4/16/2020

Jack Hale

Cylinder Detection Work Log

This document serves as a log for the work done to implement a cylinder detection algorithm using RANSAC. Please note that there were some difficulties balancing efficient coding with thorough documentation, and as a result not all debugging is included. This log is formatted in a stream of consciousness way, and should not be considered for it's grammatical or illogical errors.

When initially starting the thought was to complete the RANSAC portion of EX4 so that a successful cylinder detection algorithm could be found and implemented into a python script for analysis with the output of our turtlebot2i environment.

This approach was abandoned due to research into available python packages and the differences a successful algorithm would need between the two languages.

Further research suggested that the open3d python package would be suitable for our point cloud analysis. Attempting to install using pip on windows 10 as described in the release docs did not work for python version 3.8.0. The package was successfully implemented using conda instead. As a result the majority of implementation and scripting for this vison system was done in the Spyder python environment before full implementation and testing with the turtlebot simulation environment.

```
PS C:\Users\Jack> pip install open3d==0.9.0
ERROR: Could not find a version that satisfies the requirement open3d==0.9.0 (from versions: none)
ERROR: No matching distribution found for open3d==0.9.0
PS C:\Users\Jack>
```

http://www.open3d.org/docs/release/getting_started.html

A not insignificant amount of time was spent reading documentation on the open3d package, literature on robotic vision methods such as RANSAC and ICP, and familiarizing ourselves with the lecture notes. Of interest was the paper:

Q.-Y. Zhou, J. Park, and V. Koltun, Fast Global Registration, ECCV

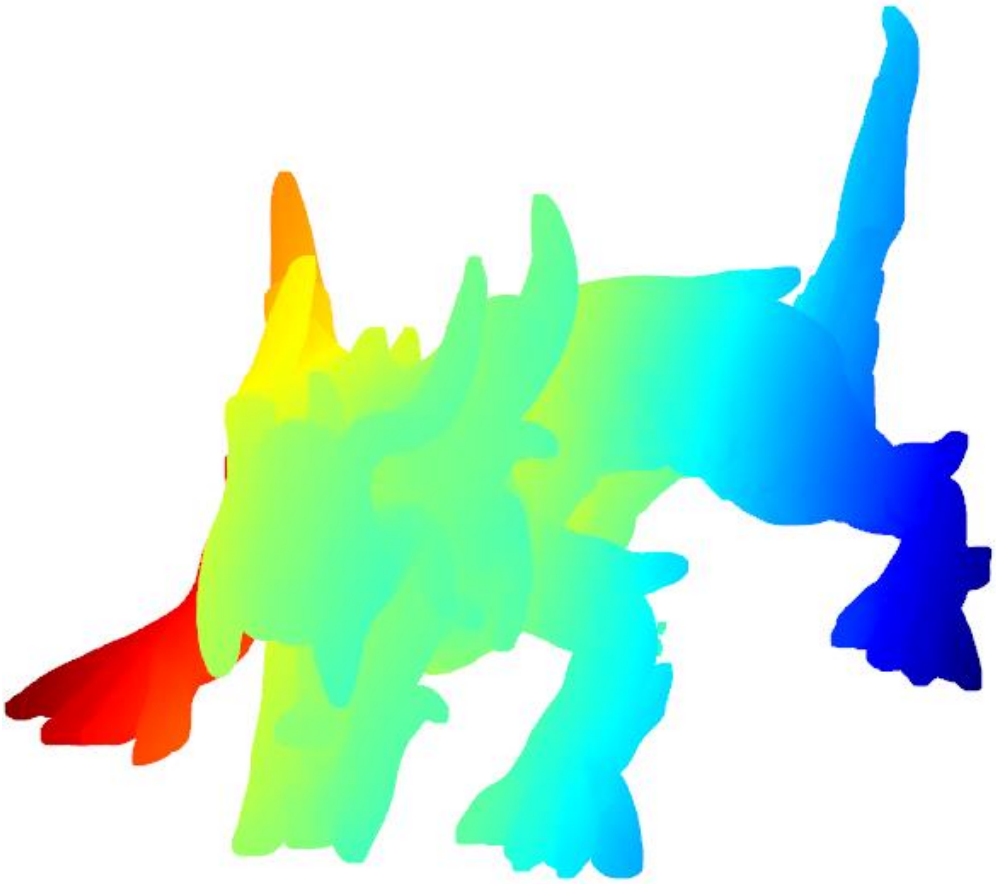
Which describes a method of fast global registration. This algorithm is meant to be significantly faster than when RANSAC global registration and local refinement ICP are implemented for feature recognition. Results of this paper suggested that the fast global registration algorithm was significantly more efficient than the previously used two step process.

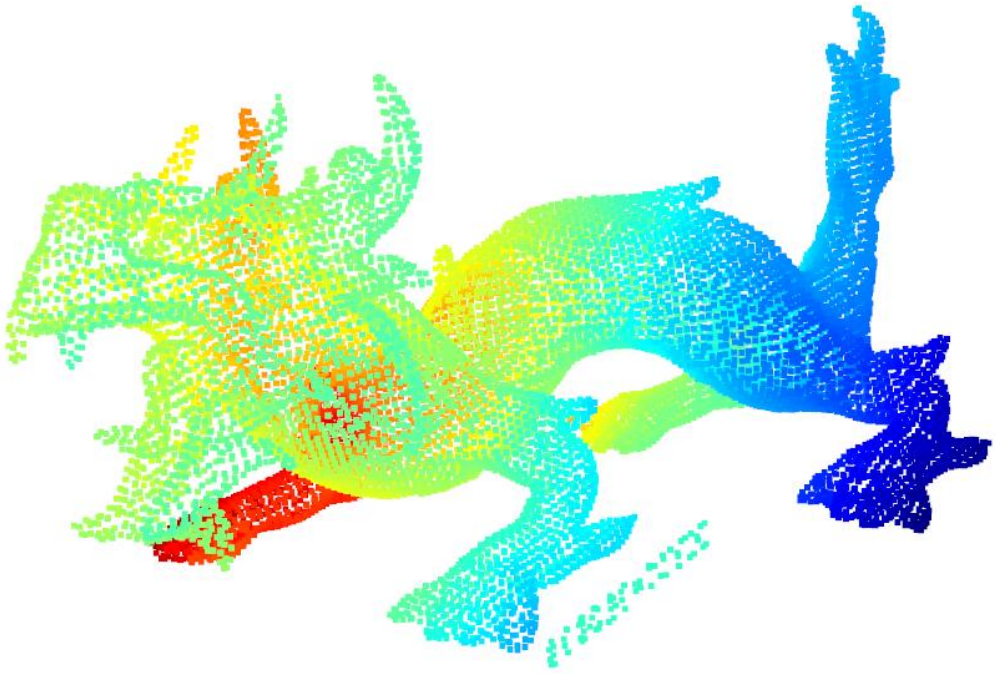
Further readings of the implementation of the fast global registration and general registration methods in the open3d library makes it seem that similar implementation would require prior initialization where the general cylinder shape is extracted from the point cloud seem in our environment, compared to a target point cloud of known location with reference to our vehicle position, and the resulting transformation from previously researched registration methods will allow us to determine distance and orientation of the cylinder with respect to our robot. Finding geometries in the point cloud was researched in this paper, which further describes use of RANSAC in general shape detection. The included algorithm is described as robust and scalable, which while not entirely necessary for the simulation environment we are using would be useful for further development of our project.

https://cg.cs.uni-bonn.de/en/publications/paper-details/schnabel-2007-efficient/

To test implementation of our cylinder detection we needed sample data sets. Some were found here http://www-graphics.stanford.edu/data/3Dscanrep/

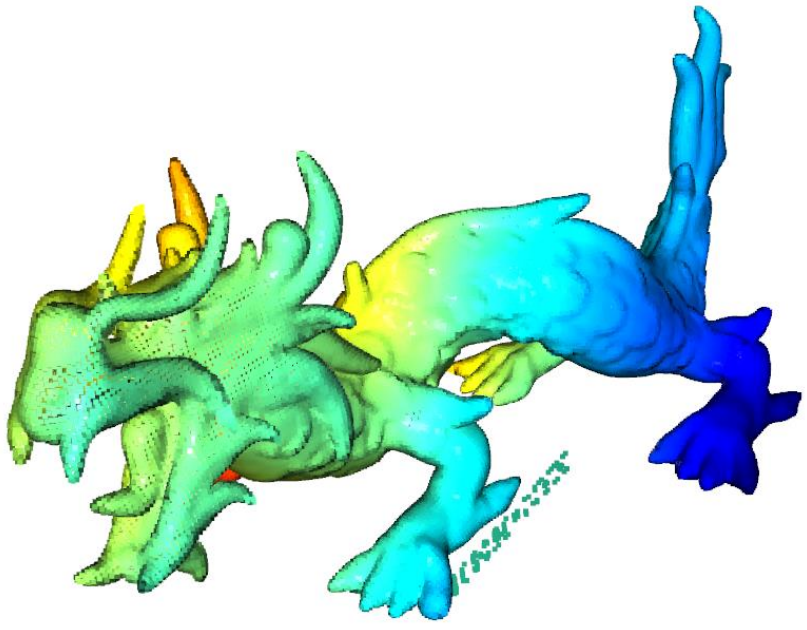
This data set was used to test cylinder detection algorithms. First, the initial point cloud was down sampled with a voxel size of 2 for ease of calculation. This voxel size was arbitrary and decided iteratively for this specific data set to optimize keeping features with the minimum number of points.

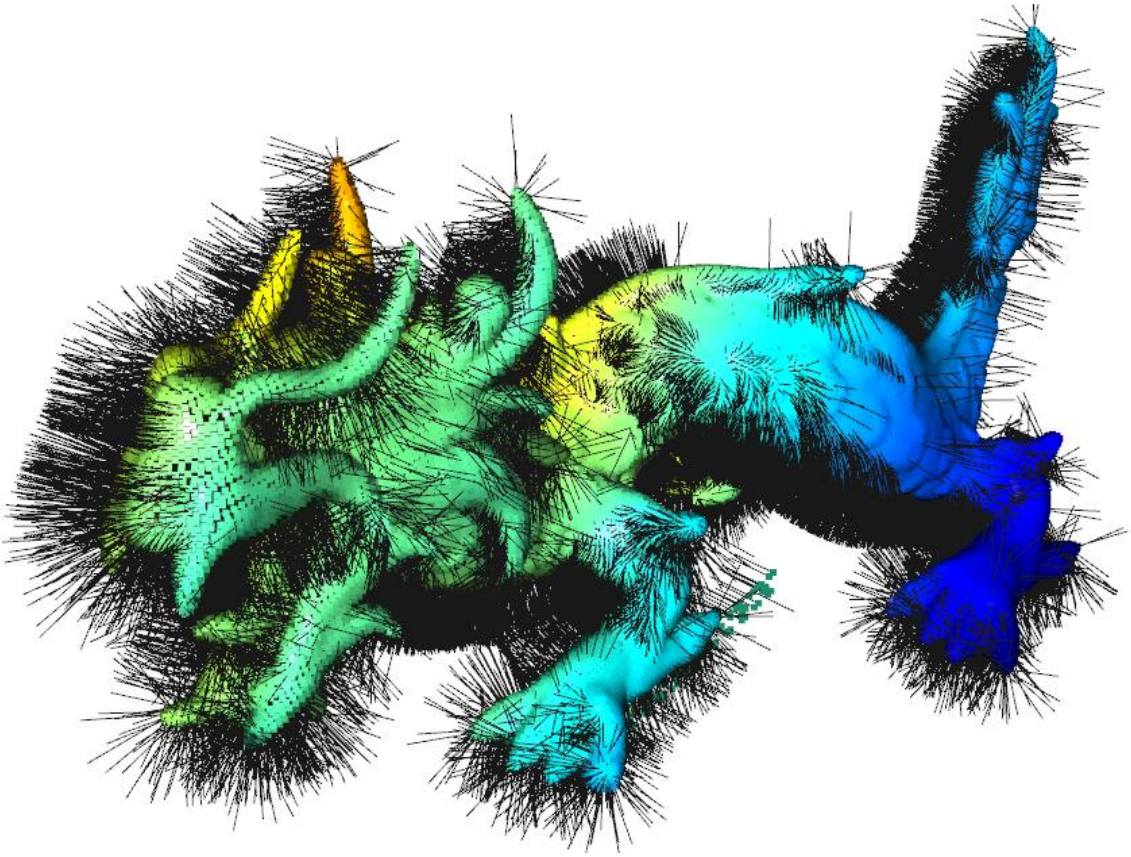




The goal is to successfully define portions of the dragon's legs as cylinders.

When calculating the surface normal's it was found that the voxel size we specified was too large for meaningful calculation radius. Voxel size was reduced to 1 and the radius used for surface normal estimation was set to 2. The resulting data set can be seen below.





When calculating axis of sampled points some research had to be done into how numpy works with arrays compared to the matlab implementation. While transposing 1D arrays in matlab is trivial, doing the same in python requires the use of the numpy.newaxis method to make our vector a pseudo 2d array to be able to transpose it.

To ensure that data points were properly being transformed to the plane orthogonal to the axis of the two randomly sampled data points the calculated array was converted into a point cloud object and plotted

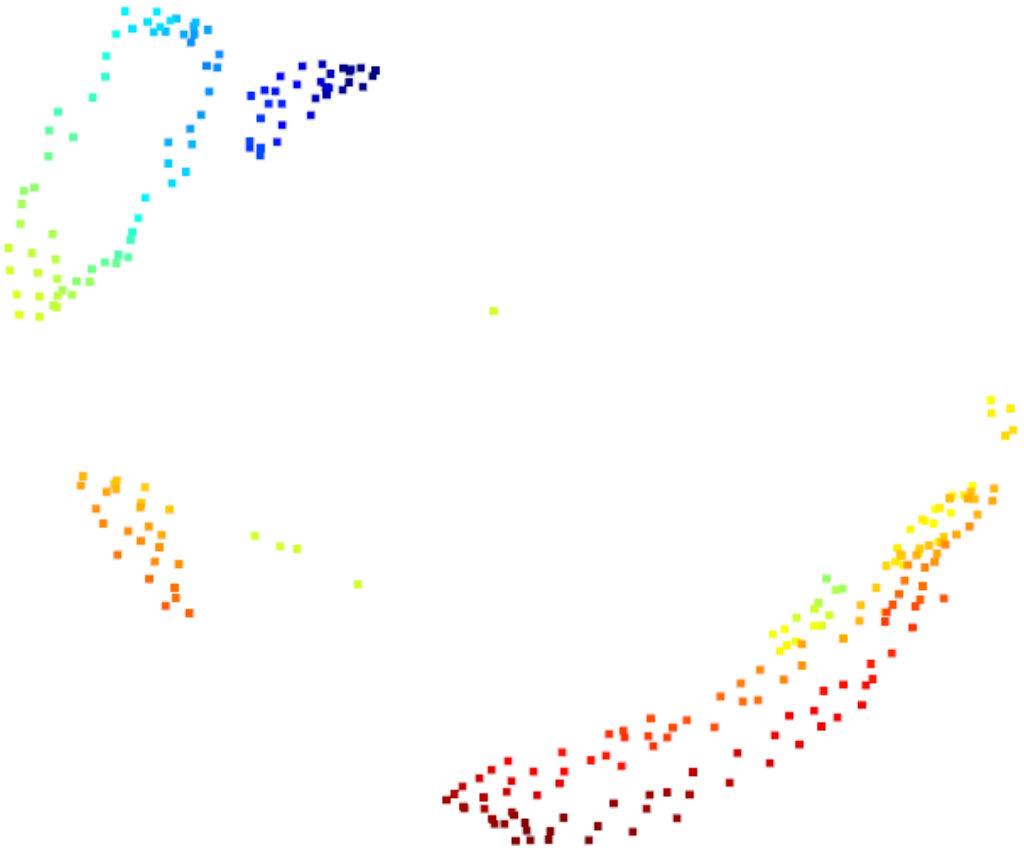
In order to determine a cylinder radius that is suitable for our data set we attempted to plot the down sampled point cloud array with matplotlib so that we could see data points around the arm and estimate reasonable cylinder sizes. When adding a matplot lib sub_scatter plot the error, “no ‘3d’ projection found” was encountered. We determined this error was because we had not imported the Axes3D package from mpl_toolkits.mplot3d. However, once this error was fixed the output was a static plot we could not interact with. Eventually it was decided to incrementally change the parameters to find the optimal setting for our practice point cloud.

Once the RANSAC algorithm was written there was significant runtime due to the highly random nature of the two sampling points. There was a low probability of finding a suitable pair. We set the voxel_size

to 1, and the radius of this downsampling to 2. Then we attempted to implement a sampling algorithm that based the second sample off the first random sample as described in

https://cg.cs.uni-bonn.de/aigaion2root/attachments/schnabel-2007-efficient.pdf

This allowed us to successfully implement an algorithm that extracts points in a cylinder as seen below



The poor spacing of these points suggest we need to tweak our parameters. Our current algorithm is too accepting of data points, where it is accepting a sample that is not composed of a cylinder, but a cylinder shape randomly cut through our sample.

Inspecting the calculated cylinder radius from the sample point and sample axis we determined that our initial code was performing incorrectly. The maximum distance between points in the point cloud was calculated to be less than that seen in the cylinder radius. This suggests an error in the code.

We also discovered that our surface normals were mostly only the direction of one axis, resulting in many parallel lines being used to find the cylinder axis. This was because the radius we used to search nearest neighbors for surface normal was smaller than the size of voxels used for down sampling.

Upon further research this paper, https://link.springer.com/content/pdf/10.1007%2Fs00006-017-0759-1.pdf, describes more of the RANSAC process, and we tried to implement sampling from more localized points in the cloud and caps on the number of iterations

In hopes to use our ransac algorithm on a more realistic sample, we downloaded the default matlab point cloud object3d.mat as a .ply file. Loading this into our script we had to heavily change our parameters due to a change in scaling.

A good portion of time was spent changing from one large test script to separate python functions that could be called as pleased. One issue seen was a runtime error when attempting to use the visualize_pcd function to convert our calculated array into a point cloud object. After some investigation we found that it was caused by a misshapen array, where the array being passed had dimensions 3xn instead of nx3. We implemented a check for the function where if an array was 3xn we would take the transpose before attempting to plot it.

We were seeing errors where if no suitable cylinder was found we get an error message. We added a check to instead pass false so that instead of failing the program we could print a reason for failure

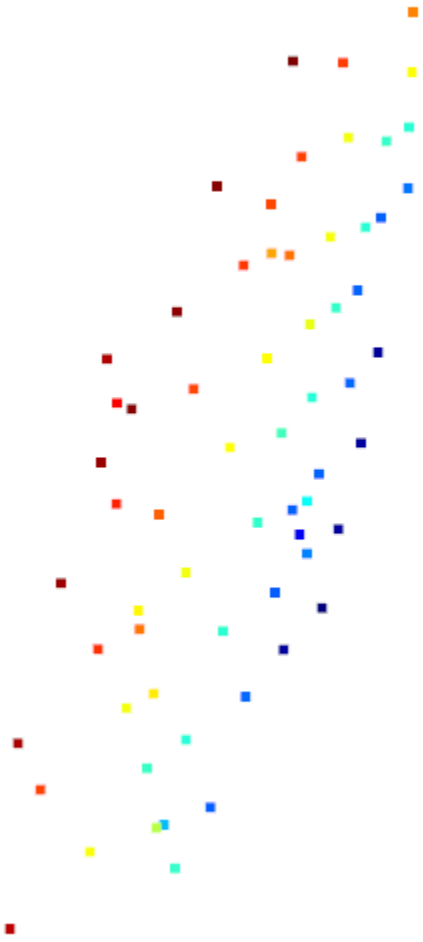
A good portion of time was spent fleshing out the individual python functions to be able to take different modes, and dynamically change parameters so that they were more robust. This time also includes writing thorough documentation and comments in all functions and improving readability.

A large error could potentially be due to our calculation of the center of the hypothetical cylinder. For a large portion of testing the normal unit vector for the sampled axis was multiplied by the projection matrix to get a center point on the orthogonal plane. This may be an incorrect assumption. We also noticed that when outputting the viable points, we were using the full down sampled point cloud instead of the local group. This resulted in the incorrect index being used and the wrong points being passed from the function.

We recalculated the center point by projecting the sampled point onto the orthogonal plane, multiplying the specified radius by the surface normal, and adding them together. However, using this method we

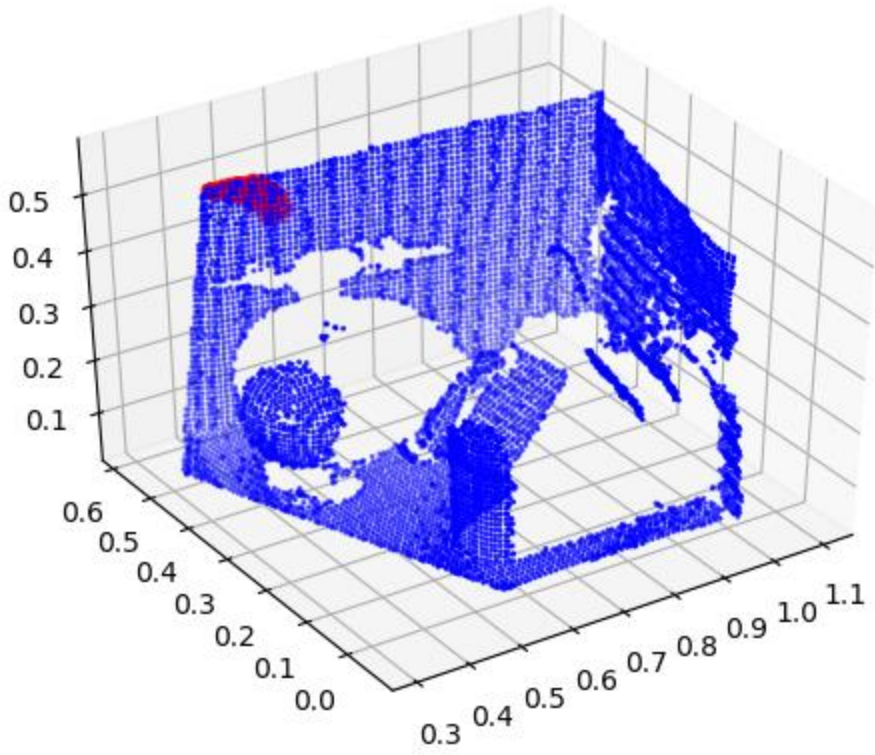
still were calculating distances that would be greater than the farthest distance between two points in the point cloud. This was checked using our max_distance function. This shows that there is an error in our math.

However, when running we do occasionally get data that could look correct, where it is not a cross section of the entire cloud.



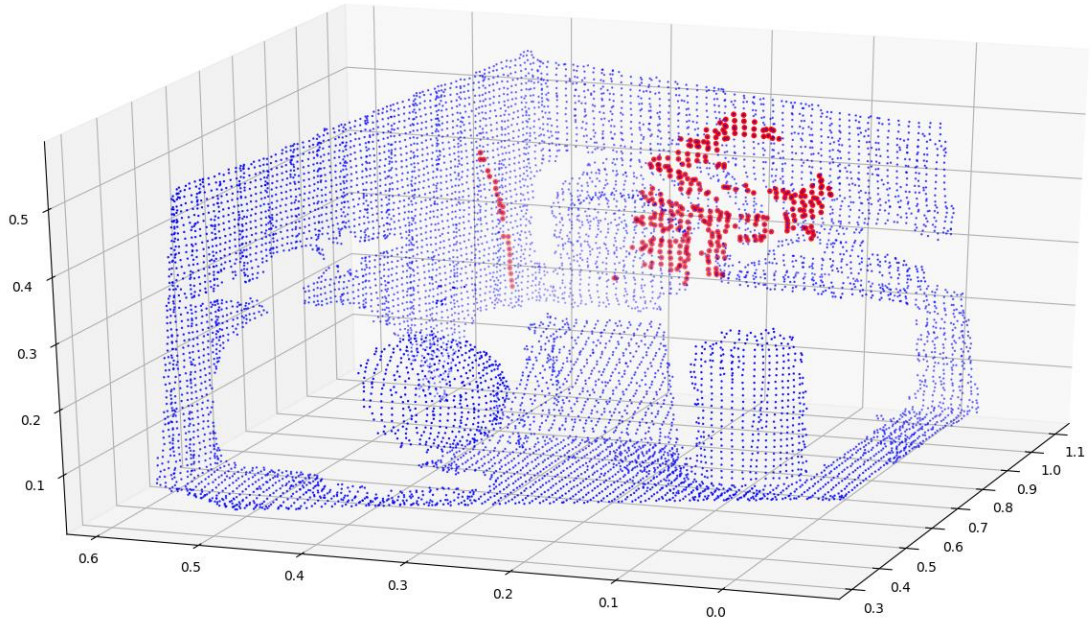
To calculate the dynamic radius of the cylinder, we determined that we must find the intersection of the two sampled surface normal on the orthogonal plane. We did this by solving the over determinate system of equations set by the sampled points and their surface normals. Solving this we obtained the amount we must move on the surface normal to reach the point that would be in the center of the cylinder.

Using matplot lib in python to compare the location of our detected cylinder and the actual point cloud we see the below plot

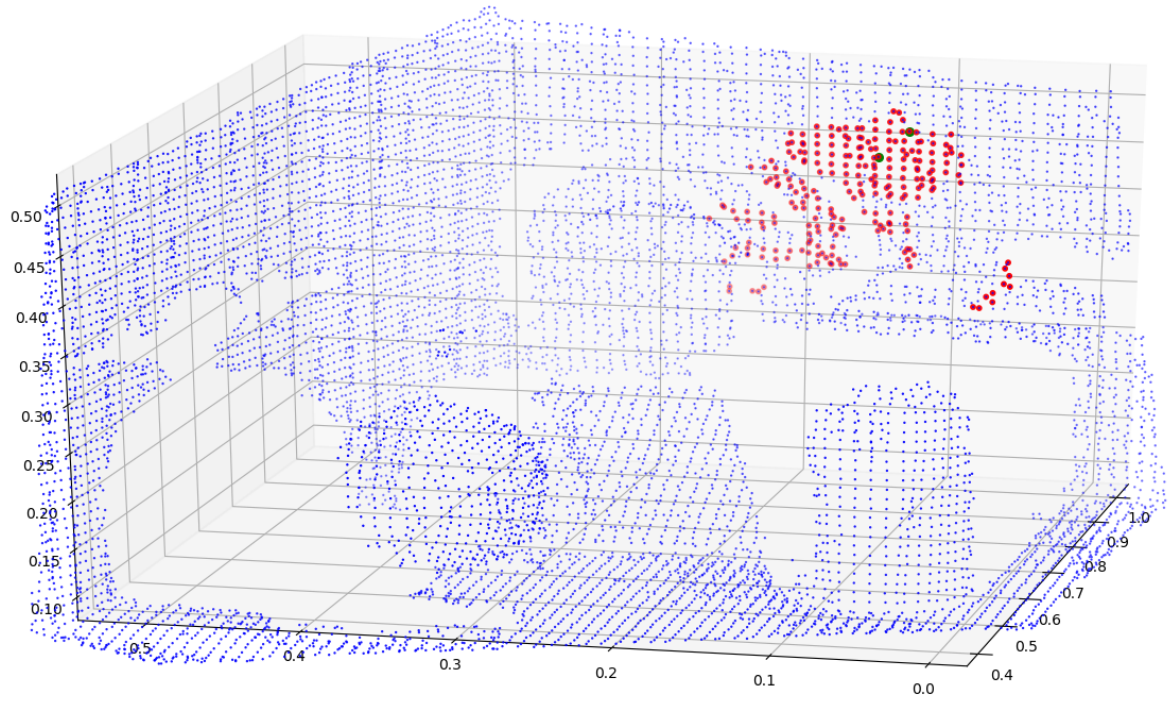


Where the red section is what we detected. This surface is consistently detected as our cylinder, suggesting that there is a fundamental error in our algorithm.

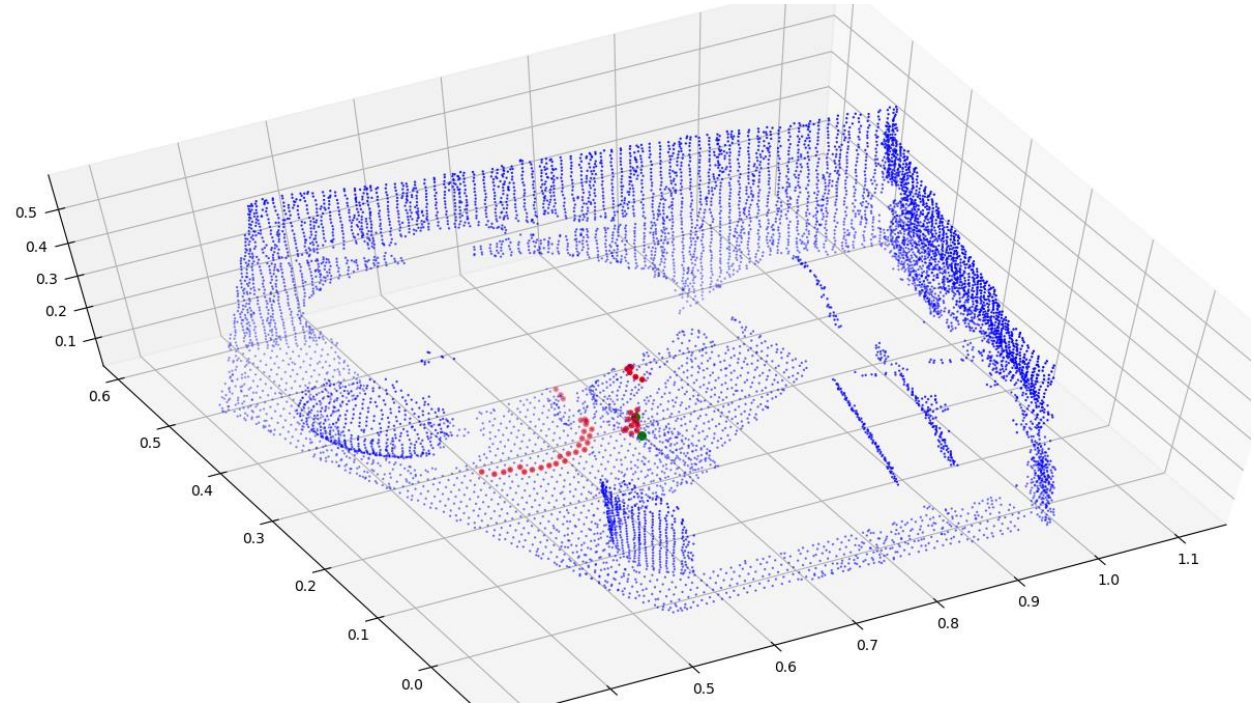
We found an error in the shape of matrices when finding all the points close to our sample. This was fixed and wildly improved the efficiency in our algorithm, but still showed a flaw in our method



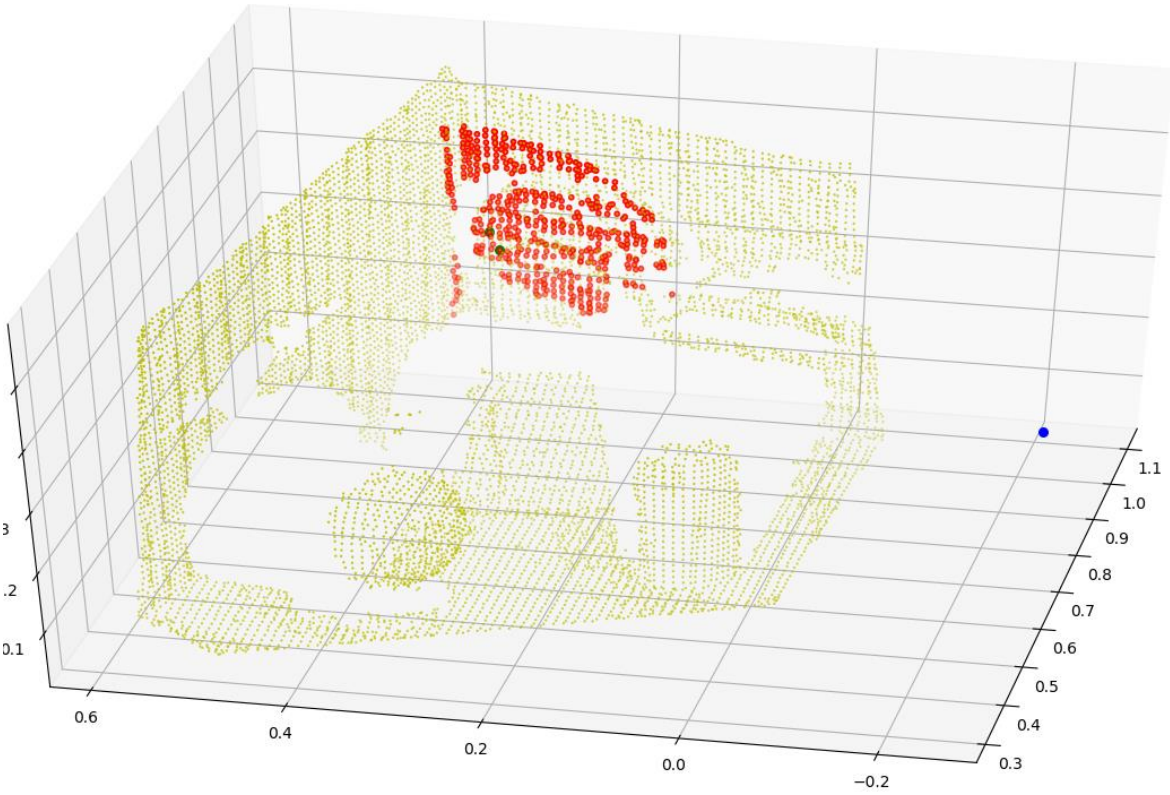
Taking the two ‘successful’ sample points from our ransac algorithm and plotting them over the point cloud we hope to root cause the error in our calculations. The sample points are seen in green below.



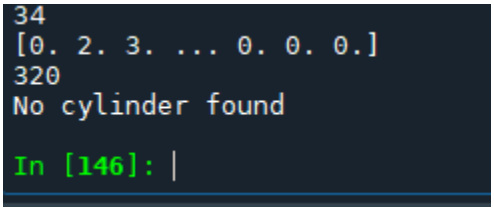
Increasing inlier requirement seems to decrease the number found. Going from 200 to 500 inliers results in the plot below.



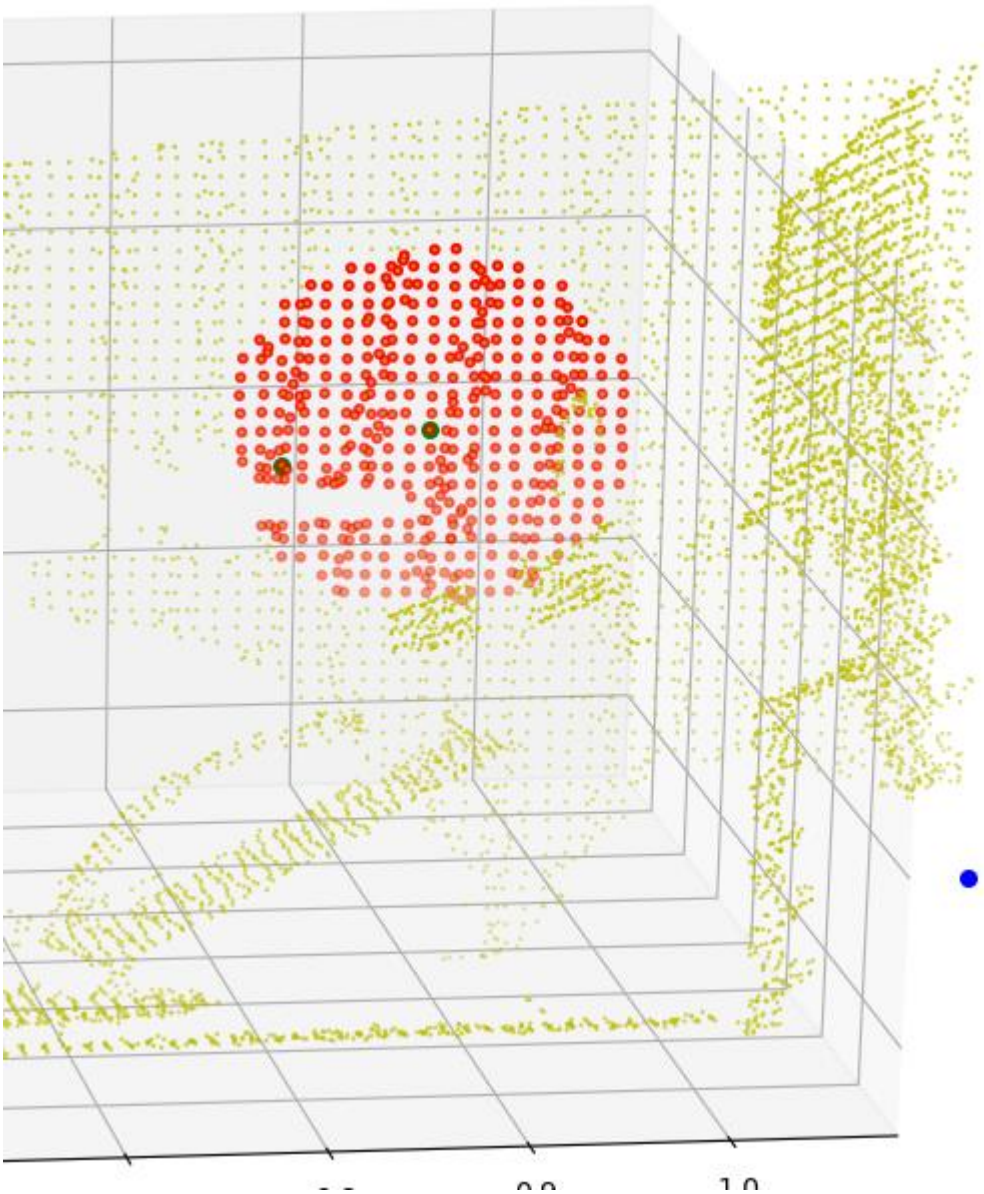
Further trying to determine where our errors lie, we plotted the calculated center of our 'detected' cylinder, seen in blue below.



Clearly there is an issue with our inlier detection. Our center detection is based off of the crossing point for the surface normals if our sample points. If all included points are in the same plane as the sample points and don't circle the hypothetical center then we might be able to narrow down our error. When adding code to tell us where in the algorithm we are, we found that we were not running the code for the specified number of attempts before assuming there was no cylinder. Further analysis shows that despite finding the indices for points that would make a cylinder our code was saying that there was no cylinder

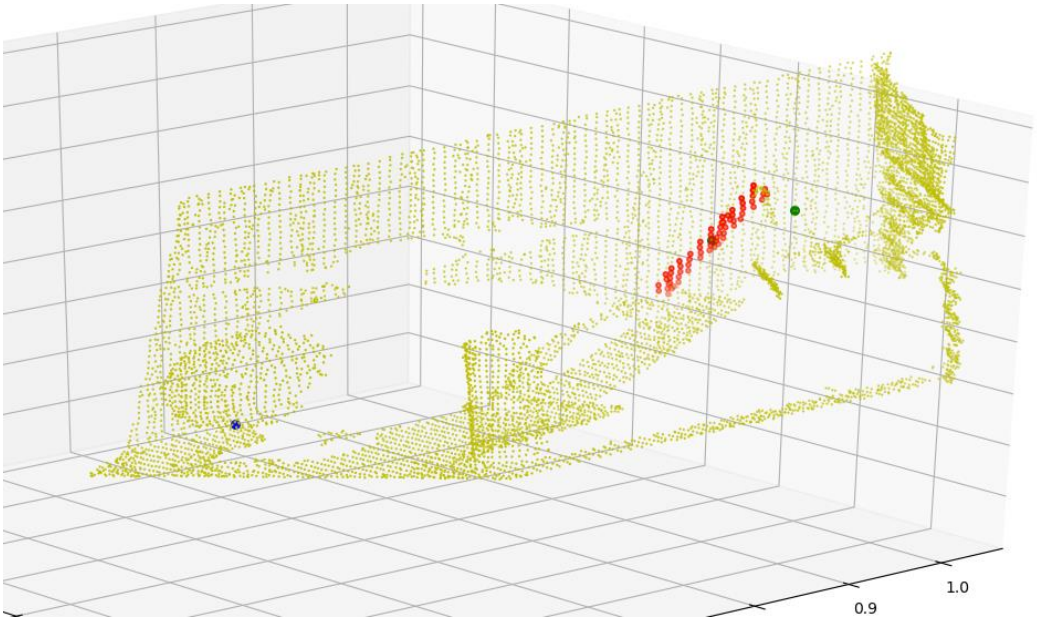


Where the top array is the cylinder indices, and 320 is the number of viable points found. The error was actually caused by our main script. When checking for a cylinder value we used cylinder.all()==0. Changing to np.mean(cylinder)==0 fixed the error. This fixed the error of prematurely cancelling our search for a viable object. Currently we do have circular output.

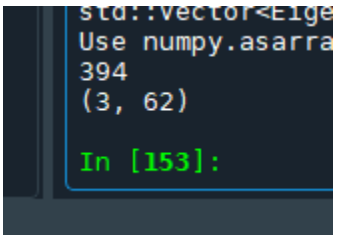


But the output is a filled in circle instead of a ring. We will attempt to fix this issue, which will lower the usual number of inliers found and possible improve the algoritm.

Though this isn't always the case, as seen below, where the output is more of a line.



This is with the inlier number set to 400. Which this solution doesn't seem to find. Checking the shape of our 'viable cylinder' this is correct. With 394 found inliers but only 62 points.

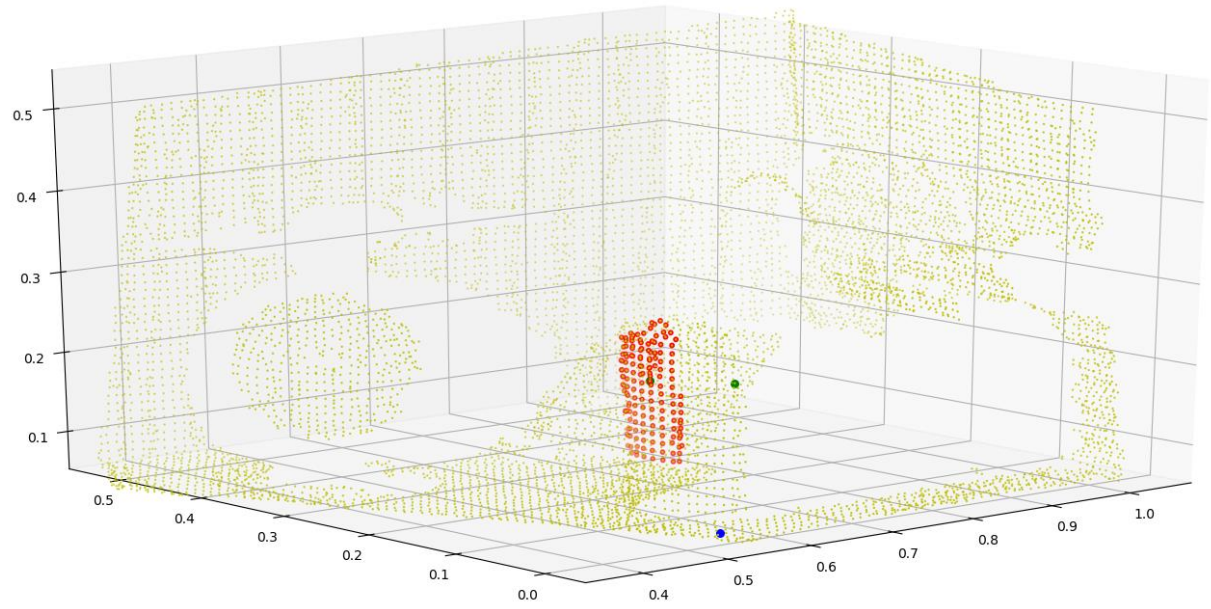


In reality. This discrepancy is because we resused the index variable to turn the actual indices for the points in our cylinder_index array into integers. This reassignment resulted in the change in length.

A new error seen is when trying to set the cylinder radius we are looking for. This results in the function saying, 'too many values to unpack (expected 4)'. To root cause this we are working through the ransac function to see where it get's stuck, since we don't get a line that can be causing the error.

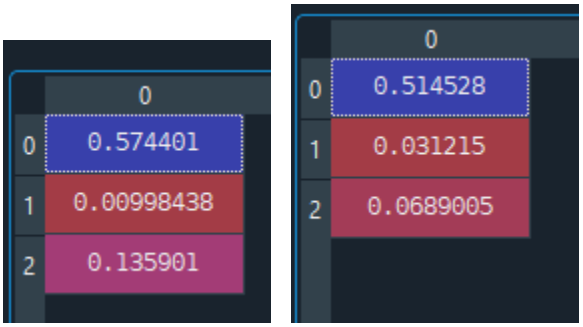
Outputing numbers for each loop it seems that the error is from not being able to find a cylinder in the given number of attempts. So the error is caused by our failure condition.

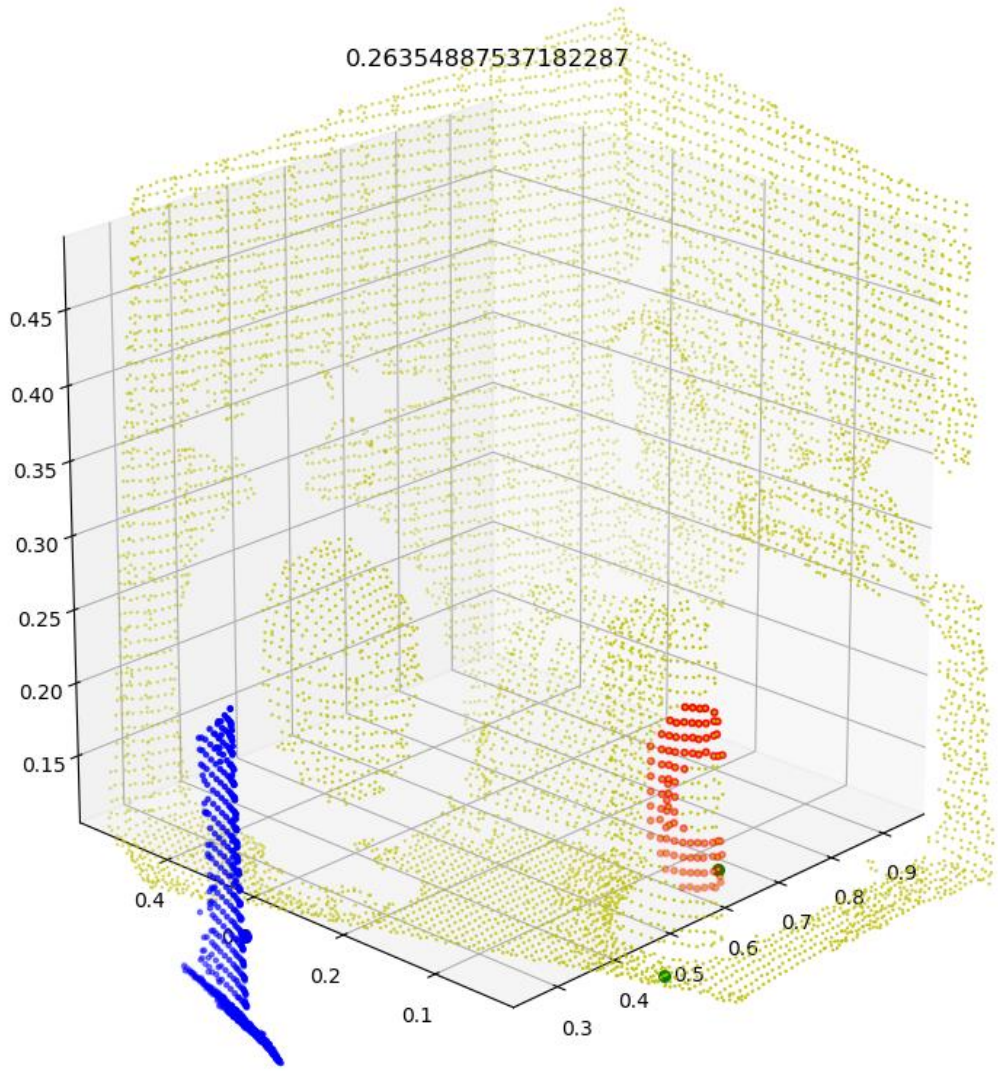
However, using our dynamic radius setting we did manage to detect the cylinder. This suggests that our math works out as long as the correct points are sampled, or the correct parameters are set.



To only accept realistic cylinders as viable geometries we set a cap on the calculated cylinder radius to 0.2. This results in significant run time and often not finding any solution at all. We must find a way to optimize our search for points. We'll try to increase the down sampling.

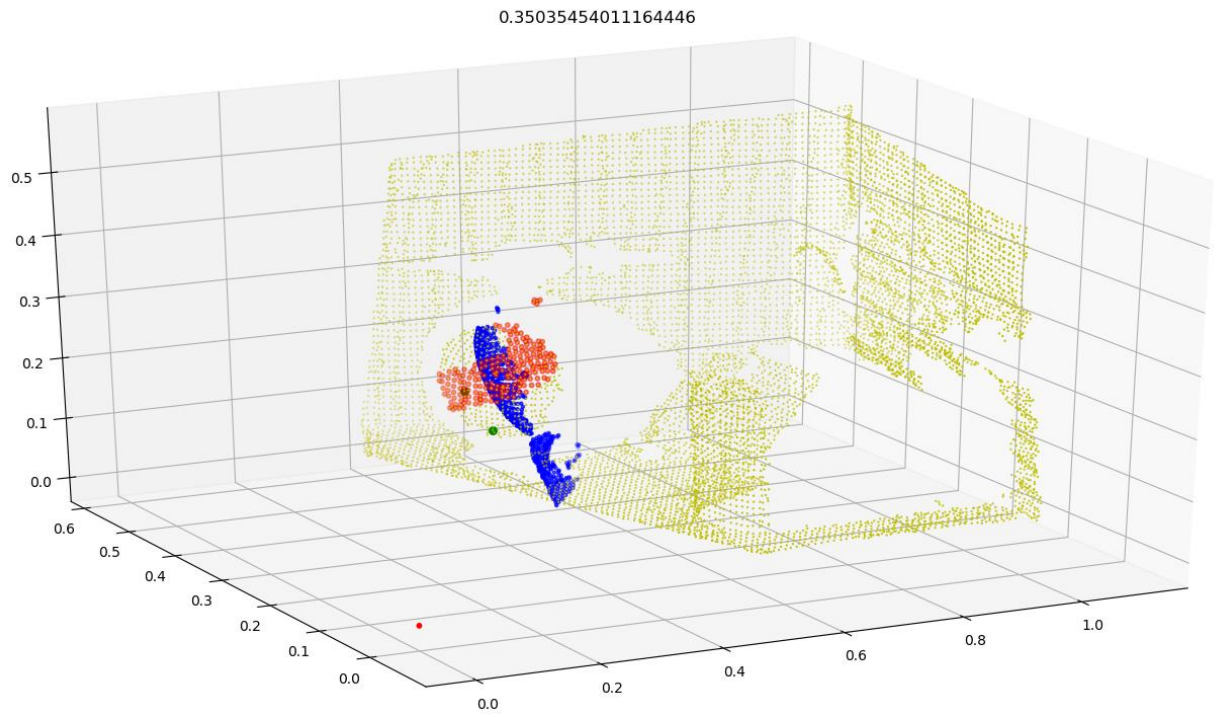
Running the algorithm many times we found a viable point for the cylinder. We will try to force a sample to pick this and another to compare our calculations of a cylinder vs the actual cylinder dimensions



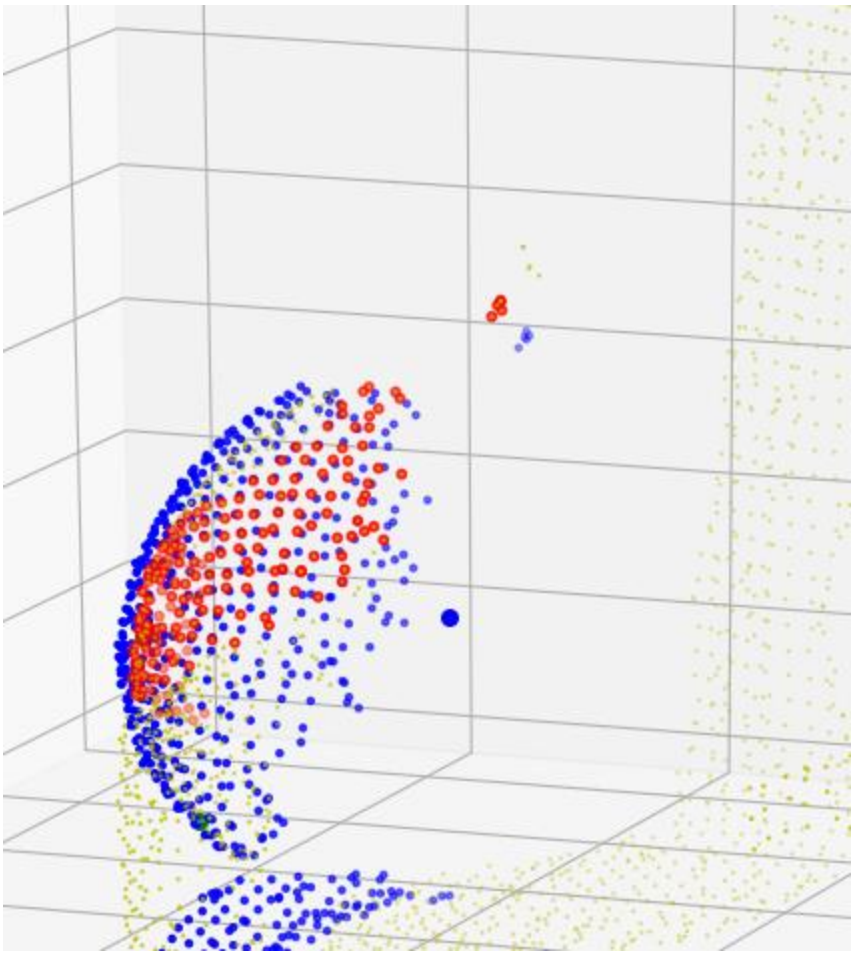


However, we needed to find the index of the successful sampled points to properly reuse it with the correct calculated surface normal. We wrote the finding_index script to do this. However, I script was having trouble finding the sampled point. We then realized that it was because the for loop we were using only looped 3 times instead of the whole down sampled point cloud because we were using the length of the cloud array to set the number of loops. Transposing the array fixed the issue and we found that our desired point had index 6287. However, when using this point our ransac algorithm was stuck in an infinite loop and wouldn't find any viable candidates, even with a drastically lower inlier threshold. Attempting the other point found we got an index of 3431. This also resulted in no viable candidates being found. However, using these points, our index searching script, and the 3D plotting tools we already developed we can iteratively plot points to determine if they are in an optimal position for cylinder detection.

Quickly running the algorithm again we get a detection of a cross section of the sphere.

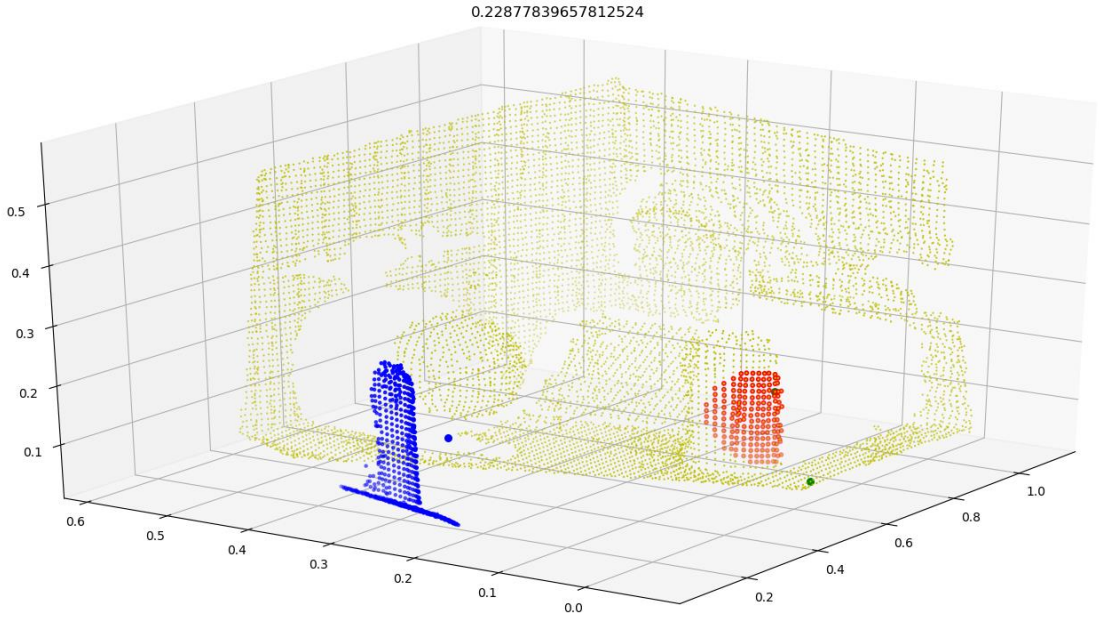


The red part is our calculated points, and the flat blue is a projection of the points found on the orthogonal axis. The title is the calculated cylinder radius. While this value doesn't seem correct, we should also notice the distance of the projection from the calculated center point.



Where the large blue dot is our center. This plot shows that our distance calculations and allowable offset from the radius is not working correctly.

Immediately after we once again detected the cylinder, thought the axis was off due to the second sample point not being on the cylinder.



The first and second sampled points can be seen below. Once again, we'll try finding the cylinder point index and using that to force a detection.

The index in our down sampled point cloud of the first point is 5685. However once again this is stuck finding a second sample point that is viable. We'll find the index of the second point above and force a repeated detection. The index of this second point is 6993. Forcing our sample points in this way then resulted in the error shown below.

```
  File "C:\Users\Jack\OneDrive - Northeastern University\Jack
\Document's\Spring 2020\Robotics\Project\main.py", line 67, in
<module>
    [cylinder, first, second, center, radius, x_plane] =
ransac(down_pcd, normals_pcd, static_radius=0)

  File "C:\Users\Jack\OneDrive - Northeastern University\Jack
\Document's\Spring 2020\Robotics\Project\ransac.py", line 152,
in ransac
    a = np.concatenate((-sample_normal,second_normal), axis=1)

  File "<__array_function__ internals>", line 6, in
concatenate

AxisError: axis 1 is out of bounds for array of dimension 1


In [24]:
```

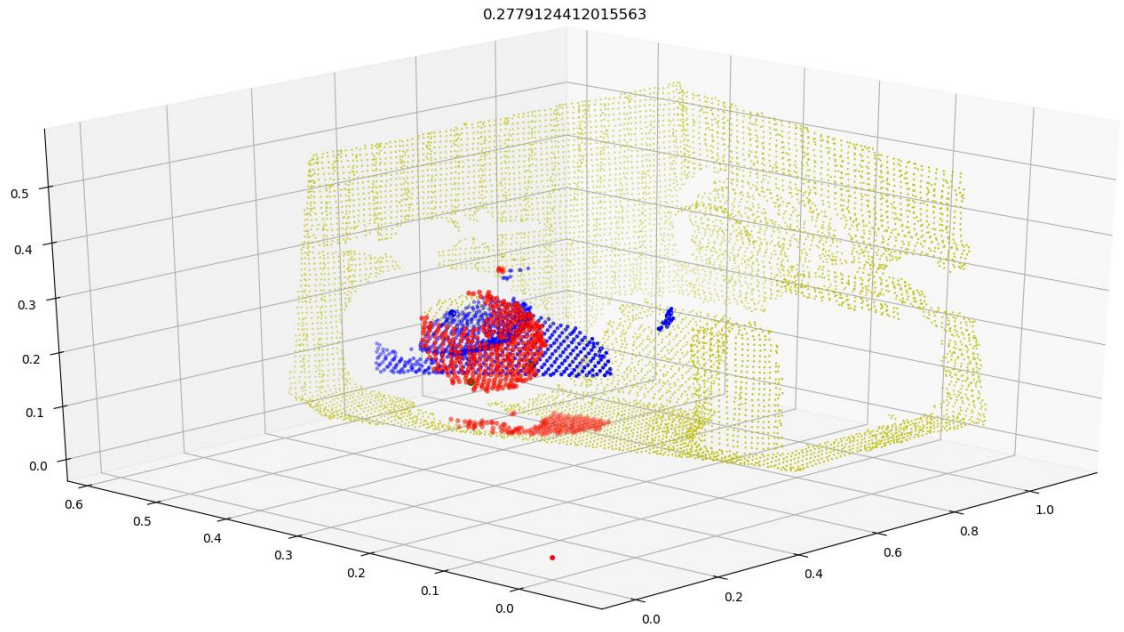
We found that in forcing an index value the output surface normal vectors were transposed. However, transposing these values on line 152 did not fix the error. This is because we cannot transpose 1D arrays in python like in matlab, so we must add an axis through the numpy.newaxis method.

To not have to fix all matrix dimensions in the program we reverted it to the original random points method. However, the ransac function now gives the below error.

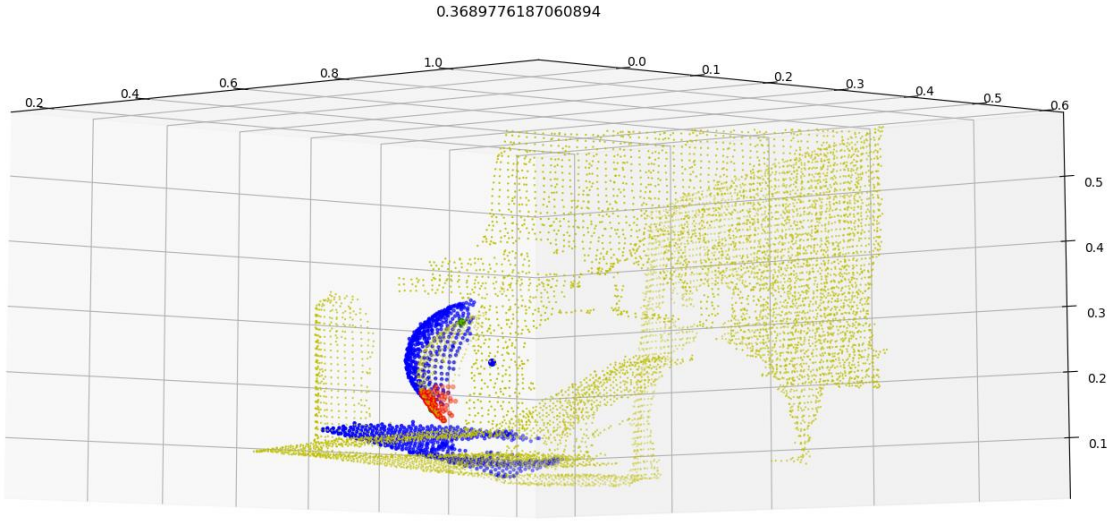
```
    [cylinder, first, second, center, radius, x_plane] =
ransac(down_pcd, normals_pcd, static_radius=0)

  File "C:\Users\Jack\OneDrive - Northeastern University\Jack
\Document's\Spring 2020\Robotics\Project\ransac.py", line 153,
in ransac
    x = la.lstsq(a,b, rcond=None)

  File "<__array_function__ internals>", line 6, in lstsq

  File "C:\Users\Jack\Anaconda3\lib\site-packages\numpy\linalg
\linalg.py", line 2228, in lstsq
    raise LinAlgError('Incompatible dimensions')

LinAlgError: Incompatible dimensions
```

This error was caused by previously transposing the surface normal in our ‘b’ matrix after we had the out of bounds error above.

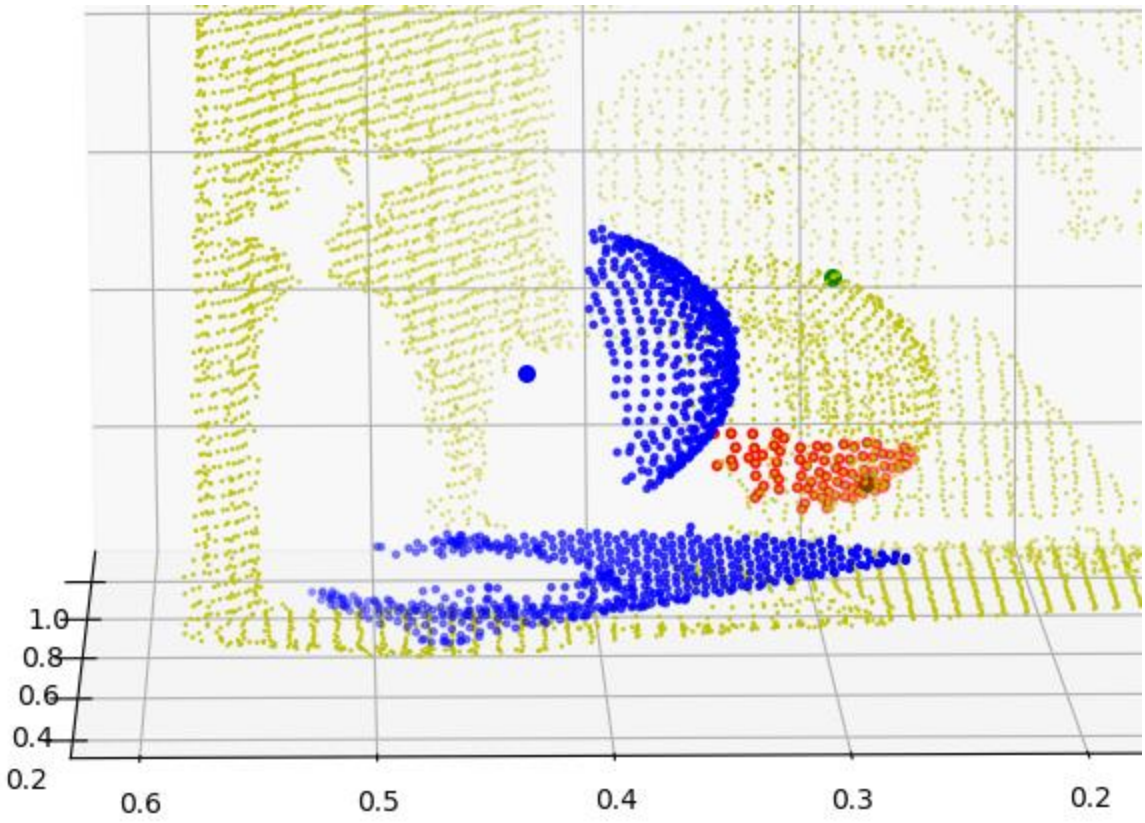
The inconsistency in our functions is remarkable, and we must determine if the error is coming from our algorithm’s calculations, or not strict enough parameters.



However, two results in a row detected our sphere, with the second showing a correct trimming of values, where the blue is all points in a local region about the sampled point projected on the orthogonal plane, while the red are the actual found suitable points.



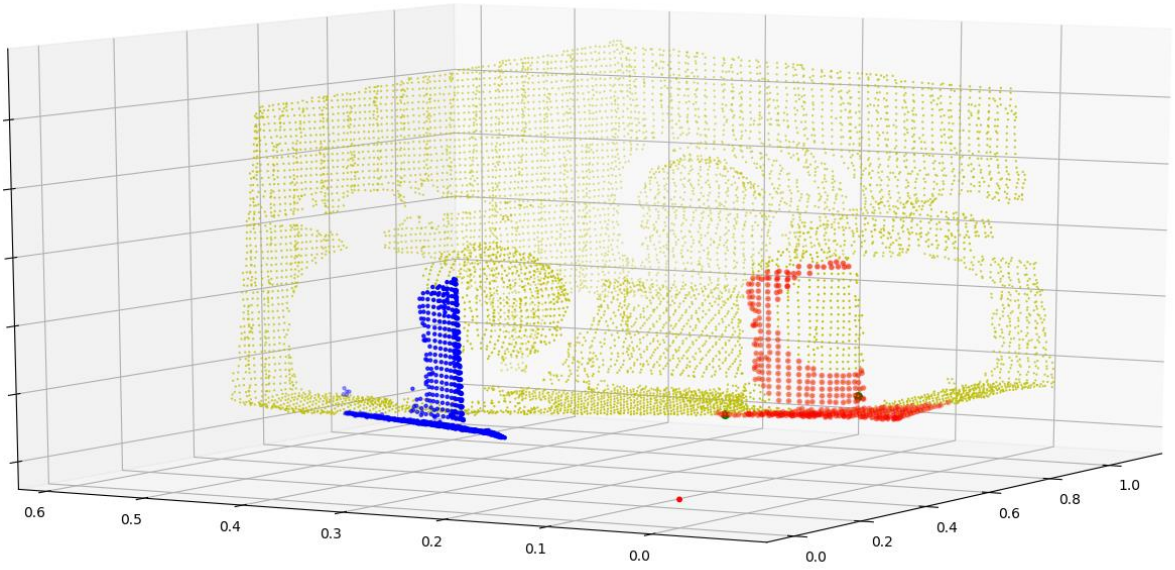
After further analysis however, it seems that these sphere values are not the correct distance about the center in the orthogonal plane, where the distance from the center varies much greater than that specified in the parameters file



Actually, this is the correct behavior. The blue points are not the valid candidates, but all potential ones in the local region.

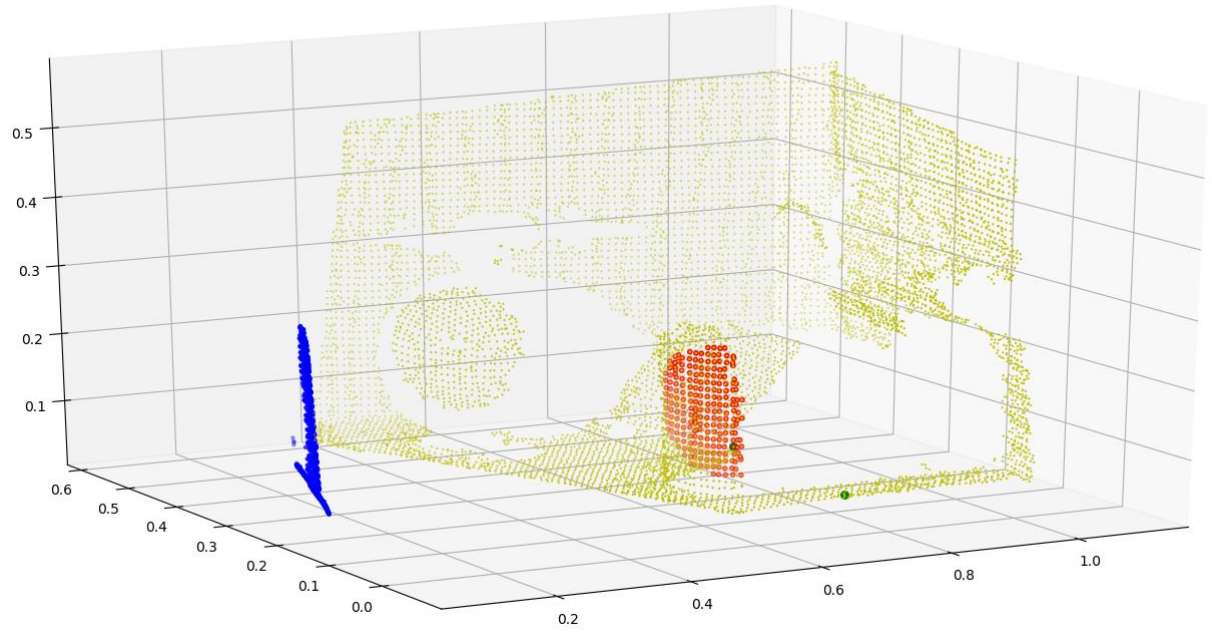
Once again we have technically detected the cylinder. However, the orthogonal plane is perpendicular to what is expected, and therefore this detection, and possibly previous ones, were mostly due to luck.

0.366180447138184 65



Though the past couple of runs of the code have shown siginificanlty more frequent detection of the curved surfaces, once again finding the cylinder and sphere. The sphere is more often the correct orthogonal plane, while the cylinder is not.

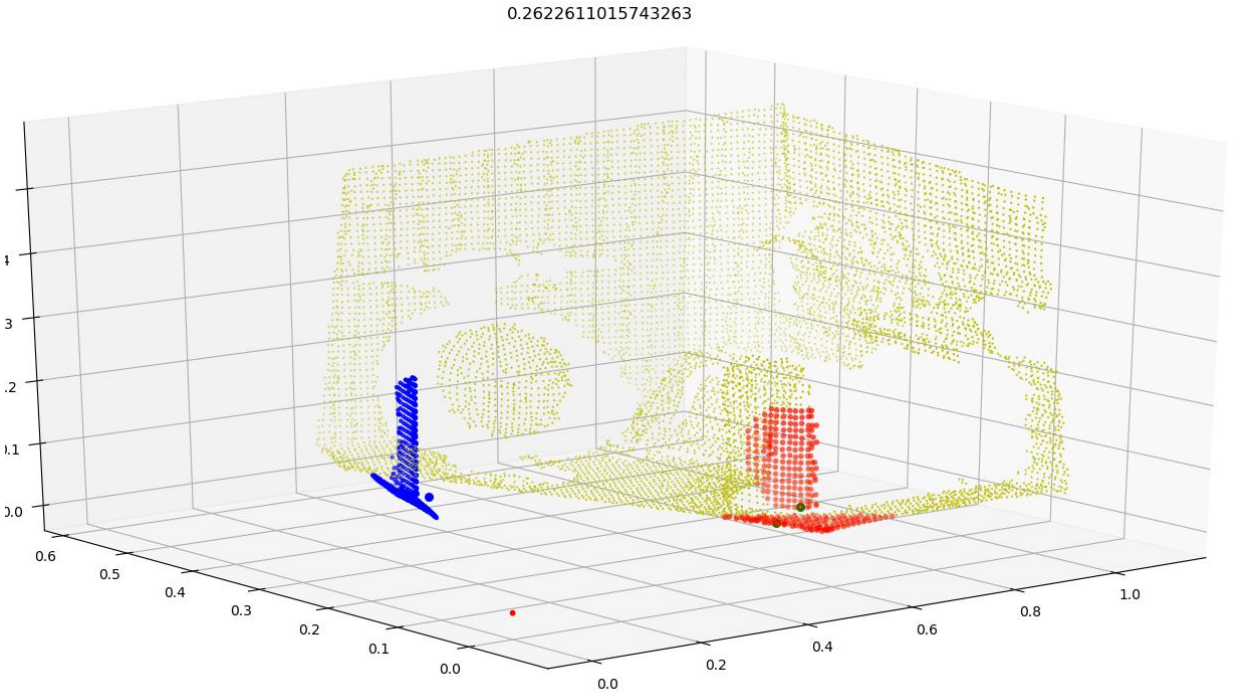
0.2662582317789745



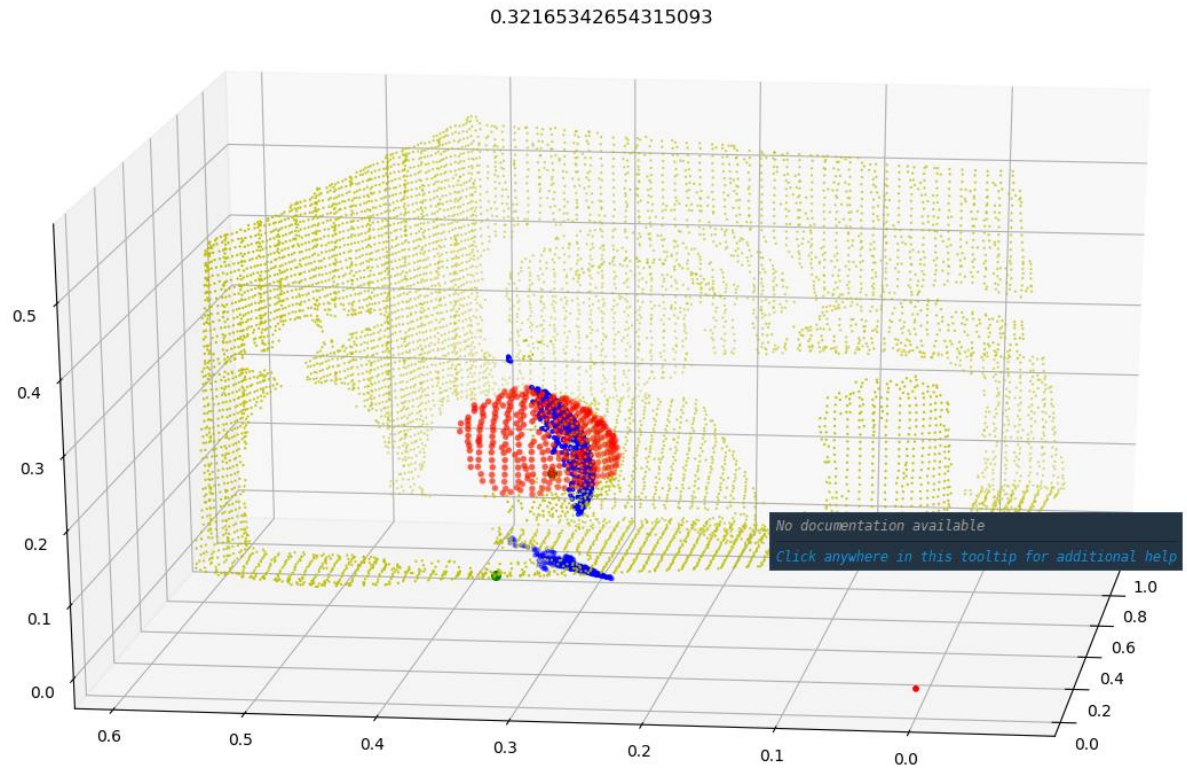
It should be noted that the two sample points often appear to be significantly farther apart than they should be as defined by the local_group setting in parameters. The first thought was that the two point

location matrices to calculate this distance did not have the right orientation, resulting in a 3x3 instead of 3x1 or 1x3 matrix. This is incorrect and there was no issue with the matrix arithmetic.

Our functions seem to be detecting the cylinder and sphere much more frequently with lower inlier's required, however this cylinder is still a result of luck given that the projected plane is not correct



Detection of the sphere is also partially luck, where below the second sampled point is not on the sphere but the calculated surface normal resulted in a well placed central axis.



The current parameters very often detected the curves of the sphere or cylinder. The specifications are listed below:

voxel_size = 0.01 # Size of voxel boxes used for down sampling

radius = 0.05 # Radius to search for surface normals

max_nn= 30 # Number of nearest neighbors to consider for surface normal calculations

""" Parameters for cylinder detection """

cylinder_radius = 0.09 # Commented if the cylinder radius is defined from the sampled point to center axis

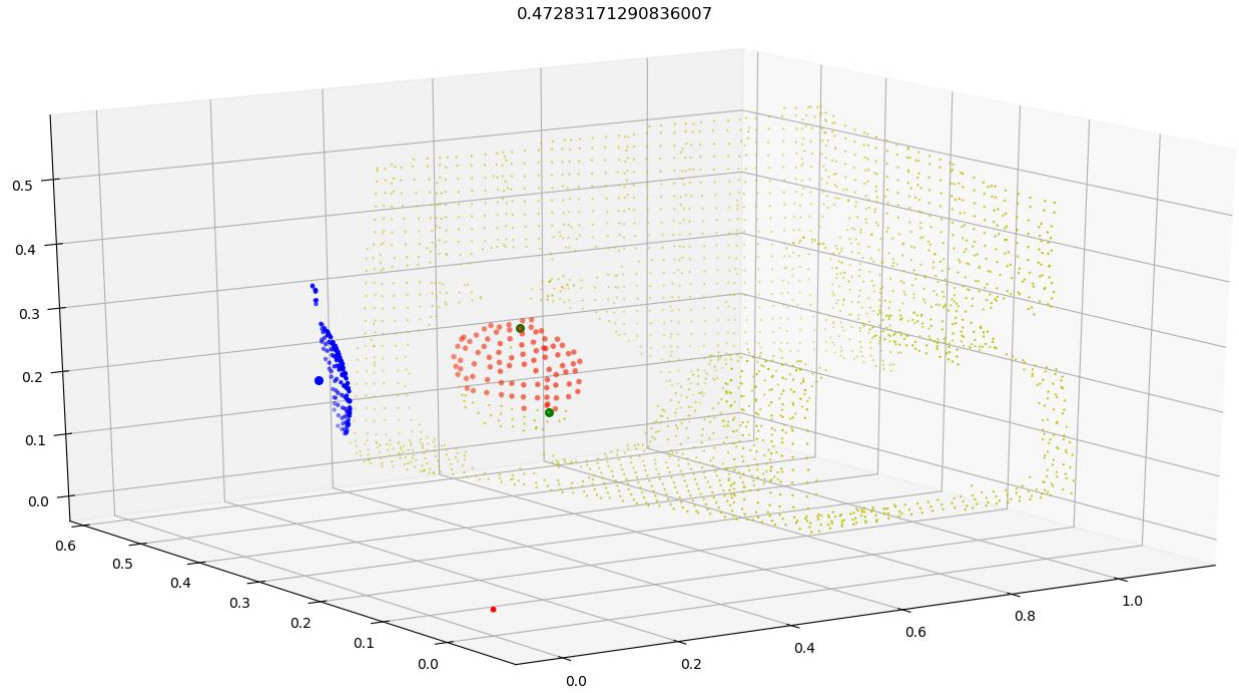
inlier = 30 # inlier threshhold

num_rounds = 300 # Number of rounds tested

delta = 0.03 # Noise allowed about cylinder radius

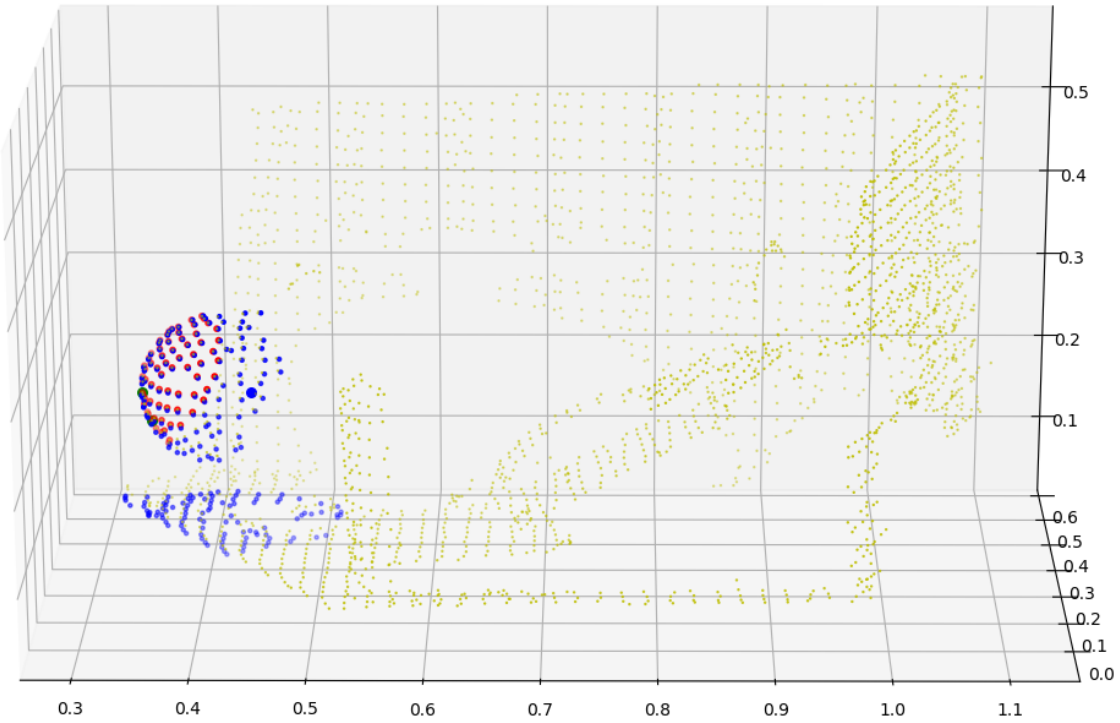
local_zone = 0.2

Increasing our downsampling by doubling the voxel size drastically increased how quickly our algorithm would run. Locating a sphere resulted in:

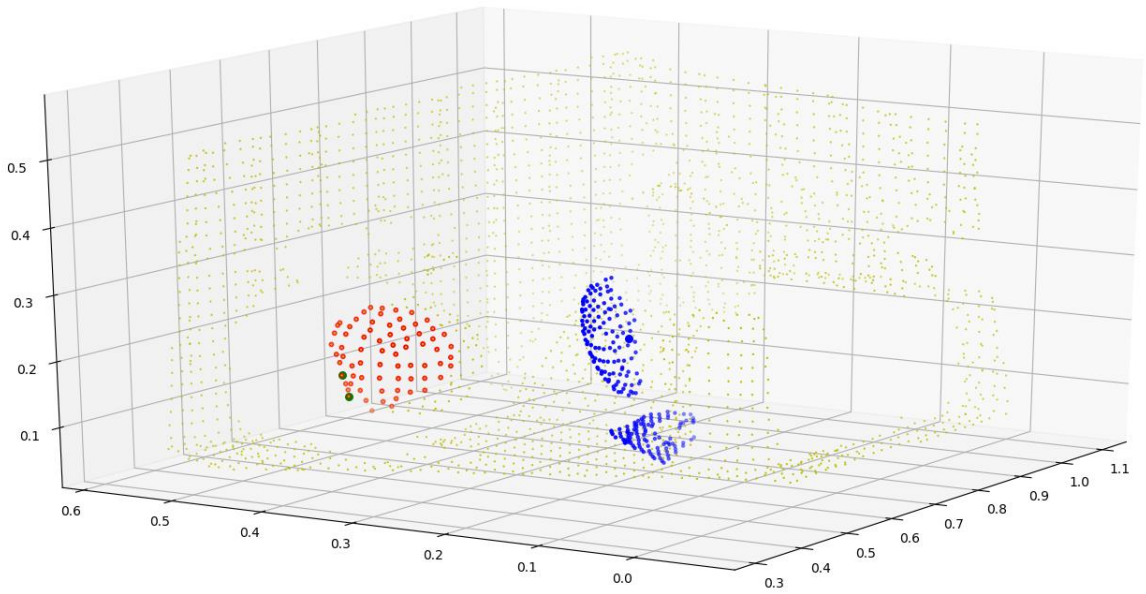


Overlapping the found viable points with the projected plane we see that the radius is not always correct between these points and ‘center’.

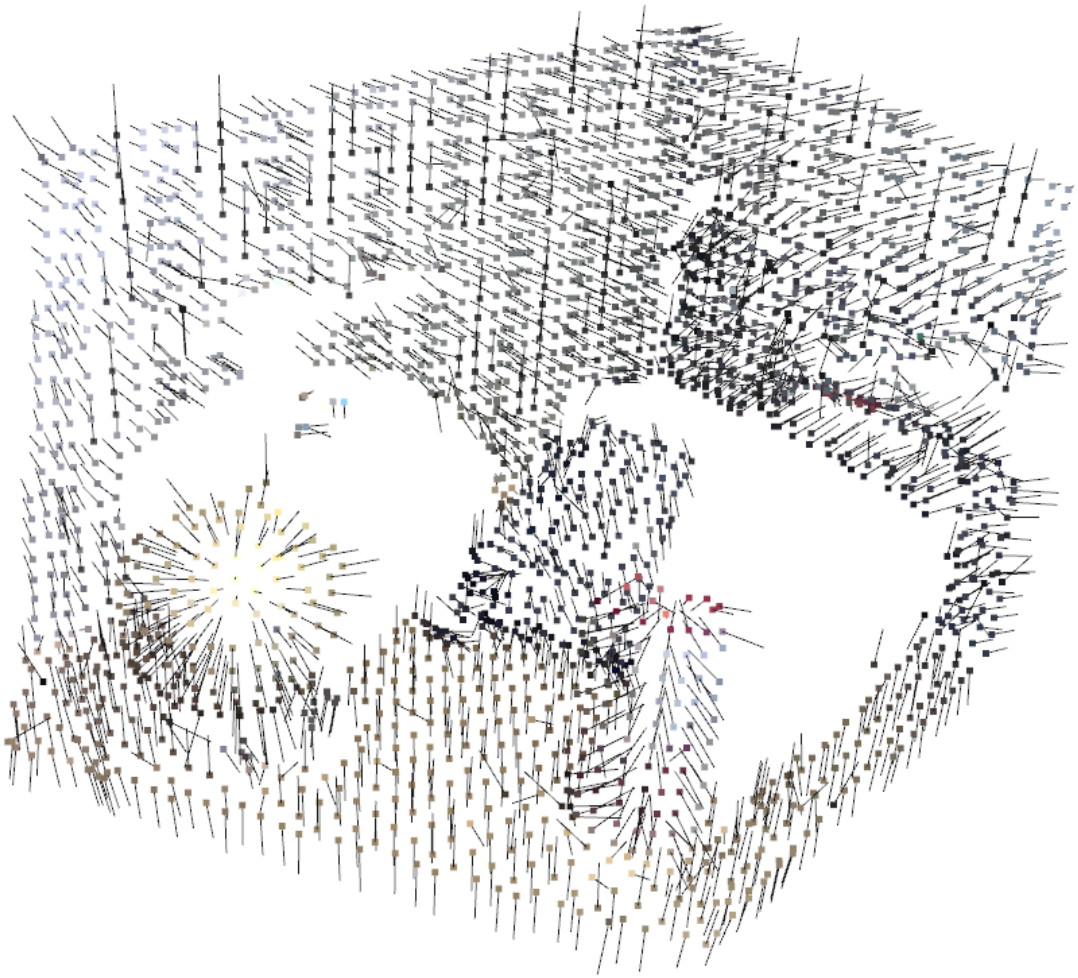
0.5597980396402837



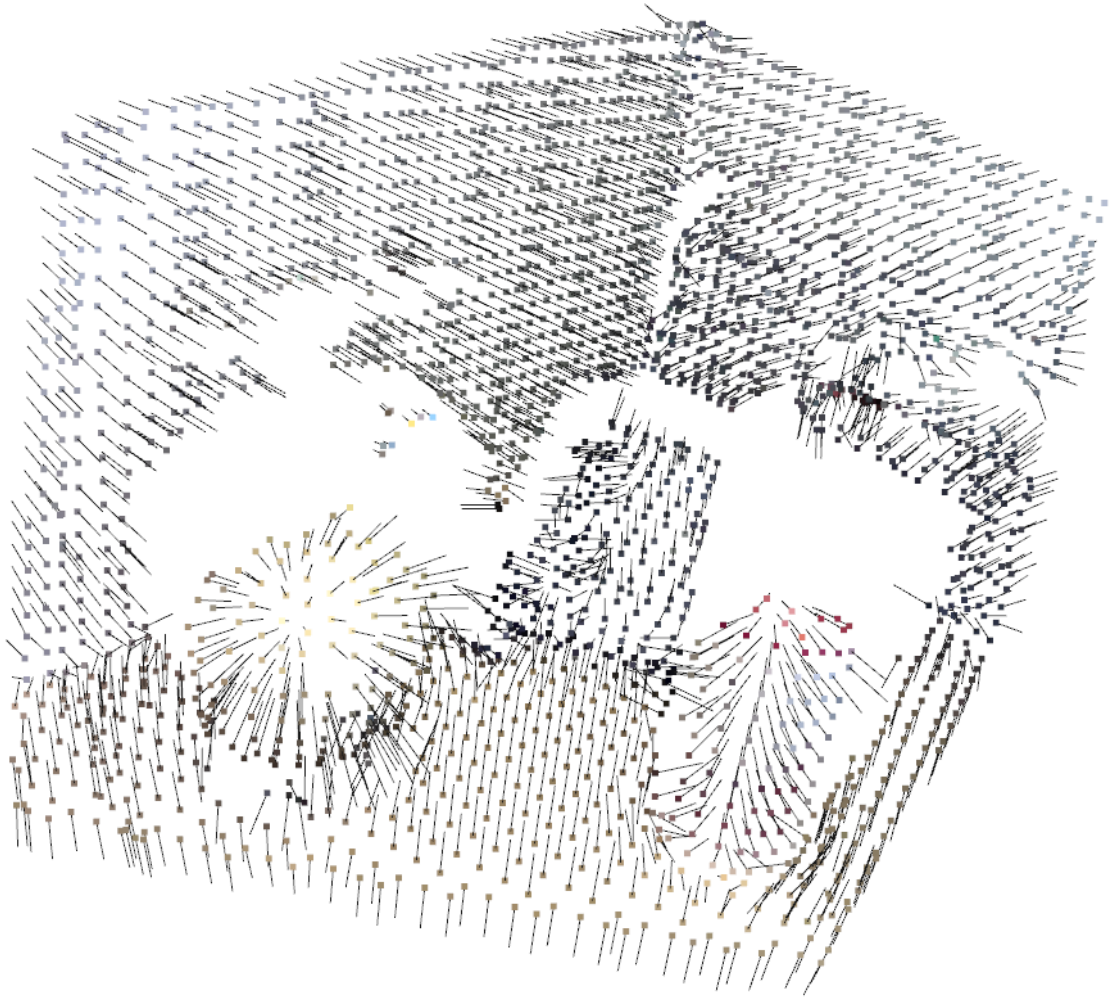
0.5597980396402837



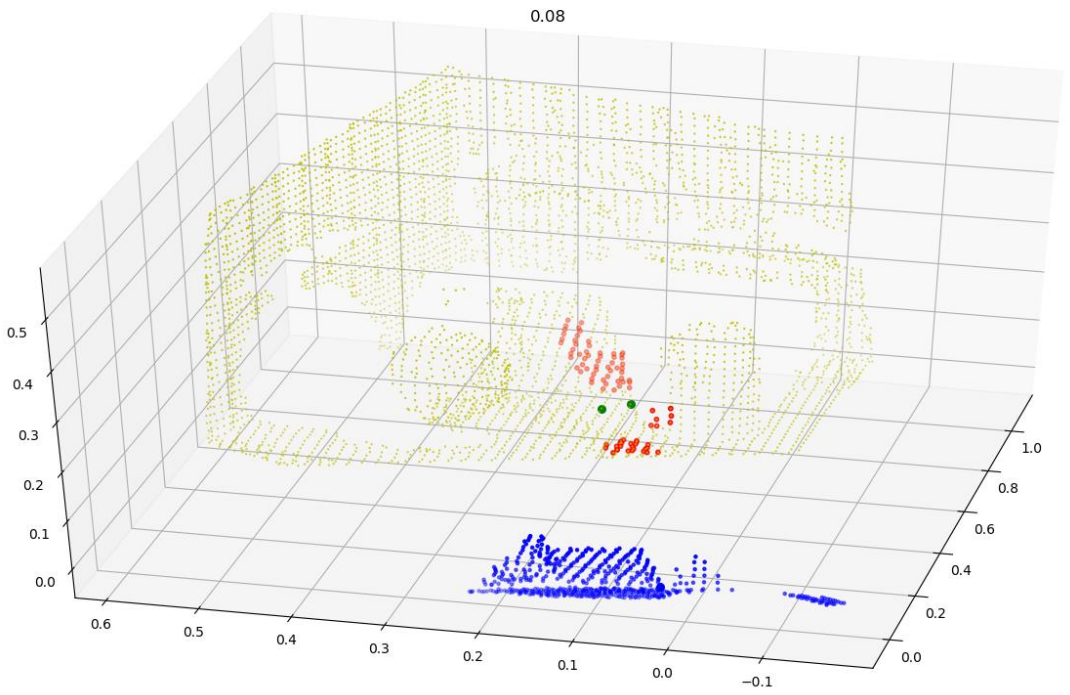
When looking at the surface normals of our downsample we can see that the surface normals point in often the incorrect direction. We'll tweak our downsampling and surface normal calculations to see if this can be improved and if it will help improve the consistency of our algorithm.



Increasing the radius of points considered to 0.05 drastically improved our surface normal calculations.



When not dynamically setting the radius, it is still possible to run into these errors of non cylinders being found,



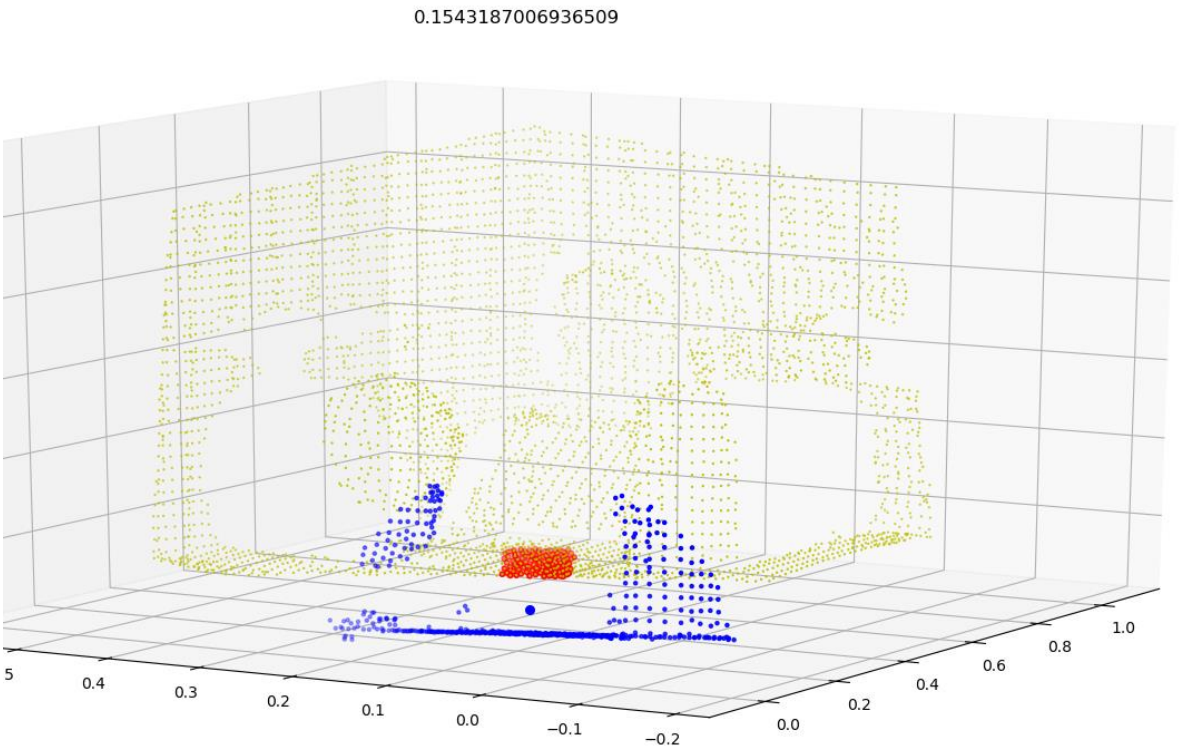
Running iteratively through searching for cylinders with radius' ranging from 0.05 to 0.1, we'll increase the required number of inliers.

Unfortunately tweaking the parameters was not very successful. It should be noted that for static radius's our algorithm fails to find the cylinder between 0.05 and 0.1, though we aren't sure if that's what the actual radius should be. Using the dynamic radius calculations, the large issue is that incorrect normals result in a cross section of all points in a zone being use and inconsistent and incorrect results. We'll try implementing an outlier check function from the open3d library to see if that can improve our surface normal behavior. Even then, if two points are taken from a corner, one of the wall and one on the floor, their theoretical axis would be parallel with the opposite wall, and all points in our radius range would be valid, along with all along the floor and wall. This makes it very difficult to exclude errors through just inlier specification, and this problem may not even be solved if we were using static cylinder radius searching.

Reading through outlier removal documentation it doesn't seem helpful for our use case for now. Reading through the documentation of KDTree is also interesting as it may be useful to implement for finding nearby points instead of using loops and linear algebra to calculate the distances between points. This implementation would be non-trivial since we've used numpy arrays throughout our code and the KDTree functions require point cloud objects. Improving efficiency is also a secondary goal after getting successful detection, which we haven't been able to achieve reliably.

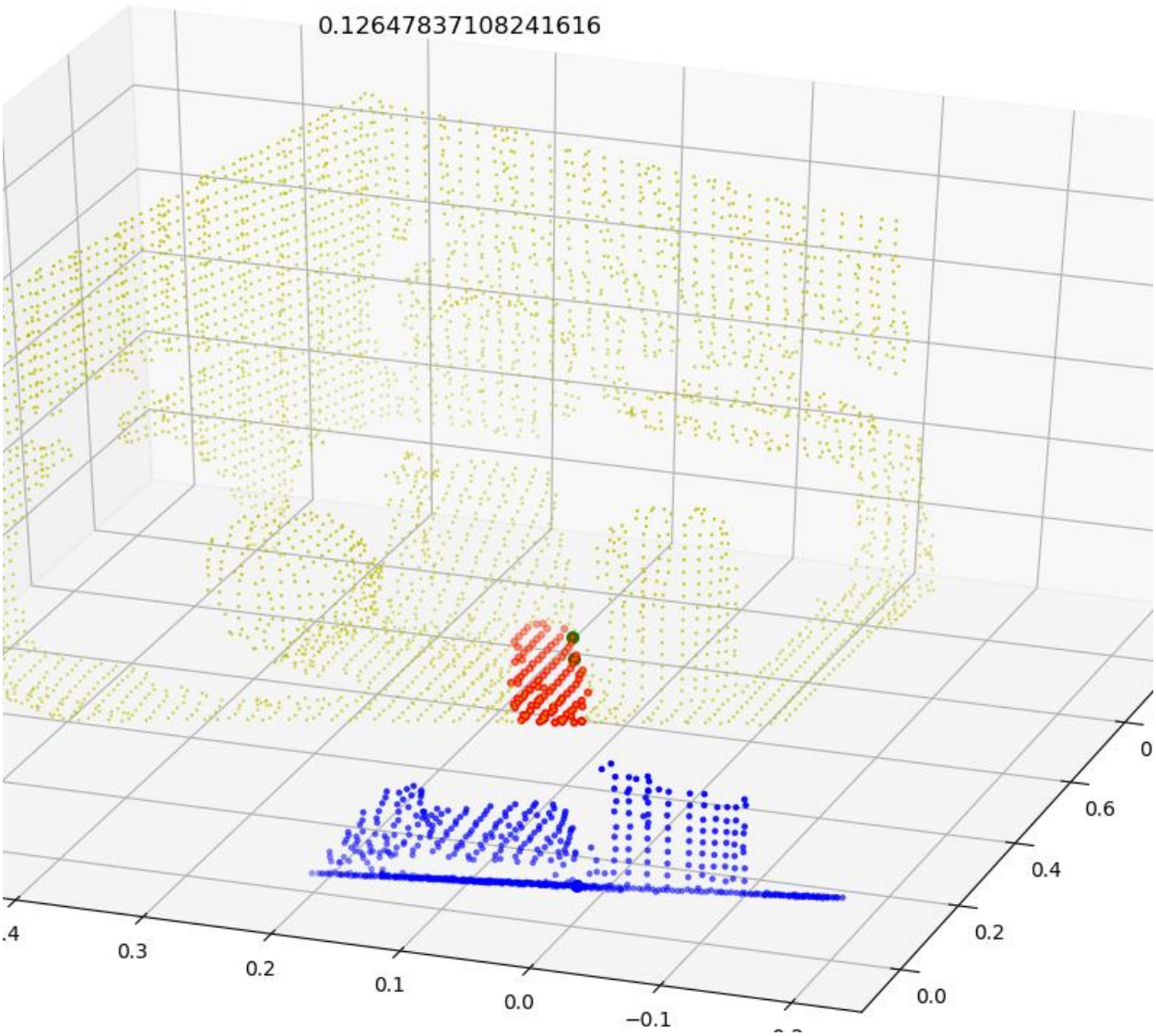
Some research online has led us to *programming computer vision* with python by Jan Solem. Skimming through this work though the relevant computer vision is mostly dealing with static images and not much information on the use of depth point clouds, which we are using for our simulation environment.

Running our code until a ‘cylinder’ is fainlly found, at 100 required outliers we find the plot below:



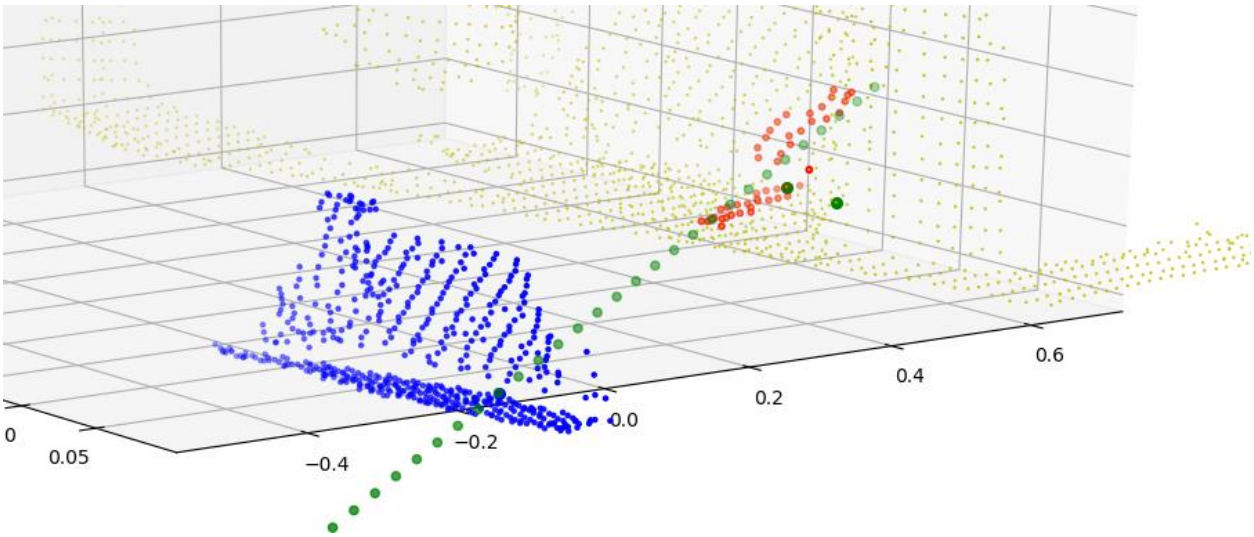
This highlights a central issue with our parameters and an issue overall wit the ransac algorithm. The number of inliers found are roughly the number found if we saw the true cylinder. With slightly off surface normals we can detect things that are not cylinders as being cylindrical. Potentially decreasing the delta (how far off a point can be from the cylinder radius) can help search for more curvature. This plot above was found with delta=0.01.

However, when reducing this number to 0.005, we actually receive the same viable points, after many runs.



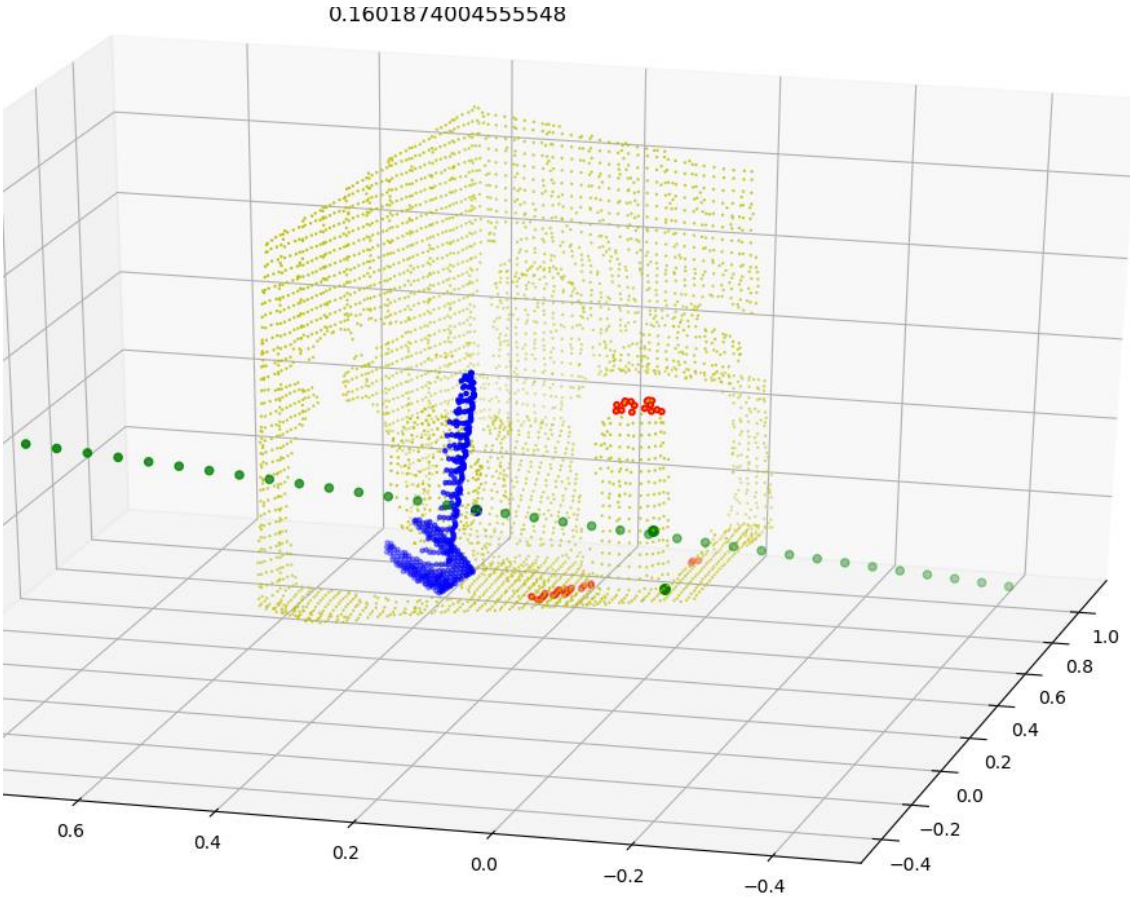
This suggests that we need to change something else about our parameters if after thousands of random samples we keep choosing this flat plane of points. However, in this case our 'viable' points when projected on the plane seem to form a line that the center point falls on. This suggests a fundamental error in our calculation of either the projected distance between the center and the points or of the center location.

To visualize how our axis relates to the points found we drew a line through our calculated center point using the unit normal of our 'axis'. The plot below highlights the error in our distance calculations for distance from a point to the axis.



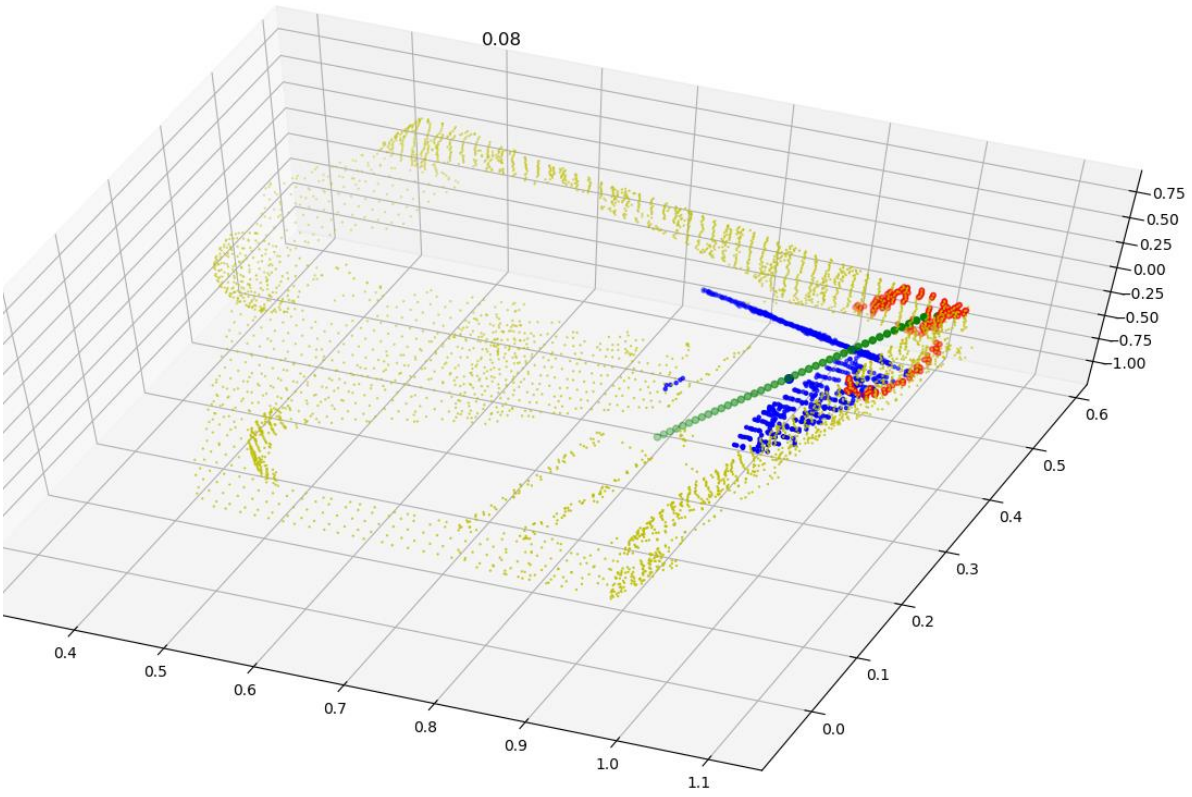
Where many red points lie closely to the axis running through the point cloud. This error was caused by a mismatch of matrix arithmetic when calculating the distance, where subtracting the point from the center resulted in a 3x3 matrix instead of a 3x1.

Fixing this error we had improved results, though still have issues with surface normal being incorrect and providing the wrong cylinder axis location.

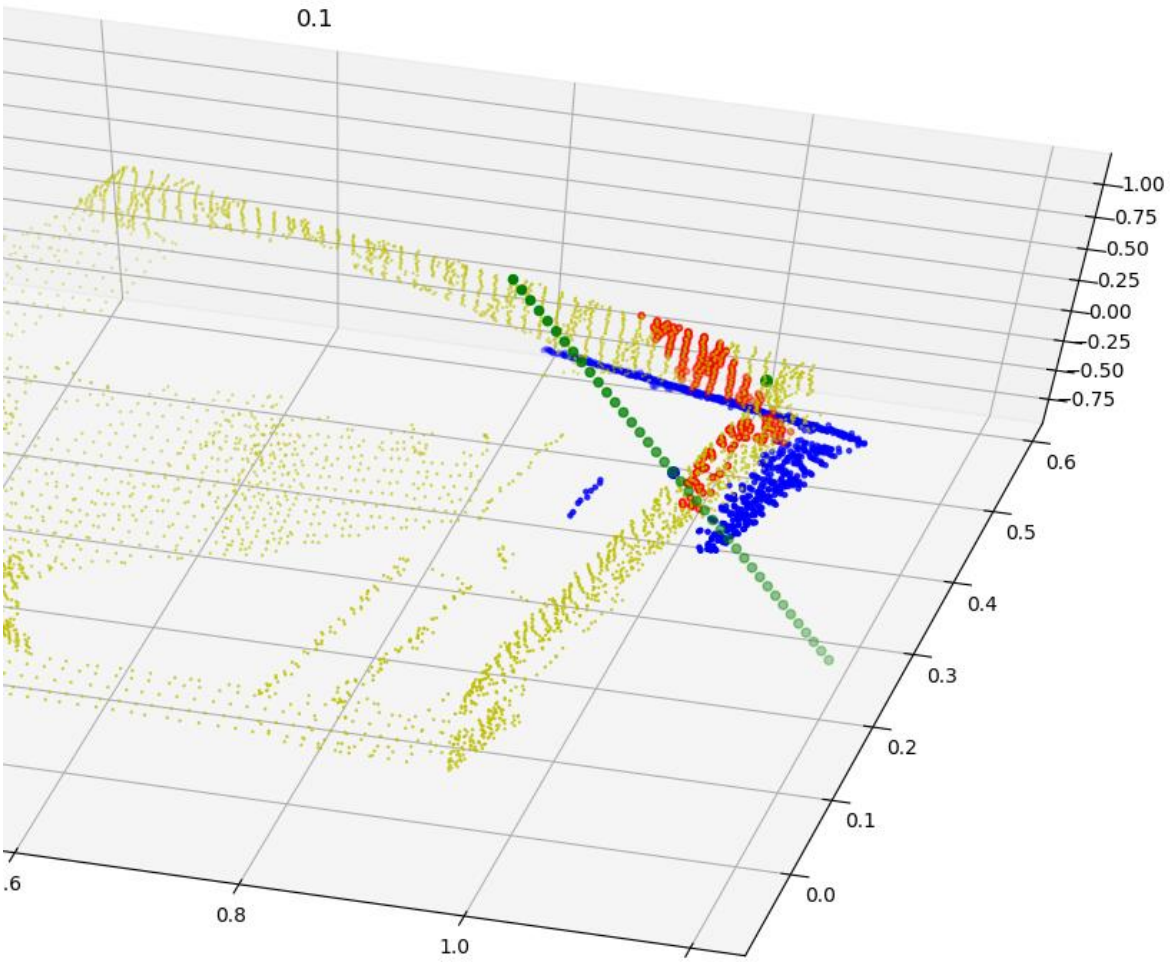


Fixing this error did remove our distance issues but did not improve our cylinder detection.

Now, using static radius for our candidates now produces ‘cylinders’ though they have the errors we were concerned about.



This corner was also detected for 0.09 and 0.1 radius sizes.



To force a cylinder detection we'll find the index of another point on our cylinder.

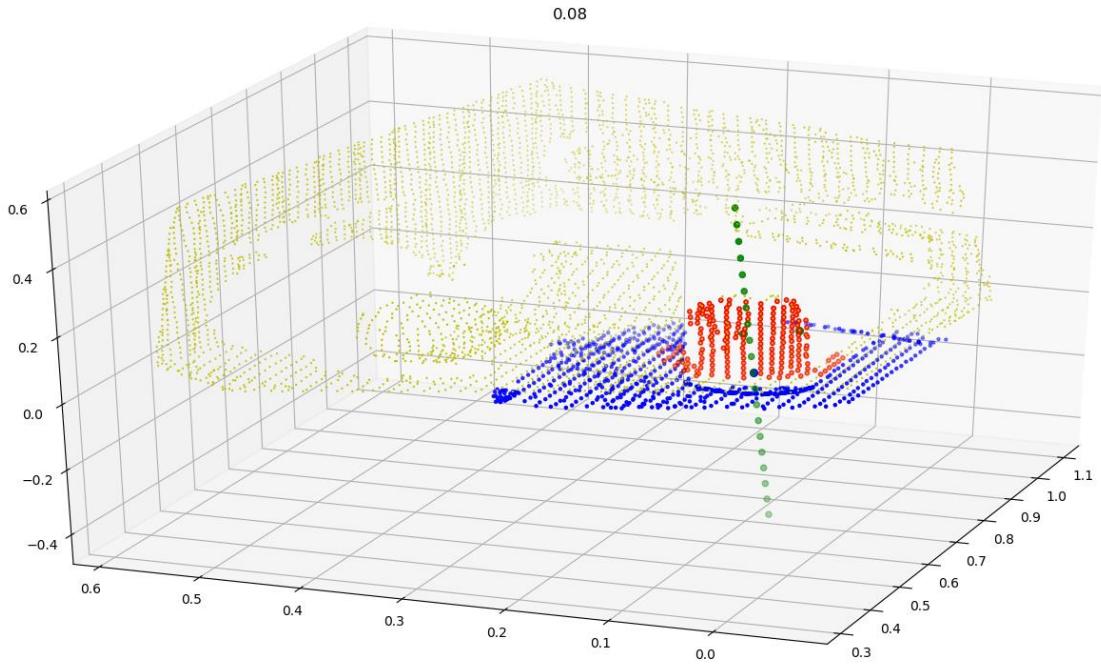
first - NumPy object array

	0
0	0.566255
1	0.0140978
2	0.133984

Which is at index 4030. Another point we found on the cylinder before has coordinates, [0.568777,0.0131362,0.211776]. We have since changed the down sampling so the previous found index is no longer valid and this point no longer exists. Using the finding index function we added a 0.005 offset to find the nearest similar point and found one at index 616.

However, when intentionally selecting these points no cylinder is found. Once removing these as our sample points we couldn't get the program to recognize any viable shapes. Plotting these points they are almost directly above each other in the cylinder. To try and find another point we iteratibvely moved one of them, and used our index finder to determine a point that was about where we desired. This was one with an index of 1390, And coordinates [0.530089, 0.0563328, 0.209143].

Forcing this cylinder detection our algorithm correctly found the cylinder, as expected. However it would only detect when using the static radius mode.



Using the dynamic mode, we found that we were calculating a radius of 1.345, which is incorrect. This suggests an error with our distance calculations.

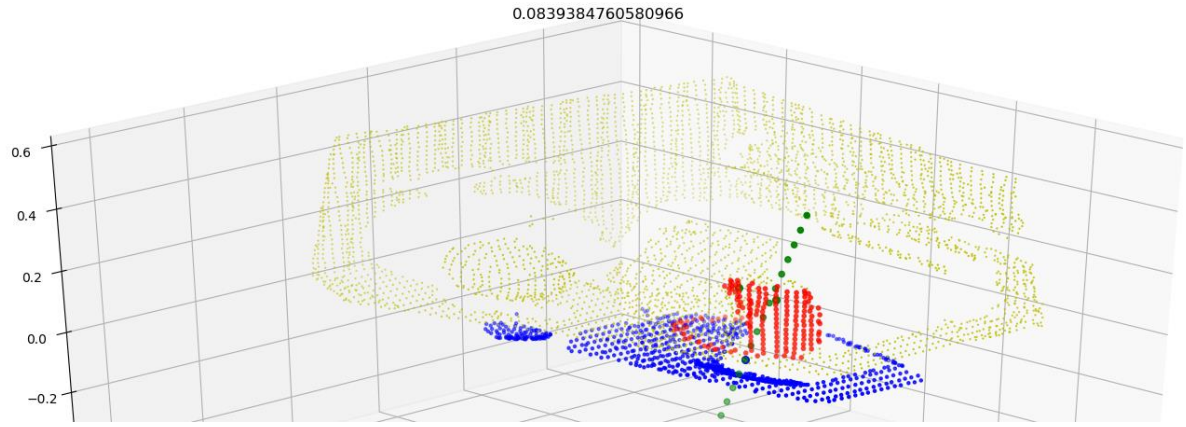
Below is the calculated center from the static cylinder detection.

center - NumPy object array

	0
0	0.623139
1	0.073261
2	0.0285591

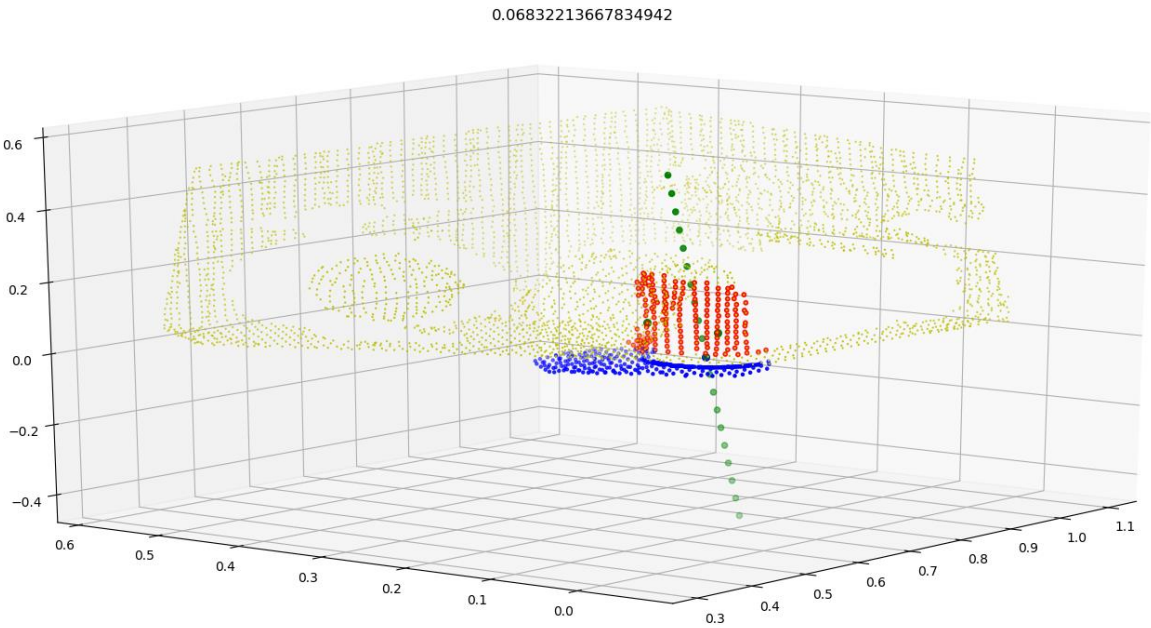
The calculated center for the dynamic cylinder detection was [0.62016005, 0.06925147, 0.02852059]. This shows a slight difference with our center calculation. We solved this issue by realizing that the offset value in our linear algebra was the cylinder radius. Replacing our projection calculation with this fixed the error.

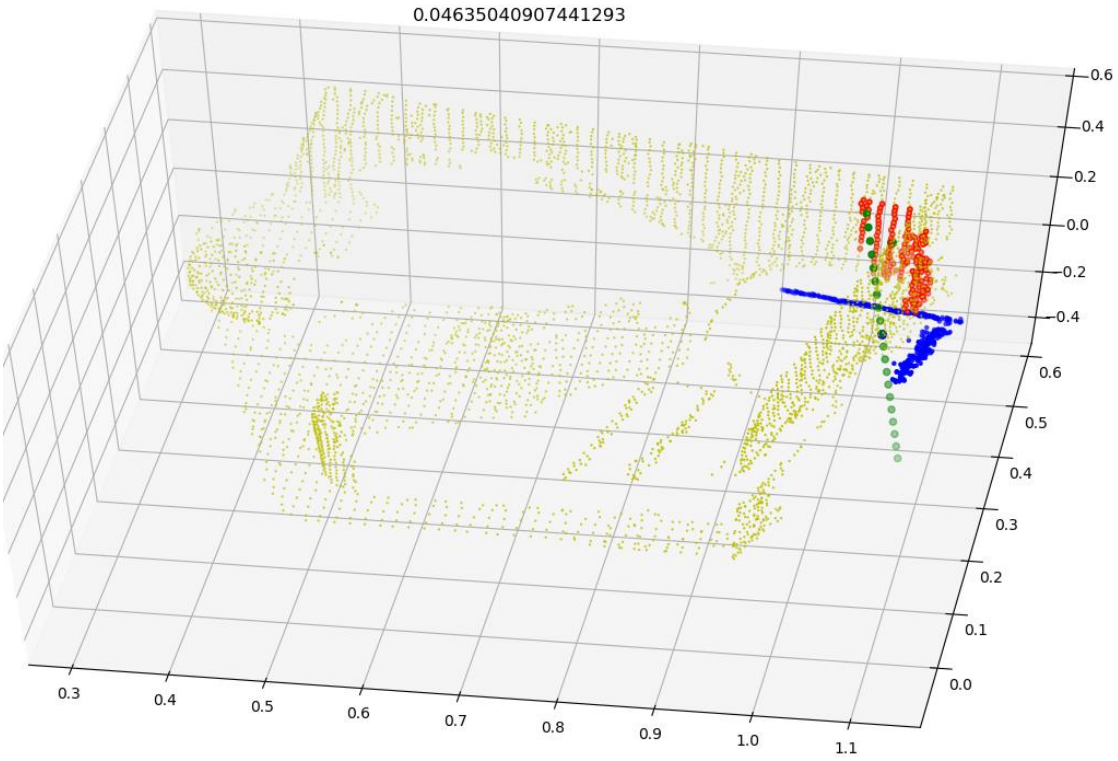
After fixing this we have found the cylinder, though still inconsistently.



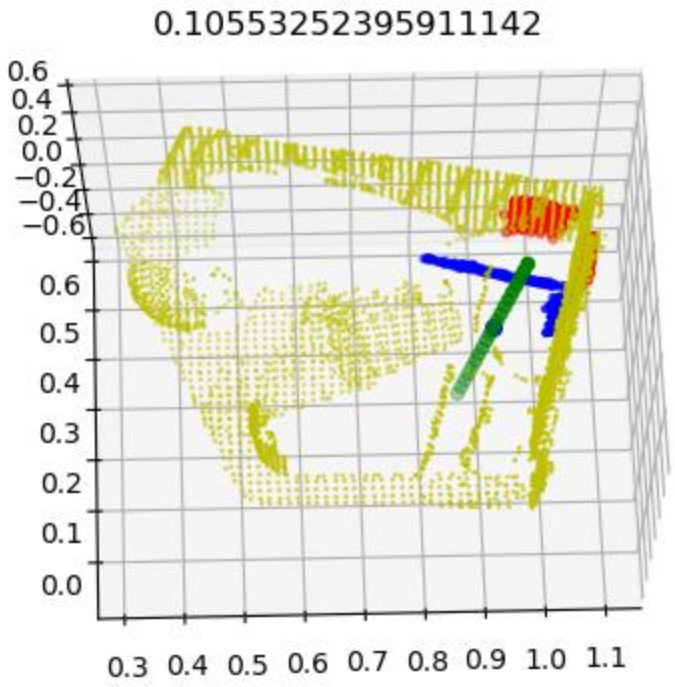
We will use our successful cylinder candidate to adjust parameters to try and filter out false positives.

Unfortunately, the threshold to detecting the cylinder still allows for other ‘objects’ to be detected. If time allowed we’d want to run a second recognition algorithm on the outputted shape and check to see that it really is cylindrical.





Above is an example of a viable candidate with our current settings. We were going to attempt to set a minimum cylinder radius to filter out the walls. However, below it can be seen that the wall is sometimes detected with slightly larger radius's.



With the current settings we often detect the cylinder.

""" Parameters to change depending on point cloud size """

voxel_size = 0.015 # Size of voxel boxes used for down sampling

radius = 0.1 # Radius to search for surface normals

max_nn= 30 # Number of nearest neighbors to consider for surface normal calculations

""" Parameters for cylinder detection """

cylinder_radius = 0.08 # Commented if the cylinder radius is defined from the sampled point to center axis

inlier = 175 # inlier threshold

num_rounds = 100 # Number of rounds tested

delta = 0.01 # Noise allowed about cylinder radius

local_zone = 0.14 # How far the second randomly sampled point can be from the first

If we had a second local method that verified the shape of the outputted point cloud we would likely have more success. It can not be understated how significant the radius length calculation error was, where many causes for mis-identification could be traced back to that bug.