These results were obtained with initial (x, y) coordinates within the range of (-5.0, 5.0) with an epsilon value of 1.0. Run time was recorded in elapsed time because user time excludes time threads are waiting on listen. I decided that conveying this information when making comparisons with different number of threads was meaningful.

I optimized the algorithm so that range of the random offset per mutation narrowed down as the distance error became smaller. Before optimization I had non-continuous mutation offsets. In other words for distance error over 500 mutate the coefficients by a random number between (-100,100), for distance error over 100 mutate by a random number in range (-50, 50) so on and so forth. I compared execution time for both implementations with fixed coordinates of (-3.2317, -1.1965), (-1.875, 4.4629), (-1.1634, -0.51425). Because the optimized version was making very small mutations when it reached distances below one digit I anticipated that the optimized version would actually have more coefficient guesses (i.e. iterations). However iterations for optimized code recorded 11392 compared to 1009490 for the un-optimized version. Another observation to note was that the number of times that the "best" coefficients was updated was greater in the optimized version. The runtime was greatly reduced from 19.50 seconds to 0.26 seconds and considering the fact that the number of iterations were reduced to a tenth, the result seem respectable.

Changing the coordinate range from (-5.0, 5.0) to (-30.0, 30.0) increased runtime for finding best coefficients. However, this was mostly due to starting mutations with a much larger distance error. Although the number of mutations for a wider range was much larger, once it reached a smaller distance error mutations showed similar behavior and approximately had similar runtime from that point on. Iteration of the whole list that stores thread execution time and sorting this list was inevitable to compute the median and the mean. However, adding the statistics to the program did not greatly affect the runtime of the program for degrees that I have tested. I could see a one or two second pause when the program reached the line for sort. But I noticed that this time obviously took longer as I tested with higher degrees which had much larger iterations. I could assume that sorting could have a significant effect for testing with polynomials with 5 or more degrees.

With consistent coordinates for two, three and four degree polynomials I compared execution time for different number of threads. Local machine supports 4 cores and 8 threads. One obvious observation was that as the number of threads increased it took less time to compute similar number of iterations.

Threads	Degree	Coordinates	Iteration count	Runtime
1	2	(-3.2317,-1.1965), (-1.875,4.4629), (-1.1634, 0.51425)	20342 19596 6948	0:00.55 0:00.55 0:00.20
2	2	Fixed	21411 20944 24882	0:00.33 0:00.31 0:00.39
3	2	Fixed	13835 33207 18717	0:00.16 0:00.42 0:00.23
4	2	Fixed	17379 22780 28823	0:00.21 0:00.25 0:00.34
8	2	Fixed	13899 29319 48335	0:00.23 0:00.64 0:00.86

For third degree polynomials iteration counts increased dramatically hence the increase in runtime. But again we can observe that increase in threads made faster runtime. It was interesting to see that the average time spent per iteration was lowest with a single thread with an approximate average of 5 microseconds per iteration. Another thing to note was that the maximum time taken among the iterations was higher with more number of threads. More specifically with a single thread maximum time among iterations did not exceed 100 microseconds but this value increased as the number of threads increased. In execution with 20 threads this value nearly reaches 10,000 microseconds (i.e. 10 milliseconds). This signifies overhead for thread execution as well as even concurrency among threads after a certain number of spawned threads. There were no other remarkable differences between the statistics other than that the mean value was always consistent with a value less than 20 regardless of thread and degree.

Threads	Degree	Coordinates	Iteration count	Runtime
1	3	(2.3669,2.4816), (1.7451,-1.1468), (3.6583,-0.24945), (-1.6479,1.7633)	478528 479016 71885	0:12.72 0:12.39 0:01.86
2	3	Fixed	1082491 730198 457537	0:15.31 0:10.67 0:06.99
3	3	Fixed	528022 1985290 1624626	0:05.59 0:21.12 0:18.67
4	3	Fixed	1697175 1030672 1034843	0:16.88 0:10.86 0:09.05
8	3	Fixed	3343420 942762 2736554	0:31.73 0:10.02 0:25.49

From fourth degree polynomials it was too time consuming for multiple iterations so I observed runtimes for different coordinates and its statistics with more than 8 threads. In my implementation I had a threshold to jump out of the local mutation range if iterations exceeded a certain number. For a fourth degree polynomial I had to higher the threshold from 600,000 to 10,000,000 as it was reached too frequently causing frequent re-computation and thus slowing total runtime.

Threads	Degree	Coordinates	Iteration count	Runtime
15	4	(1.0076,-3.2571), (-0.12352,-3.5917), (3.5507,-0.9414), (-3.1692,-4.2172), (3.841,1.9308)	75107905	7:08.26
15	4	(0.83022,0.43449), (3.2278,2.9405), (-0.51317,-2.8508), (-1.0703,-1.5596), (4.6714,2.055)	12490216	1:05.01
8	4	(-2.9664,-4.3602), (1.3492,-4.3841), (3.7887,2.7499), (0.21845,4.8722), (-1.9638,2.5002)	1035697	0:07.06

For polynomials over four degrees, maximum time of thread mutations reached as high as 5829 microseconds. Regardless, mean runtime for these iterations still remained under 20 microseconds. An interesting observations was that having a perfectly consistent scaling for the mutation ranges was in fact inefficient. For four or five degree polynomials where the initial distance error that would on average start at a 5 digit number it took significantly longer even to reach a distance error of a 3 digit. A hybrid implementation with my original inefficient version actually produced better results where I had constant scalars for discontinuous ranges. In other words for a distance error over 500 I had random mutations with numbers in range (-80, 80), distance error between 500 and 100 a scalar of 0.0005 that would be multiplied to the distance error and to the range of (-100, 100) and for distance error less than 100 an even smaller scalar multiple of 0.0002. Although it was very randomly mutating when above distance error of 500 leaving it to luck was faster than playing it safe.

Although I could not record a result for a five degree polynomial it took approximately 40 minutes to reach a "best" distance error of 2.24508. Running the program for higher degree polynomials the threshold to break out of the local mutation range was a double edged sword. Although I am mutating the coefficients with a random number within a very small range after reaching a certain distance error, it is still random. So if I was unlucky enough I would break out of the local mutation range one iteration before finding a better coefficient. But at the same time, for polynomials over three degrees I might be stuck at a point that could not reach my epsilon value or consumes a significant amount of time just to break out of that point by making very small mutations. Setting the optimal scalar values according to distance errors, an adequate threshold tolerance was the mathematical challenge in optimizing my implementation.