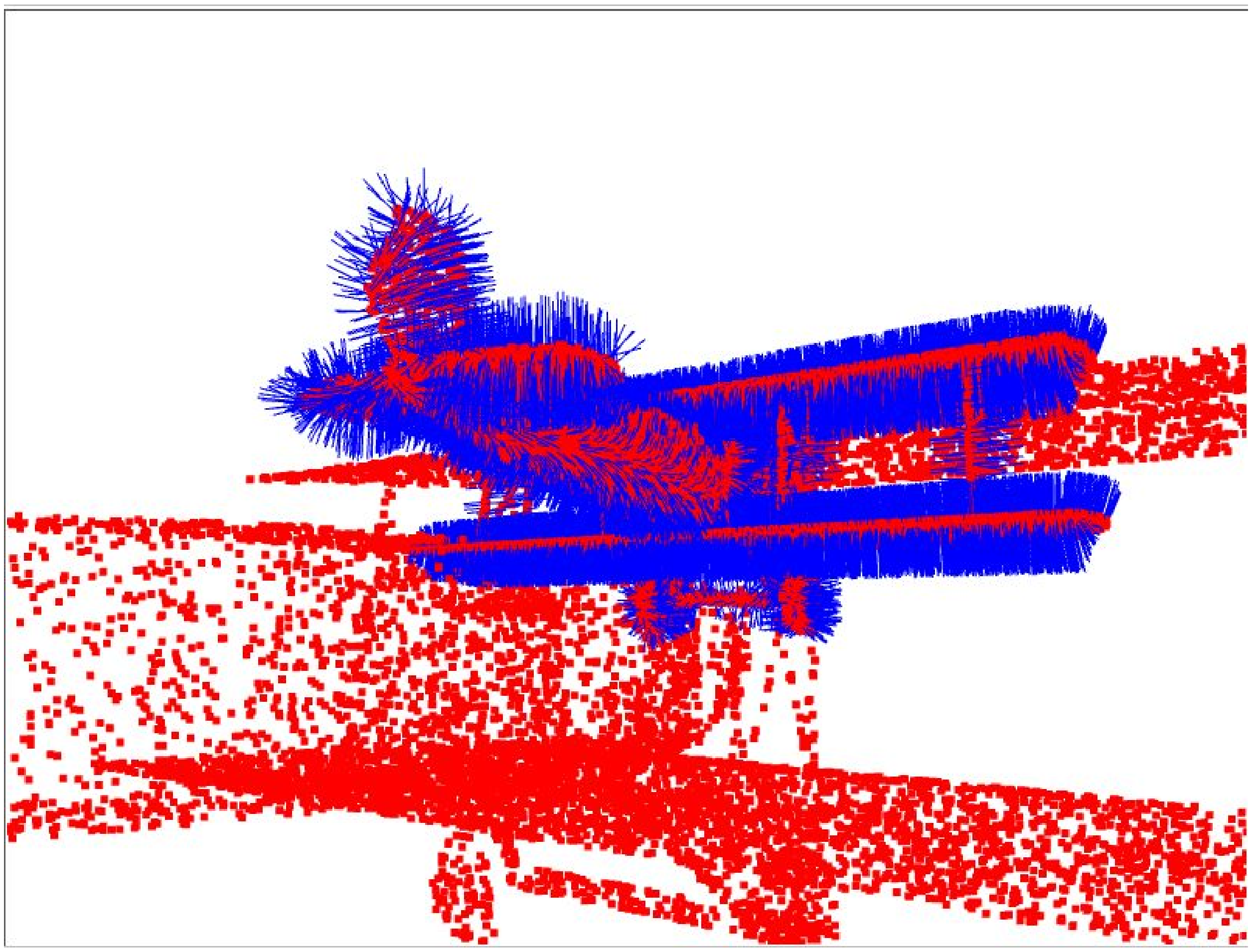
CS 290
Group Assignment 2: “Shape Google”
Group Members: Zachary Bears, Natalie Chanfreau, John Lorenz

I. Mean Center/ RMS Normalize Point Clouds

To implement this, the centroid of all of the points was first found then all the points had the centroid subtracted from them to center the image. Next, all of the RMS distances were calculated and the scale factor was pulled out from the calculation. Every point was then multiplied by this scaling factor to make the total RMS distance equal to 1. The result of this on the biplane image can be seen in the image below.

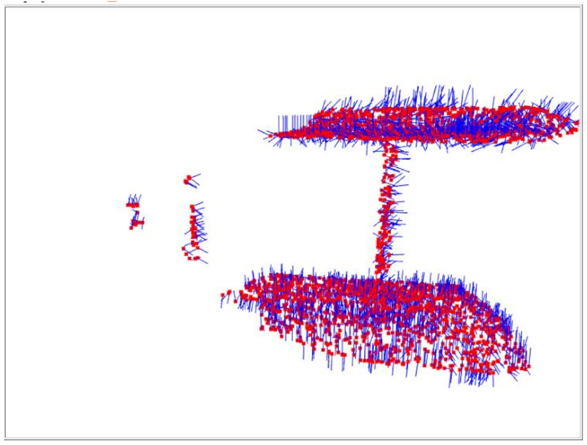


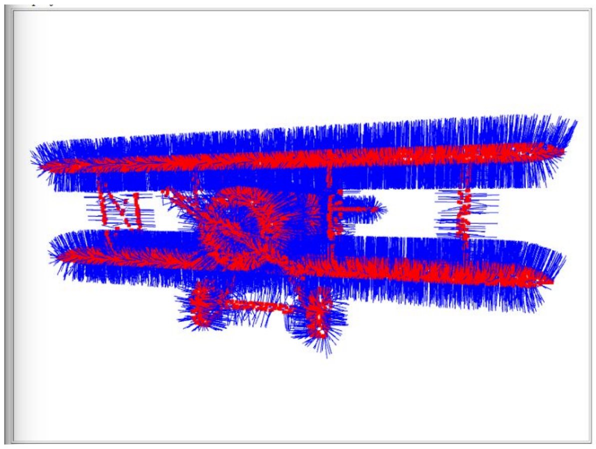
II. Shell Histograms

In order to get shell histograms, we followed a very simple process of first using the np.linalg.norm method to get the distance of each point from the origin. We then used the np.histogram method to make the histogram (If you wanted us to make the histogram by hand, we implemented that in getShapeHistogramPCA. This method was just much more succinct with the simple command).

III. Shell Histograms + Sorted Sectors

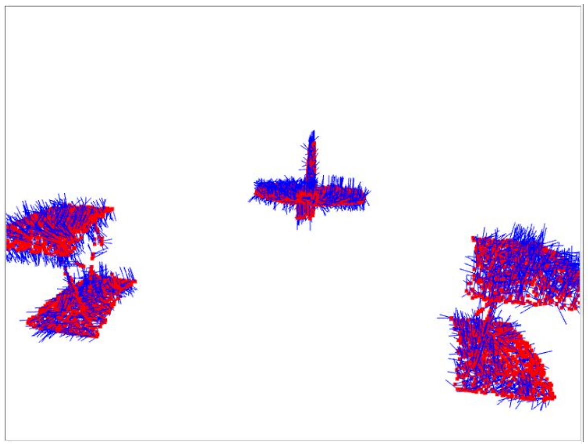
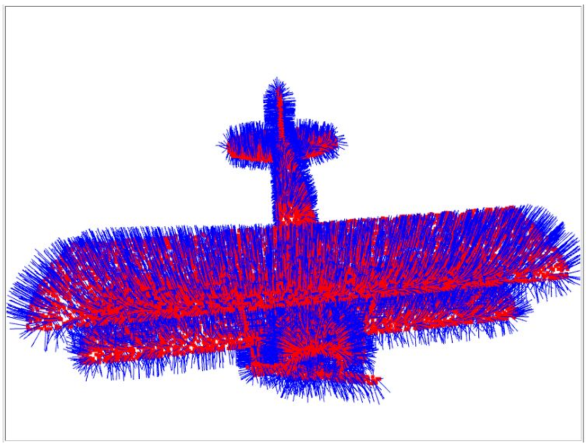
In order to get the shell histograms cut into sectors and sorted by size, we first took the dot product of the transpose of Ps with all the points in SPoints. This gave us a dot product matrix with a row for each Ps and a column for each SPoint shell. We then made a column matrix of the maximum index in each row of the matrix. This gave us a column (Ps items long) of indexes of sector points that each point in Ps aligned best to. We then used these indexes to select elements in Ps whose dot product best aligned with a given sector. By creating a histogram using these points and combining into the overall histogram, we were able to make a matrix of histograms for each sector and shell. Finally, we sorted the sectors in descending order per row, thus sorting by the greatest sector. The successful cutting into sectors can be seen in the image below. On the right is the original plane. On the left is the entirety of a sector (not split into shells). As you can see, the cutting of the wing clearly starts in the plane's center and grows outward like a sphere slice.





IV. Shell Histograms + PCA Eigenvalues

This method created a linear spacing of distances from the sphere center to divide the sphere into shell. It then took the distances from each point to the center of the sphere using the np.linalg.norm method. Finally, it went through and selected points from Ps based on their position in the shell (the outer shell was elected to contain all points beyond its radius) using Numpy selection techniques. Finally, PCA was done on the points and the eigenvalues were taken and sorted. They were finally added to the histogram. The image below shows a shell being selected from the plane, confirming that shell selection behaviour works.

V. D2 Distance Histogram

This was implemented by generating random indices for points and going through the samples and doing a distance calculation using np.linalg.norm of their difference. If the distance was less than the max distance, it was added to an array of distance. Finally, a histogram was created of all the distances and returned using the np.histogram method.

VI. A3 Angle Histogram

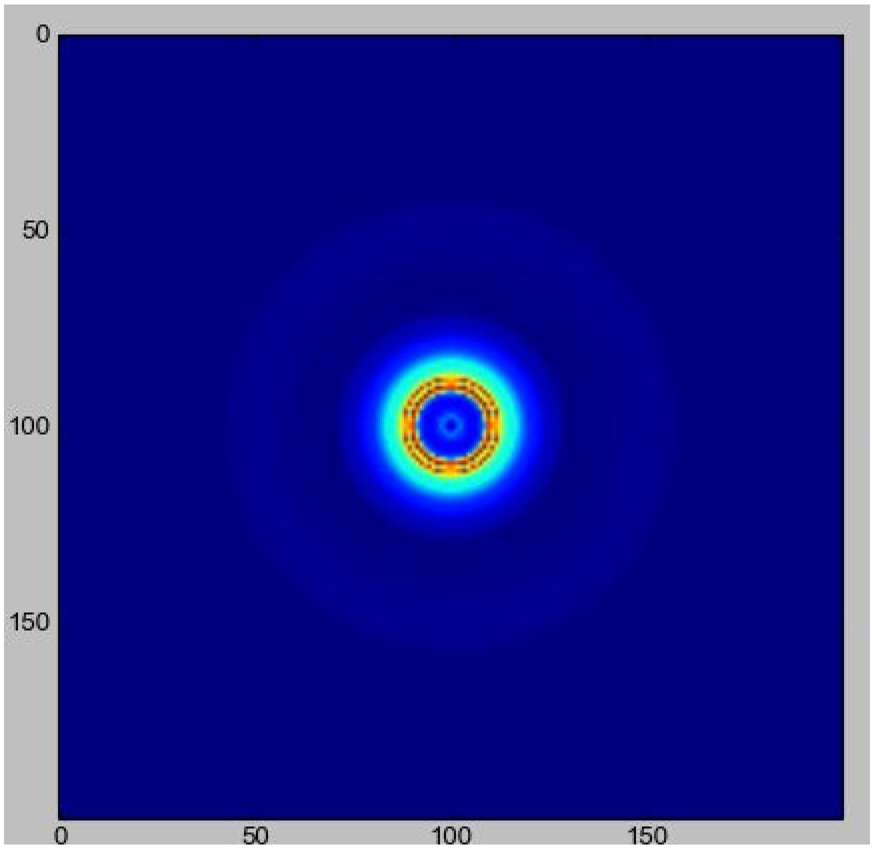
This method used the np.random.random_integers method to generate a list NSamples long of three points to compare the angles with. We will call these points a, b, and c. B-A and B-C were dotted and divided by their distances multiplied together to get the angle between them. Edge angle cases were also checked to make sure we were never dividing by 0 or returning invalid results. Finally, a histogram was created using the np.histogram method.

VII. Extended Gaussian Image

First, the PCA axes were computed by matrix multiplying the points with their transpose, taking the eigenvalues of the points, and sorting them. A rotation matrix was then created and the normals were multiplied by it to give rotated normals. Finally, a similar method to that in the sector matching was followed in that the distances were calculated with the sphere points and a maximum selecting method was used to select the normals were selected by the sectors that they were closest to sharing a direction with. The count of each of these point selections were added to a histogram and returned.

VIII. Spin Images
The spin image of the plane was calculated be first finding the PCA axes of the points through methods detailed above. The points were then rotated such that they were aligned with their greatest axis of rotation along the Z axis. Then, the points were compressed flat using numpy.histogram2D, which counted up the number of points in each bin in the XY plane. This

process was repeated as the points were spun about the XY plane, and added for all the angles in the theta divisions to a histogram, making the metric rotationally invariant. An example spin image for biplane0.off can be seen in the image below, which matches the example image provided in the assignment specification.
.



IX. Spherical Harmonic Magnitudes

This was the only part of the assignment that we did not complete.

X. Normalizing Histograms

There is a function *normalizeHists(AllHists)* that inputs a K x N matrix of histograms and outputs a K x N matrix of normalized histograms. It does so by dividing each column by the sum of its elements. It does the computation without any loops, simply calling the *np.sum* method and doing a matrix division.

XI. Euclidean Distance

The function first calls *normalizeHists(AllHists)* to normalize the input. First, aa is found, which holds the dot product of each histogram with itself. Then, bb is found, which is identical to aa. Next, ab is computed through the matrix multiplication of the transpose of normalized $AllHists$ and itself. Thus, each entry $ab_{i,j}$ is the dot product of the i-th and j-th vectors in the original normalized matrix. Now, D^2 is computed by applying broadcasting and the equation

$\left\|\overline{a}-\overline{b}\right\|^2 = \overline{a}\cdot\overline{a}+\overline{b}\cdot\overline{b}-2\overline{a}\cdot\overline{b}$. The square root is taken, which results in the distance matrix. Overall, the computation is done without Python loops.

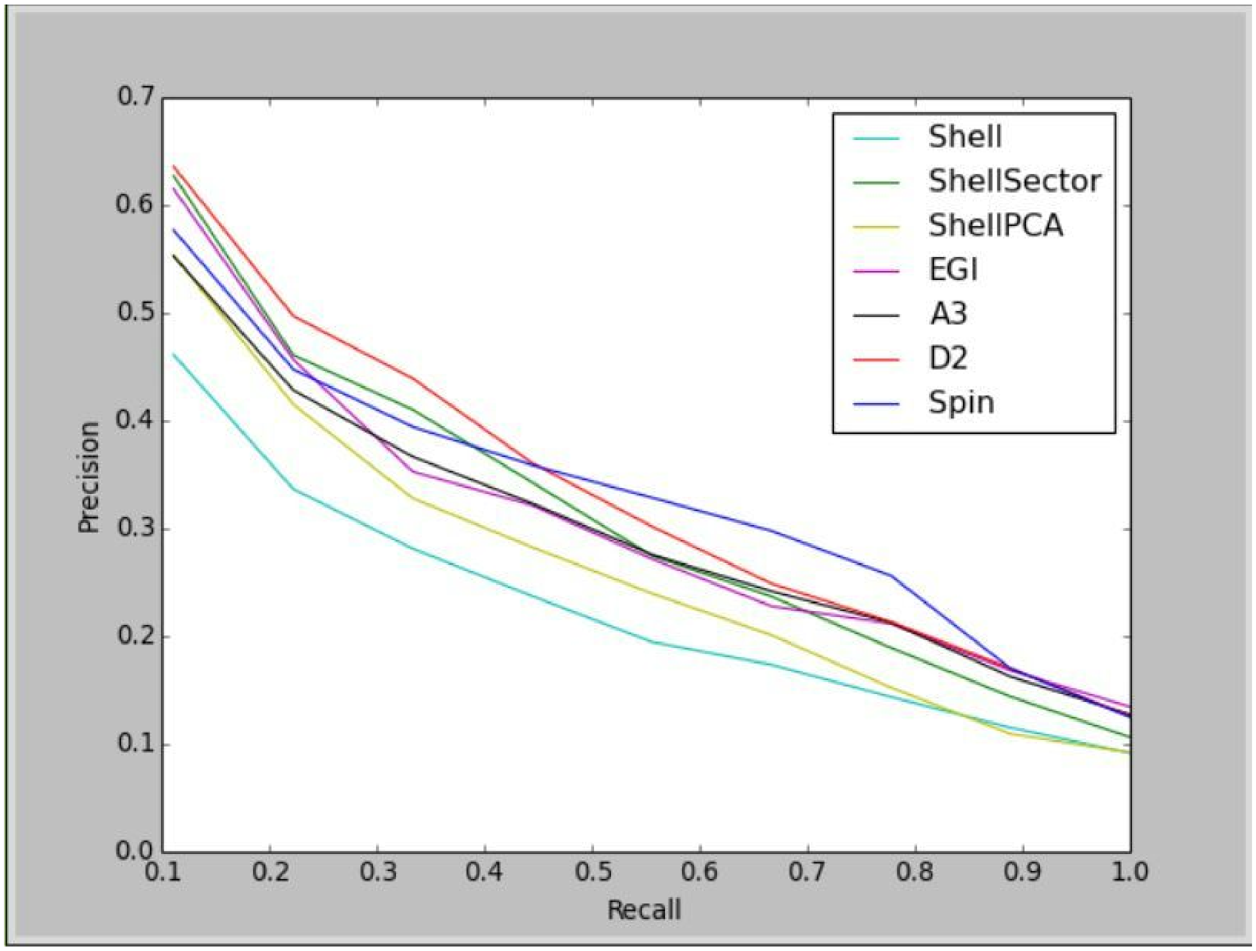
XII. Cosine Distance
The function first calls *normalizeHists(AllHists)* to normalize the input. The D matrix is found by computing matrices $numerator$ and $denominator$, which are then divided. $numerator$ is computed through the matrix multiplication of the transpose of normalized $AllHists$ and itself. Thus, each entry $numerator_{i,j}$ is the dot product of the i-th and j-th vectors in the original normalized matrix. The norms of the normalized histograms are found using *np.linalg.norm* and stored in a row vector $norms$. $denominator$ is then computed through the matrix multiplication of the transpose of $norms$ and itself, and holds all the permutations of products of pairs. The distance matrix is 1 minus the $numerator$ element-by-element divided by $denominator$. Overall, the computation is done without Python loops.

XIII. Chi Square Distance
The function first calls *normalizeHists(AllHists)* to normalize the input. First, the matrix sub is calculated, where each entry is 1D array $sub_{i,j}$ that holds the difference between the i-th and j-th vectors in the original normalized matrix. This matrix is squared element-by-element to form sub^2. This forms the numerator inside the sum in the given equation for Chi Square Distance. Then, the matrix add is calculated, where each entry is a 1D array $add_{i,j}$ that holds the sum of the i-th and j-th vector histograms in the original normalized matrix. This forms the denominator inside the sum in the given equation. Then div is formed by element-by-element dividing sub^2 by add. All that is left is to do the summation per histogram, so the distance matrix is found by finding the sums of the 1D arrays in each entry of div (and then dividing the whole matrix by 2). Overall, the computation is done without Python loops.

XIV. 1D Earth Mover's Distance
The function first calls *normalizeHists(AllHists)* to normalize the input. The h^C equation is found for each histogram by creating $histsC$, a matrix that holds the h^C vector for each of the histograms. This is computed by using a built-in numpy function, *np.cumsum*. Similar to the process for the Chi Square Distance, the matrix sub is calculated, where each entry is 1D array $sub_{i,j}$ that holds the difference between the i-th and j-th vectors in $histsC$. The absolute value is taken of this matrix to form $absSub$. This forms the expression within the summation, so all that is left to do is the summation per histogram, which is done by finding the sums of the 1D arrays in each entry of $absSub$. Overall, the computation is done without Python loops.

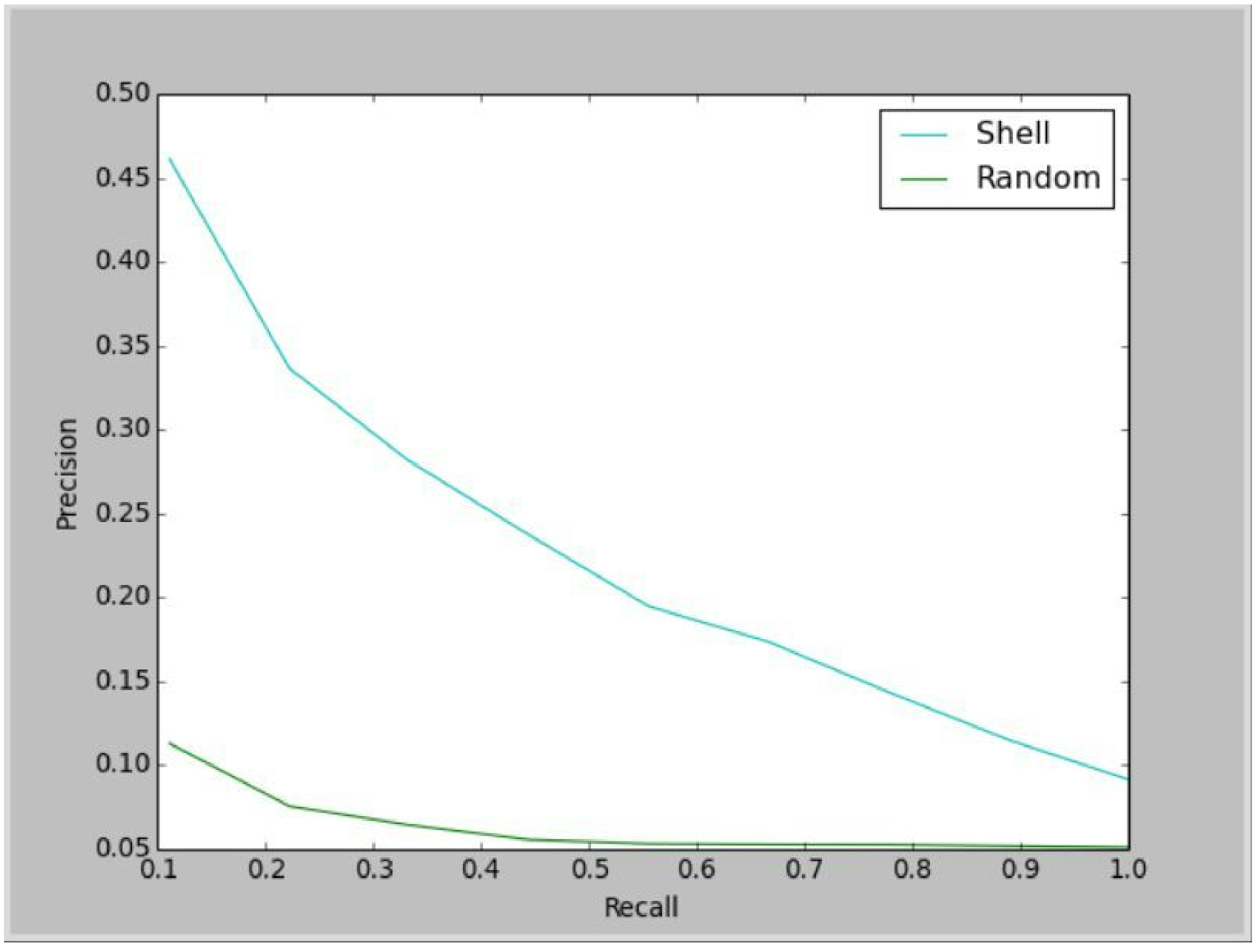
XV. Performance Evaluation

Precision-Recall Graph for All Descriptors using Euclidean Distance:



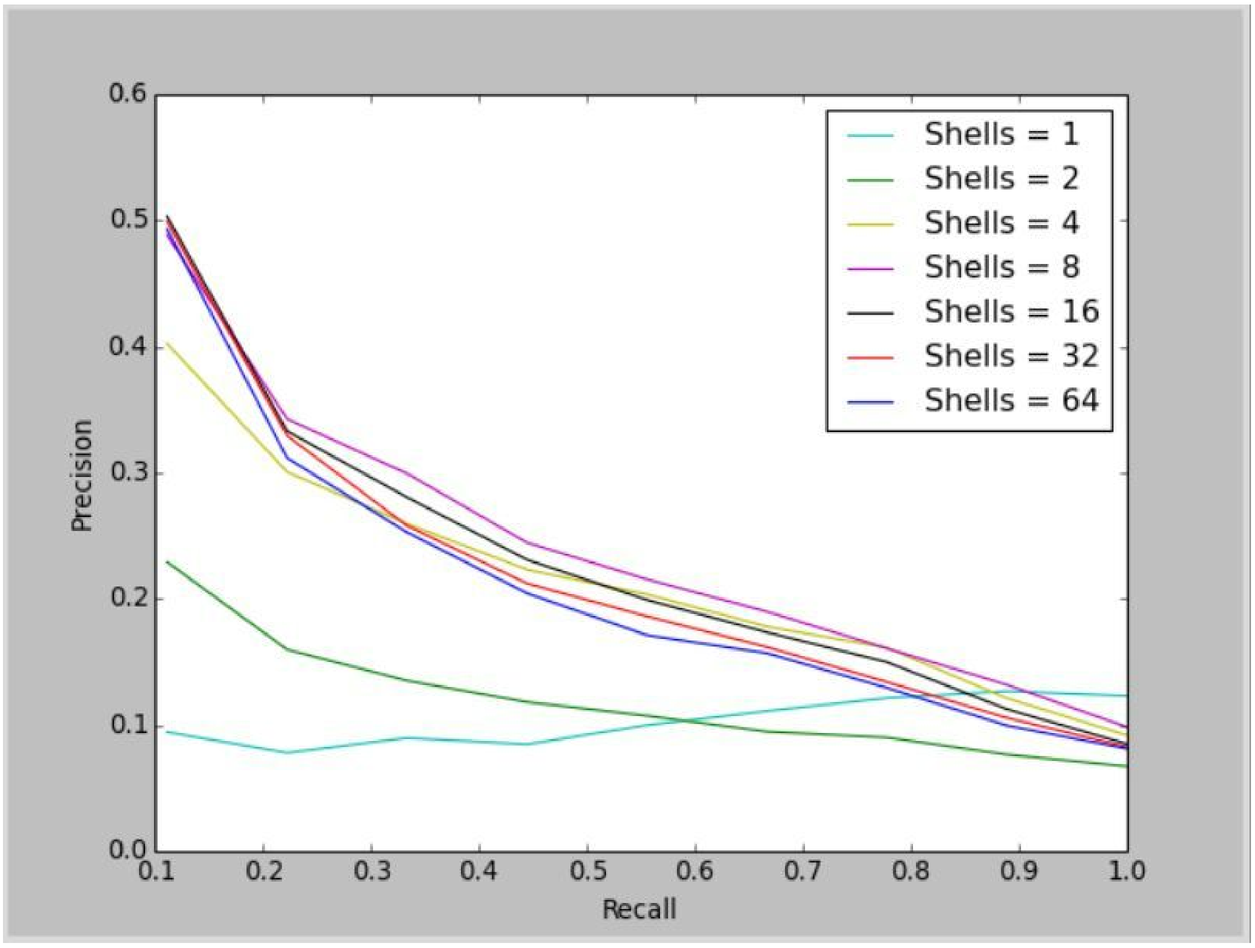
The different descriptors all perform fairly similarly. As expected, the simpler Shell computations are the lowest performers. This is because of its high tolerance for object variation around the shell. As can be seen in the graph above, adding shell sectors greatly improves the metric. ShellPCA lowers relative effectiveness but is still a great improvement over individual shells as it still uses the same amount of space as the traditional shell method. Surprisingly, D2 performs rather well in comparison to the others, although it also had the longest computation time due to taking 100,000 samples. As the recall goes out, the Spin image comparison overtakes D2 to become the most accurate descriptor.

Comparison to Random:



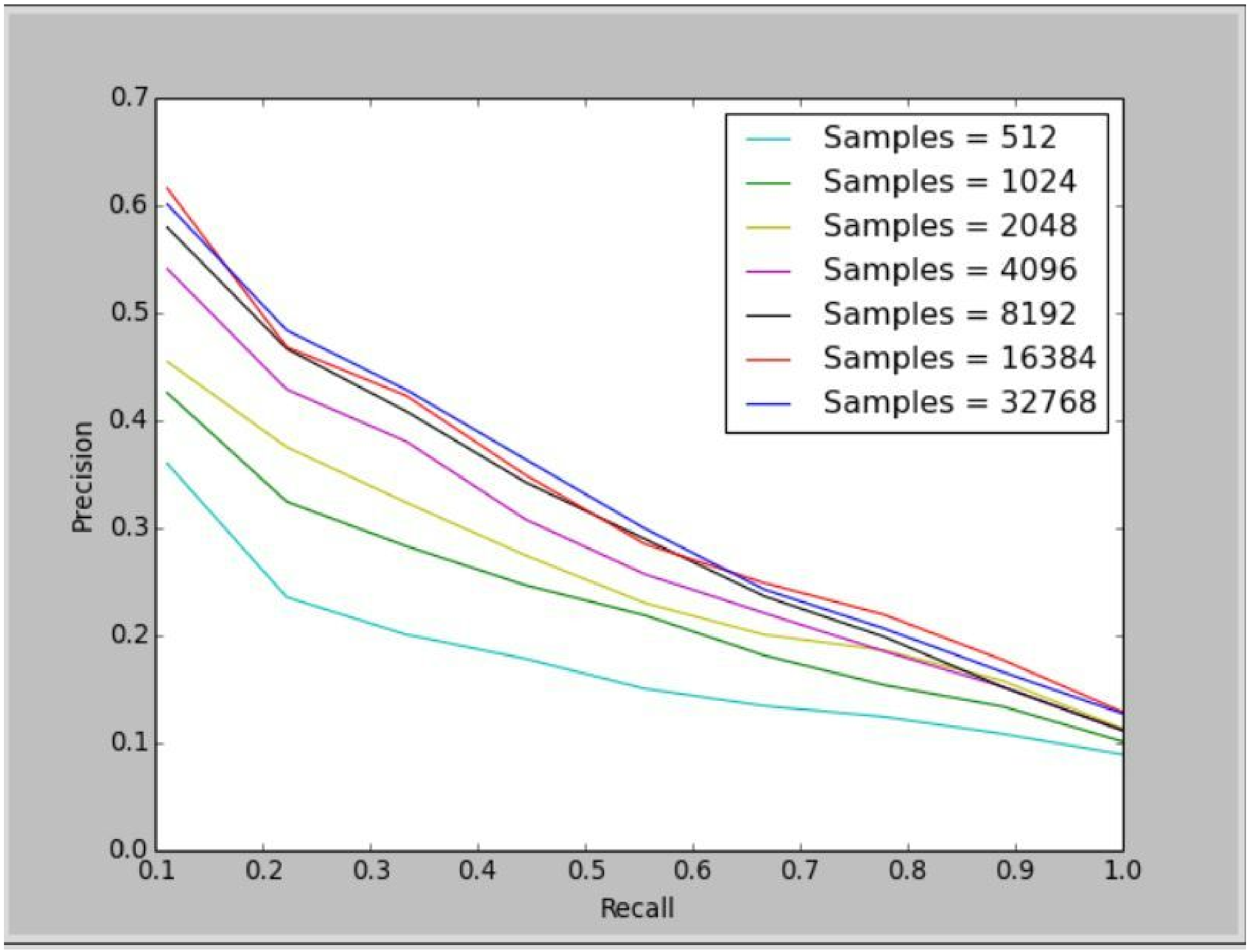
If histograms are generated using numpy.random, and have no correlation to the actual point clouds, the performance is extremely poor. Even the worst performing metric, Shell, severely outperforms the Random metric, which has almost no precision as the recall increases towards 1.0.

Varying Numbers of Shells:



Having only one or two shells does very poorly. If all of the points exist in only a single shell, then there is no differentiation between any of the shapes. Based on our chosen parameters, the optimal number of shells for this algorithm is 8. With too few shells, not enough differentiation is made. However, as the number of shells goes above 8, there is too much variation between even shapes that should be recognized as the same thing. Because of this, there are false negatives, resulting in a lower score.

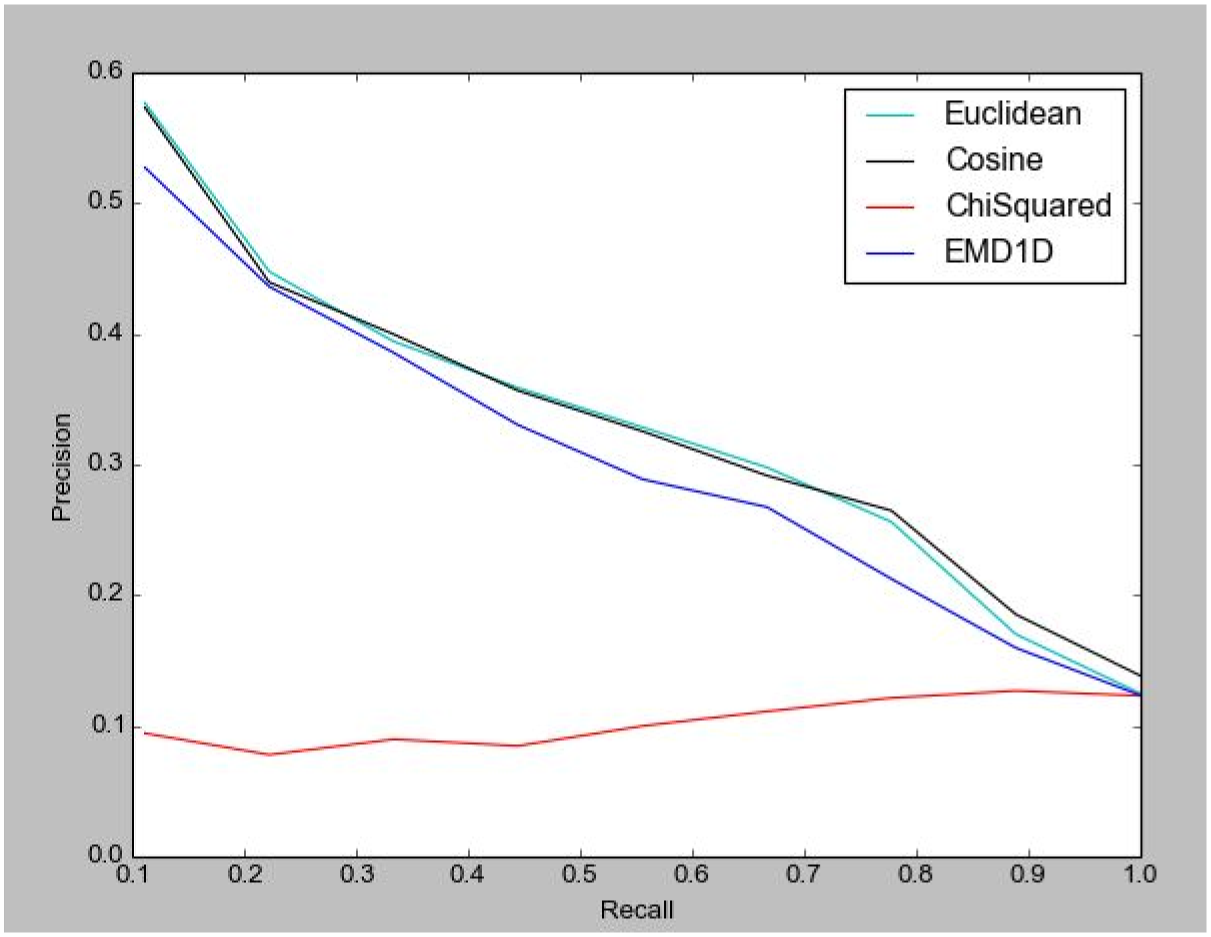
Varying Numbers of Samples in D2:



As the number of samples increases, the performance improves. However, this improvement is limited, and by the time the number of samples reaches about 10,000, the improvement has stopped. This is expected behavior, as continuing to sample beyond a certain point will not give better characterizations of the shape, only more data points to analyze. In addition, the amount of time each computation took scaled linearly with the number of samples, with 400,000 samples taking almost 15 minutes to complete.

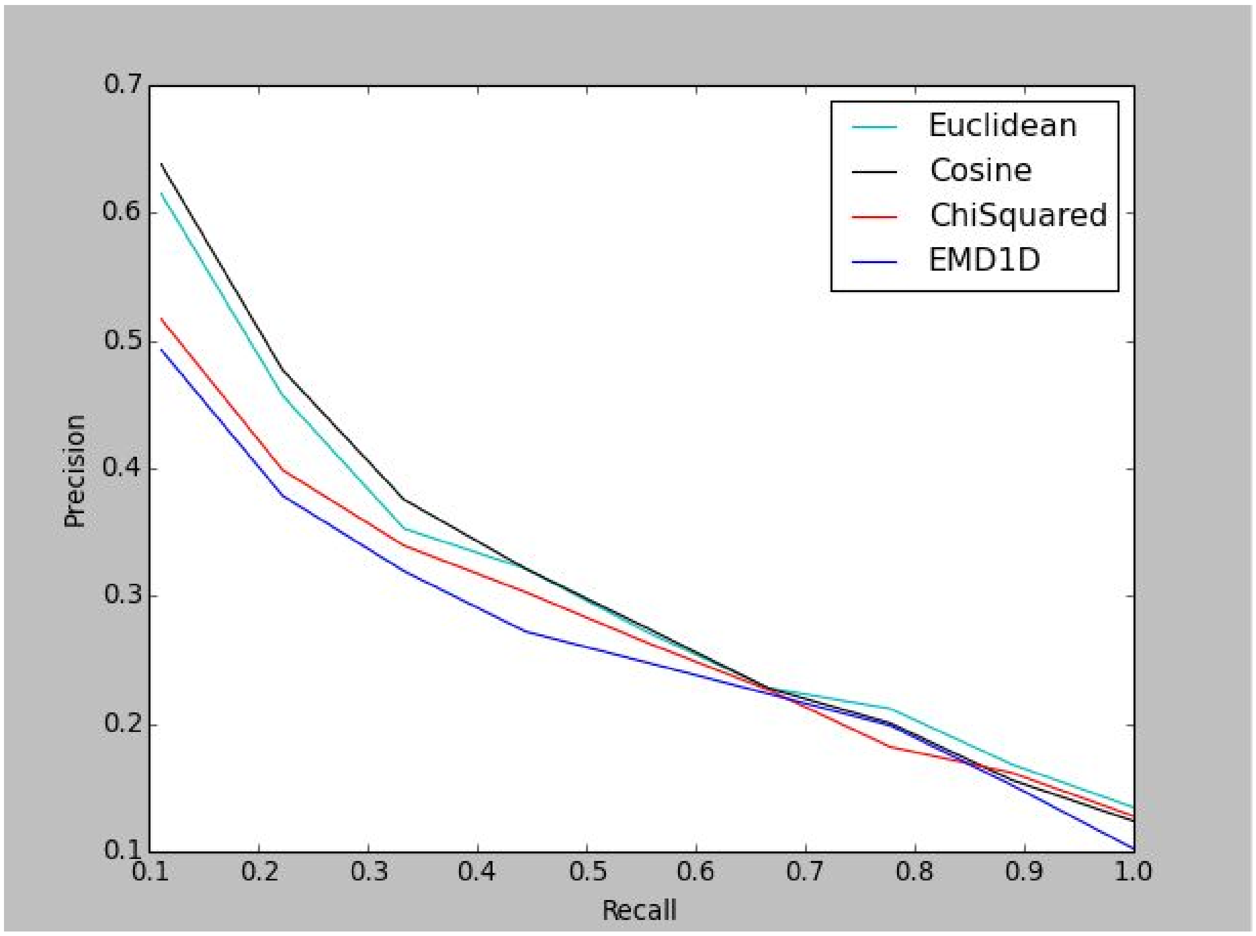
Distance Metric Comparison:

Distance Metric Comparison Using Spin Images



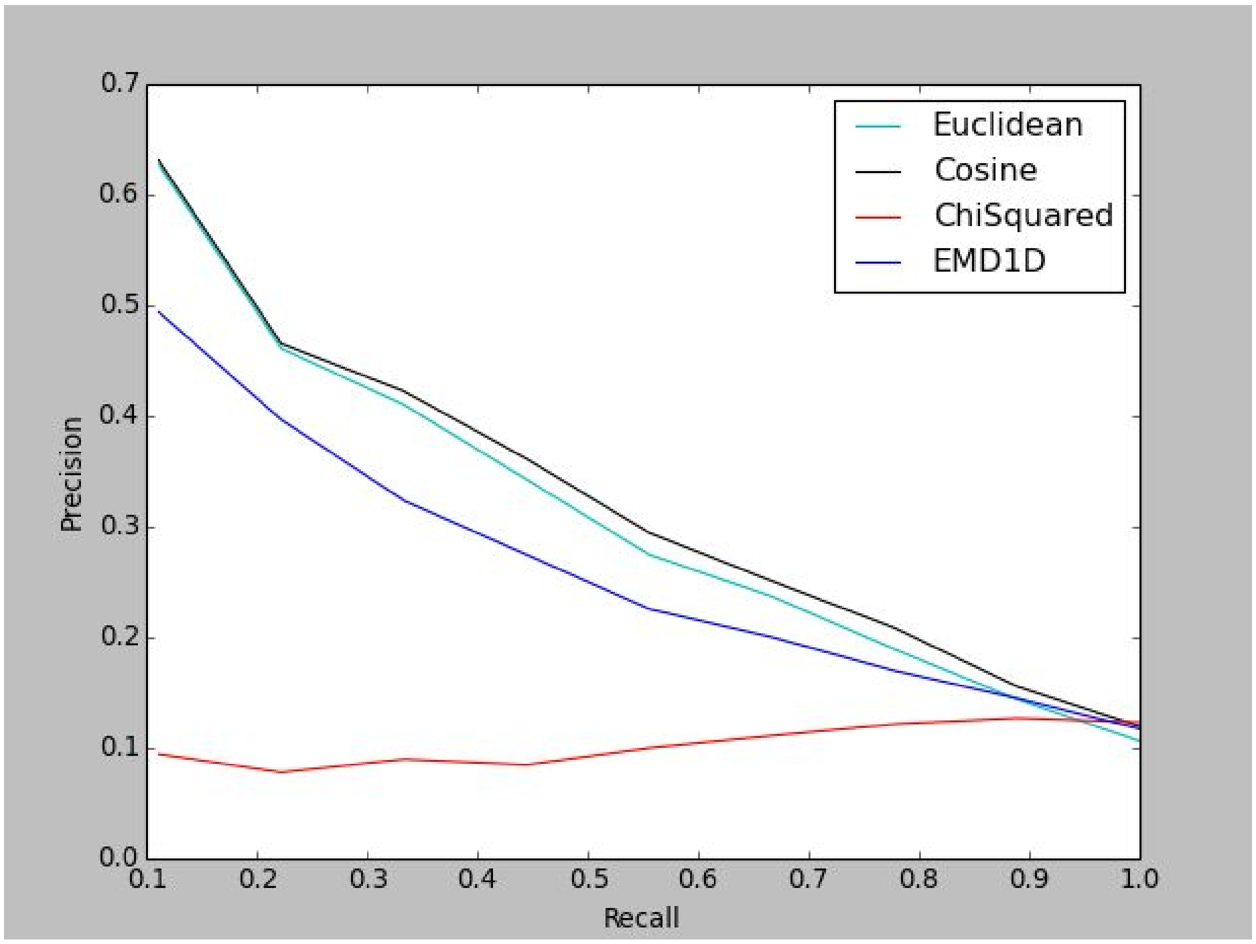
The graph above is computed using the Spin Images feature. From this plot, it appears that the Euclidean distance and Cosine distance metrics perform best. The 1D Earth Mover's Distance also performs well, although it is slightly worse than the other two. The Chi Square distance metric performs very poorly. This is likely because it divides by $h_i[k] + h_j[k]$, since when this is removed, it performs similarly to the other three.

Distance Metric Comparison Using Extended Gaussian Image



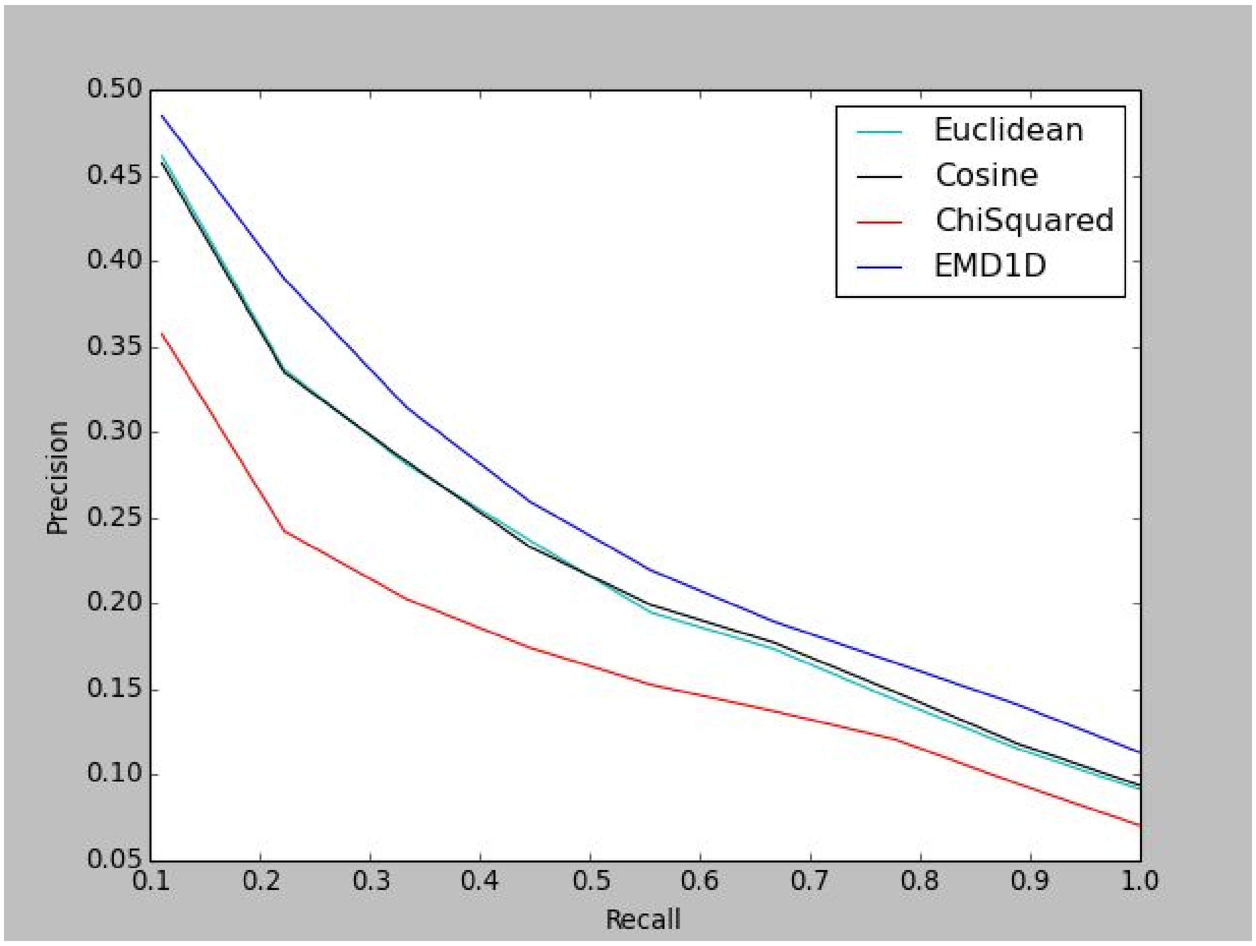
The graph above is computed using the Extended Gaussian Image feature. Like for the Spin Images feature, from this plot, it appears that the Euclidean distance and Cosine distance metrics perform best. The 1D Earth Mover's Distance and the Chi Square distance also perform well, although they are slightly worse than the other two. It is interesting to note how much better the performance of Chi Square for this feature is than for the Spin Images feature.

Distance Metric Comparison Using Shell Sector



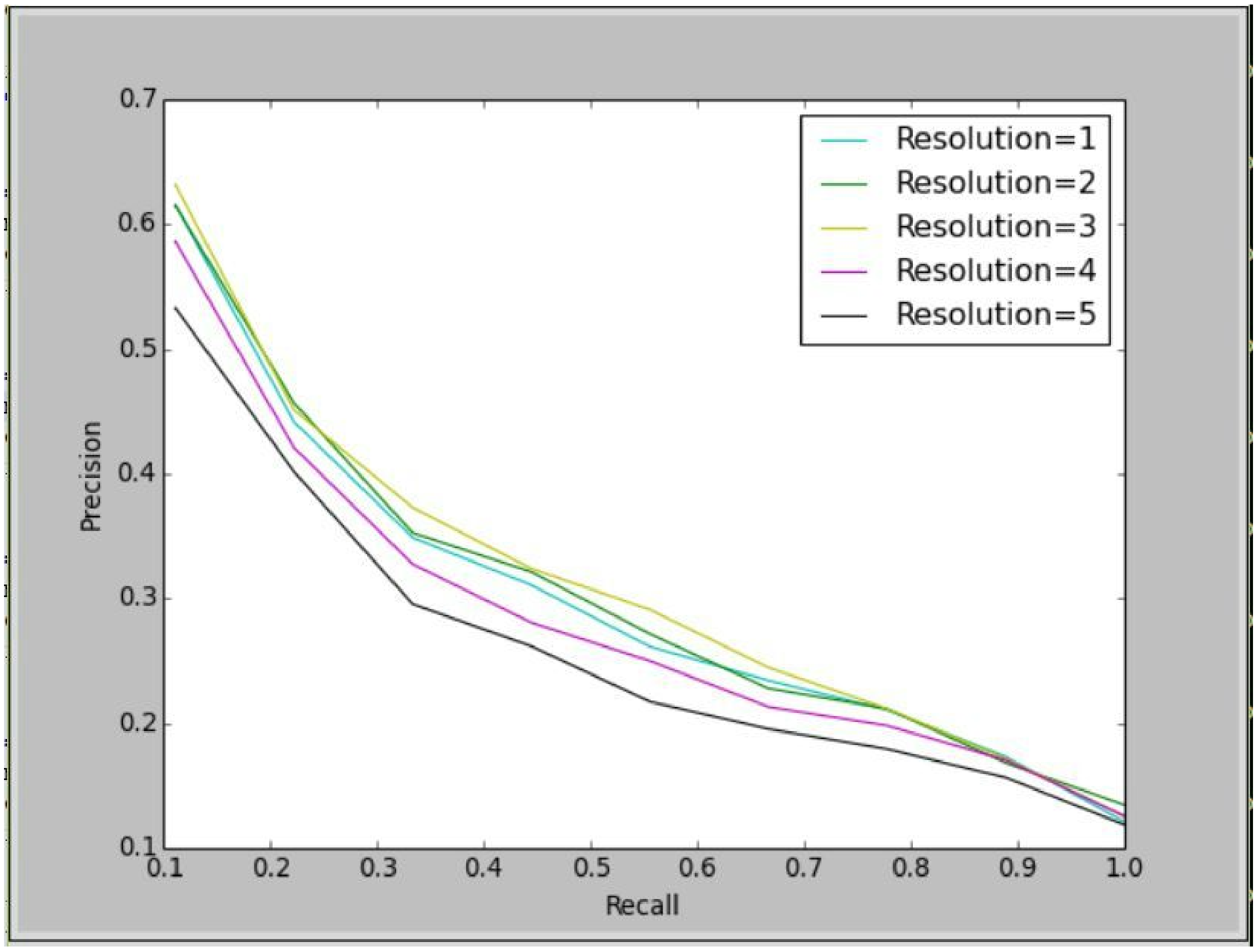
The graph above is computed using the Shell Sector feature. Like for the Spin Images and the Extended Gaussian Image features, from this plot, it appears that the Euclidean distance and Cosine distance metrics perform best. The 1D Earth Mover’s Distance also performs well, although it is slightly worse than the other two. Like for the Spin Images feature, the Chi Square distance metric performs very poorly.

Distance Metric Comparison Using Shells



The graph above is computed using the Shell feature. Unlike for the other features, the 1D Earth Mover's distance metric performs best. Similarly to for the other features, the Euclidean distance and Cosine distance metrics perform well, although not as well as the 1D Earth Mover's distance metric. The Chi Square distance metric does not perform as well as the other metrics, but it does not perform as poorly as it does for the Shell Sector feature and for the Spin Images feature.

Extended Gaussian Image Sphere Samples:



Increasing the resolution of the sampling of the sphere linearly gives an exponential increase in the number of samples that are taken. However, having more or less samples does not provide a large performance increase. Once resolution gets too high (resolution = 5), then even the small variations between shapes causes them to be classified as different objects. The best performing resolution is resolution = 3.