forall\ i\ \in N$. This is done in the same way with a linear system $Ax = b$, only x_i and x_i' are reversed. This "backward" transformation will be used to sample the original image being warped and place the samples appropriately into the transformed warped image. Note that if values were sampled directly from the original image, many portions of the polygon above would most likely be overwritten with multiple values, or sparsely populated. This means there would be empty spaces where no pixel mapped directly onto the warped polygon. Here is an example of what a direct transformation might look like:



To correctly sample the warped image values, the backward transform must be used. To get the appropriate domain in x,y of the warped polygon, I created a mask. I tried a few approaches to creating the x, y domain. First, I attempted to write logic in python to connect vertices of a polygon. I felt this approach was the best in terms of precision, however, I felt it was more practical to approximate the domain for x and y by blurring with convolution and then thresholding (this effectively spreads sparsely populated pixels to create a solid shape when thresholded). I picked some magic numbers, but with testing much larger images at higher resolutions, the approximate domain in the blurred and thresholded. I implemented this with a square 11x11 kernel, convolved with the warped image populated with values of 255. I created a threshold for values above zero. Here is the resulting polygon approximating the domain of the backward transform to sample original image values:



I then iterated over this “domain”, mapping backward onto the original image using the backward transform matrix **P_back**, sampling the original image and placing those values onto the warped polygon. Here is the result when I place the target image and warped image on the same canvas (warped at the front):



Because my domain polygon isn't exact, I must check whether each value maps correctly onto the original. Coordinates that don't map backward are ignored. Note that there are many flaws with just overlapping the two images. In subsequent sections I will discuss my approaches to blending. However, first, an analysis of the role the number N correspondences plays in getting effective warpings.

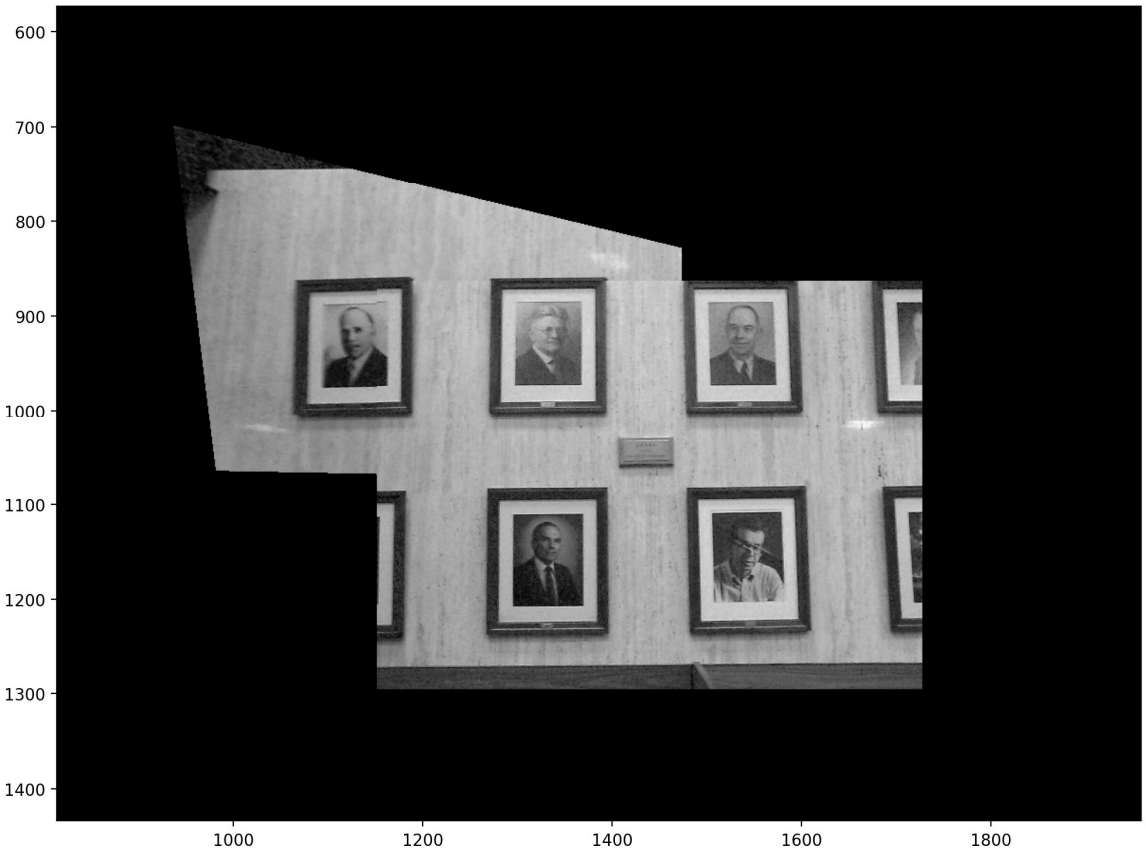
Analysis of Correspondences:

To measure effect of warping as a function of correspondences, I reduced the number of correspondences in the example image (w3c.png to w0c.png) to 3 correspondences. The result was a polygon so big that it would not map on my canvas. I incremented the correspondences to 4. Here was the result:

Note that the warped image is closer to the original image meaning its perspective is similar. Therefore, it does not align as well with the target. Next, I increment the correspondences to 5.


700
800
900
1000
1100
1200
1300
1400
1000
1200
1400
1600
1800

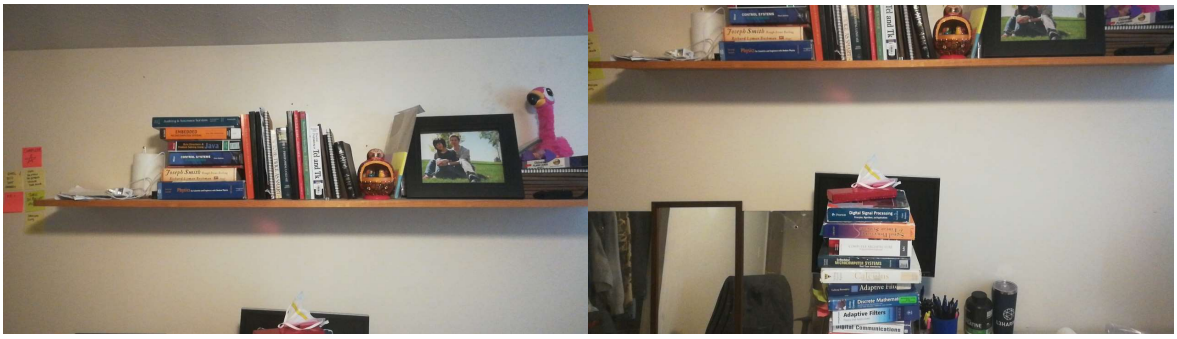
Here are the original 6 correspondences:

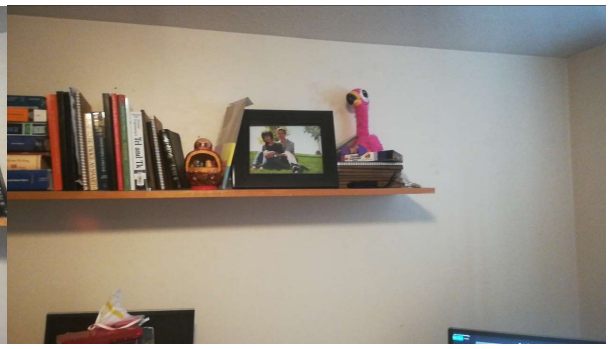
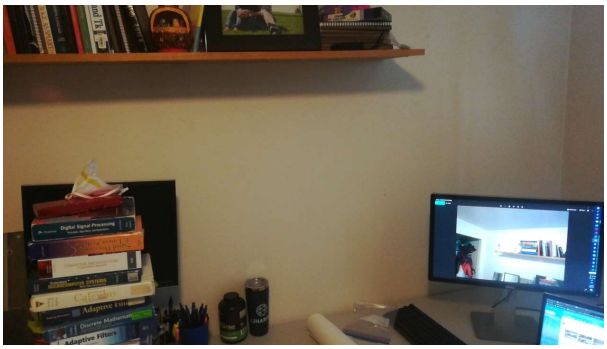


It’s clear that the alignment of perspective is proportional to the number of correspondences.

Panoramic Experiment:

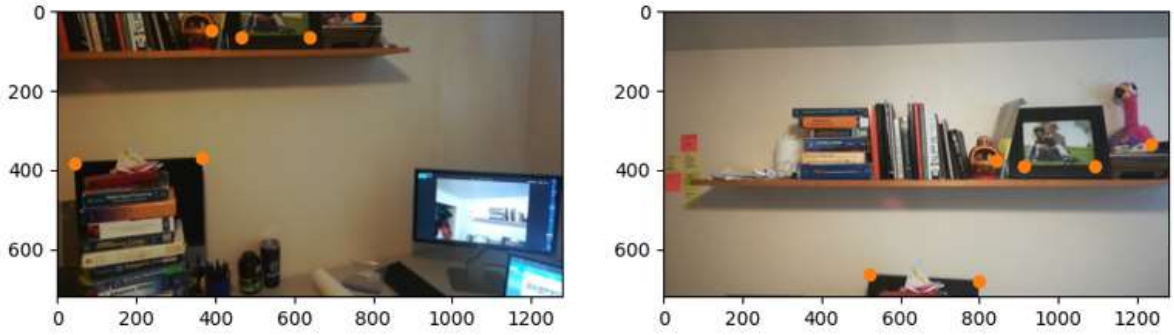
I took some panoramic photos of my office and shelf. I did my best to hold the camera in the same location while only allowing rotations in the x,y,z axis. Here are the images below:



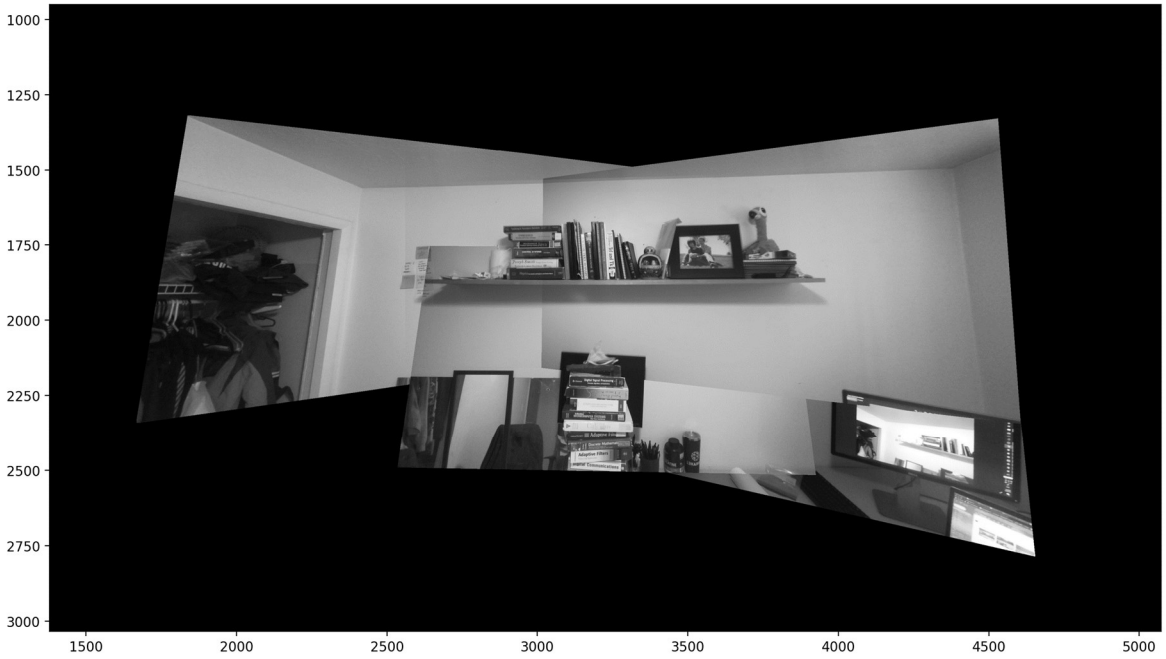
I then placed correspondences in a json file called **shelf_params.py**. I did notice that you could get by with 5 correspondences depending on the photo, however, on some photos, I needed more than 6 to get overlapped images to reach a reasonable amount of precision (edges visually aligned). It was also important to not concentrate all correspondeces in a dense area if possible, spreading them out helped the warped image align. I also discovered one point that I had entered incorrectly, which dramatically distorted one of my warped images so accuracy is very important. I'm not experienced in photography, but it is interesting to see the lighting changes toward the edges of the lense. Not sure what causes this, does the lens of a camera attenuate light at edges because of more extreme refraction? This posed a challenge when trying to blend the image boundaries. Here are the correspondences below:


0
200
400
600
0
200
400
600
800
1000
1200



The image below is the first pairing of warped images all targeting the center photo. It is hard to tell, but my Gaussian based Alpha blending approach is noticable in a few places. Unfortunately, the boundaries of images are still quite visible. I discuss this later in the *Blending Aproaches* section.


1000
1250
1500
1750
2000
2250
2500
2750
1500
2000
2500
3000
3500
4000
4500
5000


1000
1250
1500
1750
2000
2250
2500
2750
3000
1500
2000
2500
3000
3500
4000
4500
5000

Although the transformations are fairly well aligned, especially in the center, there is much to be desired on the image boundaries with blending.

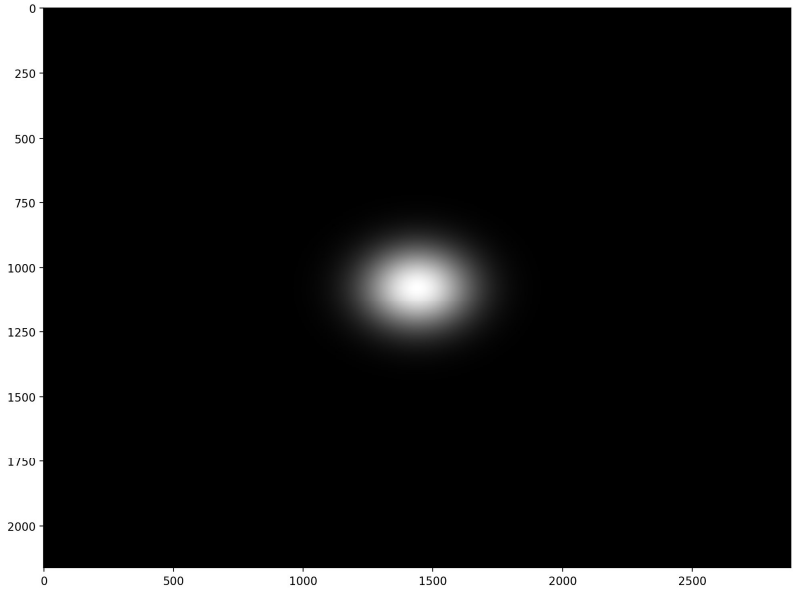
Approaches to blending:

I experimented quite a bit with simple averaging of overlapping images, however, this seemed to make the images quite distorted and noisy. I settled on using a form of Alpha blending. I read about Alpha blending techniques online, however, I wanted to slowly taper the blending to make transitions less noticeable. I used a Gaussian centered on the canvas (location of the 'target' as I have coined it) to control the contribution of the warped image and the target image. To describe this technique intuitively, the Gaussian curve has a peak (unity) at the center and 100% of grayscale contributions come from the target image. As the coordinates move away from the center, descending the Gaussian curve, the contribution of the target approaches zero and the warped image contribution approaches 100%. A rough mathematical description is as follows:

$$Canvas\_value_{x,y} = (\text{target}(x, y) * \text{norm}(0, .1, x, y)) + (\text{to\_warp}(x, y) * (1 - \text{norm}(0, .1, x, y))).$$

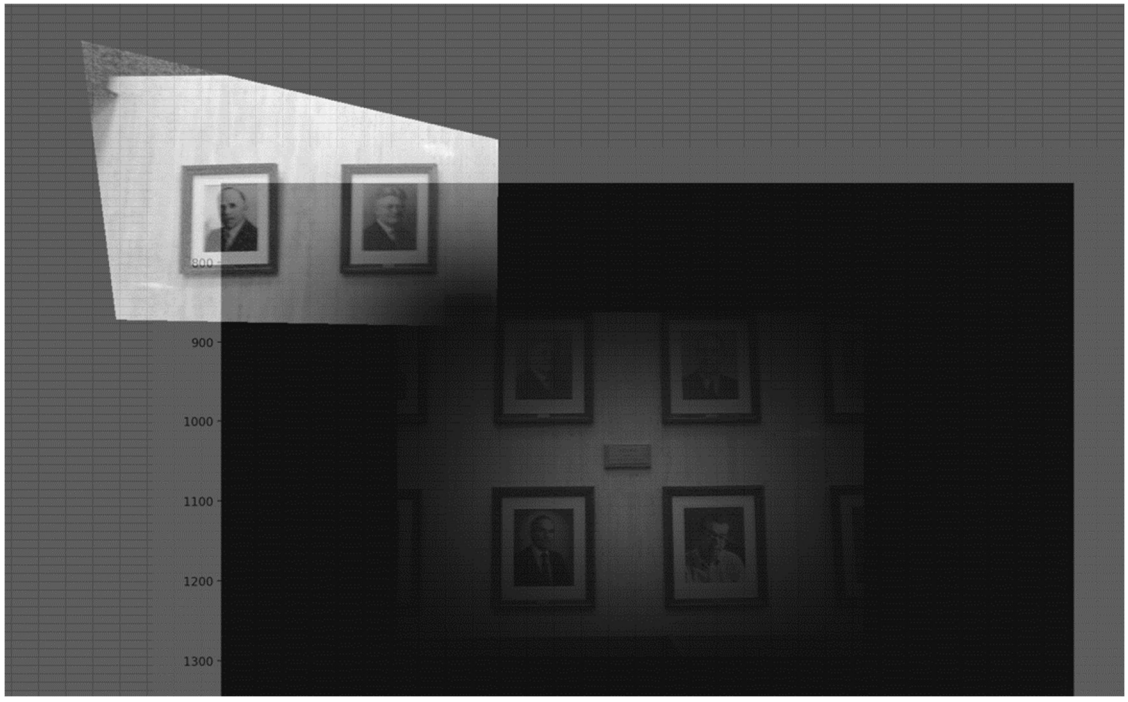
This results in some distortion and blurring and doesn't fully resolve disjointed boundaries (particularly further from the center), however, it does create a smooth center of the mosaic. I chose a sigma value of 0.1. More experimentation with this sigma value might lead to better results, I didn't have enough time to extensively experiment. One additional note, in the case that all images overlap in the center, this means the center image will not be placed in the center as it has already been blended into the canvas. In the provided data set (portraits), the center image (target), was not obscured or overlapped completely ball all other images. This required my algorithm to create masks which describe the space shared by two overlapping images. These 'union' masks allowed me to place the images on one canvas and blend them.

Here is the sigma = 0.1, zero mean Gaussian I chose for Alpha blending centered in the canvas.


0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500

Here is the Gaussian applied to the target.

To illustrate this Alpha blending using a Gaussin, here are two image transparencies (adjusted opacity), to show the Gaussian applied to the target and a negative of the Gaussian applied to the warped image.



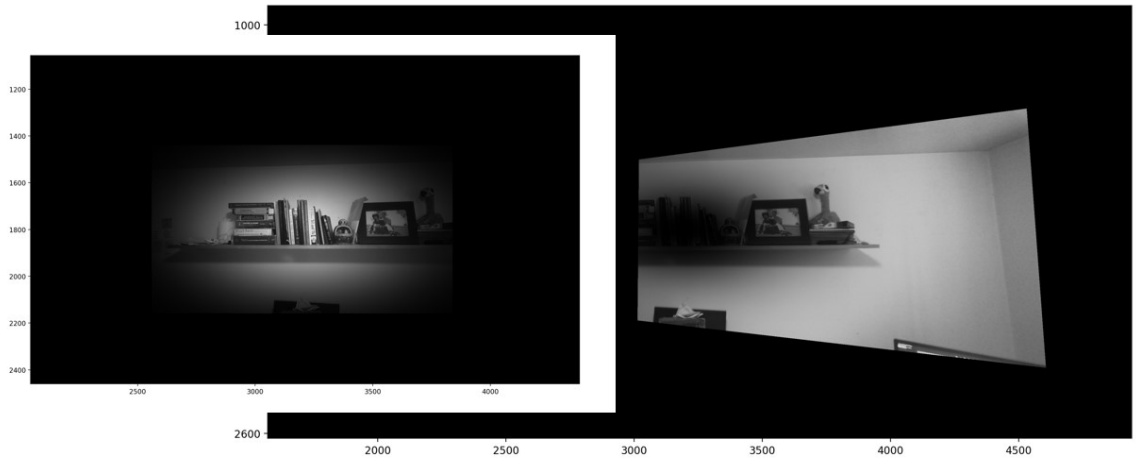
As the images are aligned on the canvas, you can see the complementary nature of the Gaussian and negative Gaussin to blend the two images moving from the center outward.

If you look closely, you can see the blending (causing some blurring) at the center of the image where the target has been blended into the two warped images being built around the target.

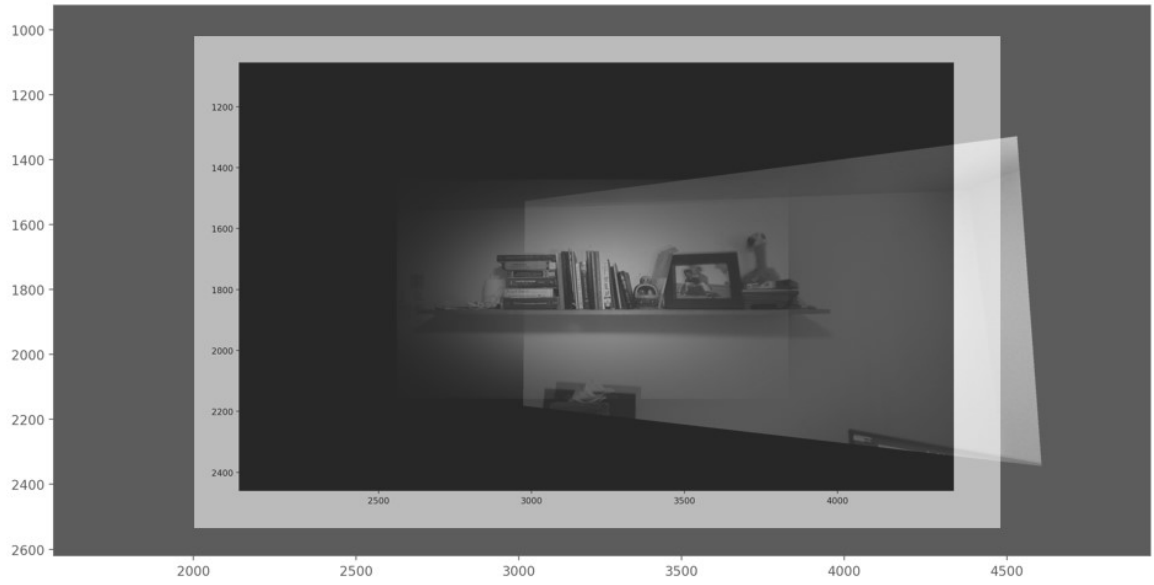
I placed a red arrow to point out the boundary of the blending that is visible. Note that a symmetric gaussian in x and y is not ideal, an improvement would be to stretch the Gaussin appropriately in each dimension. This assymetry leads to uneven application of the blending which can reveal boundaries.

Here is the final cropped canvas/mosaic.

Here is the same Alpha blending using a Gaussian on my panoramic images. These are transparencies placed in Excel to show the complementary nature of the blending.



Here are the aligned transperencies:



Here is the final crop of the mosaic below. Notice the red arrows pointing to the triangle in the ceiling. I point this out because this is the only part of the image with the original center image (target). You can also faintly see the boundary of the center image blended into the left hand warped image. Also not the hard edges in the lower center image. This is a flaw in my algorithm that I wasn't able to resolve without extensive blurring and averaging. These hard boundaries are also the case with the top right hand warped image. Gaussians with larger sigma values might mitigate this issue. Additionally, placing a floor under the Gaussian (target always makes a contribution in Alpha blending) might also be a good approach to resolve this.