CS 6640 Project 4

Travis Allen, u1056595

December 6, 2021

Contents

1 Preliminaries

I chose to do project 4a: Image Mosaicing. The code for this project can be found in the `proj_4.py` and `functions.py` files. `proj_4.py` contains the actual algorithm developed to make a mosaic of greyscale images whose correspondences are documented in a `.json` file. `functions.py` contains some useful functions that would have otherwise crowded the `proj_4.py` file. My solution to this project relies on `numba` for some parts, so you must have that installed to run my code. I use it because it dramatically speeds up the run time of the feathering algorithm. I have also included `.json` files for the images shown in this report, as well as the images themselves.

2 Methods

To build the mosaics, I set up a system of equations as follows:

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

Where each (x_i, y_i) is a known correspondence point in the base image and each (x_i', y_i') is a known correspondence point in the image to be transformed, and each p_{ij} is unknown. The building of this equation occurs in `proj_4.py` lines `97-119`. Then, I solve this system with the singular value decomposition. This process is performed in the `svd_solve()` function in `functions.py` lines `7-25`. This gives the coefficients $p_{i,j}$ to form the following transformation matrix:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{23} & p_{23} \\ p_{31} & p_{32} & 1 \end{pmatrix}$$

Each homogeneous coordinate in the image is multiplied by this transformation. The result is a non-homogeneous coordinate, so it is normalized by the third coordinate in each coordinate vector so that the resulting coordinates are homogeneous. The result of this process is the locations of each intensity with regards to the base image (non-primed coordinates in the above equation). This is implemented in the `transform()` function in `functions.py` lines `28-56`.

At this point, I am able to use these tools to build mosaics of perspectively equivalent images with known correspondence points. I heard some talk among other students in the class that a way to improve the quality of the mosaic is to introduce some interpolation to fill in the black lines created in the mosaics shown in later parts of this report. I interpret these lines as the "contours" of the warp, and they are produced because after both the multiplication of each coordinate by the transformation matrix, and the normalization of each coordinate to a homogeneous coordinate, the result is a floating point number and not an integer. Because array indices must be integers, a decision must be made here about how to treat these floating point indices. I tried using the `numpy.ceil()`, `numpy.floor()`, and `numpy.round()` functions, and found the results to be very similar, so much that they were indistinguishable to a casual observer. In all cases, these methods leave black lines that are visible on the image. **There was no mention of a need to implement a more sophisticated interpolation method than these in the project instructions, so I haven't done that.** However, I recognize that it would likely add to the quality of the mosaic, so I've described my hypothetical procedure for implementing interpolation below.

If I had more time to complete this project, I would implement the following scheme for interpolation. First I would take the untransformed image and apply the transformation matrix P to each of its coordinates, and then normalize them in exact accordance with the procedure described above. Then, I would determine the smallest integer bounding box of this transformed image (essentially the shadow of the transformed image) and transform the coordinates that fit inside that box with the inverse transformation matrix, P^{-1}. The result of this process are "new" floating point coordinates. It seems to me (informed by my limited foray into this method)

that determining the bounding box and the pixels that sit inside it is the crux of this method. Then, I would take the original integer coordinates of the untransformed image and use those and their associated intensities to create a 2D interpolation function with `scipy.interpolate.interp2d()`. I would then use this 2D interpolation function to map the "new" floating point coordinates to their respective locations on the canvas. This process is shown in the following diagram for a marginal increase in clarity.

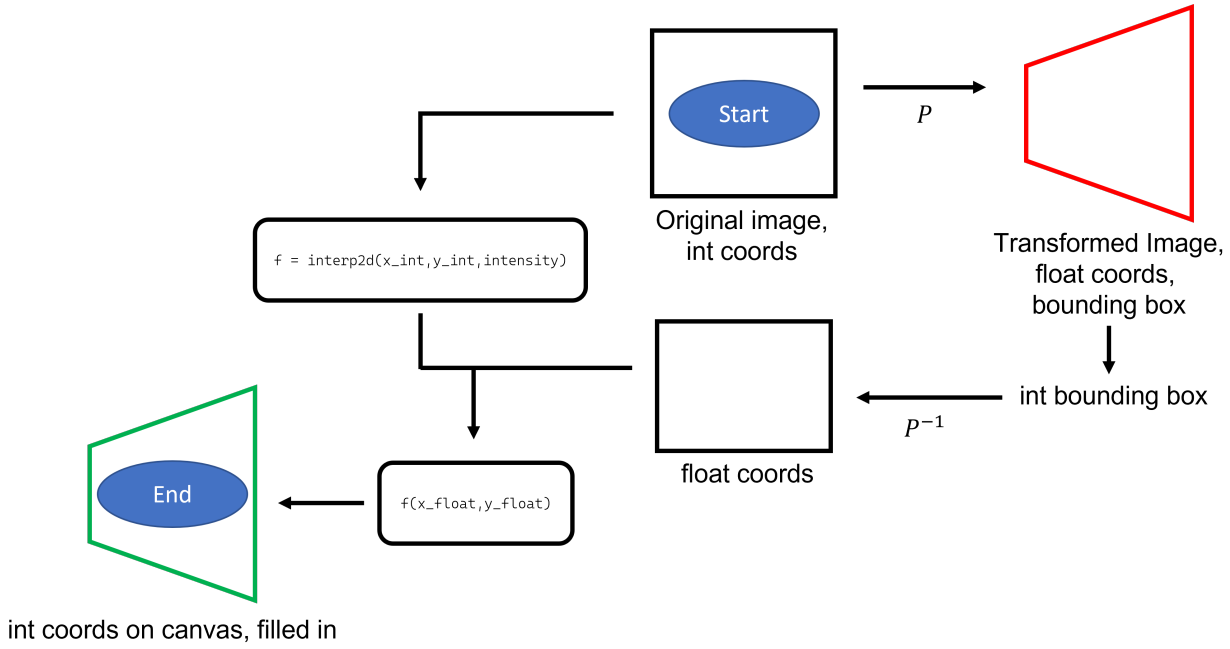

Figure 1: Proposed interpolation procedure.

3 Experiments

3.1 Given Dataset

I took the given images and formed the following mosaic with the procedure described above.

Figure 2: 4 perspectively equivalent images made into a mosaic.

The large change in perspective of these images amplifies the black lines caused by the rounding method described above. These lines are most apparent in the upper left corner of the image.

3.2 Panoramic Images

I used the following panoramic images to see how the algorithm performs on them. This is using 10 correspondence points.

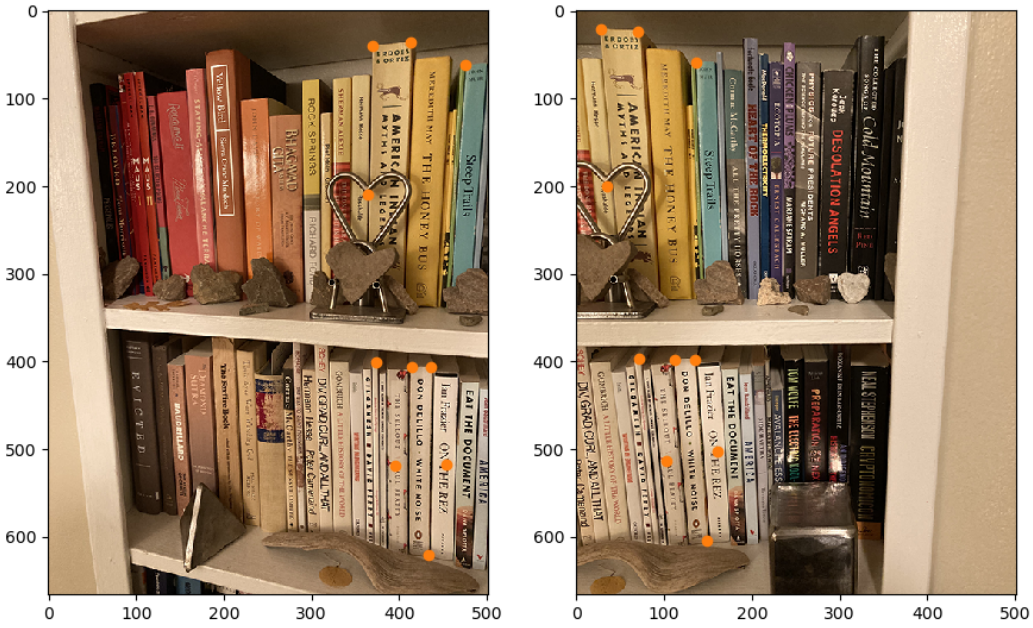

Figure 3: Bookshelf with 10 correspondences. I picked this image because it contains many corners, so accurately picking correspondences was straightforward.

Figure 4: Mosaiced bookshelf with 10 correspondences.

Once again, the large perspective change amplifies the effect of the rounding method to introduce many warp contours.

3.3 Planar Images

I used the following planar images to see how the algorithm performs on them. Here I used 12 correspondence points, shown below.

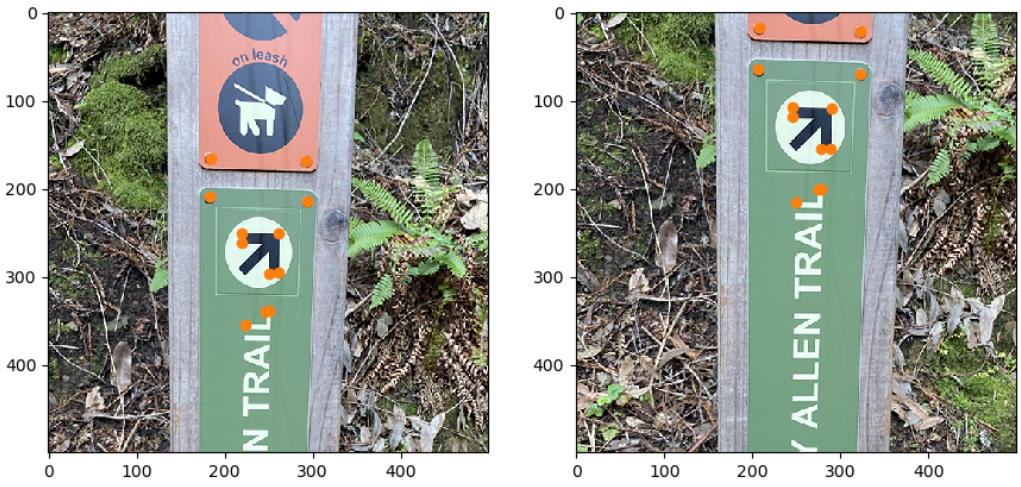

Figure 5: Trail sign with 12 correspondences. I picked this image because it contains many corners, so accurately picking correspondences was straightforward.

Figure 6: Mosaiced trail sign with 12 correspondences.

Here, the minimal warping and minimal perspective change results in fewer contour lines and a less confusing image.

3.4 Number of Correspondences

I experimented with the number of correspondence points on both the planar and panoramic images.

3.4.1 Planar Images

I tried to use 4 correspondence points on the planar image shown above. I chose the number 4 because it is the theoretical minimum number of correspondence points needed to solve for the transformation. This worked with varying degrees of success. Some combinations of 4 of my total 12 correspondence points produced a mostly valid transformation, shown below. However some combinations of correspondence points did not work as well and would have produced a very large and ugly image. I attribute this to the correspondence points being chosen by hand, so they were not extremely accurate.

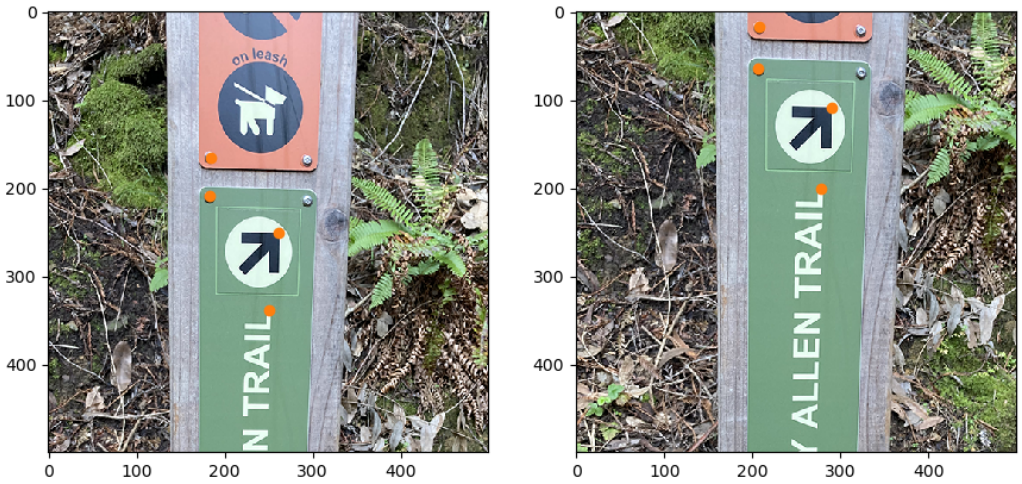

Figure 7: Trail sign with 4 correspondences.

Figure 8: Mosaiced trail sign with 4 correspondences. Things don't line up very well with so few correspondences.

Shown below again for clarity's sake is the same pair of images but constructed with a transformation informed by 12 correspondence points. This seemed to act as a filter on my imprecise correspondence point picking and generally make the image look better.

Figure 9: Mosaiced trail sign with 12 correspondences. This lines up much better.

3.4.2 Panoramic Images

I attempted to use 4 correspondence points on the panoramic images shown above, but I was unsuccessful. I encountered the same issue I described in the section on planar images where the transformation that was produced made too large of an image (orders of magnitude larger than the original images), so I added one correspondence point and successfully made a mosaic with 5 points. I attribute the lack of success to the large perspective change between the images. This is shown below.

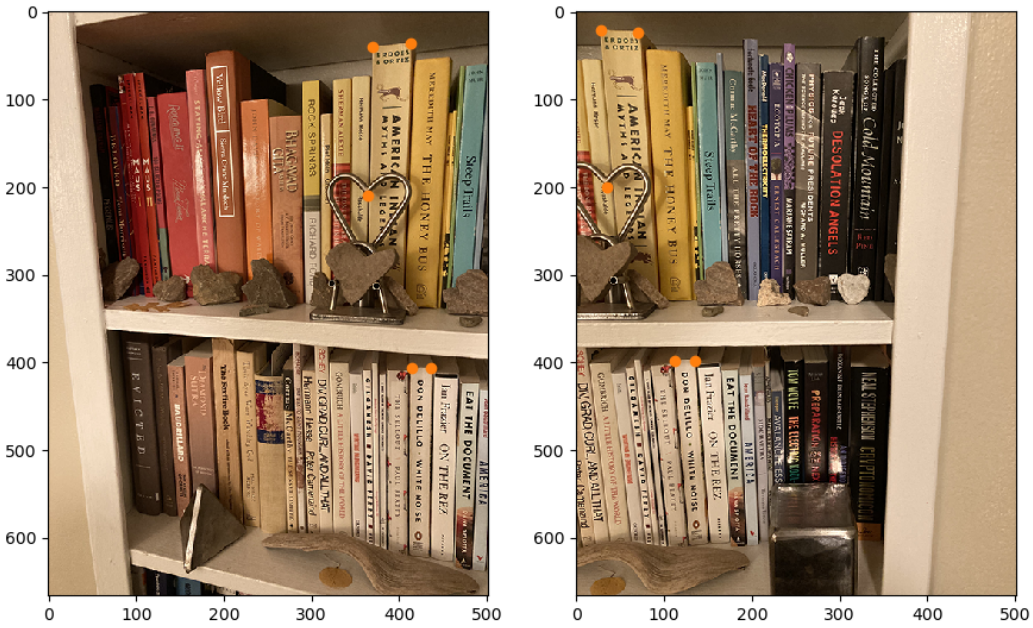

Figure 10: Bookshelf with 5 correspondences. I picked this image because it contains many corners, so accurately picking correspondences was straightforward.

Figure 11: Mosaiced bookshelf with 5 correspondences. Notice the misalignment at the bottom left corner of the overlapping image.

Shown again for clarity are the results from above with 10 correspondence points. In general, the image lines up slightly better, especially near the bottom.

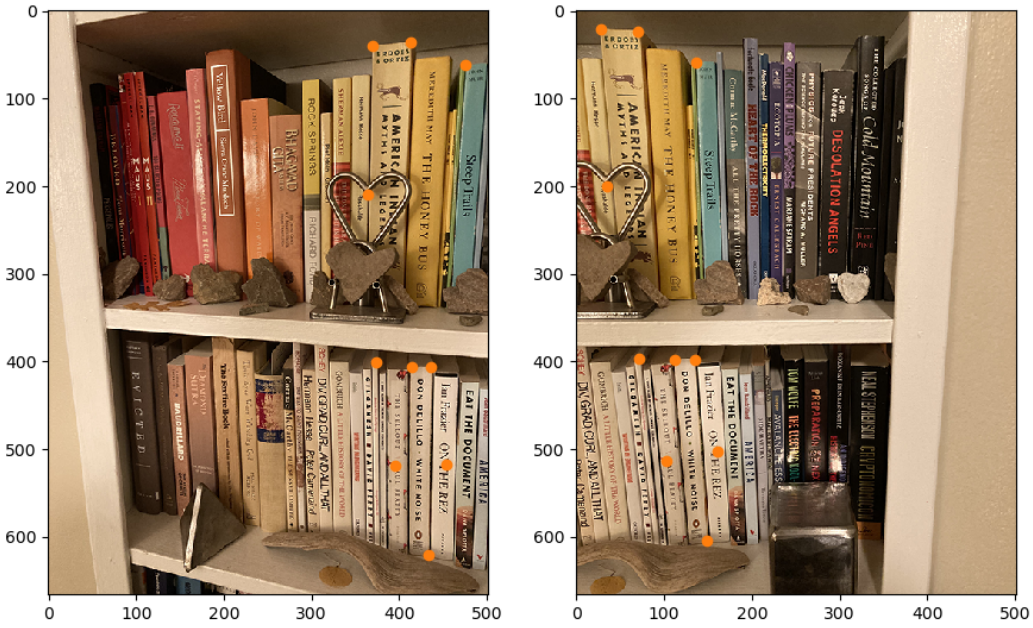

Figure 12: Bookshelf with 10 correspondences.

Figure 13: Mosaiced bookshelf with 10 correspondences.

4 Questions

4.1 How many control points does it take to get a 'good' transformation between images?

My results in the preceding sections indicate that having more correspondence points results in a better image mosaic. Solving a system with SVD provides the least-squares solution, so this comports with our intuition regarding least-squares solutions and the central limit theorem, wherein the "quality" or "truth" of a prediction improves with the number of data points that are used to make it. From my own initial experimentation it seems that 8 or greater correspondence points capture enough of the truth of the perspective equivalence to construct a convincing image mosaic.

4.2 How does the algorithm behave at the theoretical minimum of the number of control points?

The theoretical minimum number of correspondence points is 4 because that is the minimum possible number of points that allows the matrix below to have full rank. If this matrix has at least rank 8 and the number of rows is equal to the rank, then the system has an analytical solution. If the system has at least rank 8 and the number of rows is greater than the rank, then the system has a least-squares solution.

$$\begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3x_3' & y_3x_3' \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4x_4' & y_4x_4' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3y_3' & y_3y_3' \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4y_4' & y_4y_4' \end{pmatrix} \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{21} \\ p_{23} \\ p_{23} \\ p_{31} \\ p_{32} \end{pmatrix} = \begin{pmatrix} -x_1' \\ -x_2' \\ -x_3' \\ -x_4' \\ -y_1' \\ -y_2' \\ -y_3' \\ -y_4' \end{pmatrix}$$

The algorithm is temperamental when used with 4 correspondence points, and I attribute this to human error when picking correspondence points, as well as the inevitable lack of alignment/registration between the world frame and the camera frame. This means that physical objects in the world may get mapped to different fractions of pixels, and then as a result get rounded differently. The algorithm was particularly temperamental when I tried to use 4 correspondence points on the images of the bookshelf, so I ended up having a practical minimum number of 5 correspondence points. With just 4 points the transformation stretched out the second image to several orders of magnitude larger than it originally was, which was either unachievable with the current canvas size or indecipherable as an image with a canvas large enough to permit the image.

4.3 From your experiments, how does the accuracy of the control points affect the results?

To test this, I swapped two control points in one of the two images being mosaiced. I did this for an image that had 10 control points to begin with so that this experiment didn't completely derail the procedure. The results are shown below and they are wildly different from the results in the parts above. This indicates that the method is indeed sensitive to the accuracy of the control points, but not as bad as one might think. At first I only swapped one control point, but the results were not too bad. I imagine that this is due to the least-squares solution acting as a filter on outliers like this.

Figure 14: Mosaic with swapped control points. Swapping only two pairs of close-by points caused this effect.

5 Details

5.1 Contrast

A good algorithm should automatically adjust for major intensity differences.

To handle this I used the `skimage.exposure.match_histograms` function to match all of the histograms to the first image that I read in to the program. Below are some results with and without histogram matching. This seemed like the most logical way to solve this problem.

Figure 15: Mosaic with no histogram matching. Differences in intensity are very obvious and distracting.

Figure 16: Mosaic with histogram matching. Differences in intensity are much less severe and are in general less distracting.

5.2 Feathering

To blur the edges, I moved an averaging box filter around the edges of each warped image that I placed. The results are minimally effective but they are shown below. The code to execute this is located in the `feather()` function in the `functions.py` module included with this report. First, I show a representative path that the filter would move around to blur, then I show the image. Note that all of the images shown until this point have had their edges feathered with this method.

Figure 17: Representative path of an edge blending filter. This follows the perimeter of the overlaid image.

Figure 18: Edge blended mosaic. Results are minimally noticeable but add to the legibility.

5.3 Image Size

Initially, for n number of images, I make a canvas that is $n + 1$ times the size of the largest image so that I have enough room to work with when placing images in the mosaic. However, this often results in a canvas which is much larger than it needs to be. To return the canvas to a more reasonable size for viewing once the mosaic is complete, I execute the following procedure. I search through the large canvas to find the first and last rows and columns which contain only zero elements. I do this by using the `numpy.sum()` function on each row and column and checking to see if it is equal to 0. The canvas consists of all zeros before I place images on it, so this method works by assuming that a row of all zeros contains no image information. I perform it this way because I figure that `numpy`'s vectorization is faster than my own implementation of computing the sum or individually inspecting every element in the image. This is implemented in the `trim_canvas()` function in `functions.py`.

6 Notes on Implementation

- You must use my `read_json.py` file.
- You must use my `mosaic_params.json` file, and my other `.json` files provided with this report.
- If you choose to use a separate `.json` file, ensure that the ordered pairs of the correspondences match the format in my images and parameter files.
- Ensure that there is a folder called `test_images` in the directory that the python files are located.
- I have included a folder called `test_images` in my submission in the correct location. It contains the base images I have used in this report.