hw3

Name: Ruffin White
ID: A53229660

February 23, 2018

Contents

HW3 : Math for Robotics

Author: Ruffin White
Course: CSE291
Date: Feb 23 2018

1.

Consider the following differential equation over the interval $(0, 1]$:

$$\frac{dy}{dx} = \frac{1}{x^2(1-y)}$$

with $y(1) = -1$

- Obtain an exact analytical solution to the equation. In the following solve for $y(0)$ even though in theory the equation is not defined for $x = 0$.
- Implement and use Eulers method to solve the differential equation numerically. Use a step size of 0.05. How accurate is your numerical solution?
- Implement and use a fourth-order Runge-Kutta method to solve the differential equation numerically. Again, use a step size of 0.05. Again, how accurate is your numerical solution?
- Implement and use a Richardson extrapolation to solve the equation, again with a step size of 0.05. How accuracy is your solution compared to the analytical solution?

We'll start by create from the intergrator base class to provide the simple bolerplate in running out OED solvers. See notebook for source code.

Next, well overload the basic OED framwork to utilize our own custom OED solver implementations.

Finally, we'll implement each method by providing the integration formulation called upon by solver base class. This way we can seperate the routine for the solver execution from the details for the ODE we'd like to solve for.

As a unit test for our implementations, we'll derive the value of Euler's number, e, the famous irrational mathematical constant approximately equal to 2.71828, by consider the following differential equation over the interval $[0, 1]$:

$$\frac{dy}{dx} = e^x$$

with $y(0) = 1$
Here we'll assumbe a $dt = 0.0005$.

Euler's method

```
t: 1.0004999999999453, y: [ 2.71896137]
```

Runge Kutta 4th order

```
t: 1.0004999999999453, y: [ 2.71964131]
Loops:  2001
```

Richardson Extrapolation

```
t: 1.0, y: [ 2.7176028]
Loops:  2042
```

And just for comparison with scipy's own explicit runge-kutta method of order (4)5 due to Dormand & Prince: #### dopri5

```
t: 1.0004999999999453, y: [ 2.71964131]
```

Now that we've checked our solvers are working, let us return to the original ODE:

$$\frac{dy}{dx} = \frac{1}{x^2(1-y)}$$

We rewrite this into a first order separable ODE that has the form of $N(y)dy = M(x)dx$, where $N(y) = y - 1$ amd $M(x) = \frac{1}{x^2}$:

$$(y-1) \cdot \frac{d}{dx} = -\frac{1}{x^2}$$

We solve for this using the property that $\int N(y)dy - \int M(x)dx$, up to some constant c.

$$\int y - 1dy = \int -\frac{1}{x^2}dx$$

For the right hand side we apply the power rule $\int x^a dx = \frac{x^{a+1}}{a+1}$ given $a \neq -1$

$$\int -\frac{1}{x^2}dx = -\int \frac{1}{x^2}dx = \frac{1}{x} + c_1$$

For the left hand side we apply the sum rule $\int f(x) \pm g(x)dx = \int f(x)dx \pm \int g(x)dx$

$$\int y - 1dy = \int ydy - \int 1dy = \frac{y^2}{2} - y + c_2$$

Combining the constants leaves us with:

$$\frac{y^2}{2} - y = \frac{1}{x} + c_1$$

Isolating for y results in two forms:

$$y = \frac{x - \sqrt{2c_1x^2 + x^2 + 2x}}{x}, \quad y = \frac{x + \sqrt{2c_1x^2 + x^2 + 2x}}{x}$$

For this ODE, we can see from the analyitcal solution that y becomes unbounded as $\lim_{x\to\infty}$. We can see this refleced when applying our solvers to the given ODE and initial conditions.

Euler's method

```
t: -4.999999989802416e-05, y: [ -1.67541946e+19]
```

We can see that with euller's method, we're still infitly far away from infity, bit we to converge quite quickly to the solution. If we know switch to using the Runge Kutta method:

Runge Kutta 4th order

```
t: -0.0004999999999453736, y: [ -2.08921240e+20]
```

We can see with the same sepsis we about an order of magnitude greater, and quickly approaching the numerical limit of our floating point computation. Yet, still quite far from the accutial solution.

2.

Consider a predator-prey dynamics such as the simple Lotka-Volterra model

$$x' = f(x)$$

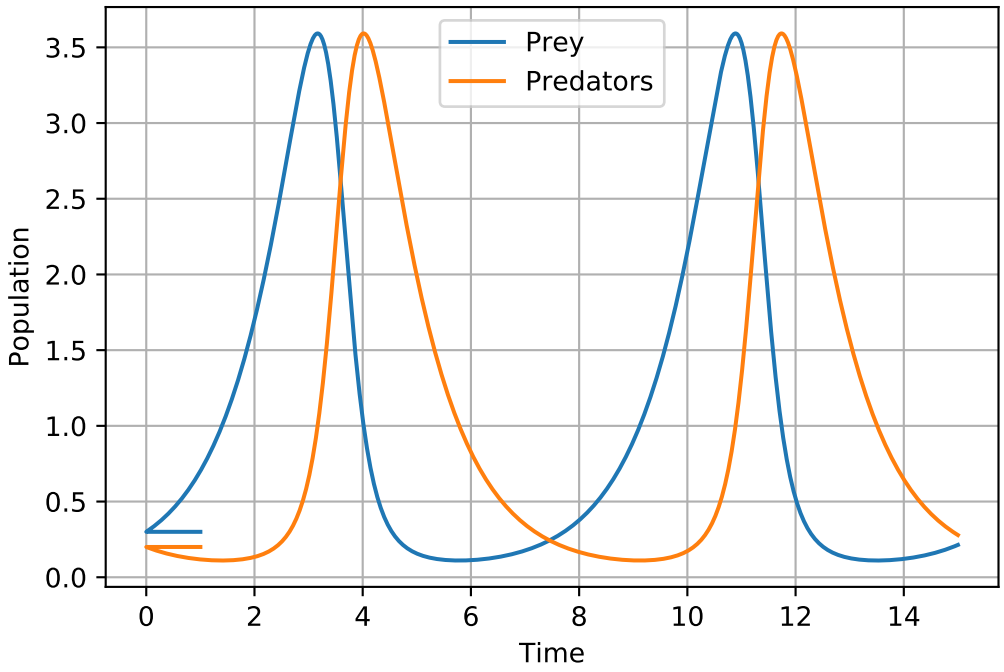
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} Preypolution \\ PredatorPolution \end{pmatrix}$$

$$f(x) = \begin{pmatrix} (b - px_2)x_1 \\ (rx_1 - d)x_2 \end{pmatrix}$$

Without predators, the prey population increases (exponentially) without bound, whereas without prey, the predator population diminishes (exponentially) to zero. The nonlinear interaction, with predators eating prey, tends to diminish the prey population and increase the predator population. Use your Runga-Kutta to solve this system, with the values $b = p = r = d = 1$, $x_1(0) = 0.3$, and $x_2(0) = 0.2$.

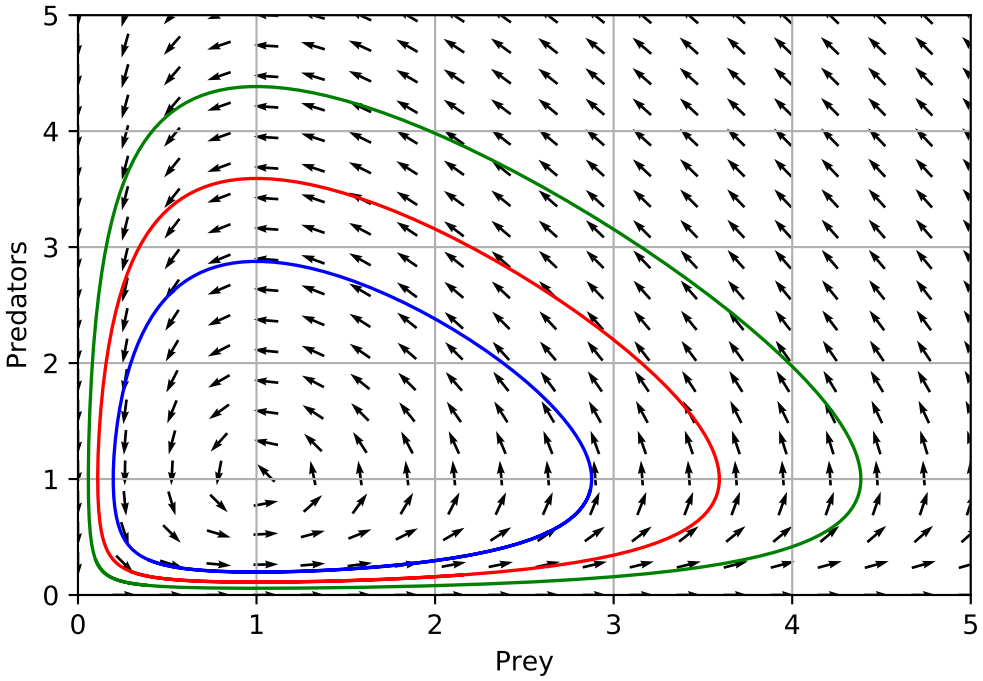
First, to use our Runga Kutta solver, we'll need to formulate the jacobian into callable function for the solver to evaluate upon every step in the itteration. This time we fomulate our state as a vector of x_1 and x_2.

Once our initial conditions are set, we can iteratbly advance the solver and record the states of x, the populations of predators and prey respectively.

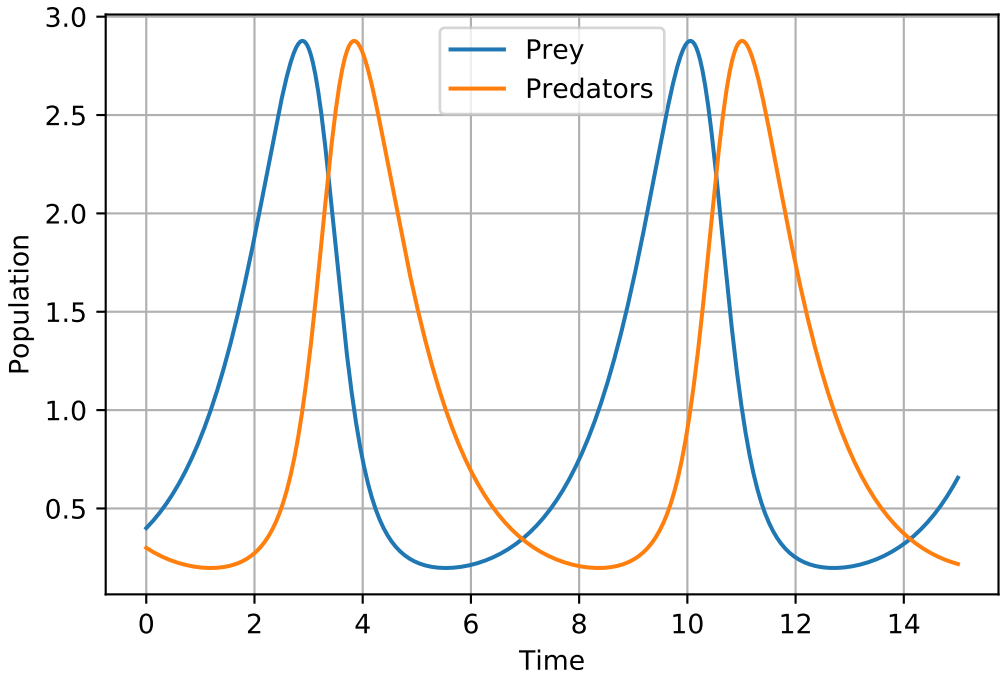


From this plot, we can clearly see the cause and effect relationship between predator and prey populations, as noted by the consistent phase shift between the two populations periodic cycles. At $t = 0$, we see the prey population begin to spike due to low number of predators. As the predators food supply become more plentiful over $t = 3$, the population of the predators also begins to rise as well. However, due to the logistic dynamics of each species, the population of predators soon overtakes that of the prey, resulting in unsustainable growth of the preditors and emdate decline of the prey.

This cycle and growth the decline continues for both. It should be noted however the athough the peaks and troughs for populations size of each species are phase shiffted, the envelopes of growth and decay are not necessarily identical. This gives a realistic dynamic to modeling these species relationships in nature, given that small prey animals are quicker to breed and recover but also quick to parrish, while larger predators are slower to mature while also slow to starve.



As shown from this vector field for the ODE using our solver, we can see that changing the initial conditions can only change the amplitude of the periodic system, given the system is autonomous, i.e as time progresses, populations will follow a closed curve. Again shown also by the same plot previously by with altered initial conditions, now will a lower peak population for both predator and prey:



3.

We have multiple robots that can generate point clouds such as those coming from a Kinect camera. In many cases we want to use the robots to detect objects in its enviroment. We provide three data files:

(a) Empty-Table.txt which containts a data for an empty table

(b) Cluttered-Table.txt contains point cloud for a cluttered table

(c) Hallway.txt

Each file has the point cloud file in a format with each line contains $x_i y_i z_i$

- Provide a method to estimate the plane parameter for the table. Test it both with the empty and cluttered table. Describe how you filter out the data from the objects. You have to be able to estimate the table parameters in the presence of clutter.
- Describe and show how the method can be generalized to extract all the dominant planes in a relatively empty hallway.

Note: See notebook for embedded 3d figure videos not rendered from the PDF.

For this approach, we'll use the RANSAC (RANdom SAmple Consensus) algorithm to help us fit a model of a plan to our 3D point cloud data. RANSAC is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set. RANSAC is enahrtly non-deterministic algorithm producing only a reasonable result with a certain probability, meaning that we may not get the same quality of a solution from one run to another. In this case well use it to help solve four linear and regression problem, i.e fitting a best fitted plane equation to our noisy point cloud data.

The algorithm splits the complete input sample data into a set of inliers, and outliers. We'll use this to help drive our search for planes via iterative filtering. In addition to simple RANSAC, we'll try leveraging

cluster techniques to segment plane hyptheiss from RANSAC in order to acquire more localized plane segmentation. This prevent over zellious filtering when removing inlier points from one iteration to the next, as far off point in space with in the plane model parameters may in fact not truly constitute point to the same surface we'd like to approximate in reality. Finally, we'll take the resulting largest cluster and use RANSAC again with a tighter residual threshold to acquire a finer estimate of the plane equation from just the segmented surface data. Points not found within the largest plane cluster are thrown back into pool of random points for the next interactive search for planes to sample from, enabling us to recursively root out smaller and smaller set of points forming plane surfaces.

We utilized pandas to help us read in the 3d points and pre processed them, like removing the large number of zeroed points that would otherwise inversely bias our RANSAC approach to find plane only intersection the origin.

We'll also need a plotting method to visualize our results, and show the performance of the plane segmentation and estimated plane models fitted. We'll also leverage matplotlib animation features to rotate the plots in video, as single perspective 3d scatter plots are hard to accurately read.

Next we'll define a function that takes in a dataframe of points and returns an fitted estimate model and abides the plane equations we need to invoke. Essentially this is a linear regression where we'd like to find the coefficients in the plane equation $a * x + b * y + c * z + d = 0$. However, we can set one of the coefficient to one, and merely solve for the remaining three, e.g set $c = 1$ for z and solve for a, b, d. Later, we'll extract the outlier and inlier masks from the model to cascade filter our search for planes.

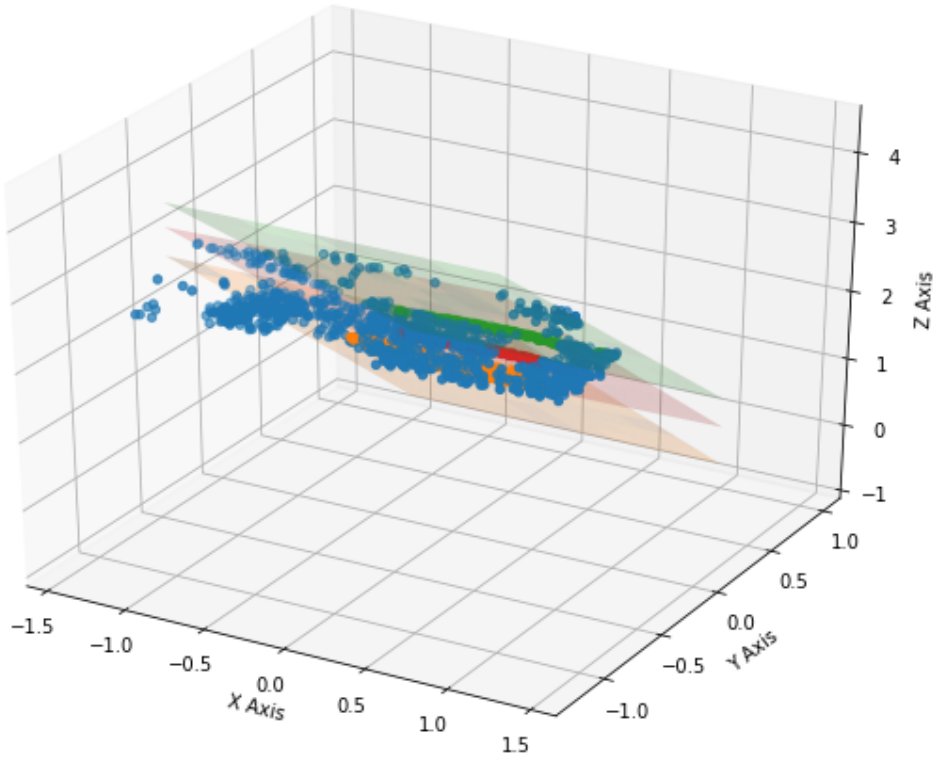
For our clustering approach, I've settled on a DBSCAN, a clustering method from vector array or distance matrix. DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. It clusters by finding core samples of high density and expands clusters from them. This method is good for data which contains clusters of similar density, but also scales well with the number of samples, provides us a simple knob for controlling maximum distance between two samples for them to be considered as in the same neighborhood, and lastly works well with labeling planner segments as a single class, as shown in sklearn's comparison annalise.

To implement the cascading filters, we'll need to do some bookkeeping to keep track of labeled points and points still available for the sample pool. To do this, we simply add a label dimension to our point cloud data and use pandas array broadcasting camaplities to utilise index masks while manipulation our data frame, or point cloud. Additionally, we take the oopertuitu to print out the refined plane equations dour our iterative search.

Empty Table

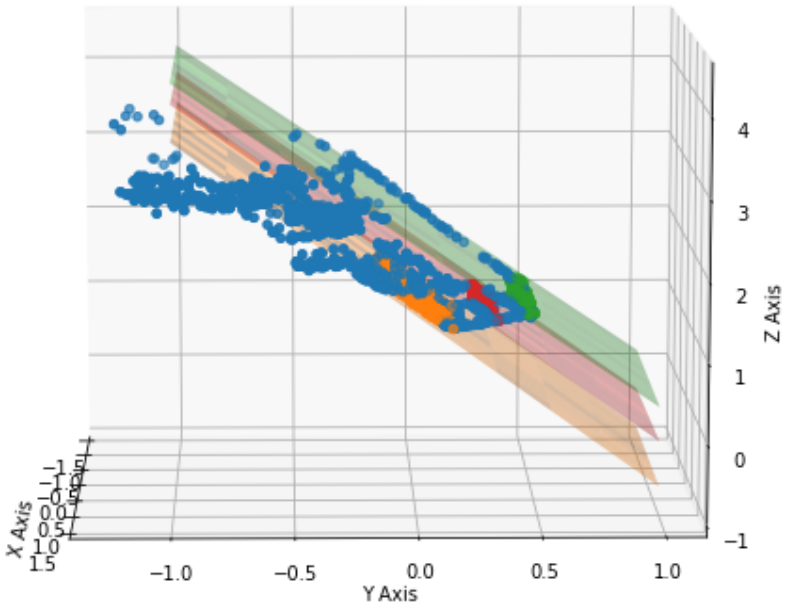
For the empty table, we know the screen to be quite favorable, as the planes are clear, uncluded, and quite flat. Additionally, they are quite close to the sensor, thus minimise the optical distortion incurred by the relatively cheap structured light depth camera. For far of distances, the noise and distortion proves troublesum to fit modeles to, given the non gaussian noise, as shown later in the hallway data set.

```
Plane Equation #1:
-0.002647782485180663 * x +
-2.1510026603638193 * y -
1 * z + 1.3972229528568754 = 0
Plane Equation #2:
-0.011318173521879856 * x +
-2.013669166694114 * y -
1 * z + 2.269985493890659 = 0
Plane Equation #3:
0.030213697072725115 * x +
-2.0988650434754947 * y -
1 * z + 1.8840585955508171 = 0
```



Above we can see a 3D scatter plot of the point cloud with segmented segmented planes colorized by differently with intersection planner surfaces rendered from the model parameters, also colored respectively. Blue is held as the common color for remaining outliers. Orange is often the most dominant plane, in this case the top surface of the table, with red, green, etc follow as smaller segments, in this case the shelf and floor partially visible from the robot's downward perspective.

```
<IPython.core.display.HTML object>
```

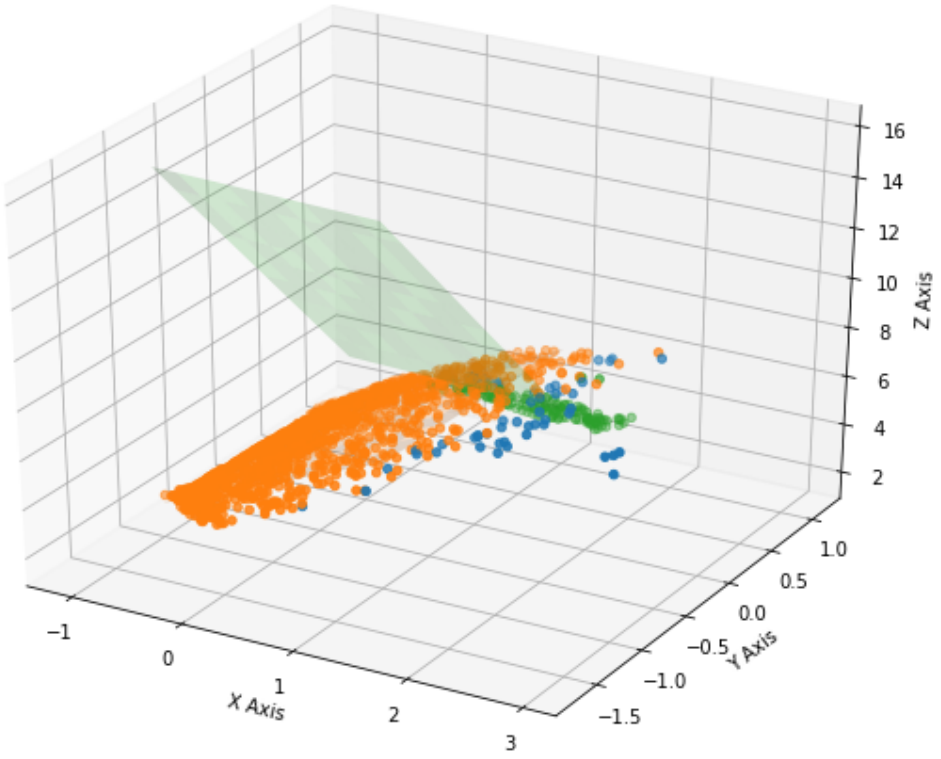


Hallway 02c

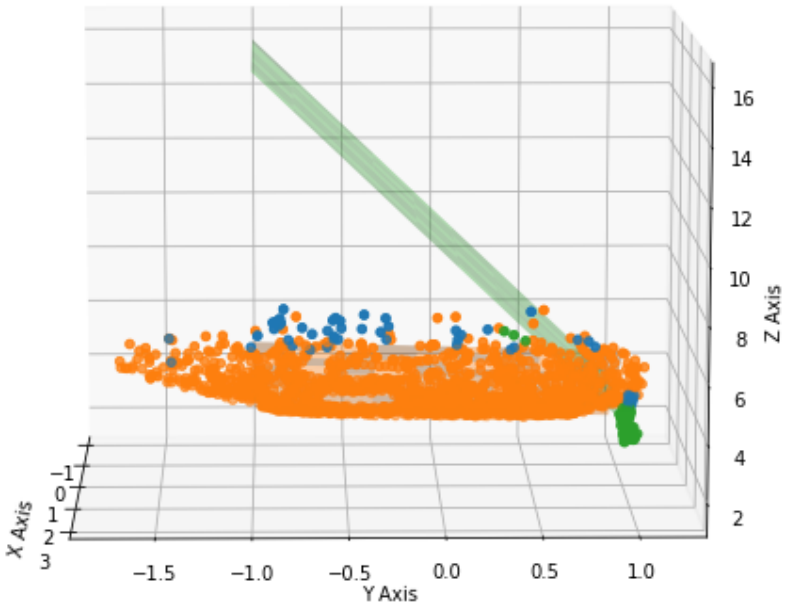
Next we have the halway. This data set proved troublesome, as the dominant plane is quite noisy further out in the distance along the wall surface. Not only does the angular resolution come into play, given the far point are also far from each other, additionally the poor rate of return given the incident angle is unfavorable for the active sensor relying on strongly returned signal already attenuated by distance.

Additialy the second plane, only partly visible from the plot, is quite elongated. Although is somewhat orthogonal to the dominant plane, its magor eagane vegort is not nearly orthonormal to it either. This results in a poor fit for the second plane, given RANSAC global sampling of points and often erroneously fitting with the outliers along the dominant plane. Short of building a proper KD tree to help guide the RANSAC algorithm to pick sampel point always more closer together as neighbors, I'm not sure tuning the residual threshold or other hyper perampates would be much help. Below is a fitting with our fancy clustering technique disabled:

```
Plane Equation #1:
1.8709241703269925 * x +
-0.16651227731391552 * y -
1 * z + 4.00043796709516 = 0
Plane Equation #2:
-0.14001813885976916 * x +
-6.167907996941239 * y -
1 * z + 9.477260800236055 = 0
```



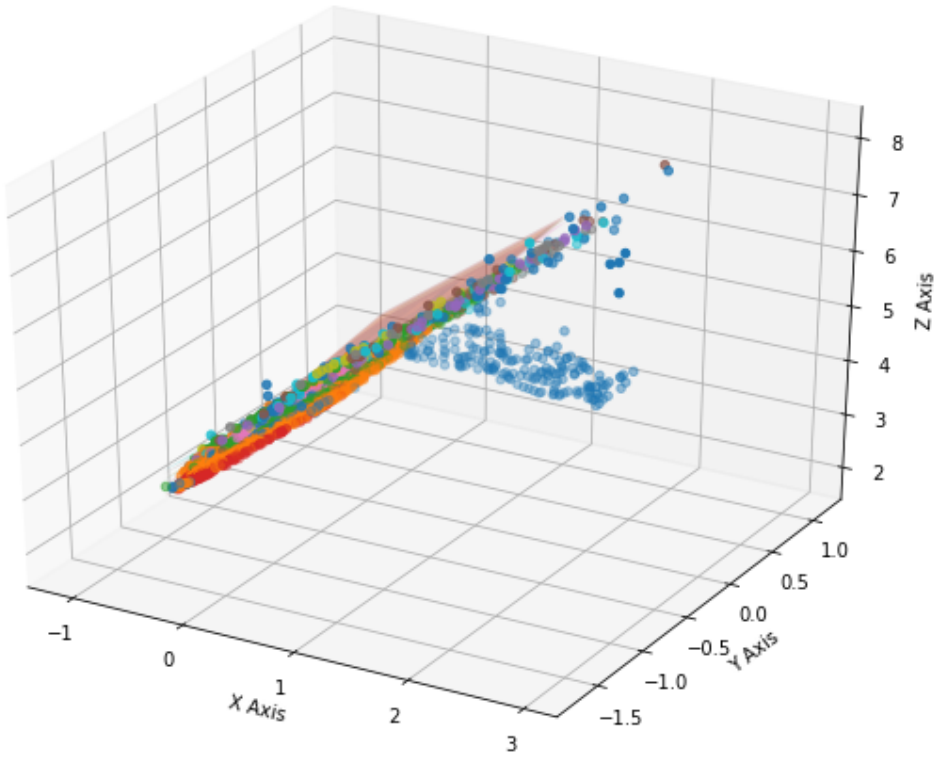

```
<IPython.core.display.HTML object>
```



Bellow is the hallway with a more aggressive segmentation search, attempting to classify the small portion of the floor. Again, there is perhaps much to optimize with respect to the interplay of residual threshold for our initial search and maximum distance between two samples defining a cluster. Presently I just have the setting of one parameter linearly scaling the other. Additionally, the residual threshold for this screen necessitate a much larger value due to the greater scale of the cloud. Adapting to this vaering degree of scale and resolution is perhaps key to achieving geral scene performance.

```
Plane Equation #1:
2.059917442130371 * x +
-0.16834866903525286 * y -
1 * z + 4.131644116851463 = 0
Plane Equation #2:
1.831760042604685 * x +
-0.1542210747043695 * y -
1 * z + 4.02569921741556 = 0
Plane Equation #3:
2.109603601712195 * x +
-0.16315788098338765 * y -
1 * z + 4.121477677216296 = 0
Plane Equation #4:
1.6602385947198484 * x +
-0.10028469352154191 * y -
1 * z + 4.042175030770252 = 0
Plane Equation #5:
1.8121689337437181 * x +
```

```
-0.15628798078575806 * y -
1 * z + 3.967899893508245 = 0
Plane Equation #6:
1.964105008390767 * x +
-0.17692157456397906 * y -
1 * z + 4.160228495852603 = 0
Plane Equation #7:
1.6485055542970604 * x +
-0.2010579275034385 * y -
1 * z + 4.038966059553072 = 0
Plane Equation #8:
1.7616786566864897 * x +
-0.14496598636722552 * y -
1 * z + 4.105795413628762 = 0
Plane Equation #9:
1.6850308489558288 * x +
-0.2299481353127553 * y -
1 * z + 4.106745313698084 = 0
Plane Equation #10:
1.6657405624793091 * x +
-0.19093383585116944 * y -
1 * z + 3.9424988310349685 = 0
```




```
<IPython.core.display.HTML object>
```

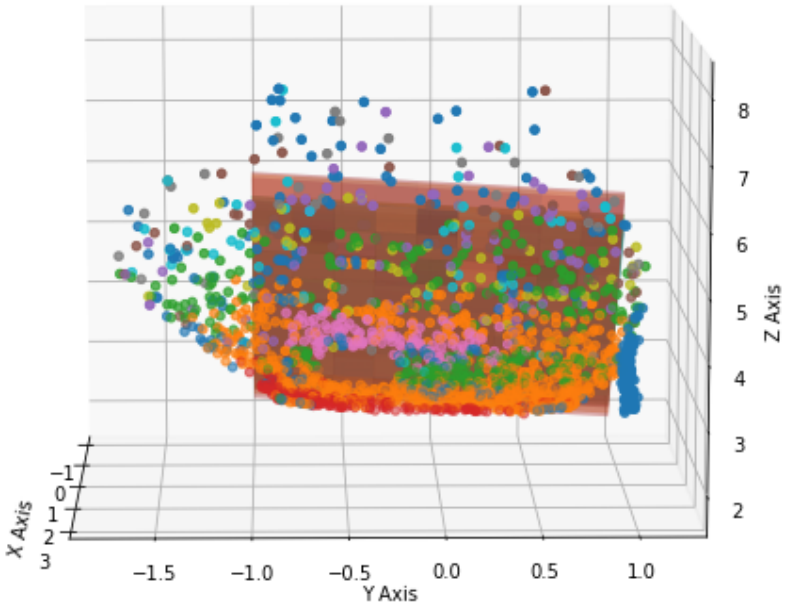
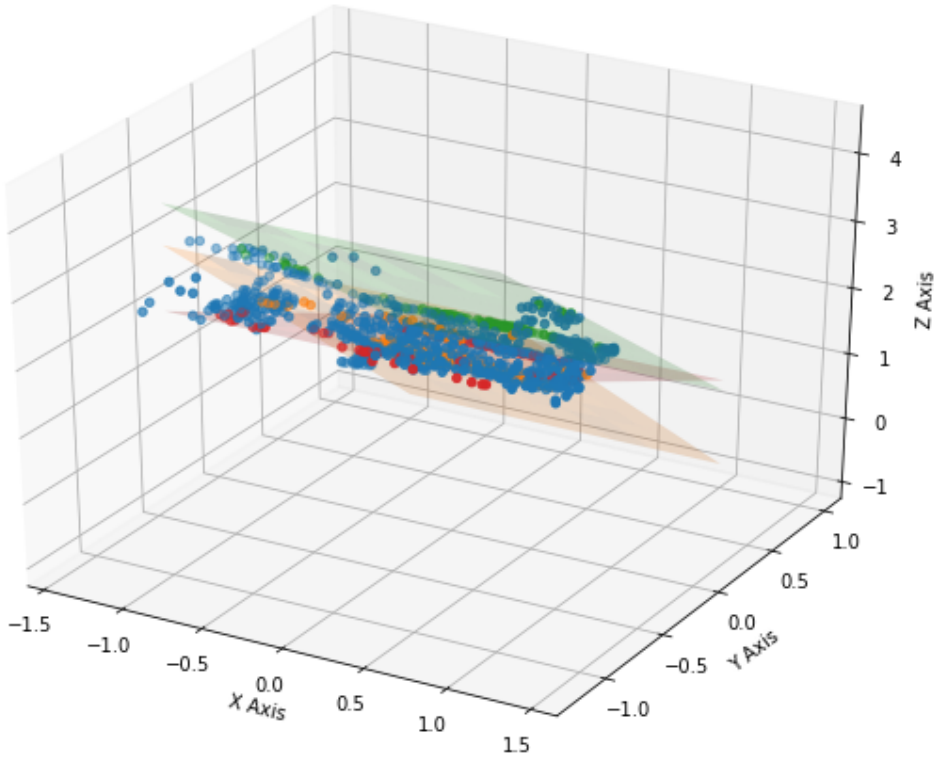


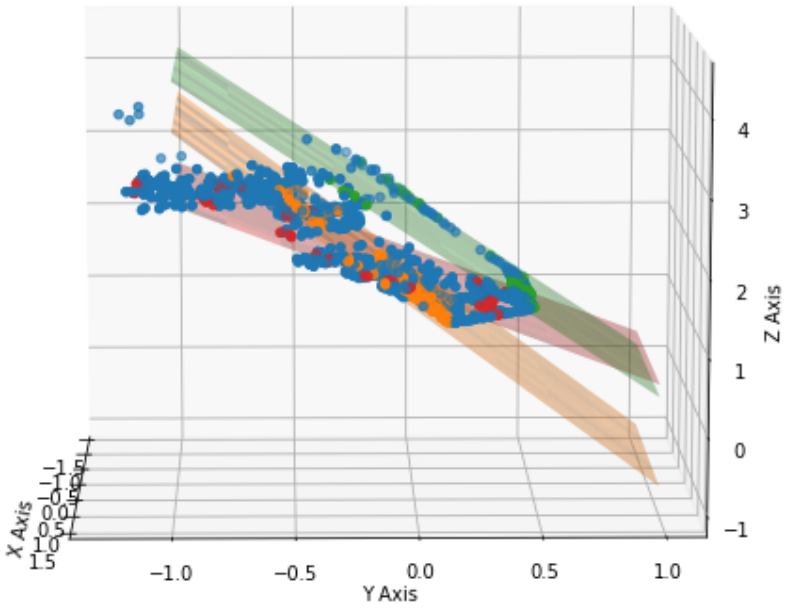
Table with Objects 2

In this scene with object cluttering the surface of the dominant plane, our revised clustering approach was still more that capable of determining equivalent parameters for the table/floor surfaces as before. Given the loser residual threshold used here to compensate for the noiser inliers, some of the surfaces, such as the shelf, suffer from more play in the model along the axis perpendicular to second domante, but substantially small eigenvector. This results in the visualization were the three planes found are no longer necessary co-planner. But if the task was merely deciphering the supporting surface for grasp and manipulation task, this approach could still be sufficient.

```
Plane Equation #1:
-0.009825493952483548 * x +
-2.2731516971287187 * y -
1 * z + 1.4009406589071756 = 0
Plane Equation #2:
0.013860482053250455 * x +
-2.020839787147342 * y -
1 * z + 2.2789287476658453 = 0
Plane Equation #3:
0.0071423285218146495 * x +
-1.1538243078626211 * y -
1 * z + 1.580598969261196 = 0
```




```
<IPython.core.display.HTML object>
```



Lastly, some remarks on performance. Although this entire pipeline is in python, and perhaps not as preffreable as C++ in the point cloud library, the RANSAC step is relatively quick in comparison to the clustering. I suspect there is plenty that could be optimized, such as more efficient uses of broadcasting, avoid recomputation of KD trees that is internally redone for each cluster evaluation, or encouraging smaller sample raduies to help support smaller plane segmentation. Perhaps the biggest improvement could come from dynamic parameter tuning, where the search pattern is adaptive to the scale or region on interest in the scene when hunter for planes.