Assignment 11 (Group)

Pratik Bhandari and Simon Swenson
pratikbhandari@email.arizona.edu
simonswenson@email.arizona.edu
Graduate Students

April 20, 2019

Group Contribution

For the assignment, the two of us worked together for each question. Every question was brainstormed and discussed with both members being involved. Similarly, the coding was done by both members in parts, one adding on to the work of the other and vice versa. The report was also worked on together as a team by using *Overleaf* as a platform for both team members to work concurrently on the report.

Part A

In this assignment we started with looking at RANSAC and its implementation on a set of data points. We applied RANSAC on the set of 300 points, out of which at least 1/4 i.e. 75 were assumed to lie on a good fit. Getting a good fit using RANSAC depends on four parameters which were set through multiple iterations of observations.
Using the following equation to find the number of iterations:

$$k = \frac{log(1-p)}{log(1-w^n)}$$

Since we are plotting lines, the value of 'n' will be 2. We used a threshold of 0.2 instead of implementing the best N% approach, and used $p = 0.99$ as the percentage of assurance. Using these parameters, the value of k was calculated to be somewhere around 72. We settled with 75 iterations.
On every iterations, two random points were selected from the dataset and the orthogonal error of every other point to the line from these two points was calculated. Using the above mentioned threshold, a set of inliers was calculated and compared to the minimum number of inliers required (%75). On every set of inliers for each iteration, homogeneous least squares was used to refit the line and calculate the RMSE and orthogonal error. Finally, after 75 iterations, the best set of inliers were used to refit the line and the respective parameters were calculated.

The final line using RANSAC had the following parameters:

- Slope, m = 1.901
- Intercept, c = −0.446
- Number of inliers = 99
- Root Mean Squared Error = 0.099348
- Orthogonal Error = 0.008231

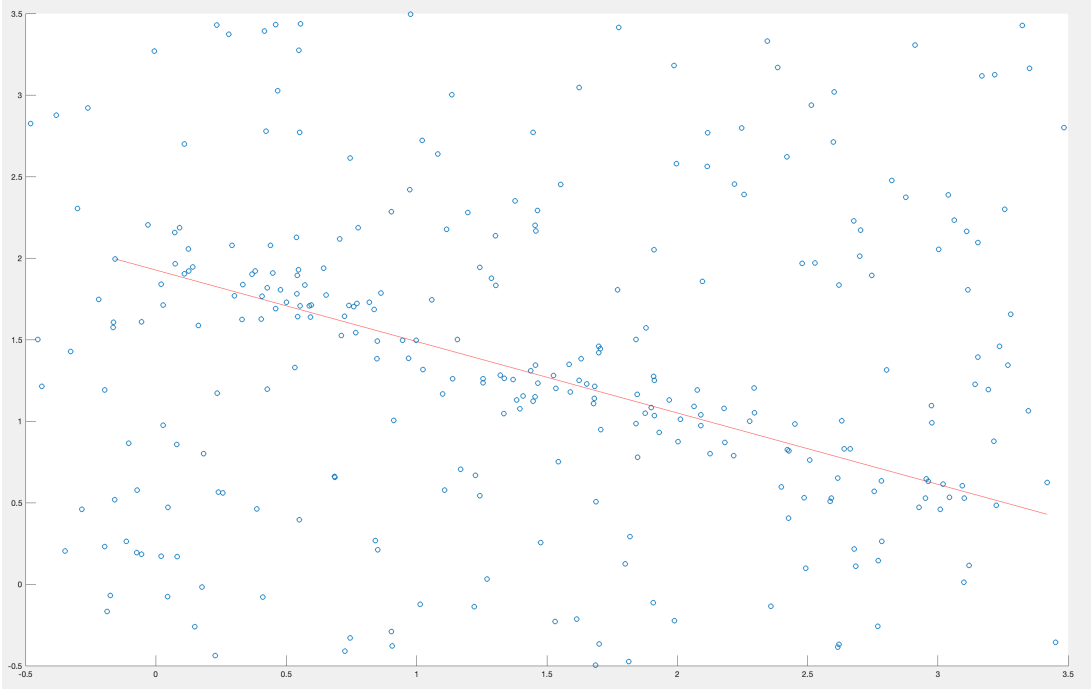

Figure 1: The image shows the scatter plot of the set of 300 points (in blue) along with the line (in red) which was determined using RANSAC. The plot shows that the line fit ignores the outliers on the top right and bottom left of the plot and fits well with the actual concentration of points around the middle.

Part B

For this part of the assignment, we began exploring homographies and one way to find good homographies (DLT). We began by calculating homographies on a varying number of random, 2-d points. Since four points is the minimum number of points needed to define a homography, we expected error to be close to 0 in that case. However, we expected that adding more random points to the data-set would begin increasing the error, since these points are generated completely randomly. This turned out to be the case, as can be seen in the table below.

Points	Error
4	0.0000
5	0.5638
6	0.9230

From there, we returned to the slide, frame image pairs that we used in the last assignment. We picked out 8 manual matches for each image pair, then computed a homography using all 8, then only 4 of the 8. While the error was slightly higher when picking only 4, the resultant transformed points were at most a pixel or two away from their actual coordinates. The actual points are hard to see, since they are mostly covered by the red transformation endpoints, but they are drawn as 3x3 yellow boxes, and you can make out a few slivers of yellow here and there, especially if you zoom in. In the slide images, we start with the same exact input points, so there is no need to use these yellow indicators. (The yellow would exactly overlap the red.)

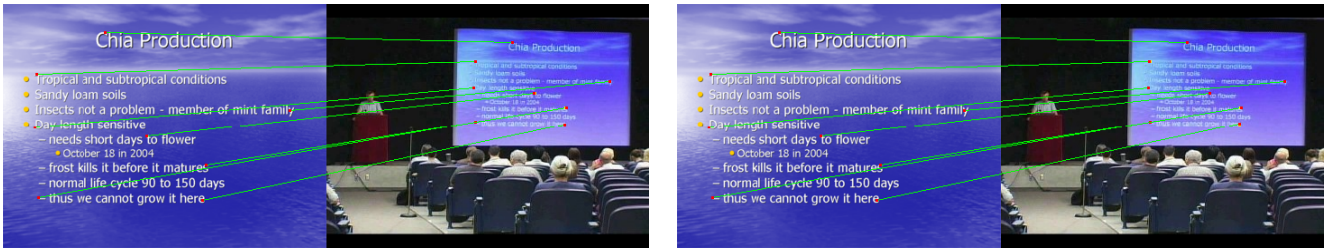

Figure 2: Homography transformations and their corresponding ground truth for the first image pair. Ground truth points are drawn in yellow in the frame images. Left - A homography computed with all 8 points. Right - A homography computed with only 4 points.

Figure 3: Homography transformations and their corresponding ground truth for the second image pair. Ground truth points are drawn in yellow in the frame images. Left - A homography computed with all 8 points. Right - A homography computed with only 4 points.

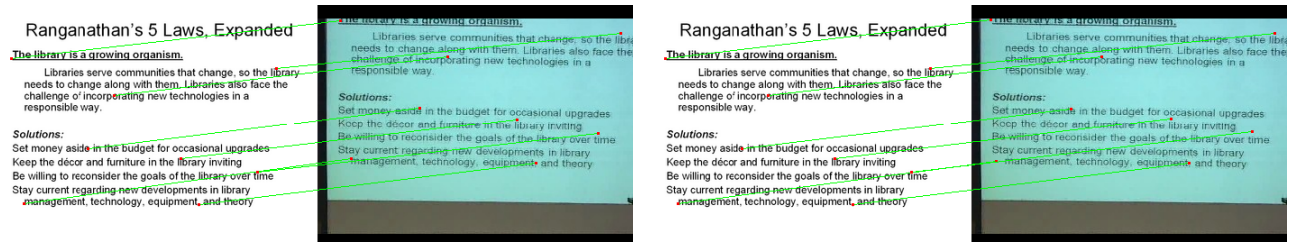

Figure 4: Homography transformations and their corresponding ground truth for the third image pair. Ground truth points are drawn in yellow in the frame images. Left - A homography computed with all 8 points. Right - A homography computed with only 4 points.

Part C

This part of the assignment combines the previous parts, A, where we implemented the RANSAC algorithm, and B, where we implemented the DLT method for computing homographies.

We continue using the now-familiar three slide, frame image pairs. Our starting point for this part is the ending point of HW10. For that assignment, our best matches were achieved using Euclidean distance and a first-second ratio of 0.7. Thus, we perform a pairwise matching of SIFT feature points between the slide image and the frame image of each image pair, ensuring that it is one-to-one to prevent the bug explained by Manujinda in his Piazza post. At the same time, we prune these results to only include feature points with a certain ratio between best match and second-best match.

As illustrated in our submission for HW10, even a ratio of 0.7 resulted in some outlier points. However, to experiment with RANSAC's ability to ignore outliers some more, we changed the ratio to 0.85 for this assignment, meaning that our pool of matching points has many more outliers. From that set, we apply

RANSAC to find the best homography and ignore the outliers. Visual inspection of the matching points before RANSAC gives us a good indicator of what parameters to use for the RANSAC algorithm. Recall the equation we discussed earlier:

$$k = \frac{log(1-p)}{log(1-w^n)}$$

Where p is the desired interval of certainty of the matching, w is the probability of a randomly selected point being an inlier, and n is the number of points that we need to pick. We settled on a value of $p = 0.99$, which ensures that we are 99% certain that the result is a good one. This gives us the number of iterations that we need for each pair of images. We presume that positions of good matches in each image will only have a small error (within a few pixels), so we choose a low value for the threshold. In other words, we are assuming a good homography will map slide points very close to their known matching frame points. Our final parameters used were as follows:

Image Pair	Number of Iterations	Match Threshold
1	14	5
2	100	10
3	20	2

This table indicates that the second image pair was the hardest to find a good homography for and the third image pair was the easiest to find a homography for. The results are below and look very good:

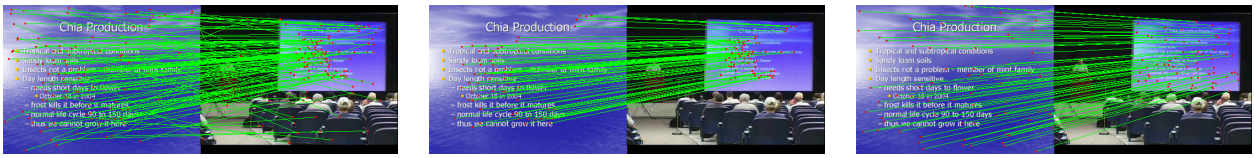

Figure 5: The first iamge pair. Left - All matching points between images after applying Lowe's ratio of 0.85. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

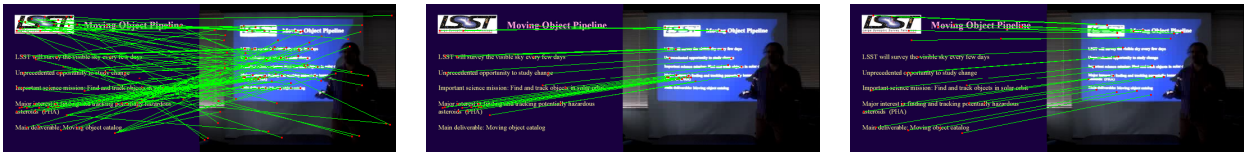

Figure 6: The second image pair. Left - All matching points between images after applying Lowe's ratio of 0.85. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

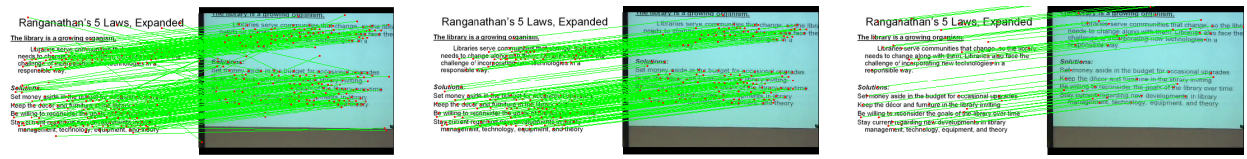

Figure 7: The third image pair. Left - All matching points between images after applying Lowe's ratio of 0.85. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

While there weren't a terrifically large number of outliers in the first and last image pairs, the second

pair is an ideal case-study into the power of RANSAC. It has quite a few outliers, and, even so, RANSAC is able to deal with it.

Then, we reconsidered the worst combination of distance metric and pruning method: χ^2 with top %. We increased the percentage here to increase the number of outliers. For all three images, we kept the top 30% of χ^2 scores.

We had to modify the RANSAC parameters from above, since there were more outliers this time around. We used:

Image Pair	Number of Iterations	Match Threshold
1	50	5
2	100	10
3	40	2

Even with the worse matches and more outliers, RANSAC provides excellent results:

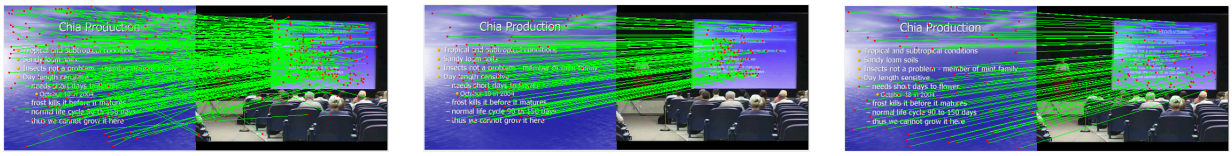

Figure 8: The first iamge pair. Left - All matching points between images after taking the top 30% of all χ^2 matches. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

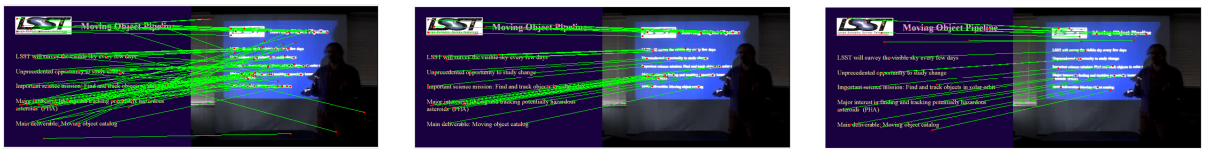

Figure 9: The second image pair. Left - All matching points between images after taking the top 30% of all χ^2 matches. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

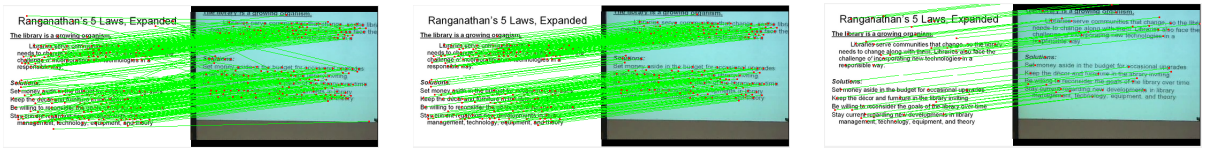

Figure 10: The third image pair. Left - All matching points between images after taking the top 30% of all χ^2 matches. Center - All inliers found after applying RANSAC. Right - A selection of all feature points from the slide image and their transformations into frame image space.

The last image, as before, did not have as many outliers, but the first two images had quite a bit of bad matches, and RANSAC was able to deal with them well.

Part D

This part of the assignment was a synthesis of many topics that we had covered throughout the class. It has similarities with an earlier assignment, HW3, in which we fit lines to data points using the homogeneous least squared error solution. However, instead of just fitting a line to a given set of points (with some noise), we had two more challenges to overcome: generating random points from background noise and several lines (with Gaussian noise) and implementing k-means to cluster each point into a number of *guessed* lines.

Generating Noisy Point Data

First, we pick a bound for point generation, since it's impossible to generate points uniformly from $[-\infty, \infty]$. We chose a bound of [0, 1] in each direction (x, y), since this is the range of MATLAB's rand function. We generate o random noise points uniformly, using rand.

This generates the background noise, but now we have to generate points for each of n lines. To do so, we again uniformly generate $2n$ random points. These will be the points used to define the lines. We calculate the parameters for each line using each pair of points. From here, for each line i, we generate m_i random points assigned to that line and apply slight noise to each point. The noise is determined by sampling a Gaussian distribution with standard deviation σ_i and mean 0. Note that m_i and σ_i may be different for each line.

The output of this step is a set of labeled points (0 for noise, $1 - n$ for line assignments) and line parameters. This gives us ground truth to test our k-means method against.

Applying K-means

From here, it is a simple matter of applying the k-means algorithm that we had previously discussed in class. However, one small caveat is that we are not clustering points according to their spatial coordinates, but according to how close they are to each line. This means that, for one part of k-means (which we'll call "chicken"), point assignments will not be based on distance to each centroid, but on perpendicular distance to each of the lines. This requires us to calculate the perpendicular distance for each point. A convenient approach might be to convert points into homogeneous coordinates and apply a transformation that collapses any point in R^2 to its perpendicular projection onto the line. However, we chose a different approach that is also easy to understand: finding the intersection of two perpendicular lines. First, recognize that, given a line in form $y = mx + b$, the vector parallel to the line is determined entirely by m. In fact, that vector is:

$$v = \begin{bmatrix} 1 \\ m \end{bmatrix}$$

Calculating the vector perpendicular to this is straightforward enough:

$$v' = \begin{bmatrix} -m \\ 1 \end{bmatrix}$$

Finally, we can calculate the y-intercept of the perpendicular line which passes through a point p:

$$b = p_y - mp_x$$

This gives us a system of equations with two unknowns (x, y), which we solve for to find the intersection point. This is the perpendicular projection of point p onto the line.

The "egg" part of k-means, finding new line parameters, can be done easily enough using the homogeneous least squares solution covered extensively in class, then converting those line parameters (a, b) to slope-intercept parameters.

We continue this algorithm until the root mean squared error (between each point and its perpendicular projection to its assigned line) converges.

Experiments and Results

We ran six experiments using this setup, changing various parameters. We tweaked the following parameters in various ways:

- o - Number of noise points
- n - Number of lines from which to generate data
- k - Number of lines fit using k-means
- m_i - Number of points to generate for each line i
- σ_i - The standard deviation for each line i

We began with an easy scenario, then slowly made things more difficult in various ways. Below is a table with the parameters that we used for six experiments. Note that, in the table, k is represented as a vector, since we tried various values for k to see how the algorithm would act:

Experiment	o	n	k	i	m_i	σ_i
1	100	5	[4, 5, 6]			
				1	250	0.01
				2	250	0.01
				3	250	0.01
				4	250	0.01
				5	250	0.01
2	50	7	[4, 7, 10]			
				1	50	0.02
				2	50	0.02
				3	50	0.02
				4	50	0.02
				5	50	0.02
				6	50	0.02
				7	50	0.02
3	50	3	[2, 3, 4]			
				1	10	0.02
				2	10	0.02
				3	10	0.02
4	50	5	[3, 5, 7]			
				1	50	0.05
				2	50	0.05
				3	50	0.05
				4	50	0.05
				5	50	0.05
5	50	5	[4, 5, 6]			
				1	50	0.02
				2	50	0.01
				3	50	0.05
				4	50	0.1
				5	50	0.001
6	250	5	[3, 5, 7]			
				1	10	0.02
				2	50	0.02
				3	25	0.02
				4	100	0.02
				5	50	0.02

We also plotted the results of each experiment. The plots are small, but they are high enough resolution, so just zoom into the PDF to view them closer. Discussion can be found in the accompanying figure caption and in the proceeding section (labeled "More Discussion").

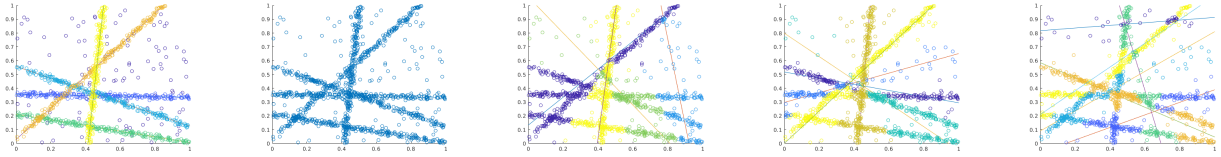

Figure 11: Experiment 1 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 4. Center-right - K-means with k = 5. Far Right - K-means with k = 6. Since we started with a very small value for each line's standard deviation, little background noise, and many line samples, we expected this to be an ideal scenario. However, even the small amount of background noise present proved to be a problem for k-means. While some lines were found easily enough, other lines were fit using background noise, and ended pointing in the direction of such noise. In addition, many of the lines intersected each other near the center of the window. This could also have caused a point of confusion for the algorithm.

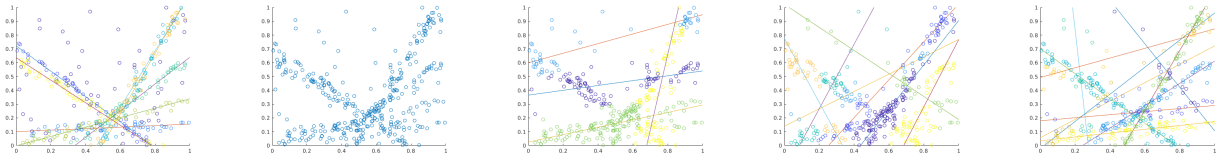

Figure 12: Experiment 2 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 4. Center-right - K-means with k = 7. Far Right - K-means with k = 10. Another difficult data-set with lines intersecting around the same spot. In addition, this data-set has even more ground truth lines (7), and each line has more noise. Can you find each line from just the unlabeled data? (It looks like only 5 distinct lines to me (Simon)!) The results are expectedly bad, considering the difficult dataset. However, despite this, some lines are found, for example, the green cluster when k = 4 and the blue, aqua, and green clusters when k = 10.

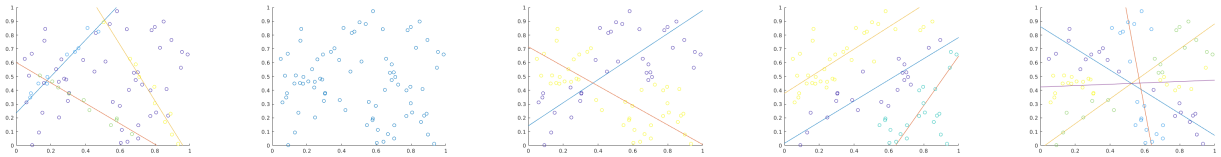

Figure 13: Experiment 3 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 2. Center-right - K-means with k = 3. Far Right - K-means with k = 4. Here, we were trying to see if K-means could discover the lines when the number of random noise points greatly outnumbers the line-generated points. I (Simon) can barely make out the original lines, but I still think they are distinguishable. However, k-means has trouble with this data-set, because it must assign background noise to lines.

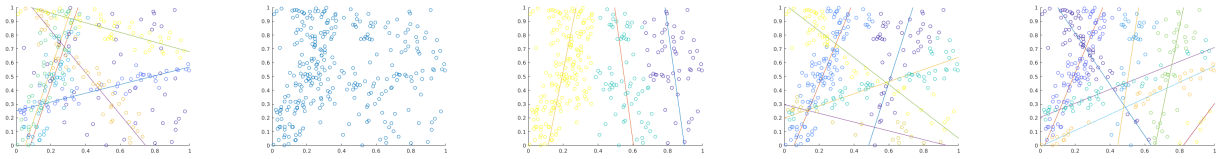

Figure 14: Experiment 4 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 3. Center-right - K-means with k = 5. Far Right - K-means with k = 7. Here, wanted to see the effect of increasing the noise for each line. I (Simon) can make out some of the lines, but it is a bit difficult. K-means does a good job of finding the cluster which were generated by two almost-identical lines, but the other fits were not so good.

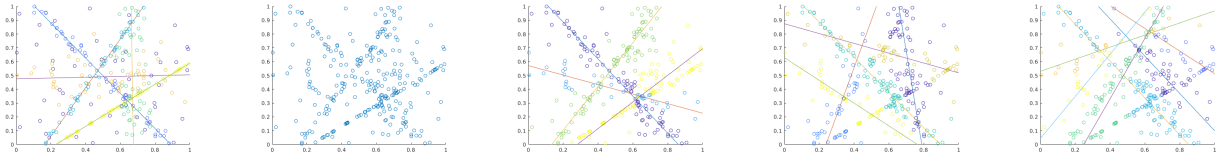

Figure 15: Experiment 5 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 4. Center-right - K-means with k = 5. Far Right - K-means with k = 6. Here, we wanted to see what would happen when we used lines with varying degrees of noise. Looking at the ground truth, two of the lines (the orange cluster with purple line and the green cluster with orange line) almost blend into the background. As expected, these could not be found. However, the experiment with k = 4 found the other three clusters.

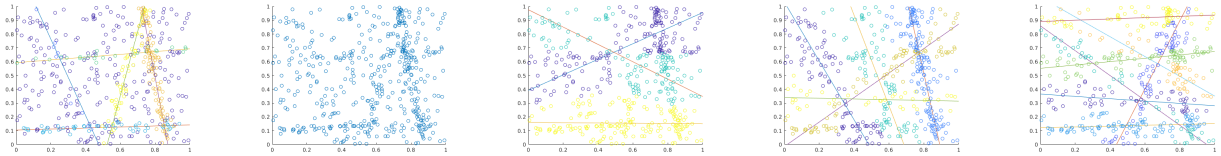

Figure 16: Experiment 6 results. Far Left - ground truth. Center-left - Unlabeled data. Center - K-means with k = 3. Center-right - K-means with k = 5. Far Right - K-means with k = 7. Here, we wanted to see what would happen when we varied the number of samples for each line. Judging by the ground truth, there are really only three clusters that have an adequate number of points, and this is reflected when we inspect the unlabeled data. None of these clusterings were particularly great, but k = 7 found the bottom, horizontal line, and k = 5 found the right, vertical line.

More Discussion

In general, it is important to note that there is a lot of randomness involved in these experiments. All lines are chosen at random. The points are generated at random. Finally, and most importantly, k-means begins with either random assignments of each point to each line or random line parameters. (We chose the latter in our implementation.) It might be possible to get better results by running k-means on the same data-set with the same value of k multiple times, then. However, this sort of a brute-force approach, and we did not consider it for our experiments. (This could be done in future work.)

The important variables are those that we outlined in the experiment table above. Each has an effect on the k-means algorithm. For example, increasing background noise will make the algorithm perform worse, since k-means *must* assign each point to the closest line. This caused several lines to be fit mostly to the

noise. (See Experiment 1, k = 5 and k = 6 results, for example. This effect was made worse when there was a large portion of the space that no lines passed through. Since all noise in that space requires a line assignment, it would tend to pull a line toward this swath of noise.

Increasing the number of lines generated had the biggest effect when lines tended to be close to each other and intersect at similar points. Experiment 2 illustrates this phenomenon. Seven lines were plotted, and all happened to have very similar intersection points, and many had similar trajectories. For lines which do not intersect any other lines within the window, I am confident that k-means would still work well in this situation.

For difficult data-sets, changing k did not have an obvious effect on the clustering. Sometimes the correct lines were found, sometimes they weren't. For such data-sets, some lines were found correctly even for non-ideal values of k (when$k \neq n$). However, for easier data-sets, we'd imagine that a non-ideal value for k would produce worse results. The reason for this is that if the clustering can be found easily, then, when $k = n$, all lines will match ground truth. If $k < n$, though, at least some points from multiple lines will be grouped to the same cluster, leading to a (potentially) horrible match. If $k > n$, then multiple lines will be very close to each other, also not ideal.

Increasing the number of data points generated for each line will increase the chance of a good match. (Consider the inverse, experiment 3, for example.) This is similar to decreasing the number of background noise points generated. The last parameter, the standard deviation for each line, also affects the effectiveness of k-means. Intuitively, increasing this standard deviation will make the line appear noisier. This will make points sampled from the line more likely to blend into the background noise and make it harder for k-means to find the actual ground truth lines.