EECE 5639 - Project 2

Image Mosaicing

Kevin Russell Eugen Feng

2023-03-10

Abstract

This research project investigates the image mosaicing technique for the purpose of creating a single mosaic from two images. The process involves finding corresponding features in two images, pairing similar corners, estimating the homography matrix, and warping an image onto the coordinate system of the second image. Our report presents a detailed exploration of the image mosaicing process, the implementation of our program, the experiments we conducted, the tuning parameters we used, the observations we found, and our conclusions.

Description

Our paper focuses on investigating the effectiveness of corner matching techniques by using the Harris Corner Detector algorithm. We leveraged this technique to detect and extract corners from both images. Similar corners were paired by applying a normalized cross-correlation technique on image patches centered at each corner to identify correspondences. The RANSAC algorithm was used to remove any outliers, and the least-squares algorithm was applied to estimate the homography matrix. Finally, we used the homography matrix to warp one image onto the other, which created a single panoramic view. Our results showed that our approach achieves high accuracy in identifying corresponding corners and produces high-quality image mosaics.

Implementation

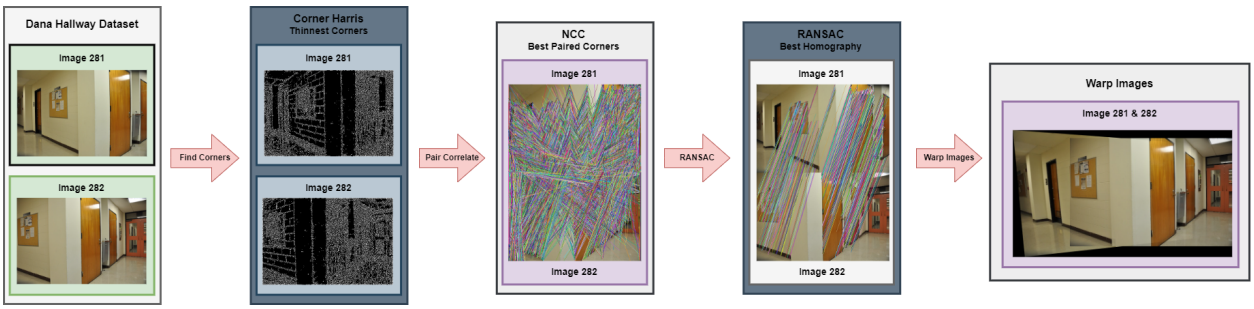

Figure 1: *Image Mosaic Process*

Harris Corner Detector

In this study, we explored the process of implementing and optimizing the Harris Corner Detector for the purpose of image mosaicing. We calculated the M matrix by convolving the image derivatives and multiplying matrices. The r equation, $r = \left|M\right| - k \sum_{i=1}^{n} M_{ii}$, was then derived from the M matrix and applied a threshold to obtain the brightest corners for both images. Non-max suppression was used to thin out the corners and find the highest peak. The resulting thin corners are shown on a black canvas (see Figure 1). Through experimentation, we were able to optimize the Harris Corner Detector parameters for more accurate corner matching and ultimately improve the overall quality of the image mosaicing process.

Normalized Cross-Correlation

To find normalized cross-correlation, we extracted 5x5 image patches in two lists of thirty thousand patches per list. These patches had normalized cross-correlation calculated by first subtracting the mean of each patch in f and g, following the equation $f = f - \frac{\sum_{j=1}^{n}\sum_{k=1}^{m} f_{ijk}}{n}$ and $g = g - \frac{\sum_{j=1}^{n}\sum_{k=1}^{m} g_{ijk}}{n}$. Next, f and g were normalized by their respective standard deviation by using the equations $\hat{f} = \frac{f}{\sqrt{\sum_{j=1}^{n}\sum_{k=1}^{m} f_{ijk}^2}}$ and $\hat{g} = \frac{g}{\sqrt{\sum_{j=1}^{n}\sum_{k=1}^{m} g_{ijk}^2}}$. Matrix multiplications and summations were performed to calculate the normalized cross-correlation, as given by equation $ncc = \sum_{k=1}^{n}\sum_{l=1}^{m} \hat{f_{ijklm}} \hat{g_{ijklm}}$. For color images, the NCC values are averaged for the second dimension to reduce the channels to one. To find paired matches, a threshold was applied, and the most highly correlated correspondences were paired and stored in a list.

RANSAC and Homgraphy Matrix

To estimate the homography matrix, we used the RANSAC algorithm by randomly sampling points from both lists of paired correspondences. We implemented Hartley's Precondition by normalizing the correspondences, creating the A matrix with four corresponding points, and leveraging SVD to find H matrix with equation $H = T_2^{-1} S_2^{-1} H_{norm} S_1 T_1$. A prediction is made by multiplying the H matrix with all paired correspondences, normalizing the points, calculating the distance using the equation $\sqrt{\sum_{i=1}^{n} (y - y_0)^2}$, applying a threshold to the distance, and counting how many inliers are found within the band. The algorithm selects the best homography by greedily finding the band with the most inliers. We ran the algorithm until the desired number of iterations, N, were met, and we applied the least squares regression, $\sum_{i=1}^{n} (y - y_0)^2$, before returning the best homography.

Warping

In order to stitch together two images, we utilized the warp perspective function from the cv2 package. However, it was necessary to create a new canvas to accommodate both images. A translation was performed to fix image two's coordinate system on the center of the new canvas' coordinate system. Image one is warped onto the new canvas with image two stitched onto the canvas. We utilized the addWeighted function from the cv2 package to blend the images seamlessly, which results in a single cohesive mosaic image.

Extra Credit

In the extra credit assignment, we implemented a technique to warp one image into a specific region within the second image. To achieve this, we utilized the mouse click points feature from the matplotlib library to obtain four coordinates that define the frame. Next, we calculated the homography matrix and warped the first image onto the second image's coordinate system. We then stitched the second image on top of the first

image within the frame region, and blended the two images together using the composite function from the Python Imaging Library (PIL).

Experiments

In our experiments, we explored the tuning of the normalized cross-correlation (NCC) and RANSAC algorithms in the context of image mosaicing. To maintain consistency with the original image size of 340x512x3, we manually tuned the Harris Corner Detector and focused on tuning NCC and RANSAC. To do this, we implemented a method that involved iterating through potential settings for both algorithms. However, to avoid overfitting, we utilized cross-validation to split the corners after NCC into 5 training and 5 test datasets. RANSAC was then trained on the training dataset, and predictions were calculated for accuracy using the true positive formula of $\frac{\sum_{i=1}^{n} TP_i}{n}$ to calculate the number of inliers. We selected the best accuracy and found a tuning that was on average accurate. In total, we conducted 1440 experiments by iterating through the different parameters for NCC and RANSAC. It is important to note that this method may be susceptible to overfitting, and thus, we manually vetted the results to ensure the accuracy of the tuning.

Harris Corner Detector

For Harris Corner Detector, we experimented with various parameters. Throughout our experiments, we consistently used a 3x3 block filter and a k value of 0.04 for Harris Corner Detector. We did manually tune the threshold by trying six different threshold percentages:

- 0.01, 0.02, 0.03, 0.04, 0.05, and 0.06

We assessed the performance of the detector by examining the resulting black and white canvas, ensuring that the right number of corners were captured without introducing too much noise. In total, we conducted six trials to tune the method, and determined that the best threshold is 0.04 as shown in Figure 2.

Figure 2: *[Harris Corner Algorithm] A picture of a hallway with doors. (left) Threshold 0.01 was used that yielded a noisy canvas with too many corners. (center) Threshold 0.04 was used that had a balanced picture of corners. (right) Threshold 0.06 was used where all corners were cleaned out*

Normalized Cross-Correlation

In order to tune the normalized cross-correlation method, we iterated through the following threshold values:

- 0.93, 0.94, 0.95, 0.96, 0.97, and 0.98

Our experimentation showed that the threshold value for NCC had a significant impact on the homography matrix, as a lower threshold yielded more paired corners while a higher threshold yielded fewer. When the

threshold was set too high, there were not enough correlated points to train the RANSAC algorithm, which results in poor estimation. Conversely, a lower threshold allowed too many outliers to be captured, which leads to a less accurate homography matrix estimation. We determined the optimal threshold value of 0.93 to be used for NCC during our experiments.

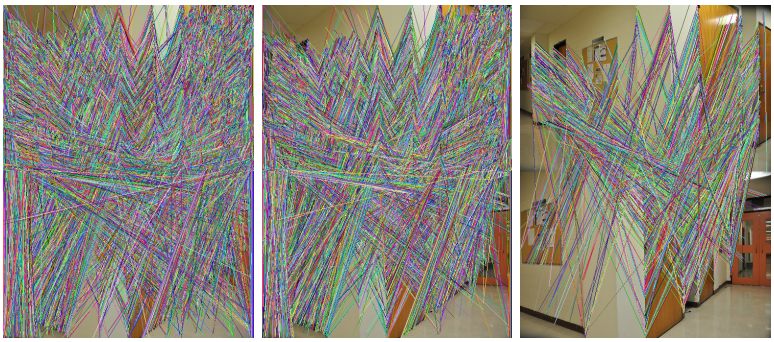

Figure 3: *[NCC] A picture of a hallway with doors and correlated points. (left) Threshold 0.93 was used that yielded alot of correlated points. (center) Threshold 0.97 was used that had a balanced amount of correlated points. (right) Threshold 0.997 was used where very few correlated points were used*

RANSAC and Homgraphy Matrix

To tune the RANSAC algorithm, we varied the RANSAC threshold from 0.5 to 10, the number of iterations from 500 to 5000, and the number of sampled points from 3 to 6. We found that higher RANSAC thresholds yielded better results when used with lower NCC thresholds. Additionally, higher number of iterations improved accuracy whereas a lower number of iterations decreased accuracy. We also found that four sampled points consistently provided better results. Our experiments had a NCC threshold of 0.93, a RANSAC threshold of 5, 5000 iterations, and four sampled points, which yielded the best accuracy.

Figure 4: *[NCC] A picture of a hallway with doors and corresponding points. (left) NCC Threshold is 0.93, RANSAC threshhold is 0.5, RANSAC iterations is 500, and 3 sample points. (center) NCC Threshold is 0.93, RANSAC threshhold is 5, RANSAC iterations is 5000, and 4 sample points. (right) NCC Threshold is 0.95, RANSAC threshhold is 10, RANSAC iterations is 5000, and 4 sample points*

Warping

To assess the accuracy of the tuned RANSAC algorithm, we evaluated the quality of image warping. Overfitting can be a concern when tuning RANSAC, therefore it is important to evaluate the homography matrix estimation. We found that the combination of NNC threshold at 0.93, RANSAC threshold at 5, RANSAC iterations at 5000, and RANSAC sample points at 4 produced the best warp compared to the other tuned parameters. From the results, it was observed that the left image in Figure 5 had the worst warp, with the

first image being far away from the coordinate system of the second image. On the other hand, the center picture had the best warp while the right picture was overfitted, resulting in images that were excessively warped.

Figure 5: *[NCC] Two pictures glued together as a mosaic. (left) NCC Threshold is 0.93, RANSAC threshhold is 0.5, RANSAC iterations is 500, and 3 sample points. (center) NCC Threshold is 0.93, RANSAC threshhold is 5, RANSAC iterations is 5000, and 4 sample points. (right) NCC Threshold is 0.95, RANSAC threshhold is 10, RANSAC iterations is 5000, and 4 sample points*

Observations

In our research, we observed the effect of tuning the NCC and RANSAC algorithms on the accuracy of our image mosaicing technique. Through our observations, we found that increasing the NCC and RANSAC thresholds generally led to more inliers, but increasing the RANSAC threshold above 5 resulted in overfitting. Our results, as shown in Figure 6, indicate that a NCC threshold of 0.93 yields the highest accuracy on average, while thresholds of 0.94 and above lead to overfitting. In addition, we found that increasing the RANSAC threshold beyond a certain point also led to overfitting, as seen in the accuracy drop after a threshold of 5. These findings highlight the importance of carefully tuning parameters to avoid overfitting in image mosaicing.

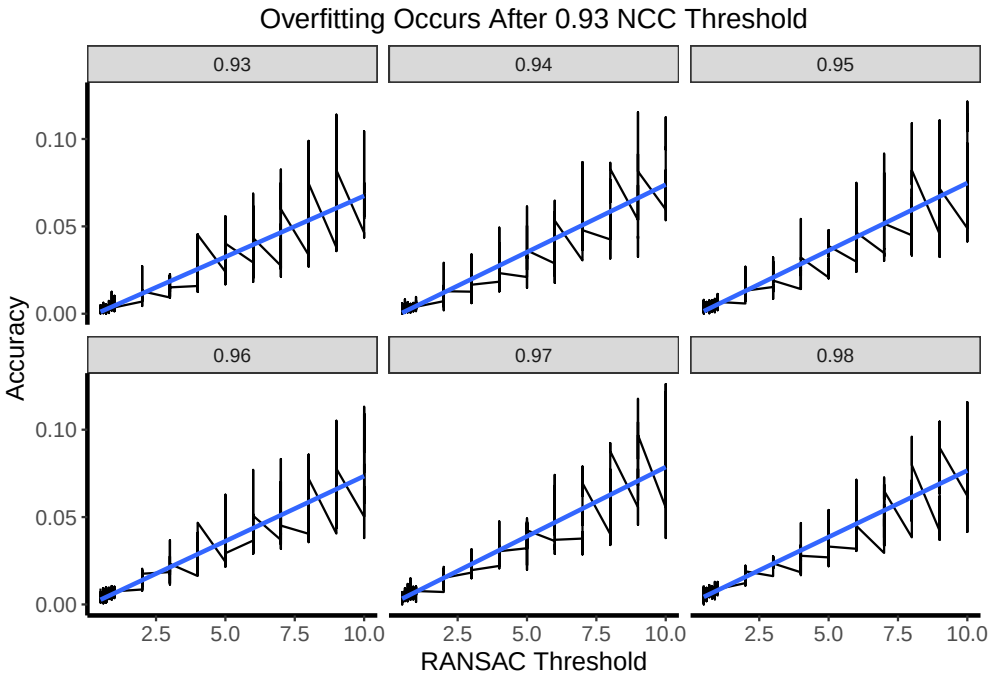

Figure 6: *A comparison of NCC threshold and RANSAC threshold by accuracy*

We investigated the tuning of RANSAC max iterations and sample points to determine their effect on accuracy. By comparing accuracies, we observed that the accuracy remained relatively constant between 500 to 2000 iterations, but increased slightly when the iterations were set to 5000, as shown in Figure 7. This suggests that higher iterations can result in better accuracy. To further validate this observation, we manually tested various iterations and found that max iterations at 15000 to 20000 tend to have a better fit than iterations below 5000.

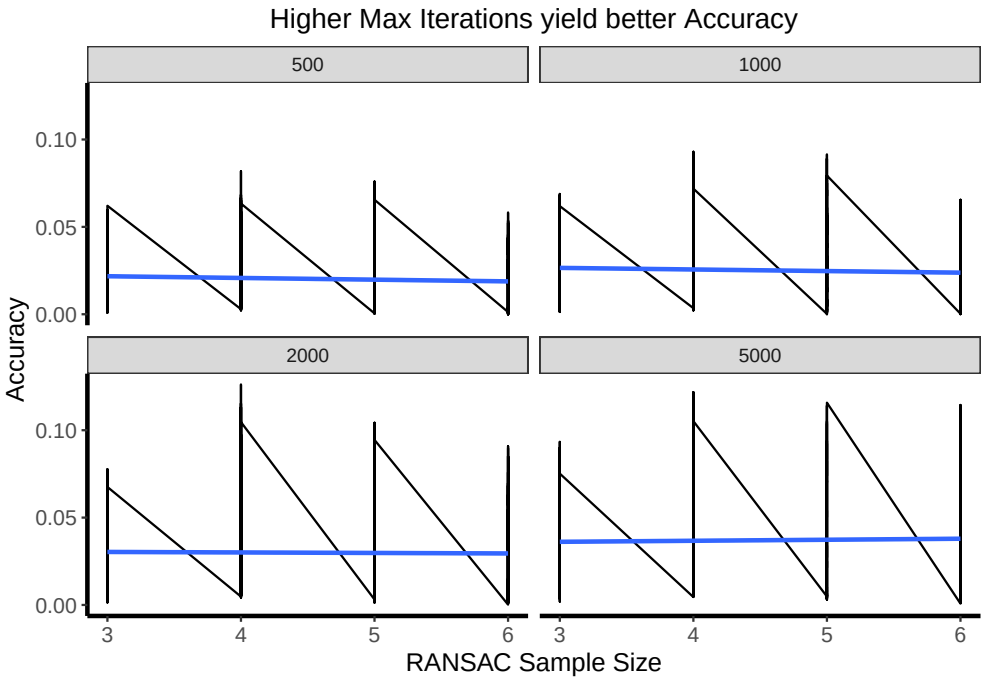

Figure 7: *A comparison of RANSAC iterations and RANSAC sample size by accuracy*

Extra Credit

In addition to our primary objective of creating image mosaics, we also explored the task of warping a single image into a frame region of another image for extra credit. This was accomplished by applying the same methodology used in creating mosaics. We were able to successfully warp several images into interesting planes, demonstrating the versatility of our approach.

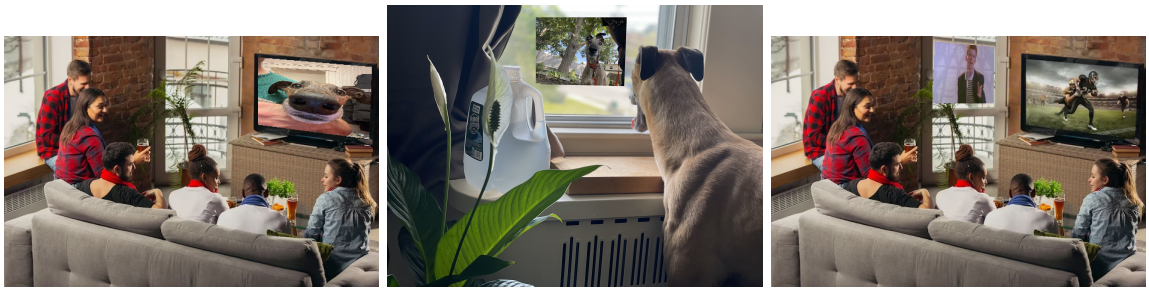

Figure 8: *(left) A family watching TV that shows a close up of a dog. (center) A greyhound staring at a poster on the window of a greyhound standing in the woods. (right) warped image onto window*

Conclusion

Overall, we successfully implemented image mosaicing by finding corresponding features, pairing similar corners together, estimating the homography matrix, and warping one image onto the other's coordinate system. We manually tuned the Harris Corner Detector, and used cross-validation to tune the normalized cross-correlation and RANSAC algorithms. Our experiments showed that the optimal parameters were NCC threshold at 0.93, RANSAC threshold at 5, RANSAC iterations at 5000, and RANSAC sample points at 4, resulting in the best warp accuracy. We observed that overfitting can occur when increasing RANSAC threshold above 5 or NCC threshold above 0.95. Furthermore, we found that RANSAC max iterations at 15000 to 20000 tend to have a better fit than iterations below 5000. Our results suggest that these parameters are optimal for image mosaicing and can be applied in various computer vision applications.

Appendix

```
from numpy import zeros_like, array, float32, ndarray, ones, zeros, sqrt as np_sqrt,\
    arctan2, pi, argwhere, column_stack, copy as np_copy, newaxis, vstack, hstack, linalg,\
    random, where, column_stack
from PIL import Image, ImageChops
from sklearn.model_selection import KFold
from PIL.Image import fromarray, open as pil_open
from os.path import isdir, exists, join
import os
from typing import Callable
from math import ceil
from random import randint
import numpy as np
import torch
from copy import deepcopy
import cv2
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from pandas import DataFrame


'''
    Image Sequence Folder Directories with Image Sequence start photo and end photo
'''

# Default Image Settings
default_img_dirs = {1: {
    'img_base_dir': r'.\DanaHallWay1',
    'filename_prefix': 'DSC_',
    'filename_ext': 'JPG',
    'filename_zfill': 4,
    'image_id_start': 281,
    'image_id_end': 282,
    'block_size': (3, 3),
    'k': 0.04,
    'patch_size': 5,
    'thresh_perc': 0.04,
    'match_corner_thresh': 0.7,
    'match_corner_chunk_size': 250,
    'black_white_patches': False,
    'ransac_threshold': 0.1,
    'ransac_max_iter': 1000,
    'ransac_sample_size': 4,
    'tune_homography': True},
    2: {
    'img_base_dir': r'.\DanaOffice',
    'filename_prefix': 'DSC_',
    'filename_ext': 'JPG',
    'filename_zfill': 4,
    'image_id_start': 308,
    'image_id_end': 317,
    'block_size': (3, 3),
```

```
    'k': 0.04,
    'patch_size': 5,
    'thresh_perc': 0.01,
    'match_corner_thresh': 0.7,
    'match_corner_chunk_size': 250,
    'black_white_patches': False,
    'ransac_threshold': .1,
    'ransac_max_iter': 1000,
    'ransac_sample_size': 4,
    'tune_homography': True}}

# Testing Image Settings
test_img_dirs = {1: {
    'img_base_dir': r'.\DanaHallWay1',
    'filename_prefix': 'DSC_',
    'filename_ext': 'JPG',
    'filename_zfill': 4,
    'image_id_start': 281,
    'image_id_end': 282,
    'block_size': (3, 3),
    'k': 0.04,
    'patch_size': 5,
    'thresh_perc': 0.04,
    'match_corner_thresh': .93,
    'match_corner_chunk_size': 250,
    'black_white_patches': False,
    'ransac_threshold': 5,
    'ransac_max_iter': 20000,
    'ransac_sample_size': 4,
    'tune_homography': False},
    2: {
    'img_base_dir': r'.\DanaOffice',
    'filename_prefix': 'DSC_',
    'filename_ext': 'JPG',
    'filename_zfill': 4,
    'image_id_start': 308,
    'image_id_end': 309,
    'block_size': (3, 3),
    'k': 0.04,
    'patch_size': 5,
    'thresh_perc': 0.01,
    'match_corner_thresh': 0.7,
    'match_corner_chunk_size': 250,
    'black_white_patches': True,
    'ransac_threshold': .1,
    'ransac_max_iter': 1000,
    'ransac_sample_size': 4,
    'tune_homography': True}}


# Global Settings
image_setting = test_img_dirs[1]
```

```
class ImageMosaic(object):
    """
    Class with all the necessary functions to create an Image Mosaic.
    We decided to use the Corner Harris method, Normalized Cross-Correlation,
    RANSAC, least squares homography, and planar warping with blending
    """

    def __init__(self,
                 img_base_dir: str,
                 filename_prefix: str,
                 filename_ext: str,
                 filename_zfill: int,
                 image_id_start: int,
                 image_id_end: int):
        """
        Initialize Image Mosaic class that controls various functions to create an
        image mosaic

        @param img_base_dir: Base directory for sequence of images dataset
        @type img_base_dir: String
        @param filename_prefix: Prefix name for filename of sequence of images
        @type filename_prefix: String
        @param filename_ext: Extension for filename
        @type filename_ext: String
        @param filename_zfill: Number padding for image ID (i.e. zfill 4 is 0000)
        @type filename_zfill: Integer
        @param image_id_start: Starting ID for image number pad
        @type image_id_start: Integer
        @param image_id_end: Ending ID for image number pad
        @type image_id_end: Integer
        """

        # Ensure that img_base_dir is an existing path and valid directory
        assert isdir(img_base_dir) and exists(img_base_dir)

        # Initialize parameters
        self.sobel_x = array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
        self.sobel_y = array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
        self.dialate_kernel = ones((3, 3))
        self.img_dir = img_base_dir
        self.filename_prefix = filename_prefix
        self.filename_ext = filename_ext
        self.filename_zfill = filename_zfill
        self.image_id_start = image_id_start
        self.image_id_counter = image_id_start
        self.image_id_end = image_id_end
        self.convolve_sum_func = lambda x, y: (x * y).sum()

    def create_mosaic(self,
                      block_size: tuple = (3, 3),
                      k: float = 0.04,
                      patch_size: int = 5,
                      thresh_perc: float = 0.01,
```

```
                match_corner_thresh: float = 0.7,
                match_corner_chunk_size: int = 1000,
                black_white_patches: bool = False,
                ransac_threshold: float = .1,
                ransac_max_iter: int = 1000,
                ransac_sample_size: int = 4,
                tune_homography: bool = False,
                ncc_thresholds: list = [.93, .94, .95, .96, .97, .98],
                rans_thresholds: list = [.5, .6, .7, .8, .9,
                                         1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                rans_iters: list = [500, 1000, 2000, 5000],
                rans_samples: list = [3, 4, 5, 6]):
    """

    @param block_size:
    @type block_size: Tuple
    @param k:
    @type k: Float
    @param patch_size:
    @type patch_size: Integer
    @param thresh_perc:
    @type thresh_perc: Float
    @param match_corner_thresh:
    @type match_corner_thresh:
    @param match_corner_chunk_size:
    @type match_corner_chunk_size:
    @param black_white_patches:
    @type black_white_patches:
    """

    def find_homograph(patches1,
                       patches2,
                       corners1,
                       corners2,
                       ncc_thresh,
                       ran_thresh,
                       ran_iters,
                       ran_sample,
                       train=False):
        c1, c2 = self.match_corners(patches1,
                                    patches2,
                                    corners1,
                                    corners2,
                                    threshold=ncc_thresh,
                                    chunk_size=match_corner_chunk_size,
                                    black_white=black_white_patches)

        if train:
            accuracy = list()
            kf = KFold(n_splits=5, shuffle=True, random_state=42)

            for train_indices, test_indices in kf.split(c1):
                train_c1, train_c2, test_c1, test_c2 = c1[train_indices],\
```

```
                                                    c2[train_indices],\
                                                    c1[test_indices],\
                                                    c2[test_indices]

                train_h, _, _ = self.ransac(train_c1, train_c2,
                                            threshold=ran_thresh,
                                            max_iter=ran_iters,
                                            sample_size=ran_sample)
                if np.any(train_h):
                    test_acc = self.ransac_predict(train_h,
                                                   test_c1,
                                                   test_c2,
                                                   ran_thresh,
                                                   method='test')
                    accuracy.append(test_acc)

            return np.mean(accuracy)
        else:
            H, ind, inliers = self.ransac(c1, c2,
                                          threshold=ran_thresh,
                                          max_iter=ran_iters,
                                          sample_size=ran_sample)
            return H, ind, inliers, c1, c2

    # find corners for first image in sequence
    print('Finding corners for Starting Image: %s' % '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter).zfill(
                self.filename_zfill),
            self.filename_ext))
    prev_image = self.get_next_image()
    print('Image Shape:', array(prev_image).shape)
    prev_corners = self.find_corners(prev_image, block_size, k, thresh_perc)
    prev_corners = column_stack((prev_corners[:, 1], prev_corners[:, 0]))

    # Get patches for first image in sequence
    print('Getting patches for Starting Image: %s' % '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter - 1).zfill(
                self.filename_zfill),
            self.filename_ext))
    prev_patches = self.get_patches(prev_image,
                                    prev_corners,
                                    patch_size,
                                    black_white=black_white_patches)

    # Iterate among image sequence to process mosaic
    for img_id in range(self.image_id_start + 1,
                        self.image_id_end + 1):
        # Find corners for next image in sequence of images
        print('Finding corners for Image: %s' % '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter).zfill(
```

```
            self.filename_zfill),
        self.filename_ext))
    curr_image = self.get_next_image()
    curr_corners = self.find_corners(curr_image, block_size, k, thresh_perc)
    curr_corners = column_stack((curr_corners[:, 1], curr_corners[:, 0]))

    # Get patches for current image
    print('Getting patches for Image: %s' % '{}{}.{}'.format(
        self.filename_prefix,
        str(self.image_id_counter - 1).zfill(
            self.filename_zfill),
        self.filename_ext))
    curr_patches = self.get_patches(curr_image,
                                    curr_corners,
                                    patch_size,
                                    black_white=black_white_patches)

    # Find the normalized cross-correlation between previous image and current image
    print('Finding Best Corners via NCC for Images (%s, %s)' % (
        '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter - 1).zfill(
                self.filename_zfill),
            self.filename_ext),
        '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter - 2).zfill(
                self.filename_zfill),
            self.filename_ext)
    ))
    if tune_homography:
        best_tune = {'ncc_thresh': None,
                 'rans_thresh': None,
                 'rans_iter': None,
                 'rans_sample': None,
                 'accuracy': float('-inf')}
        trials = list()

        for ncc_threshold in ncc_thresholds:
            for rans_threshold in rans_thresholds:
                for rans_iter in rans_iters:
                    for rans_sample in rans_samples:
                        mean_acc = find_homograph(prev_patches,
                                                  curr_patches,
                                                  prev_corners,
                                                  curr_corners,
                                                  ncc_threshold,
                                                  rans_threshold,
                                                  rans_iter,
                                                  rans_sample,
                                                  train=tune_homography)
                        tune = dict()
                        tune['ncc_thresh'] = ncc_threshold
```

```
                            tune['rans_thresh'] = rans_threshold
                            tune['rans_iter'] = rans_iter
                            tune['rans_sample'] = rans_sample
                            tune['accuracy'] = mean_acc

                            if best_tune['accuracy'] < tune['accuracy']:
                                _, indices, _, corners1, corners2 = find_homograph(
                                    prev_patches,
                                    curr_patches,
                                    prev_corners,
                                    curr_corners,
                                    tune['ncc_thresh'],
                                    tune['rans_thresh'],
                                    tune['rans_iter'],
                                    tune['rans_sample'])

                                if np.any(indices):
                                    print('Best Tune:', tune)
                                    best_tune = deepcopy(tune)
                                    self.draw_corr_corners(
                                        prev_image,
                                        curr_image,
                                        corners1[indices],
                                        corners2[indices],
                                        filename_prefix=
                                        'tune_{}_nthresh_{}_rthresh_{}_riter_{}_rsample_{}_acc'.for
                                            tune['ncc_thresh'],
                                            tune['rans_thresh'],
                                            tune['rans_iter'],
                                            tune['rans_sample'],
                                            tune['accuracy']
                                        ),
                                        filedir=r'./tuning/')
                            trials.append(deepcopy(tune))
                            print('Train:', best_tune['accuracy'], tune)

                DataFrame(trials).to_csv('output.csv', index=False)
                h, indices, num_inliers, corners1, corners2 = find_homograph(
                    prev_patches,
                    curr_patches,
                    prev_corners,
                    curr_corners,
                    tune['ncc_thresh'],
                    tune['rans_thresh'],
                    tune['rans_iter'],
                    tune['rans_sample'])
            else:
                h, indices, num_inliers, corners1, corners2 = find_homograph(
                    prev_patches,
                    curr_patches,
                    prev_corners,
                    curr_corners,
                    match_corner_thresh,
```

```
                    ransac_threshold,
                    ransac_max_iter,
                    ransac_sample_size)

            self.draw_corr_corners(prev_image,
                                   curr_image,
                                   corners1,
                                   corners2,
                                   filename_prefix='color_corr')

            self.draw_corr_corners(prev_image,
                                   curr_image,
                                   corners1[indices],
                                   corners2[indices],
                                   filename_prefix='color_ransac')

            print(h)

            print('Generating warped image.')
            self.warp_image(prev_image,
                            curr_image,
                            h,
                            filename_prefix='warped_image')

            print('Starting Extra Credit')
            #With matplotlib
            extracredit1 = r'./pics/pic1.png'
            extracredit2 = r'./pics/base2.png'

            self.extra_credit(
                image1=extracredit1,
                image2=extracredit2)

    def get_next_image(self) -> Image:
        """
        Get the next image in the image sequence

        @return: Next image in the Image sequence if it exists
        @rtype: Image
        """

        # assemble file path for next image
        fp = join(self.img_dir, '{}{}.{}'.format(
            self.filename_prefix,
            str(self.image_id_counter).zfill(
                self.filename_zfill),
            self.filename_ext))

        # ensure that image exists
        assert exists(fp)

        # open image as a PIL image and return image
        img = pil_open(fp)
```

```
        self.image_id_counter += 1
        return img

    def find_corners(self, image: Image,
                     block_size: tuple = (3, 3),
                     k: float = 0.04,
                     thresh_perc: float = 0.01) -> ndarray:
        """
        Find corners in an image by calculating the r equation from
        harris corner detection.

        @param image: Image that you wish to find corners
        @type image: PIL Image
        @param block_size: block size for neighborhood in calculating Sx2, Sy2, Sxy
        @type block_size: tuple
        @param k: K value for harris corner r equation
        @type k: float
        @param thresh_perc: Percent threshold for r threshold after non-max suppression
        @type thresh_perc: float
        @return: Array of corners from Harris Corner detector
        @rtype: Numpy Array
        """

        # Copy Image
        bw_img = image.copy()

        # Convert to black and white image
        if bw_img.mode != 'L':
            bw_img = bw_img.convert("L")

        # make as numpy array and find the derivatives by applying sobel filters
        bw_img = array(bw_img)
        dx = self.convolve(bw_img, self.sobel_x)
        dy = self.convolve(bw_img, self.sobel_y)

        # Calculate dx2, dy2, and dxy
        dx2, dy2, dxy = dx ** 2, dy ** 2, dx * dy

        # Find Sx2, Sy2, and Sxy
        neighborhood = ones(block_size)
        neigh_dx2 = self.convolve(dx2, neighborhood)
        neigh_dy2 = self.convolve(dy2, neighborhood)
        neigh_dxy = self.convolve(dxy, neighborhood)

        # Calculate R equation
        r = (neigh_dx2 * neigh_dy2) - (neigh_dxy ** 2) - k * ((neigh_dx2 + neigh_dy2) ** 2)

        # Find magnitude and direction by using dx2, dy2, dx, and dy values
        magnitude = np_sqrt(dx2 + dy2)
        direction = arctan2(dy, dx) * 180 / pi

        # Find the non-max suppression for r
        r = self.non_max_suppression(r, magnitude, direction)
```

```
        # copy r, save image as non-bright r equation after non-max suppression
        r_bright = np_copy(r)
        fromarray(r).convert("L").save('unbright_corner_{}{}.png'.format(
            self.filename_prefix,
            str(self.image_id_counter-1).zfill(
                self.filename_zfill - 1)))

        # Make a bright version of the r equation after non-max suppression and save img
        r_bright[r > thresh_perc * r.max()] = 255
        r_bright[r <= thresh_perc * r.max()] = 0
        fromarray(r_bright).convert("L").save('bright_corner_{}{}.png'.format(
            self.filename_prefix,
            str(self.image_id_counter-1).zfill(
                self.filename_zfill - 1)))

        # return corners by thresholding r
        return argwhere(r > thresh_perc * r.max())

    @staticmethod
    def non_max_suppression(image: ndarray,
                            magnitude: ndarray,
                            direction: ndarray) -> ndarray:
        """
        Thins corners to get rid of non-max items by suppressing it in the image

        @param image: Array representing a gray picture
        @type image: Numpy Array
        @param magnitude: Magnitude calculated by using sqrt(dx2 + dy2)
        @type magnitude: Numpy Array
        @param direction: Directions calculated by arctan2(dy, dx) * 180 / pi
        @type direction: Numpy Array
        @return: Picture that is non-max suppressed (thinning of corners)
        @rtype: Numpy Array
        """

        # generate list of angles to suppress non-max items
        angles = array(list(range(1, 14, 2))) * (360.0 / 16)

        # iterate image with i, j indices
        for i in range(1, magnitude.shape[0] - 1):
            for j in range(1, magnitude.shape[1] - 1):
                # apply magnitude value to neigh1, neigh2 according to the direction and
                # where that direction points to between the below angles
                if (0 <= direction[i, j] < angles[0]) or (157.5 <= direction[i, j] <= 180):
                    neigh1 = magnitude[i, j + 1]
                    neigh2 = magnitude[i, j - 1]
                elif angles[0] <= direction[i, j] < angles[1]:
                    neigh1 = magnitude[i + 1, j + 1]
                    neigh2 = magnitude[i - 1, j - 1]
                elif angles[1] <= direction[i, j] < angles[2]:
                    neigh1 = magnitude[i + 1, j]
                    neigh2 = magnitude[i - 1, j]
                elif angles[2] <= direction[i, j] < angles[3]:
```

```
                    neigh1 = magnitude[i + 1, j - 1]
                    neigh2 = magnitude[i - 1, j + 1]
                elif angles[3] <= direction[i, j] < angles[4]:
                    neigh1 = magnitude[i, j - 1]
                    neigh2 = magnitude[i, j + 1]
                elif angles[4] <= direction[i, j] < angles[5]:
                    neigh1 = magnitude[i - 1, j - 1]
                    neigh2 = magnitude[i + 1, j + 1]
                elif angles[5] <= direction[i, j] < angles[6]:
                    neigh1 = magnitude[i - 1, j]
                    neigh2 = magnitude[i + 1, j]
                elif angles[6] <= direction[i, j] < angles[7]:
                    neigh1 = magnitude[i - 1, j + 1]
                    neigh2 = magnitude[i + 1, j - 1]
                else:
                    neigh1 = 255
                    neigh2 = 255

                # if the two neighbors values are smaller than the image magnitude then...
                if magnitude[i, j] >= neigh1 and magnitude[i, j] >= neigh2:
                    # apply magnitude to image at that location
                    image[i, j] = magnitude[i, j]
                else:
                    # apply 0 at that image location
                    image[i, j] = 0

        # return image
        return image

    def get_patches(self,
                    image: Image,
                    corners: ndarray,
                    patch_size: int = 5,
                    black_white: bool = False) -> ndarray:
        """
        Find patches centered at the corners given a defined patch size

        @param image: Image to get patches from
        @type image: PIL Image
        @param corners: Array of corners to find patches of images
        @type corners: Numpy Array
        @param patch_size: Size of patch for each corner
        @type patch_size: Integer
        @param black_white: (default False) Convert Image to black and white
        @type black_white: Bool
        """

        # Copy Image
        bw_img = image.copy()

        # Calculate padding size
        pad_h, pad_w = patch_size // 2, patch_size // 2
```

```
        # Convert to black and white image
        if bw_img.mode != 'L' and black_white:
            bw_img = bw_img.convert("L")

        rgb_img = self.pad(array(bw_img), (pad_h, pad_w))
        patches = list()

        for corner in corners:
            patches.append(
                rgb_img[pad_h + corner[1] - patch_size // 2:pad_h + corner[1] + patch_size // 2 + 1,
                pad_w + corner[0] - patch_size // 2:pad_w + corner[0] + patch_size // 2 + 1])

        return array(patches, dtype=object)

    @staticmethod
    def pad(image: ndarray,
            padding: tuple) -> ndarray:
        """
        Create a padding for the image

        @param image: Image to apply padding
        @type image: Numpy Array
        @param padding: Padding dimensions (height, width)
        @type padding: Numpy Array
        @return: Image with padding added to image
        @rtype: Numpy Array
        """

        if len(image.shape) > 2:
            pad_h, pad_w = padding
            img_h, img_w, img_z = image.shape
            z = zeros((img_h + 2 * pad_h, img_w + 2 * pad_w, img_z))
            z[pad_h:img_h + pad_h, pad_w:img_w + pad_w] = image
        else:
            pad_h, pad_w = padding
            img_h, img_w = image.shape
            z = zeros((img_h + 2 * pad_h, img_w + 2 * pad_w))
            z[pad_h:img_h + pad_h, pad_w:img_w + pad_w] = image

        for i in range(pad_h):
            z[i, pad_w:img_w + pad_w] = image[pad_h - i - 1, :]
            z[img_h + pad_h + i, pad_w:img_w + pad_w] = image[img_h - i - 1, :]

        for j in range(pad_w):
            z[:, j] = z[:, 2 * pad_w - j - 1]
            z[:, img_w + pad_w + j] = z[:, img_w + pad_w - j - 1]

        return z

    def convolve(self,
                 image: ndarray,
                 kernel: ndarray,
                 padding: (tuple, None) = None,
```

```
                conv_func: Callable = None) -> ndarray:
        """
        Convolve an image with a kernel patch

        @param image: Image to use convolve function
        @type image:  Numpy Array
        @param kernel: Patch/Filter to use convolve with
        @type kernel: Numpy Array
        @param padding: Padding size for padded image
        @type padding: Integer or None
        @param conv_func: Callable function to use when applying kernel
        @return: Image to convolve
        @rtype: Numpy Array
        """

        if conv_func is None:
            conv_func = self.convolve_sum_func

        if padding is None:
            pad_h, pad_w = (len(kernel) // 2, len(kernel) // 2)
        else:
            pad_h, pad_w = padding

        output = zeros_like(image)
        pad_image = self.pad(image, (pad_h, pad_w))

        for i in range(pad_h, len(image) + pad_h):
            for j in range(pad_w, len(image[0]) + pad_w):
                patch = pad_image[i - pad_h:i + pad_h + 1,
                        j - pad_w:j + pad_w + 1]
                output[i - pad_h, j - pad_w] = conv_func(patch,
                                                         kernel)

        return output

    @staticmethod
    def match_corners(patches: ndarray,
                      patches2: ndarray,
                      prev_corners: ndarray,
                      curr_corners: ndarray,
                      threshold: float = 0.7,
                      chunk_size: int = 500,
                      black_white: bool = False) -> tuple:
        """
        Calculates the normalized cross-correlation between patches and matches
        correlated corners together and returns two lists of correlated corners

        @param patches: Image Patches to find ncc
        @type patches:  Numpy Array
        @param patches2: 2nd Image Patches to find ncc
        @type patches2: Numpy Array
        @param prev_corners: Previous Corners to find best corners
        @type prev_corners: Numpy Array
```

```
        @param curr_corners: Current Corners to find best corners
        @type curr_corners: Numpy Array
        @param threshold: (default 0.7) Threshold value to select best ncc values
        @type threshold: Float
        @param chunk_size: (default 500) chunk_size to process patches x patches2
        @type chunk_size: Integer
        @param black_white: (Default False) Find corners using black white patches
        @type black_white: Bool
        @return: Image to convolve
        @rtype: Numpy Array
        """

        def subtract_mean(image):
            if black_white:
                return image - image \
                                    .mean(axis=1) \
                                    .mean(axis=1)[:, newaxis, newaxis]
            else:
                return image - image \
                                    .mean(axis=1) \
                                    .mean(axis=1)[:, newaxis, newaxis, :]

        def find_std(image):
            if black_white:
                return np_sqrt((image ** 2)\
                               .sum(axis=1)\
                               .sum(axis=1)[:,
                               newaxis, newaxis].astype(float32))
            else:
                return np_sqrt((image ** 2)\
                               .sum(axis=1)\
                               .sum(axis=1)[:,
                               newaxis, newaxis, :].astype(float32))

        # initialize parameters
        c_output = list()
        c_output2 = list()
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # remove mean from f and g
        f = subtract_mean(patches)
        g = subtract_mean(patches2)

        # calculate standard deviation for f and g
        f_std = find_std(f)
        g_std = find_std(g)

        # calculate f_hat (f_hat = f / f_std) and g_hat (g_hat = g / g_std)
        if black_white:
            f_hat = torch.tensor((f / f_std).astype(float32)[:,
                                 newaxis, :]).to(device)
            g_hat = torch.tensor((g / g_std).astype(float32)[newaxis,
                                 :, :]).to(device)
```

```
        else:
            f_hat = torch.tensor((f / f_std).astype(float32)[:,
                                 newaxis, :, :]).to(device)
            g_hat = torch.tensor((g / g_std).astype(float32)[newaxis,
                                 :, :, :]).to(device)

        # calculate number of iterations to calculate ncc by chunk size
        num_of_iter = ceil(f_hat.shape[0] / chunk_size)

        # iterate from 1 to number of iterations
        for i in range(1, num_of_iter + 1):
            # Find max ncc by the sum of f_hat (according to chunk)
            # and g_hat for only dimensions 2 and 3
            ncc = (f_hat[i * chunk_size - chunk_size:i * chunk_size] * g_hat) \
                .sum(dim=2).sum(dim=2)

            if not black_white:
                ncc = ncc.mean(dim=2)

            # Calculate max ncc values for dimension 1 and threshold values
            ncc = ncc.max(dim=1)
            mask = ncc[0] > threshold
            ncc = ncc[1][mask]

            # if gpu available for pytorch then...
            if torch.cuda.is_available() and ncc.nelement() != 0:
                # append correlated corners for prev_corners and curr_corners
                c_output.append(
                    prev_corners[i * chunk_size - chunk_size:i * chunk_size][mask.cpu().numpy()])
                c_output2.append(curr_corners[ncc.cpu().numpy()])
            elif ncc.nelement() != 0:
                # append correlated corners for prev_corners and curr_corners
                c_output.append(
                    prev_corners[i * chunk_size - chunk_size:i * chunk_size][mask.numpy()])
                c_output2.append(curr_corners[ncc.numpy()])

        # combine both stacks of chunks together and return two numpy arrays
        return vstack(c_output), vstack(c_output2)

    @staticmethod
    def ransac_predict(h: ndarray,
                       corners1: ndarray,
                       corners2: ndarray,
                       threshold: float,
                       method='train') -> tuple:
        pred = (h @ vstack((corners1.T, ones(len(corners1))))).T
        pred /= pred[:, 2].reshape(-1, 1)

        if method == 'train':
            dist = np_sqrt(((pred[:, :2] - corners2) ** 2).sum(axis=1))
            mask = dist < threshold
            inliers = mask.sum()
            return h, inliers, dist[mask].mean()
```

```
        else:
            preds = (np.linalg.norm(pred[:, :2] - corners2, axis=1) < threshold)
            return preds.sum() / len(preds)

    def ransac(self,
               best_corners: ndarray,
               best_corners2: ndarray,
               threshold: float = 0.1,
               max_iter: int = 1000,
               sample_size: int = 4):

        best_homography = np.array([])
        best_std = float('inf')
        best_num_of_inliers = 0

        for i in range(max_iter):
            indices = random.choice(best_corners.shape[0],
                                    size=sample_size,
                                    replace=False)
            src_points, dest_points = best_corners[indices], best_corners2[indices]
            h = self.compute_homography(src_points, dest_points)

            if not np.any(h):
                continue

            homography, num_of_inliers, std = self.ransac_predict(h,
                                                                   best_corners,
                                                                   best_corners2,
                                                                   threshold)

            if num_of_inliers > best_num_of_inliers or\
                    (num_of_inliers == best_num_of_inliers and std < best_std):
                # print('Best Inlier:', num_of_inliers, ', STD:', best_std)
                best_std = std
                best_homography = homography
                best_num_of_inliers = num_of_inliers

        if np.any(best_homography):
            p = (best_homography @ vstack([best_corners.T,
                                           ones(best_corners.shape[0]).T])).T
            p /= p[:, 2].reshape(-1, 1)
            d = np_sqrt(((p[:, :2] - best_corners2) ** 2).sum(axis=1))
            indices = where(d < threshold)
            src_points, dest_points = best_corners[indices], best_corners2[indices]
            h = self.compute_homography(src_points, dest_points)
        else:
            h = np.array([])

        if np.any(h):
            homography, num_of_inliers, std = self.ransac_predict(h,
                                                                   best_corners,
                                                                   best_corners2,
                                                                   threshold)
```

```
            return best_homography, indices, num_of_inliers
        else:
            return array([]), array([]), None

    def compute_homography(self,
                           src_pts: ndarray,
                           dest_pts: ndarray):
        try:
            src_pts_norm, t1 = self.norm_points(src_pts)
            dest_pts_norm, t2 = self.norm_points(dest_pts)
            a = list()

            for i in range(src_pts_norm.shape[0]):
                x1, y1 = src_pts_norm[i]
                x2, y2 = dest_pts_norm[i]
                a.append([x1, y1, 1, 0, 0, 0, -x1 * x2, -y1 * x2, -x2])
                a.append([0, 0, 0, x1, y1, 1, -x1 * y2, -y1 * y2, -y2])

            u, d, ut = linalg.svd(a)
            h = (ut[-1, :] / ut[-1, -1]).reshape(3, 3)
            return linalg.inv(t2) @ h  @ t1
        except:
            return np.array([])

    @staticmethod
    def norm_points(points: ndarray):
        points_mean = points.mean(axis=0)
        s = np_sqrt(2) / np_sqrt(((points - points_mean) ** 2).sum(axis=1)).mean()
        t = array([[s, 0, -s * points_mean[0]],
                   [0, s, -s * points_mean[1]],
                   [0, 0, 1]])
        norm_points = t @ column_stack((points, ones((len(points),)))).T
        return norm_points.T[:, :2], t

    def draw_corr_corners(self,
                          image1: Image,
                          image2: Image,
                          best_corners: ndarray,
                          best_corners2: ndarray,
                          filename_prefix: str = 'corr_corners',
                          filedir: str = r'./'):
        def get_color():
            r = randint(0, 255)
            g = randint(0, 255)
            b = randint(0, 255)
            return r, g, b

        w1, h1 = image1.size
        w2, h2 = image2.size
        image = Image.new('RGB', (max(w1, w2), h1 + h2))
        image.paste(image1, (0, 0))
        image.paste(image2, (0, h1))
        image = array(image)
```

```
        for i in range(best_corners.shape[0]):
            x1, y1 = best_corners[i].tolist()
            x2, y2 = best_corners2[i].tolist()
            y2 = y2 + h1
            cv2.line(image, (x1, y1), (x2, y2), get_color(), 1)

        fromarray(image).save('{}{}_{}{}_{}.png'.format(
            filedir,
            filename_prefix,
            self.filename_prefix,
            str(self.image_id_counter - 2).zfill(
                self.filename_zfill - 2),
            str(self.image_id_counter - 1).zfill(
                self.filename_zfill - 1)))

    def crop(self, image: ndarray):
        x_size = image[1,:].shape[0]
        y_size = image[:,1].shape[0]
        for i in range(y_size):
            #print(sum(sum(image[i,:])))
            if sum(sum(image[i,:])) > 255:
                top_of_img = i
                #print("Top of Image:",top_of_img)
                break

        for i in range(y_size-1,0,-1):
            #print(sum(sum(image[i,:])))
            if sum(sum(image[i,:])) > 255:
                bottom_of_img = i
                #print("Bottom of Image:",bottom_of_img)
                break

        for j in range(x_size):
            #print(sum(sum(image[:,j])))
            if sum(sum(image[:,j])) > 255:
                left_of_img = j
                #print("Left of Image:",left_of_img)
                break

        for j in range(x_size-1,0,-1):
            #print(sum(sum(image[:,j])))
            if sum(sum(image[:,j])) > 255:
                right_of_img = j
                #print("Right of Image:",right_of_img)
                break
        #print('Top:',top_of_img)
        #print('Bottom', bottom_of_img)
        #print('Left', left_of_img)
        #print('Right',right_of_img)
        return image[top_of_img:bottom_of_img,left_of_img:right_of_img],\
               left_of_img, bottom_of_img
```

```
    def warp_image(self,
                   image1: Image,
                   image2: Image,
                   h: ndarray,
                   filename_prefix: str = 'warped_image'):

        image1 = array(image1)
        image2 = array(image2)
        height1, width1, _ = image1.shape
        height2, width2, _ = image1.shape

        pts1 = float32([[0, 0],
                        [0, height1],
                        [width1, height1],
                        [width1, 0]]).reshape(-1, 1, 2)
        pts2 = float32([[0, 0],
                        [0, height2],
                        [width2, height2],
                        [width2, 0]]).reshape(-1, 1, 2)
        image = np.concatenate((pts1,
                                cv2.perspectiveTransform(pts2, h)),
                               axis=0)
        [x_min, y_min] = np.int32(image.min(axis=0).ravel() - 0.5)
        [x_max, y_max] = np.int32(image.max(axis=0).ravel() + 0.5)
        h_t = array([[1, 0, -x_min],
                     [0, 1, -y_min],
                     [0, 0, 1]])
        print(x_min,x_max,y_min,y_max)
        warped_image = cv2.warpPerspective(image1, h_t @ h,
                                           (x_max - x_min, y_max - y_min),
                                           borderMode=0)
        shift = (-x_min,-y_min)
        pano_size_y, pano_size_x, _ = warped_image.shape

        #The start of blending - Average Blending
        RGBA_warped = Image.new('RGBA',
                                size=(pano_size_x, pano_size_y),
                                color=(0, 0, 0, 0))
        RGBA_warped.paste(fromarray(image2), shift)
        RGBA_warped.paste(fromarray(warped_image),(0, 0))

        newimage2 = Image.new('RGBA',
                              size=(pano_size_x, pano_size_y),
                              color=(0, 0, 0, 0))
        newimage2.paste(fromarray(warped_image), (0, 0))
        newimage2.paste(fromarray(image2), shift)

        result = cv2.addWeighted(np.array(RGBA_warped),
                                 0.1,
                                 np.array(newimage2),
                                 0.9,0)
        result = fromarray(result).convert('RGB')
```

```
        #Saves the warped and stitched image into the working folder
        #os.chdir(self.img_dir)
        result.save("{}_{}{}_{}.png".format(
            filename_prefix,
            self.filename_prefix,
            str(self.image_id_counter - 2).zfill(
                self.filename_zfill - 2),
            str(self.image_id_counter - 1).zfill(
                self.filename_zfill - 1)))

    def extra_credit(self,
                     image1: str,
                     image2: str):

        matlab_img1 = mpimg.imread(image1)
        plt.switch_backend('wxAgg')
        matlab_img2 = mpimg.imread(image2)
        img2plot = plt.imshow(matlab_img2)
        print("Please select 4 coordinates")
        mng = plt.get_current_fig_manager()
        mng.frame.Maximize(True)
        #Records mouse coordinates
        mouse_coords = plt.ginput(4)
        plt.close()


        #The following finds the corner orientation of the coordinates
        mouse_coords = sorted([(int(i), int(j)) for i, j in mouse_coords])

        left = mouse_coords[:2]
        left = sorted(left, key=lambda i: i[1])
        top_left = left[0]
        bot_left = left[1]

        right = mouse_coords[2:]
        right = sorted(right, key=lambda i: i[1])
        top_right = right[0]
        bot_right = right[1]

        PIL_img1 = pil_open(image1)
        PIL_img2 = pil_open(image2)

        PIL_img1_arr = array(PIL_img1)
        PIL_img2_arr = array(PIL_img2)
        height1, width1, _ = PIL_img1_arr.shape
        height2, width2, _ = PIL_img2_arr.shape

        #The corner points of the image where we are warping
        img_pts1 = float32([[0, 0],
                            [0, height1],
                            [width1, height1],
                            [width1, 0]]).reshape(-1, 1, 2)
```

```
        #The corner points on the second image where we want the first image warped into
        img_pts2 = float32([top_left,
                            bot_left,
                            bot_right,
                            top_right]).reshape(-1, 1, 2)

        #This returns the homography matrix since we know the src and dst points of the warped image.
        #The points are found with mouse clicks
        h_mat = cv2.getPerspectiveTransform(img_pts1,img_pts2)
        warped_image = cv2.warpPerspective(PIL_img1_arr,
                                           h_mat,
                                           (width2, height2))

        print(warped_image.shape)

        warped_mask = Image.new('RGBA',
                                size=(width2,height2),
                                color=(0, 0, 0, 0))
        warped_mask.paste(fromarray(warped_image),(0, 0))

        blended_image = Image.composite(fromarray(warped_image),
                                        fromarray(PIL_img2_arr),warped_mask)

        blended_image.save("extra_credit_image.png")


if __name__ == '__main__':
    mosaic_mod = ImageMosaic(
        img_base_dir=image_setting['img_base_dir'],
        filename_prefix=image_setting['filename_prefix'],
        filename_ext=image_setting['filename_ext'],
        filename_zfill=image_setting['filename_zfill'],
        image_id_start=image_setting['image_id_start'],
        image_id_end=image_setting['image_id_end'])

    mosaic_mod.create_mosaic(
        thresh_perc=image_setting['thresh_perc'],
        block_size=image_setting['block_size'],
        k=image_setting['k'],
        patch_size=image_setting['patch_size'],
        match_corner_thresh=image_setting['match_corner_thresh'],
        match_corner_chunk_size=image_setting['match_corner_chunk_size'],
        black_white_patches=image_setting['black_white_patches'],
        ransac_threshold=image_setting['ransac_threshold'],
        ransac_max_iter=image_setting['ransac_max_iter'],
        ransac_sample_size=image_setting['ransac_sample_size'],
        tune_homography=image_setting['tune_homography'])
```