A Point Feature Matching-based Approach To Real-Time Camera Video Stabilization

Alvin Kim
Department of Electrical Engineering
alvink@stanford.edu
Juan Manuel Camacho
Department of Electrical Engineering
jcamach2@stanford.edu

Abstract—Unintentional and unwanted movement while holding a video camera can cause undesired jitters in the final output of the digital video. Video stabilization algorithms aim to align the frames in the video stream so that the entire video appears to be stabilized. However, these techniques differ significantly in the choice and complexity of image processing methods employed, and therefore, will vary in their performance and reliability. While some of these algorithms work robustly off-line, they may not be suitable for real-time processing, which, for example, is the case for automotive vision tasks. In this paper, we describe how we implemented a video stabilization algorithm using feature matching and image transform techniques that can work well in a real-time setting. We were able to test it on an Android smartphone, and managed to perform real-time video stabilization while holding and slowly moving the phone's camera.

Index Terms—Video Stabilization, FAST, ORB, Homography, Keypoints, Rotation, Translation, Real-Time

1 Introduction

The quality of digital video sometimes suffers from undesired effects such as image blurring, image distortion, uneven contrast, among others [2]. Unstable movement of hand-held devices such as smartphones, for example, can lead to visible, annoying jitters in the final video output stream. To automatically improve the quality of these digitally-acquired videos without the need for manual intervention from the user, applications employ camera video stabilization techniques.

The process of video stabilization aims to remove the unwanted movement of the video by repositioning and rotating the frames so that the video looks stable [1]. Although there exist numerous different methods for achieving this effect, video stabilization frameworks usually implement these 3 stages in sequence: motion estimation, motion smoothing and, finally, motion compensation [2]. Due to the interest in deploying video stabilization algorithms in power-constrained devices (including hand-held devices such as smartphones, for example), its imperative to develop methods that are not prohibitively computationally-intensive but that still yield reasonable results. Hardware-based methods to augment the camera lens and sensors' capabilities in acquiring images have been proposed [4]. However, these solutions are too expensive and too cumbersome to add on to ordinary cameras [4]. Therefore, a completely software-based solution is preferred.

In this paper, we describe our real-time Camera Video Stabilization algorithm, which we managed to implement in an Android smartphone. First, we start out by discussing related work that has been done in this field of study, and then proceed to motivate our choice of using standard feature detection and matching techniques. In the next section, we give background information in keypoint detection, feature matching and image transforms with the homography model. In the method overview section, we explain in detail the overall flow of our video stabilization pipeline. Finally, we discuss our experimental results and summarize our main challenges in implementing a computationally-intensive video processing algorithm in a real-time setting.

2 Related Work

Video stabilization has been a well-studied topic in the field of digital image processing, and many researchers have proposed widely different methods for achieving so. A set of these methods, for example, employ a technique called optical flow detection, which is the idea of detecting motion of objects between consecutive frames [6]. More specifically, optical flow detection models a 2D displacement vector using a Taylor series approximation, and then solves this system of equations to find the unknown $\frac{dx}{dt}$ and $\frac{dy}{dt}$ (displacement rate along the horizontal and vertical direction respectively) [6]. Liu et al. have used an approximation model of optical flow to perform video stabilization where they smoothed feature trajectories by smoothing motion vectors collected around the same pixels [7]. For optical flow detection to work, however, it's required to assume that the brightness of an object doesn't change across consecutive frames [7].

Another group of researchers at National University of Defense Technology (NUDT) have made the observation that over short-time intervals, homography transformations can capture frame motion without significant error accumulation [4]. Using this observation, Dong et al. used a

homography transform that is updated at each new frame for motion estimation and a novel Kalman filter for motion smoothing [4]. Given that for video recordings of a scene it's safe to assume that there will not be constant large translations of the camera, we will use the idea of employing a homography transform.

Our camera-video stabilization method has been largely inspired by the work of Kulkarni et al. [2]. We apply keypoint corner detection to detect unique features of each scene captured by the camera, estimate an image transform between consecutive frames, and warp each frame into the perspective reference of the previous frame [2]. However, while Kulkarni et al. managed to perform video stabilization with this method, they only applied it to video files that were stored on disk. We aim to apply a variant of this method to real-time video recordings without the constant need to save and load long video files.

Before proceeding to the description of our method, we first show what the desired end-result is of video stabilization. Figure 1 shows the state of a video sequence before stabilization (upper row) and after stabilization (lower row). We can see how in the second row, the center of the two people's figures are located around the same coordinate in each frame. This gives the illusion of a stabilized video stream, even though the camera slightly moved around during recording.

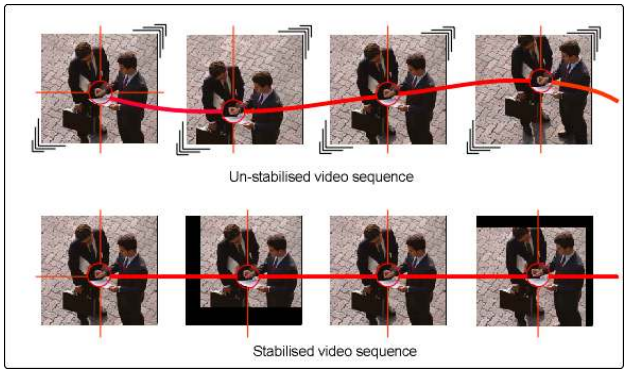

Fig. 1. Result of Video Stabilization [5]

3 Background Information

As mentioned before already, we decided to use keypoint corner detection and matching as our first two steps in our video stabilization pipeline. Corners are usually well-localized along multiple directions, which makes them good candidates as distinctive features of an image. Using these 'interest' points, we can then compute local information, also known as descriptors, about the area around those points, which then can be used for vision tasks such as image correspondence and simple object recognition.

Many methods have been developed to compute keypoint detection and descriptors, but the most well-known algorithms are SIFT and SURF [8]. However, although SIFT and SURF keypoint descriptors can be used for robustly estimating the homography transform between two frames that capture the same scene, they can be overly expensive for real-time video stabilization [8]. For this reason, we opted to use ORB, a computationally-efficient alternative to SIFT and SURF [8].

ORB, which stands for Oriented FAST and rotated BRIEF, is an algorithm for computing binary feature descriptors of an image. It first uses the FAST method for keypoint detection, which has been shown to work well in a real-time setting [8]. Figure 2 shows how FAST decides whether or not to classify a pixel point as a distinctive keypoint. Despite certain limitations in the robustness of FAST (for example, edges tend to get strong responses), this can be dealt with by augmenting the FAST method with the Harris corner test and pyramid schemes for scale [8].

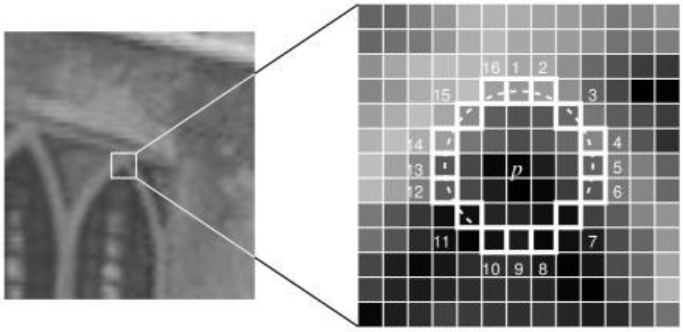

Fig. 2. Example of FAST Keypoint detection. A point in the image is considered an 'interest' keypoint if it's intensity value is either greater than or less than all the intensity values of its neighbor set of contiguous (typically 16 contiguous points). [9]

To compute the image's descriptors, ORB uses the BRIEF descriptor method with an extra step to make the descriptors invariant to in-plane rotation [8]. As a initial step, ORB computes a simple measure for corner orientation: the intensity centroid. This can be done by first computing the moments of each image patch as formulated in the following way [8]:

$$m_{p,q} = \sum_{x,y} x^p y^p I(x,y)$$

The centroid of the patch is then defined as [8]:

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$$

We can then construct the vector from the corner's center to the centroid, and compute its orientation as in the following [8]:

$$\theta = atan2(m_{01}, m_{10})$$

Bradski et al. explain more in depth how to improve the rotation invariance of the intensity centroid measure in the original ORB paper, but this is the metric they used [8].

Now, the ORB method needs to compute the BRIEF descriptors, which is described as a bit string description of a patch. This bit string encodes information about the relative intensity between two points in each pair of points that constitute a patch p; more formally, it uses the following test [8]:

$$\tau(p,x,y) := \begin{cases} 1 & p(x) < p(y) \\ 0 & p(x) \geq p(y) \end{cases}$$

where $p(x)$ indicates the intensity of the pixel located at point x. A feature vector of n bits can then be constructed for each feature point [8]. To make BRIEF descriptors invariant to rotation, Bradski et al. propose to steer the points in each image patches "according to the orientation of keypoints," which is computed as described above [8].

Although there are still other issues to deal with (potential loss in variance between good descriptors), Bradski et al. have proposed greedy-based algorithms for solving those [8]. Therefore, the authors of the ORB paper claim that ORB is comparable to SIFT and SURF in terms of accuracy, but with less computational burden [8]. Figure 3 shows the effectiveness of ORB in matching the same keypoints between two images.

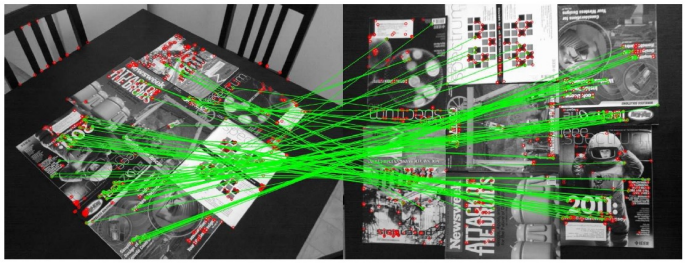

Fig. 3. ORB Detection and Matching. The image shows that ORB can be a good alternative to SIFT and SURF descriptors [8]

Once we have feature descriptors of different frames, we want to compute a transform that estimates the motion between consecutive frames. Dong et al. have noted that either 2D and 3D models can be used for motion estimation [4]. However, while 3D scene models can yield more accurate results, they are also more expensive to compute [4]. Also, since we can assume that there will not be rapid motion changes between camera frames, it suffices to use a 2D model. Therefore, homography transforms are appropriate for the task of video stabilization.

We recall that a homography transform represents motion of planar surface under a perspective projection. More formally, for each point $[x, y, 1]^T$:

$$[x', y']^T = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} [x, y, 1]^T$$

Each value of h_{ij} represents a degree of freedom in the projection.

A special case of the homography transform is the affine transform, which is formally defined as:

$$[x', y']^T = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} [x, y, 1]^T$$

With the affine model, we are able to capture Euclidean rotation and translation, for example [3]. We will explain in the next section why we decided to try both the general homography model and the affine model in our implementation of video stabilization.

4 Overview of our Initial Video Stabilization Method

As mentioned, we have strongly inspired by the image processing pipeline proposed by Kularni et al. [2]. However, since we are implementing the video stabilization in real time, we had to make slight modifications to the algorithm to be able to stabilize the video frame by frame and display it. The speed of our video stabilization algorithm was a high priority. Since other methods typically performed video stabilization on a saved video, a fast runtime of their algorithm was not as essential as long as the results were accurate, as they could display the stabilized video when they finished.

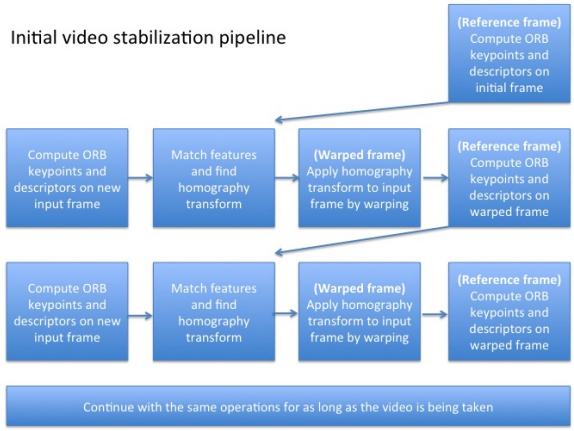

Fig. 4. Our first implementation of our video stabilization algorithm. Note that OpenCV uses RANSAC inside the findHomography function for eliminating outliers

We found the features and calculated the descriptors of our initial frame, and then saved them. Let us refer to this frame as ReferenceFrame. We then read in the next frame, and calculate the new frame's features and descriptors. We can then match features between our consecutive frames, and we then use those matched features to compute a homography transform. We apply a warp on our current frame using the transform we just calculated, and we display this newly warped image, let's called WarpedFrame. We then compute the features and descriptors again on WarpedFrame, and save those results for our next frame. Therefore, we are essentially making our WarpedFrame now our ReferenceFrame for when we read in the next frame.

We used OpenCv functions for all steps of our algorithm. However, upon testing our implementation, we noticed that the stabilized video's framerate was quite poor and there was a lot of lag and jittering from frame to frame. Therefore, our implementation was too computationally expensive and slow. Based on the recommendation given in [3], we decided to change our homography transform into an affine transform, which was still supported by OpenCv. We would lose 3 degrees of freedom, but since we are still able to rotate and translate the image and since we assumed those would be the basic necessary operations, we figured choosing the quicker affine transform would yield superior results. We will explain later why this change however may have been unnecessary.

5 Overview of our revised Video Stabilization Method

Despite our change from using a homography transform to using an affine transform, we noticed that our resulting video still had a poor framerate and lag. After consulting with our advisor Jean-Baptiste Boin, we measured how long our algorithm spent during each step, and we noticed that the most time-intensive was by far the keypoint and descriptor extraction. As mentioned in the previous section, we were essentially extracting this information twice for each frame as we needed it for first creating the transform, and then we needed the keypoints and descriptors on WarpedFrame to prepare for the next frame's transform. If we could find a way to reduce the amount of these operations, then we could drastically improve our speed. Therefore, we came up with this revised algorithm below.

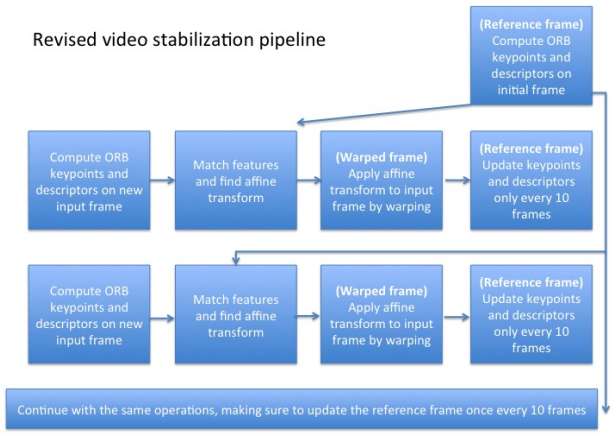

Fig. 5. Our revised implementation of our video stabilization algorithm. We now use an affine transform and compute the keypoints and descriptors of the warped frame only once every 10 frames.

The core concept is very similar to our original algorithm. Let's consider the first frame again. We consider this frame to be ReferenceFrame again and save its keypoints and descriptors. We go on to the next frame and extract the same information so that we can find the affine transform and apply a warp to the previous image. However, now instead of finding the keypoints and descriptors of this new WarpedFrame, we instead do nothing and keep our ReferenceFrame keypoints and descriptors as the ones we found from the first frame. Therefore, when we get our third frame and want to feature-matching, we match it with the keypoints from ReferenceFrame which in this case is still the first frame. We chose to only update ReferenceFrame once every 10 frames. This change meant that instead of finding the keypoints and descriptors 20 times for every 10 frames, we only have to find them 11 times for the same 10 frames. And since the operation of finding these keypoints and descriptors were the most time intensive, we saw significant improvements in the stabilized video's framerate.

Now we will discuss the assumptions we made when making these changes. First of all, we realized that we are sacrificing accuracy for our improved speed, but the improvements in the stabilized video viewability made it seem like a worthwhile exchange. We are also assuming that the video-taker is largely fixated on a particular object and is not randomly moving the camera around nor rapidly panning the camera at wide ranges. Therefore, the difference between frame 1 and frame 10 should not be too great as it would just be standard hand jitters, allowing the results of the transformed frames to remain relatively accurate. We considered this a reasonable use case, as oftentimes when taking video you are not moving the camera too violently but rather focusing it on a particular object, and we chose to optimize for this situation. We will elaborate later on changes we could make to our algorithm to make our stabilization more robust to movement.

Fig. 6. Result of our Video Stabilization Implementation when moving the phone's camera from right to left and rotating it at a 45° angle. This is from our revised method shown in figure 5.

6 Evaluation and Results

First of all, let us look at the timings of each step of both our initial algorithm and then our revised algorithm. We will also include the timing of our final algorithm which we will discuss soon.

We can see that there is a big jump between our initial algorithm and our revised algorithm in terms of timing, while not as big of a jump to our final algorithm. This is

	Initial algorithm	Revised algorithm	Final algorithm
Total average time per frame (seconds)	.0261	.0155	.0146

Fig. 7. The average amount of time spent per frame on a 5 second long video recording. The final algorithm will be explained in our next section

because the operation that takes the most time is by far the descriptor calculations, as that consistently takes the longest out of the operations around .01 seconds. Therefore, in our first implementation when we were calculating descriptors twice for every frame, we got the a much slower result when compared to our second algorithm where we calculated it twice only once every 10 frames. In our final algorithm we managed to cut out calculating the descriptors twice at all, but since we still need to do it once per frame, the timing benefits were less pronounced.

It is more difficult to evaluate the accuracy of our algorithm as we were working in real time. One possible test we could have accomplished was to test our algorithm on a recorded video that has a set amount of jitter. That way, it would mimic the camera holder causing the video to be unstable but it would be done in a controlled way. We could overlay video stabilization of our algorithm with a more standard version to compare the differences. However, by pure qualitative analysis we were quite happy with the result of the stabilization, both in how well it stabilized the frames and also how smoothly it displayed it. Figure 6 shows what the phone's camera was displaying with video stabilization turned on.

Despite our improved speed, our particular implementation had a few weaknesses. First of all, it did not deal well with rapid motions, as if you aimed the camera and then quickly rotated it 180 degrees, the application would sometimes crash. This crash may have been the result of being too aggressive with our distance point filtering (e.g. if matching points are too far apart, then they are discarded, and the findHomography functions do not work well with empty point vectors). In addition, when we would slowly pan from side to side, it would not rotate the stabilized video but rather show a smaller and smaller portion of the frames and show blackness instead. We will discuss how we can improve both of these bugs in our continued work section.

7 Continued Work

There is significant work that can be done to both improve robustness and speed. First, we spoke about using an affine transform instead of a homography transform. This change was made before actually timing the difference between finding the two transforms, and we assumed that the benefit in speed would be worth it. However, when actually looking at the time difference, we discovered that it was trivial when compared to our more costly operations. Therefore, we returned to using a homography transform as that would improve our robustness.

After consultation with our mentors, we also realized that our current implementation was needlessly finding the keypoints and descriptors of ReferenceFrame as described. Instead of doubling the amount of keypoint and descriptor operations for certain frames, we could utilize an implementation that guarantees these calculations once. This method follows the diagram shown here

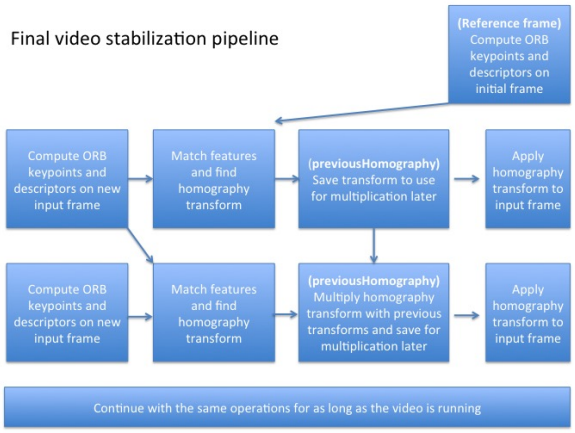

Fig. 8. Our final video stabilization algorithm. Note that we no longer need to find the keypoints and descriptors of our warpedframe as we are passing along the multiplied homography transforms.

We still find the keypoints and descriptors of each frame as we read them in. However, the key change comes with our homography transform. For the first two frames, we calculate the homography transform as normal, matching the features of consecutive frames. We save this transform as previousHomography. When we consider the third frame, we find the homography matrix between the third frame and the second frame as done before. Now however, before we warp our frame, we do a matrix multiplication of this new homography transform with previousHomography. This operation means that we are always able to take into account the changes from frame to frame.

This change actually also solves the issues we discussed in the prior session, that of the crashing and that of the rotating camera. In regards to the rotating camera, with our older implementation we would find the keypoints and descriptors of the warped frame. Therefore, if we rotated the camera, then there would be less overlap between our previous frame and our current frame, so our warped frame would be smaller. As we continued rotating the camera, then the warped frames would continue to get smaller and smaller and we would have fewer descriptors, which would lead to only being able to see part of the scene with the rest being black. Therefore, by chaining together the homography transforms of each frame as mentioned in our final implementation, we no longer lose those descriptors and we are able to display the full frame. The amount of crashing when quickly rotating the camera was also improved. We hypothesize that this is because since we are now chaining transforms, we are never trying to warp an image with an empty set of descriptors, which we believe is what would happen in our prior implementation and cause a crash. However, we are not entirely sure about this conjecture and one of the first steps we would take if continuing this application would be to concretely identify the causes and solutions for the crashes.

If we were to continue more with this project, we would

look to find more ways to both improve optimization so that the video stabilization runs smoother and we would also look to further improve robustness, mainly identifying changes so that we can deal with quick moving objects and more rapid movements of the camera.

8 Conclusion

We were able to iterate and improve on our goal of implementing a video stabilization algorithm in real time. While we were quickly able to identify the basic algorithm that we wished to implement, when we first tested our implementation we realized that we were dealing with far different constraints then explained in our readings. Since we wished to work in real time, we would have to highly prioritize the speed of our algorithm even at the expense of accuracy. In addition, we were dealing with an additional constraint of using a smartphone device which is less powerful than a computer. Therefore, our final result was finding a balance of having it accurate enough while still being pleasant to view. The most progress was made when we discovered that finding the descriptors and keypoints were the most expensive operations, and then we could identify ways to modify our algorithm to limit those operations.

This type of work is very relevant for a majority of users who record video on their phone by hand, as it is very hard to prevent shaking so some amount of video stabilization will be useful. In addition, the question of finding the most efficient algorithm would be of high interest to all smartphone producers.

9 Acknowledgements

We would like to thank the TAs for the time they spent in helping us during office hours and helping us revise our project. We also want to thank professor Bernd Girod for the lectures he's given throughout the quarter.

References

[1] Farid, Hany and Jeffrey B. Woodward. "Video Stabilization and Enhancement." Department of Computer Science, Darmouth University.
[2] Shamsundar Kulkarni et al. "Video Stabilization Using Feature Point Matching." 2017 J. Phys.: Conf. Ser. 787 012017
[3] "Video Stabilization Using Point Feature Matching." Mathworks. https://www.mathworks.com/help/vision/examples/video-stabilization-using-point-ture-matching.html
[4] J. Dong and H. Liu, "Video Stabilization for Strict Real-Time Applications," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 27, no. 4, pp. 716-724, April 2017.
[5] "StableEyes real-time image stabilisation for high-zoom surveillance cameras" [Online]. Available: https://www.ovation.co.uk/video-stabilization.html
[6] OpenCV: Optical Flow. https://docs.opencv.org/3.3.1/d7/d8b/tutorial_py_lucas_kanade.html
[7] S. Liu, L. Yuan, P. Tan and J. Sun, "SteadyFlow: Spatially Smooth Optical Flow for Video Stabilization," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, 2014, pp. 4209-4216.
[8] G. Bradski, E. Rublee, V. Rabaud and K. Konolige, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, Barcelona, 2011, pp. 2564-2571.
[9] OpenCV: FAST Algorithm for Corner Detection [Online]. Available: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_fast/py_fast.html

Appendix A
Team Member Contribution

Juan Camacho:
- Implemented the initial prototype of the algorithm
- Implemented third method
- Mostly wrote first half of report

Alvin Kim:
- Implemented the optimizations discussed here (first and second method).
- Also wrote the code for timing each operation.
- Mostly wrote second half of report.

We worked on how to improve upon the initial algorithm that we had, and tested it together. On top of that, the both of us revised the poster and report together.