Robotic Science and Systems - Exercise 5

Kevin Robb

1 OPT - Trajectory Optimization

In this section, we will use trajectory optimization (direct transcription) to find trajectories that satisfy constraints while minimizing costs. My code for parts (a) and (b) can be found in `opt.m`, and my code for parts (c) and onward is in `opt_lightdark.m`.

We consider a 2D robot with position $\boldsymbol{x} = \begin{bmatrix} x_x & x_y \end{bmatrix}^T \in \mathbb{R}^2$. It is controlled via displacement in each axis independently, $\boldsymbol{u} = \begin{bmatrix} u_x & u_y \end{bmatrix}^T \in \mathbb{R}^2$, s.t. the system dynamics are linear with zero process noise: $f(\boldsymbol{x}_t, \boldsymbol{u}_t) = \boldsymbol{x}_t + \boldsymbol{u}_t$.

The control cost for a timestep is the sum of squares of the individual components,

$$C_t = u_{x,t}^2 + u_{y,t}^2 = \boldsymbol{u}_t^T R \boldsymbol{u}_t,$$
$$\text{where } R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Since this aggregates by summing all timesteps, we can perform this in one step with our vector of commands by squaring and then summing all elements.

Note: For parts (a) and (b), we minimize the set of commands and use that to obtain a trajectory, while in later parts we minimize the trajectory of positions/variances itself.

1.a Initial Cost-Minimizing Trajectory Generation

Supposing $\boldsymbol{x}_{start} = \begin{bmatrix} 2 & 2 \end{bmatrix}^T$ and $\boldsymbol{x}_{goal} = \begin{bmatrix} 4 & 0 \end{bmatrix}^T$, we can use `fmincon` in MATLAB to find a cost-minimizing trajectory.

By storing our trajectory as a $2T \times 1$ array in the form $\begin{bmatrix} u_{x,1} & ... & u_{x,T} & u_{y,1} & ... & u_{y,T} \end{bmatrix}^T$, we can ensure only valid trajectories will be found with the equality constraints

$$A_{eq} = \begin{bmatrix} 1, ..., 1 & 0, ..., 0 \\ 0, ..., 0 & 1, ..., 1 \end{bmatrix} \in \mathbb{R}^{2 \times 2T}$$
$$B_{eq} = \boldsymbol{x}_{goal} - \boldsymbol{x}_{start} \in \mathbb{R}^{2 \times 1}$$

which ensures the necessary displacement will be achieved to move from the start to goal.

Running this for $\boldsymbol{x}_{start} = \begin{bmatrix} 2 & 2 \end{bmatrix}^T$ and $\boldsymbol{x}_{goal} = \begin{bmatrix} 4 & 0 \end{bmatrix}^T$ for $T = 20$ timesteps yields a straight-line trajectory, as we'd expect. The control command for each timestep is $\boldsymbol{u}_t = \begin{bmatrix} 0.1 & -0.1 \end{bmatrix}^T$, giving the correct displacement over 20 timesteps, and an overall cost of 0.4. This trajectory is shown in the following figure.

Figure 1: Trajectory generated for the sum of squared components cost function.

1.b Additional Cost Functions and Trajectories

When modifying the cost function to generate other trajectories, I found that most of the time the generated trajectory will make huge jumps, i.e., if my cost function is the distance from the goal, it will instantly jump to the goal, and remain there for the duration. To make these trajectories more realistic, meaningful, and differentiable from one another, I added a constraint for the maximum command allowed in one timestep.

$$A \cdot \boldsymbol{u} < B$$
$$\begin{bmatrix} I_{2T} \\ -I_{2T} \end{bmatrix} \cdot \boldsymbol{u} < B,$$

where $\boldsymbol{u} \in \mathbb{R}^{2T \times 1}$, and all elements of $B \in \mathbb{R}^{4T \times 1}$ are the maximum magnitude allowed for a control component, in this case 0.5. The double size allows us to control the maximum positive velocity and minimum negative velocity with one linear matrix inequality. (We will later move this condition into our nonlinear constraint function.)

With these constraints implemented alongside the previous, we can experiment with the cost function to generate different trajectories.

1.b.i Cost = Distance to Goal

With the cost being the sum of all timesteps, the cost at each timestep is the current distance to the goal in addition to the sum of squared components. In this case, the robot will drive at full speed towards the goal, stopping once it arrives for the duration. This is because the cost is lower closer to the goal, so it will spend as much time as it can in this region of low cost.

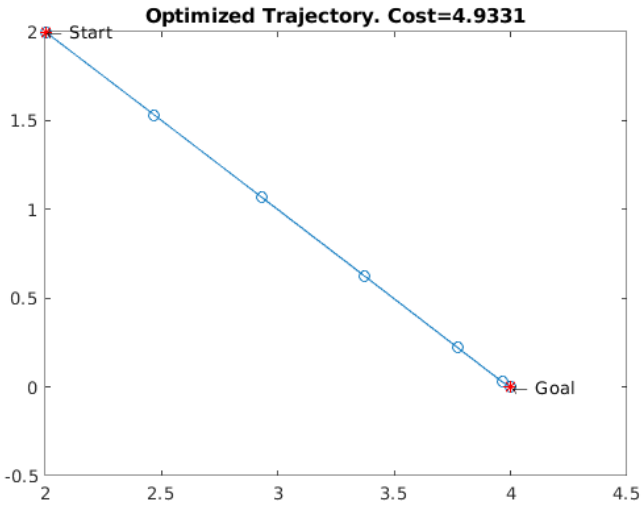


Figure 2: Trajectory generated for the "distance to goal" cost function.

1.b.ii Cost = Inverse Distance to Midpoint

Here the cost is the sum of squared components plus the inverse distance to the midpoint. This makes the cost very high along the line directly connecting the start and goal, so it takes a different route, staying as far away from the midpoint as possible while satisfying constraints and minimizing the sum of its squared command components.

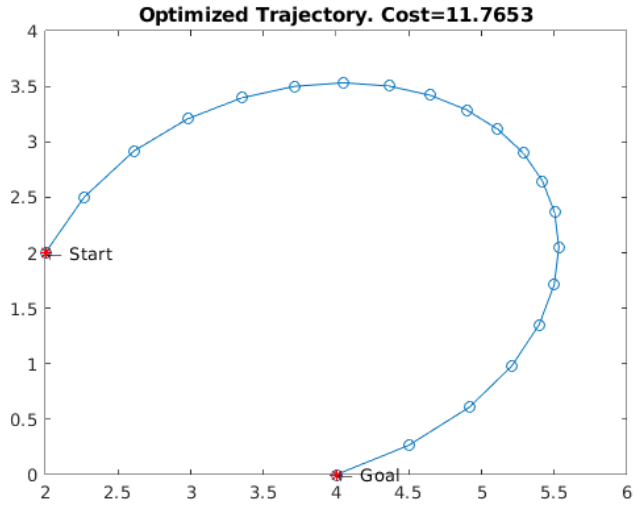


Figure 3: Trajectory generated for the "inverse distance to midpoint" cost function.

1.b.iii Cost = Distance to Arbitrary Point

Here the cost is again the sum of squared components, but with the addition of the distance to a point of our choosing, in this case $\begin{bmatrix} 2 & 4 \end{bmatrix}^T$. The robot goes to this point, spends as much time as it can there, and leaves with just enough time to make it to the goal at the end.

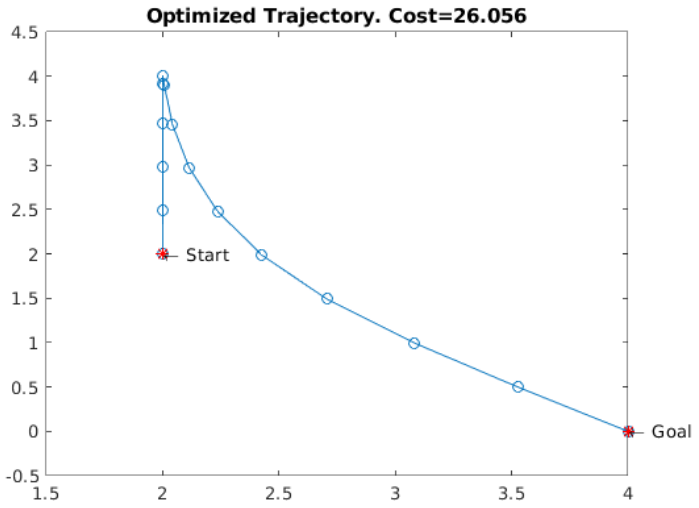


Figure 4: Trajectory generated for the "distance to a point" cost function.

1.c Light-Dark Domain Derivations

We now move to the light-dark domain in which uncertainty is a factor we must consider. The observation noise varies with x-position as $\sigma_w^2 = w(x_x) = \frac{1}{2}(5 - x_x)^2 - 0.1$. We assume for the sake of trajectory planning that we always observe the most likely outcome (so the observation step does not affect the estimated position). We store our variance in the state, so now $\boldsymbol{x}_t = \begin{bmatrix} x_t & y_t & v_t \end{bmatrix}^T$. Framing this as a Kalman Filtering problem gives us the following constraints. We assume there is a constant process noise variance, σ_v^2, s.t. the variance element in the state will always increase during the prediction step, as we expect with a Kalman filtering problem.

$$\boldsymbol{x}_{t+1}^+ = \begin{bmatrix} x_t + u_x \\ y_t + u_y \\ v_t + \sigma_v^2 \end{bmatrix}$$

$$\boldsymbol{x}_{t+1} = \begin{bmatrix} x_{t+1}^+ \\ y_{t+1}^+ \\ (1-K) \cdot v_{t+1}^+ \end{bmatrix},$$

$$\text{where } K = v_{t+1}^+ (v_{t+1}^+ + w(x_{t+1}^+))^{-1}$$

$$\Rightarrow (1-K) = \frac{\cancel{v_{t+1}^+} + w(x_{t+1}^+) - \cancel{v_{t+1}^+}}{v_{t+1}^+ + w(x_{t+1}^+)}.$$

Putting this together yields

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} x_t + u_x \\ y_t + u_y \\ \frac{(v_t + \sigma_v^2) \cdot w(x_{t+1}^+)}{(v_t + \sigma_v^2) + w(x_{t+1}^+)} \end{bmatrix}.$$

We can create a function to check for this constraint for each pair of timesteps, and use this in the nonlinear constraint parameter of `fmincon`. We will also check here that the distance moved between any two adjacent states in the trajectory is less than our accepted maximum control for commands.

We have a further constraint that the variance is always positive, and greater than some epsilon which

I set to $\epsilon = 0.0001$; we achieve this with the $A \cdot \boldsymbol{u} \leq B$ constraint. Setting A to the lower third of the $3T$ identity matrix extracts out the variance for all states, and $B \in \mathbb{R}^{T \times 1}$ has every element equal to ϵ; these values will allow us to check $A \cdot \boldsymbol{u} >= B$, so we negate both to get the required constraint for `fmincon`.

The last inequality constraint is that the final variance must be $\leq$ the variance specified in $\boldsymbol{x}_{goal}$. We check this by adding a row to the bottom of A that is all zeros, ending with a 1, and we add the goal variance to the bottom of B.

For our linear equality constraints, we will use the `Aeq` and `Beq` parameters. These constraints will be the starting position and variance, and the goal position. Since we store the trajectory as a $3T \times 1$ vector, we will initialize `Aeq` as a $5 \times 3T$ matrix of all zeros, and will put a one in positions (1,1), (2,T+1), (3,2T+1), (4,T), and (5,2T). This extracts out the first and last state entries, and we can require that this equals

$$\texttt{Beq} = \begin{bmatrix} x_{start} \\ y_{start} \\ v_{start} \\ x_{goal} \\ y_{goal} \end{bmatrix} \in \mathbb{R}^{5 \times 1}$$

to force the trajectory to begin and end in the correct positions, and use the starting variance.

1.d Implementing the Light-Dark Domain

Implementing the constraints discussed in the previous section alongside the "sum of squared commands" cost function that we have used previously yields the following results. As expected, the robot moves towards the light and stays there as long as it can to reduce its uncertainty, and moves to the goal at the end. The variance for a point is represented by the size of the red circle centered at that point.

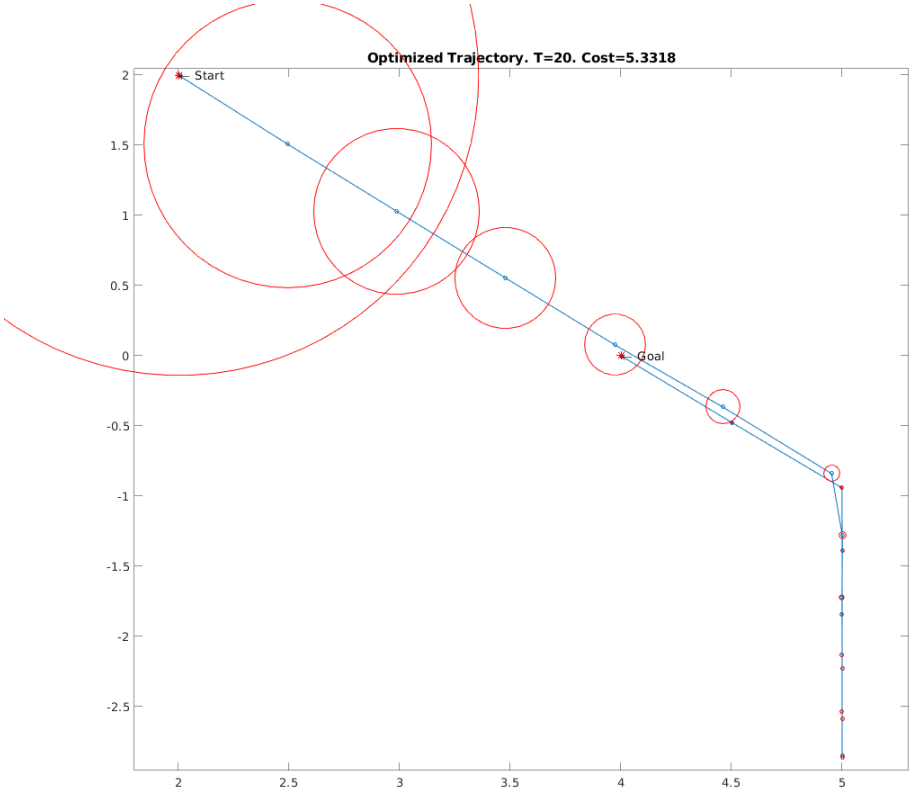


Figure 5: Trajectory generated in the light-dark domain.

1.e Light-Dark Domain with Different Parameters

Varying T, the number of timesteps the trajectory is divided into, yields interesting results, visualized and described in the following figures.

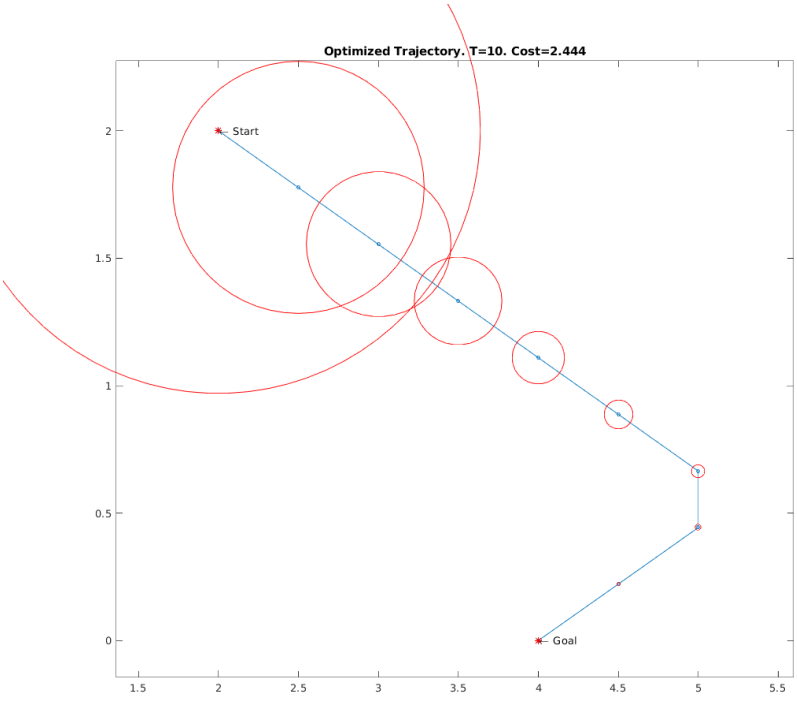


Figure 6: Reducing the number of timesteps to 10 causes the path to more directly approach the light, hug it, and then go to the goal, instead of passing through the goal and wasting time by the light as in the $T = 20$ case.

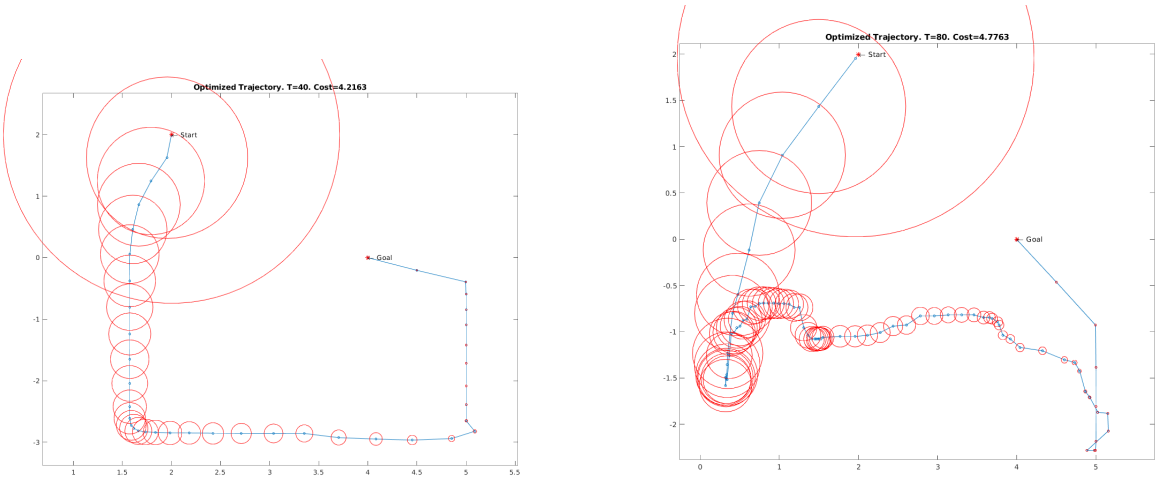


Figure 7: Increasing the number of timesteps to 40 (left) and 80 (right) gives an unintuitive path. This is caused by the constraint on the variance's lower bound, and its lack of impact on the cost. Since $v > \epsilon$ always, and the variance does not affect the cost, there is no motive to, say, approach the goal directly after waiting by the light, as opposed to wandering off just long enough that the final variance will still be under the expectation.

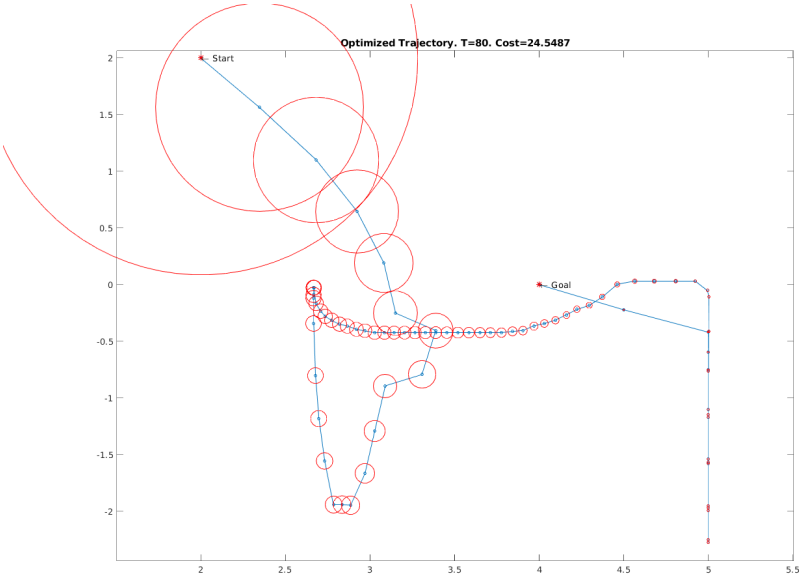
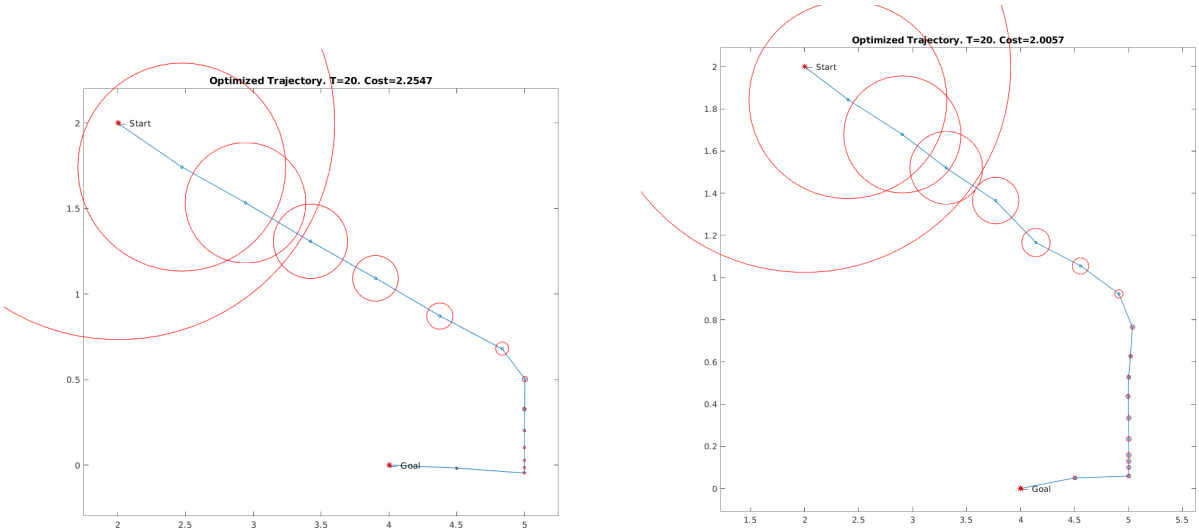


Figure 8: Naively adding the sum of all timesteps' variances to the cost gives a better but still not intuitively optimal graph.

Looking now to different configuration parameters, we can change how we initialize the trajectory to influence the final path. Up to this point we have initialized it as all zeros, but we can see that initializing instead as all ones (left) actually finds a better path with a lower cost. Initializing with all states equal to the starting state (right) improves this even further. This demonstrates that our implementation of `fmincon` is very dependent on the initial setup, and is thus not necessarily able to find the true optimal trajectory.



(a) Initialize all states with $[1, 1, 1]^T$.

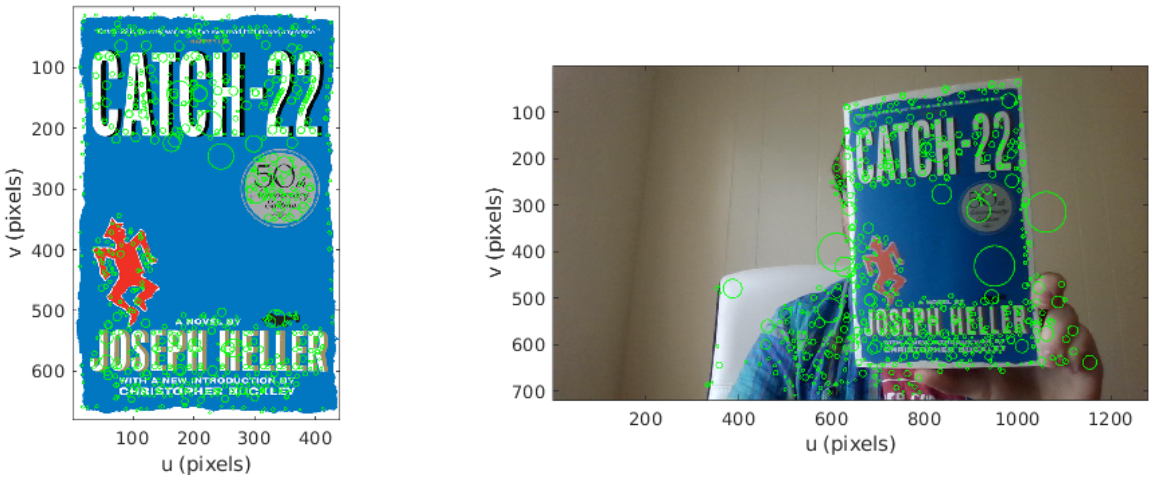
(b) Initialize all states with start.

Figure 9: Different trajectory initializations' effects on `fmincon`.

2 V1 - Book Tracking

In this section, we track the pose of a moving book cover in a series of webcam frames. I will be using the novel *Catch-22*.

Our strategy involves the use of a template image of the cover, which I obtained from the web. To ensure that we only track features across adjacent images which are part of the book cover, we do not directly compare real images, but rather use the template as an intermediary. For each image from the webcam stream (or from a folder of pre-saved images), we use `isurf` to compute its SURF features, and use `match` to find matches between these features and those on the template. This ensures that static or other moving objects in the scene will not affect our resulting homographies.



(a) Template with 800 SURF features.

(b) Real image with 800 SURF features.

Figure 10: All features obtained with the `isurf` function.

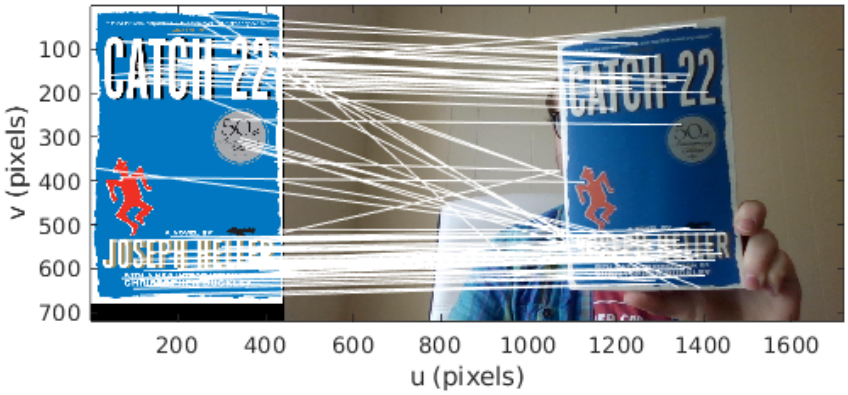


Figure 11: Matches produced between the template and a real image by the `match` function.

Using the matched points between the template and a real image, I use my custom RANSAC function to

find a homogenous transform matrix that describes the relationship between the point sets to my satisfaction. I select a random set of 40 matches, use their centroids to compute the translation of each set from the origin, and then use a singular value decomposition to determine the rotation about the origin that best describes the matches after correcting for translation. We then check this transform for all 400 matches and compute the proportion of points for which this results in an error of less than 40 pixels, and if this ratio is above 60%, we stop the loop. The last thing we do is recompute the transform using the full set of points for which the last chosen transform works well (the "inliers"). This last step improves our accuracy by taking as many points into account as is reasonable.

After the SE2 transform is returned from our RANSAC function, we can use it on the set of features in the template and display the results to get an idea of the effectiveness of our transform. We can also use it on the corner points in the template to draw a bounding box for the book cover on the real image. This allows us to show the position of the book on a live webcam feed, although to get my RANSAC function running quickly enough we sacrifice some desired accuracy of the transform to limit iterations.

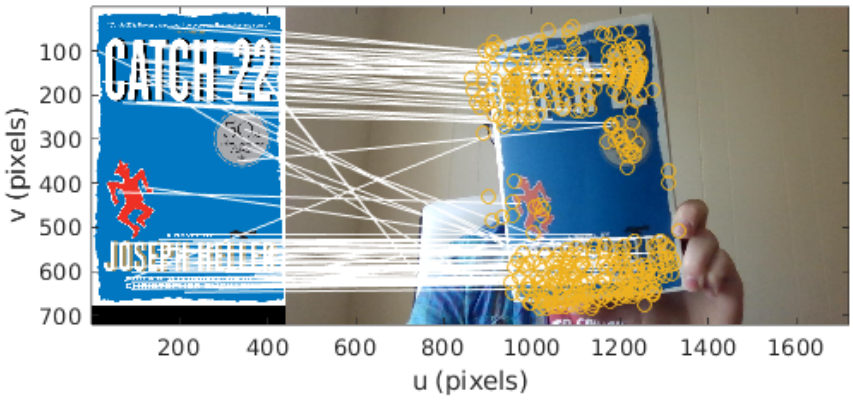


Figure 12: Template and real image matches, with features transformed from the template shown. We see it matches the real image fairly well.

Figure 13: Real book image showing features transformed from the template, and a bounding box also transformed from the template. We can see the bounding box isn't perfect but does clearly find the book in the image.

After computing the transform between the template and two consecutive images, we can compute the transform from the first image to the next by inverting the first transform and multiplying by the second. (In my script, the translation and rotation are printed to the console for each pair of images.) We can then apply this transform to features in the first image to plot correspondences between the two images.

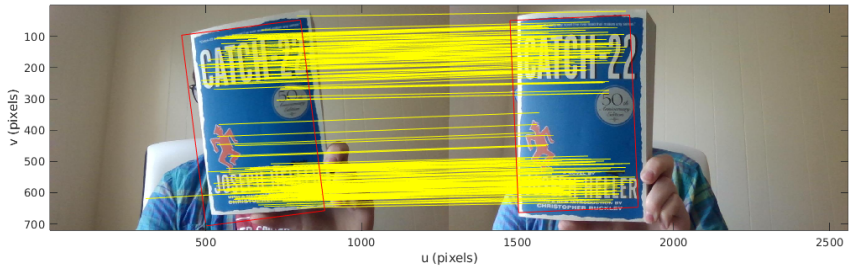


Figure 14: Features in the first image are transformed first to the template frame, then to the second image's frame, and a line is drawn between them. The bounding boxes are transformed from the template to each image for comparison.

3 V2 - ICP

In this section, we will implement and investigate the iterative closest point (ICP) algorithm for feature matching in 3D pointcloud images.

3.a Implementing ICP

My implementation of this algorithm can be found in `v2.m`. I created the primary `icp` function, as well as some helper functions:

- `estimate_transform`: Estimate the SE3 transformation between two point clouds by computing each's center of mass, centering them, and using SVD to compute the rotation submatrix.
- `transform_points`: Apply a 4x4 transformation matrix to all points in a set to get a new pointcloud.
- `nearest_pt`: Find the nearest point in a set to the given point, using 3D Euclidean distance as the metric.

3.b Testing ICP with two pointclouds

If data association between the two point clouds, ICP is trivial, as our `estimate_transform` is able to find the transformation in a single step. If data association is not known, we use the nearest neighbor in set 2 of each point in set 1 as its match, and more iteration is needed to get close enough that these matches are reasonably accurate, and the true transform can be approximated.

With my instantiation, I create a random (unitary) transform in SE(3) using the `SE3.rand().T` function in Corke's Robotics Toolbox. As a result, some transforms are harder to estimate than others. If we always use the identity transform as our initial guess for ICP, it fails in cases where the rotation is significant, and will end up in the right place but orientated incorrectly. Local minima like this can easily trap the ICP algorithm.

If we give ICP an initial (rough) guess for the transformation (computed by adding another random SE3 transformation with its magnitude reduced to the true one), it is much more capable of estimating the true transformation within 15 iterations.

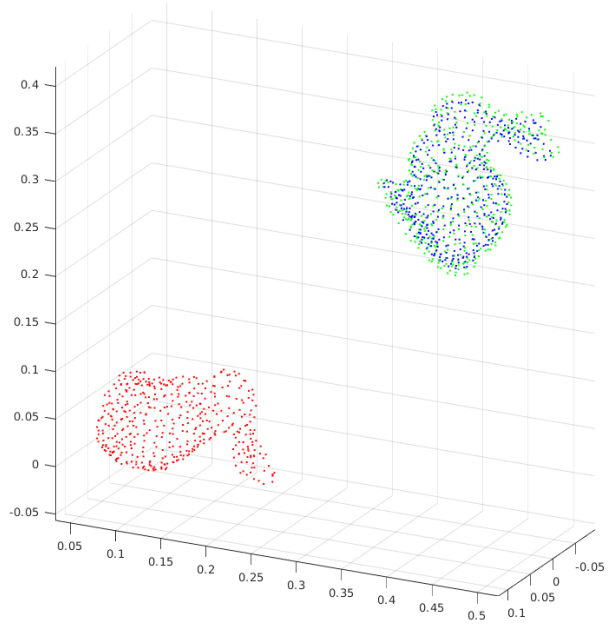
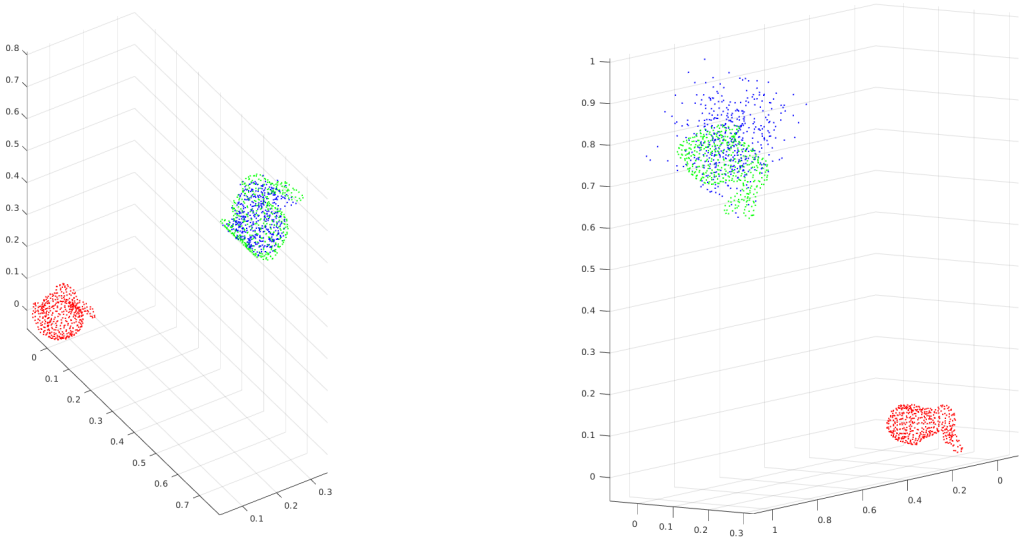


Figure 15: Original pointcloud is shown in red, and true transformed data in blue. Our estimate with ICP after converging is shown in green. We can see it basically overlaps with the blue, so it is successful.

3.c Investigating the robustness of ICP

If we alter the transformed pointcloud in different ways, we can affect the performance of ICP. If, after transforming, we add random Gaussian noise to every point, we can test at what point our algorithm is unable to find the true transform.

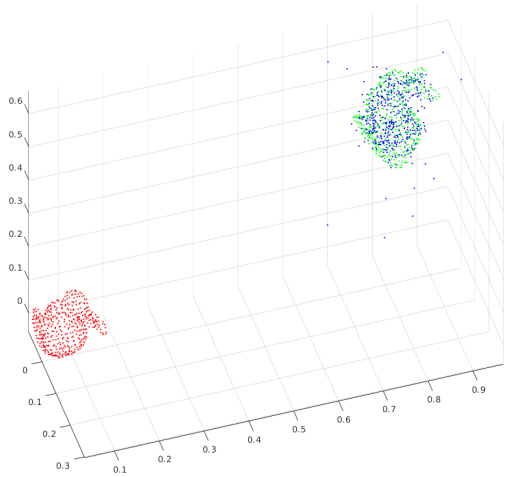


(a) 0-mean, 0.01 standard deviation Gaussian noise.

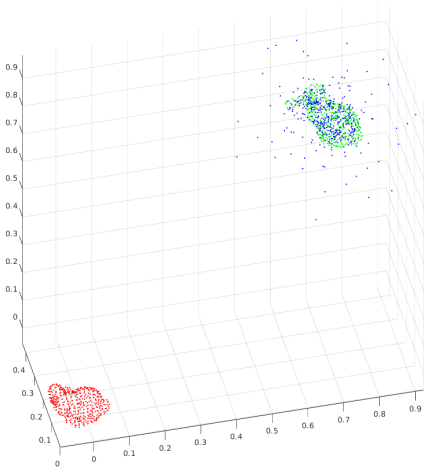
(b) 0-mean, 0.05 standard deviation Gaussian noise.

Figure 16: We can see that with some noise, our algorithm is still very effective, as in (a). If the noise becomes too large, however, it can become hard to determine the correct orientation, as any features in the pointcloud are blurred out of existence. (b) shows this beginning to happen; we can visually see where the blue bunny is oriented, but the program struggles.

Another way we can obfuscate the second pointcloud is by adding spurious points and removing some existing points.



(a) Remove 5% of points, and add that many new spurious points.



(b) Remove 33% of points, and add that many new spurious points.

Figure 17: Removing a small portion of the points, and adding a few random points, as in (a), does not affect the performance significantly, as ICP will still snap to fit the pointclouds as best it can. Even replacing 1 out of every 3 points, as in (b), does not prevent the true transform being estimated to our desired precision.

ICP is surprisingly robust when it comes to spurious or missing points, and does not degrade in consistency until about half of all points are false. The biggest source of failure remains a too high difference in the initial guess and true rotation, where ICP can easily get stuck in the wrong orientation.

3.d Extra Credit

ICP works well for images that are close, and is completely useless if the transform between two pointclouds is substantial. As I discussed above, it is very easy for ICP using the nearest-neighbors data association approach to fall into a local trap, such as an oblong point cloud being rotated 180° from where it should be.

ICP also works better for point clouds with unique shapes to them that allow the clouds to converge on an approximately correct transform. It will be very hard to align a sphere with points evenly distributed across the surface, since determining the correct orientation will be nearly impossible for ICP to achieve.

4 V3 - Implementing RANSAC

4.a Computing Surface Normals

We can use points in a neighborhood to determine the structure of the surface at that point. We simplify this by only checking points with pixel coordinates in a certain region of our point of interest, rather than computing the distances between every pair of points in our set. After checking pixel neighbors and selecting only those within a specified radius (by Euclidean distance), we use the set of neighbors to compute a sample covariance matrix. The eigenvectors of this matrix tell us the primary axes of its characteristic ellipsoid, and the eigenvalues corresponding to each vector gives us a sense of the "flatness" of the ellipsoid in that

dimension. By taking the eigenvector with the smallest eigenvalue, we get the direction most likely to be normal to the surface. If no one axis is significantly smaller than the others, i.e., if $\lambda_{smallest}/\lambda_{biggest}$ is not small, it is unlikely our point of interest actually lies on a distinguishable surface, so we assign it the zero vector as a surface normal.

4.b Using RANSAC to Find Planes

We know that a plane is defined by three points. As such, we can use RANSAC to choose such a set of three points. Many points in our cloud have `NaN` components, so we randomly choose sets of pixel coordinates until we end up with three fully defined points in space.

Given the three points, $\boldsymbol{p_1}$, $\boldsymbol{p_2}$, and $\boldsymbol{p_3}$, we can find the unit normal vector of the plane at $\boldsymbol{p_1}$ as

$$\boldsymbol{v_1} = \boldsymbol{p_2} - \boldsymbol{p_1}$$
$$\boldsymbol{v_2} = \boldsymbol{p_3} - \boldsymbol{p_1}$$
$$\boldsymbol{n} = \frac{\boldsymbol{v_1} \times \boldsymbol{v_2}}{\|\boldsymbol{v_1} \times \boldsymbol{v_2}\|}$$

We can now find the distance from any candidate point p_c to the plane as

$$D(\boldsymbol{p_c}) = (\boldsymbol{n} \cdot \boldsymbol{p_c}) - (\boldsymbol{n} \cdot \boldsymbol{p_1}).$$

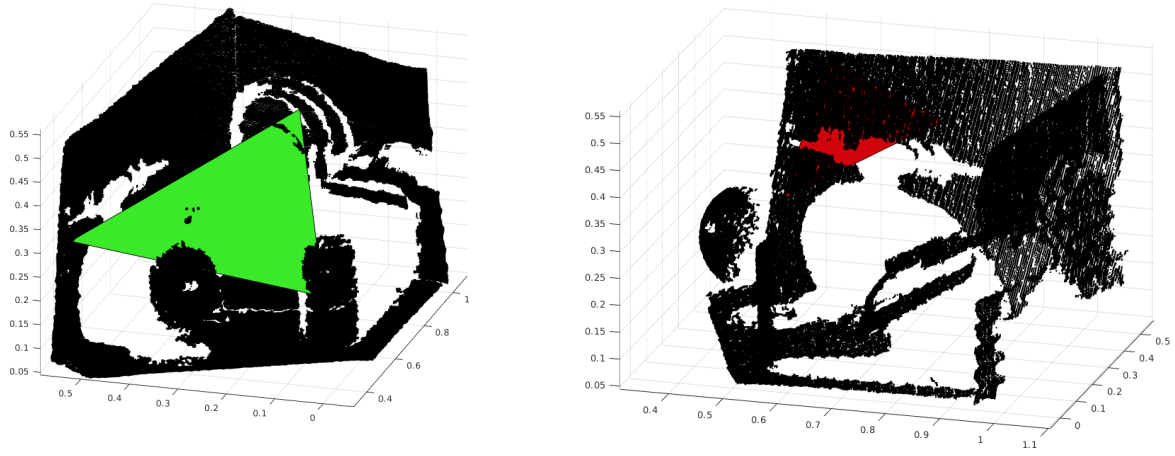
We now have a way to determine the set of points that are within some $\epsilon_{position}$ of the plane. We'd also like to check that the surface normal at a point is approximately parallel or anti-parallel to the normal vector of the plane. Let the normal vector at our candidate point be denoted $\boldsymbol{n_c}$. We know that if $\boldsymbol{n_c} \perp \boldsymbol{n}$, their dot product is zero, and that if $\boldsymbol{n_c} \parallel \boldsymbol{n}$, their dot product is one (since they are both unit vectors).

We now have a formulaic way of testing whether every point in the set is part of a plane defined by three points. To be part of the plane defined by $\boldsymbol{p_1}$ and $\boldsymbol{n}$, a point $\boldsymbol{p_c}$ must satisfy both:

$$D(\boldsymbol{p_c}) < \epsilon_{position}$$
$$1 - \|\boldsymbol{n_c} \cdot \boldsymbol{n}\| < \epsilon_{angle}$$

where in my code I've used $\epsilon_{position} = 0.05$ and $\epsilon_{angle} = 0.1$. If a suitable number of points pass this criteria (at least 20,000 in my code), we say the plane is valid, and save it. We then remove all points on this plane from future consideration by setting them to `NaN` in the pointcloud, so that RANSAC can continue searching for planes without finding the same plane repeatedly.

Using this method, we are able to identify the three planes in the provided pointcloud, as shown in the following figures.



(a) Not enough points pass our criteria, so the plane is not used.

(b) Many points pass our criteria, so the plane is used.

Figure 18: Planes formed by a choice of three random points in the cloud.

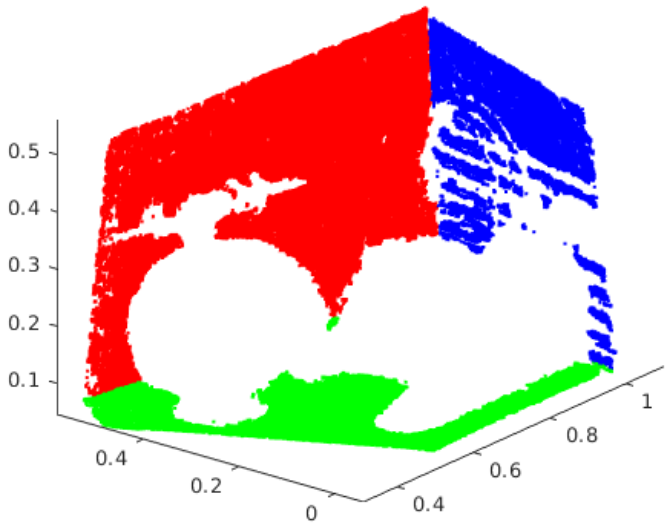


Figure 19: After running for 300 iterations, RANSAC is consistently able to categorize the three major planes in the scene.

We can tell that the green plane in Figure 19 is the table because its normal vector will be the most vertical. Similarly, we can tell the red and blue are walls, since their normal vectors are strongly horizontal.

4.c Using RANSAC to Find the Sphere

Our procedure for finding the sphere in the image is to choose a point at random (ensuring this point has no `NaN` components and has a nonzero surface normal vector) and assume this point is on the surface of the

sphere. We then sample a radius at random from the acceptable range, and compute the center of the sphere using the surface normal at the point.

$$\boldsymbol{ctr} = \boldsymbol{p} - \boldsymbol{p_{norm}} \cdot radius$$

We then check all points within a region of pixels to see if they are on the surface of this sphere. We do this by computing a position error and angle error, similarly to what was done for plane detection. If a point is really on the sphere, its distance from the center point should be approximately equal to the radius, and its associated surface normal should be approximately colinear with the vector from the center to the point. Thus, for a candidate point $\boldsymbol{p_c}$ with surface normal vector $\boldsymbol{n_c}$ to lie on the sphere, it must satisfy the following:

$$\left| \|\boldsymbol{p_c} - \boldsymbol{ctr}\| - radius \right| < \epsilon_{position}$$
$$1 - \left| \frac{\boldsymbol{p_c} - \boldsymbol{ctr}}{\|\boldsymbol{p_c} - \boldsymbol{ctr}\|} \cdot \boldsymbol{n_c} \right| < \epsilon_{angle}$$

For my case, I use $\epsilon_{position} = 0.02$ and $\epsilon_{angle} = 0.2$.

If our randomly created sphere has at least 6000 inliers, we end our RANSAC loop and recompute the radius and center using only the set of inliers. A helper function `snap_sphere` tracks the cumulative position and angle error for all inliers, and iterates a specified number of times (i.e., 30), saving the radius and center that give the lowest error. This improves the accuracy significantly, since we're taking all points into account, but is also much faster than relying on our RANSAC loop entirely, since we've limited the scope to only inliers.

All in all, my script is able to run in about 2 seconds (after the surface normals for all points have been precomputed), and finds the sphere very consistently. It determines the radius is ≈ 0.094 meters, and the center point is located at $\approx \begin{bmatrix} 0.44 & 0.35 & 0.21 \end{bmatrix}^T$.

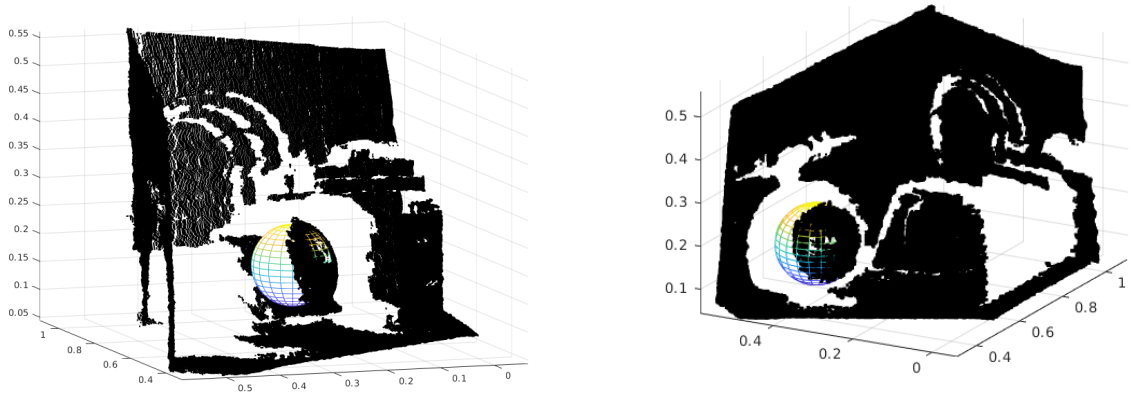


Figure 20: Sphere in the pointcloud detected by our algorithm.

4.d Using RANSAC to Find the Cylinder

We start by choosing a point with a nonzero surface normal vector at random from the pointcloud. We then choose a second point at random from within a reasonable neighborhood of the first point, and ensure its surface normal is not approximately parallel to the first point's. With these two points, $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$, and

their surface normal unit vectors, $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$, we can compute the parameters of the cylinder, assuming both points lie on the surface. The calculations that follow, resulting in a point $\boldsymbol{c}$ and unit vector $\hat{\boldsymbol{a}}$, are performed in a helper function `estimate_cylinder`.

First, we can find the central axis unit vector for the cylinder, $\hat{\boldsymbol{a}}$.

$$\hat{\boldsymbol{a}} = \frac{\boldsymbol{n_1} \times \boldsymbol{n_2}}{\|\boldsymbol{n_1} \times \boldsymbol{n_2}\|}.$$

We need to define a plane orthogonal to this axis, which requires a point. We will arbitrarily choose to make $\boldsymbol{p_1}$ a point on the plane, so our plane is defined by the following equation, where $\boldsymbol{p}$ is any point on the plane.

$$0 = \hat{\boldsymbol{a}} \cdot \boldsymbol{p} - \hat{\boldsymbol{a}} \cdot \boldsymbol{p_1}$$

More generally, the distance from any point $\boldsymbol{p}$ to the plane is

$$D(\boldsymbol{p}) = \hat{\boldsymbol{a}} \cdot \boldsymbol{p} - \hat{\boldsymbol{a}} \cdot \boldsymbol{p_1}.$$

Now we must project our two points and their normal vectors onto this plane. $\boldsymbol{p_1}$ is on the plane by definition, so let's project the others:

$$\begin{aligned}
\boldsymbol{p}_{1,proj} &= \boldsymbol{p}_1 \\
\boldsymbol{p}_{2,proj} &= \boldsymbol{p}_2 - D(\boldsymbol{p}_2) \cdot \hat{\boldsymbol{a}} \\
\boldsymbol{n}_{1,proj} &= \boldsymbol{n}_1 - D(\boldsymbol{p}_{1,proj} + \boldsymbol{n}_1) \cdot \hat{\boldsymbol{a}} \\
\boldsymbol{n}_{2,proj} &= \boldsymbol{n}_2 - D(\boldsymbol{p}_{2,proj} + \boldsymbol{n}_2) \cdot \hat{\boldsymbol{a}}
\end{aligned}$$

Our goal is to determine the point of intersection of the lines which include both surface normal vectors, and in order to do this we will perform a change of basis into the 2D plane. The first basis vector will be arbitrarily chosen as $\boldsymbol{n_1}$.

$$\hat{\boldsymbol{x}}_{plane} = \frac{\boldsymbol{n}_{1,proj}}{\|\boldsymbol{n}_{1,proj}\|}$$

We know the second basis vector must be a vector in the plane (i.e., orthogonal to $\hat{\boldsymbol{a}}$) and be orthogonal to the other basis vector $\hat{\boldsymbol{x}}_{plane}$. Thus, we can find it by taking a cross product.

$$\hat{\boldsymbol{y}}_{plane} = \frac{\hat{\boldsymbol{x}}_{plane} \times \hat{\boldsymbol{a}}}{\|\hat{\boldsymbol{x}}_{plane} \times \hat{\boldsymbol{a}}\|}$$

Now we can construct our change of basis matrix, $\boldsymbol{B} \in \mathbb{R}^{2\times 3}$.

$$\boldsymbol{B} = \begin{bmatrix} \hat{\boldsymbol{x}}_{plane}^T \\ \hat{\boldsymbol{y}}_{plane}^T \end{bmatrix}$$

And finally we can obtain our points and normal vectors in the plane's coordinate system. The points need to be relative to a point on the plane, so again we can use $\boldsymbol{p_1}$ as the origin of this coordinate system without loss of generality.

$$\begin{aligned}
\boldsymbol{p}_{1,plane} &= \boldsymbol{B} \cdot (\boldsymbol{p}_{1,proj} - \boldsymbol{p}_{1,proj}) = \boldsymbol{0} \\
\boldsymbol{p}_{2,plane} &= \boldsymbol{B} \cdot (\boldsymbol{p}_{2,proj} - \boldsymbol{p}_{1,proj}) \\
\boldsymbol{n}_{1,plane} &= \boldsymbol{B} \cdot \boldsymbol{n}_{1,proj} \\
\boldsymbol{n}_{2,plane} &= \boldsymbol{B} \cdot \boldsymbol{n}_{2,proj}
\end{aligned}$$

The center point in the planar coordinate frame $\boldsymbol{c}_{plane} = \begin{bmatrix} c_x & c_y \end{bmatrix}^T$ must be the intersection of the lines formed by $\boldsymbol{n}_{1,plane}$ and $\boldsymbol{n}_{2,plane}$. For simplicity of notation, let

$$\begin{aligned}
\boldsymbol{p}_{1,plane} &= \begin{bmatrix} p_{1,x} & p_{1,y} \end{bmatrix}^T \\
\boldsymbol{p}_{2,plane} &= \begin{bmatrix} p_{2,x} & p_{2,y} \end{bmatrix}^T \\
\boldsymbol{n}_{1,plane} &= \begin{bmatrix} n_{1,x} & n_{1,y} \end{bmatrix}^T \\
\boldsymbol{n}_{2,plane} &= \begin{bmatrix} n_{2,x} & n_{2,y} \end{bmatrix}^T
\end{aligned}$$

The center point is then constrained by the following equations.

$$\begin{aligned}
c_y &= \frac{n_{1,y}}{n_{1,x}} \cdot (c_x - p_{1,x}) + p_{1,y} \\
c_y &= \frac{n_{2,y}}{n_{2,x}} \cdot (c_x - p_{2,x}) + p_{2,y}
\end{aligned}$$

We can subtract these equations to solve for c_x.

$$\begin{aligned}
0 &= \left(\frac{n_{1,y}}{n_{1,x}} - \frac{n_{2,y}}{n_{2,x}} \right) c_x - \frac{n_{1,y}}{n_{1,x}} p_{1,x} + \frac{n_{2,y}}{n_{2,x}} p_{2,x} + p_{1,y} - p_{2,y} \\
\implies 0 &= (n_{1,y} n_{2,x} - n_{2,y} n_{1,x})\, c_x - n_{1,y} n_{2,x} p_{1,x} + n_{2,y} n_{1,x} p_{2,x} + n_{1,x} n_{2,x} (p_{1,y} - p_{2,y}) \\
\implies c_x &= \frac{n_{1,y} n_{2,x} p_{1,x} - n_{2,y} n_{1,x} p_{2,x} - n_{1,x} n_{2,x} (p_{1,y} - p_{2,y})}{n_{1,y} n_{2,x} - n_{2,y} n_{1,x}}
\end{aligned}$$

After solving for c_x, we substitute our result into either original equation to find c_y. Now that we have the center point in the planar coordinate frame, we can transform this back to 3D Cartesian space with the reverse of our change of basis matrix.

$$\boldsymbol{c} = \boldsymbol{B}^T \cdot \boldsymbol{c}_{plane}$$

We now know a point $\boldsymbol{c}$ on the cylinder's axis, as well as the direction of this axis $\hat{\boldsymbol{a}}$. This allows us to determine all of the nearby points in our pointcloud which are on the lateral surface of the cylinder. For each candidate point $\boldsymbol{p_c}$, its distance to the line formed by $\boldsymbol{c}$ and $\hat{\boldsymbol{a}}$ should approximately match the radius, and its normal vector should be approximately parallel to its displacement from the line, and orthogonal to $\hat{\boldsymbol{a}}$.

The radius implied by the previous step is

$$r = \|\boldsymbol{c} - \boldsymbol{p_1}\|$$

and the distance of a point $\boldsymbol{p_c}$ from the axis line is

$$D(\boldsymbol{p_c}) = \frac{\|(\boldsymbol{p_c} - \boldsymbol{c}) \times (\boldsymbol{p_c} - (\boldsymbol{c} + \hat{\boldsymbol{a}}))\|}{\|\hat{\boldsymbol{a}}\|}$$

so our criteria to accept $\boldsymbol{p_c}$ as a point on the lateral surface of the cylinder is

$$|r - D(\boldsymbol{p_c})| < \epsilon_{position}$$

Our second criteria, for the candidate point's surface normal vector $\boldsymbol{n_c}$, is that it must be approximately parallel to the vector joining $\boldsymbol{p_c}$ to the axis at its nearest point. This point on the axis is

$$\boldsymbol{p}_{axis} = \boldsymbol{c} - ((\boldsymbol{c} - \boldsymbol{p_c}) \cdot \hat{\boldsymbol{a}})\, \hat{\boldsymbol{a}},$$

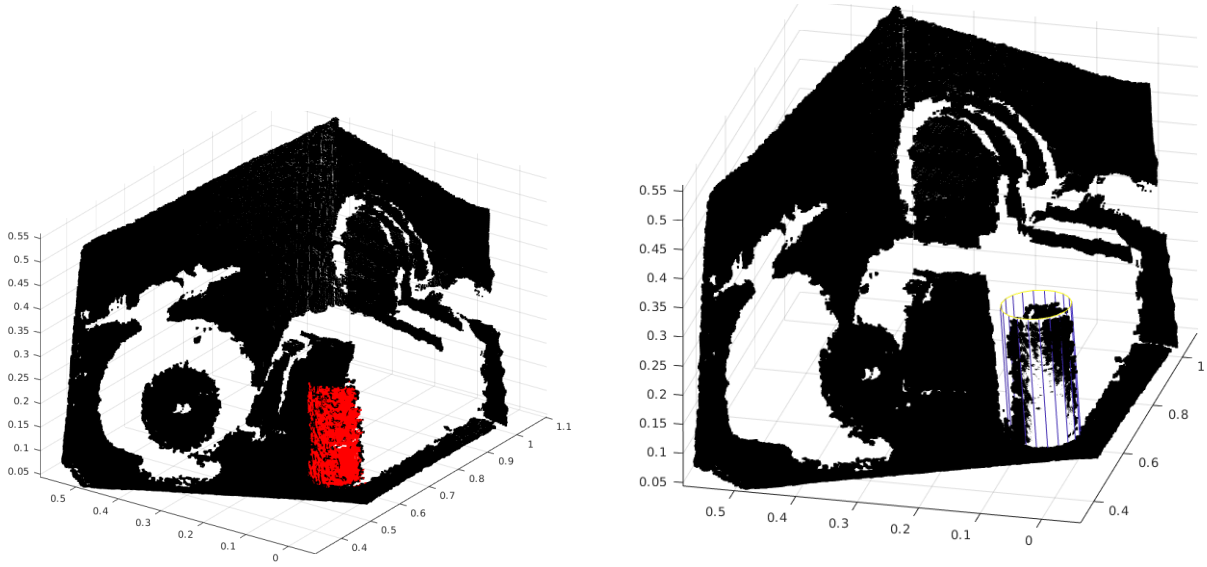
so our condition is

$$a - \left| \boldsymbol{n_c} \cdot \frac{\boldsymbol{p_c} - \boldsymbol{p}_{axis}}{\|\boldsymbol{p_c} - \boldsymbol{p}_{axis}\|} \right| < \epsilon_{angle}$$

For each successful candidate point, we keep track of the $\boldsymbol{p}_{axis}$ point furthest from $\boldsymbol{c}$ in each direction. When we have finished finding all inliers, the distance between these two extreme points is the `length`, and their midpoint is the `center`.

This algorithm is fairly successful in finding the cylinder in the scene, although it runs very slowly compared to my implementations for the planes and sphere. Even when replacing all my geometric calculations with a random sample for the radius, it still takes one or two orders of magnitude longer than the sphere. I believe this is due to the fact that we sample two points and assume both lie on the lateral surface, as opposed to sampling only one point on the sphere, so many of our iterations are doomed from the start, as one or both points may very well not be on the cylinder.

Regardless of these flaws, our results are decent. It correctly identifies the cylinder's set of inliers consistently, though sometimes the axis angle is slightly tilted. To correct for this, I've written a helper function `snap_cylinder` which will randomly perturb the axis angle, and attempt to reduce positional error in the set of inliers. This gives us fairly good results, as the following figures show.



(a) Set of inliers found by RANSAC.

(b) Final geometry returned.

Figure 21: Cylinder in the pointcloud detected by our algorithm.

Our final determination of the cylinder's parameters are:

- `center` $\approx \begin{bmatrix} 0.59 & 0.069 & 0.20 \end{bmatrix}^T$.
- `axis` $\approx \begin{bmatrix} 0.0034 & -0.011 & 0.9999 \end{bmatrix}^T$.

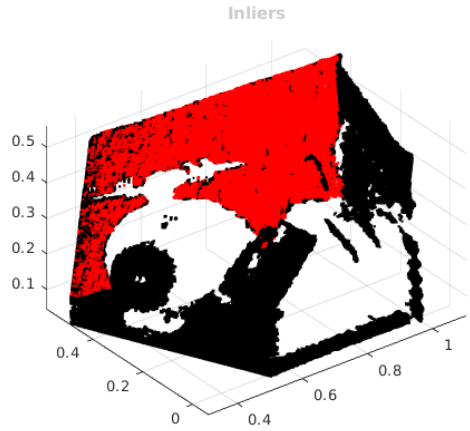
- `radius` ≈ 0.0578
- `length` ≈ 0.1835

Running All of V3 in Series

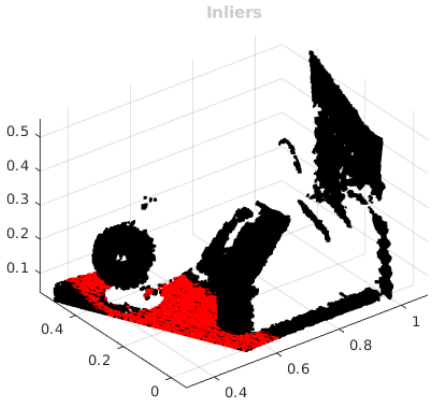
Up until now, each geometry detection algorithm was running on the full pointcloud, which is obviously redundant, since if we already know which points correspond to planes, we don't need to consider them when looking for the cylinder. Running all of these back-to-back, setting all inliers to `NaN` after detection, reduces the search space dramatically by the time we get to the cylinder, and reduces the runtime just as dramatically.

Note: Inliers in RANSAC are those within a certain error $\epsilon_{position}$. To account for points that are very close but not technically inliers, while maintaining the strictness of the original detections, we remove points within $2\epsilon_{position}$. This much more cleanly removes points on the planes and sphere without leaving many noisy artifacts.

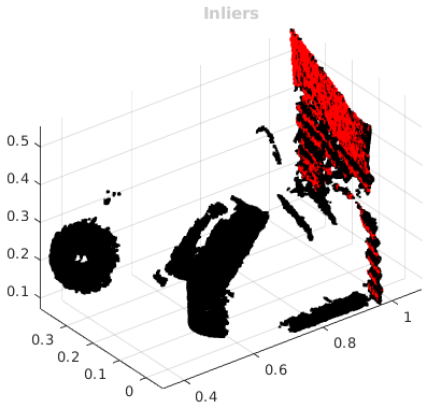
All figures produced in a full run of V3 follow.



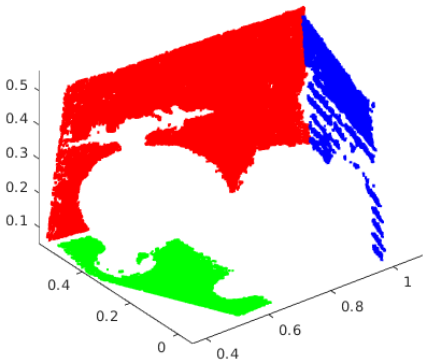
(a) Plane 1 inliers.



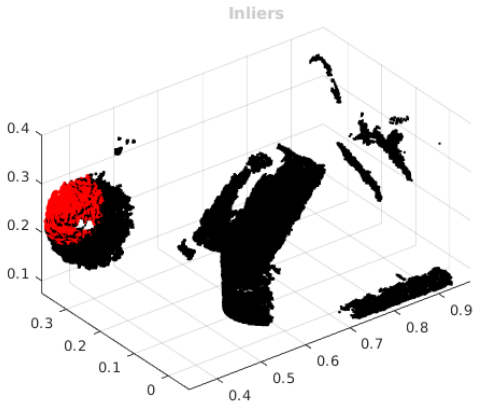
(b) Plane 2 inliers.



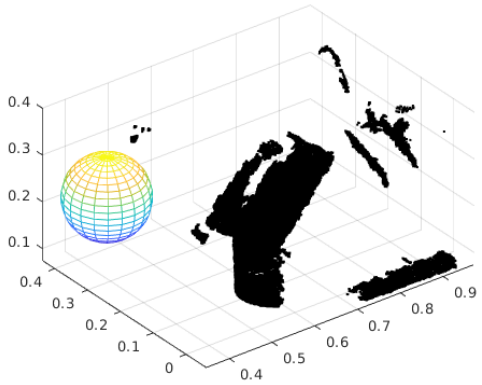
(c) Plane 3 inliers.



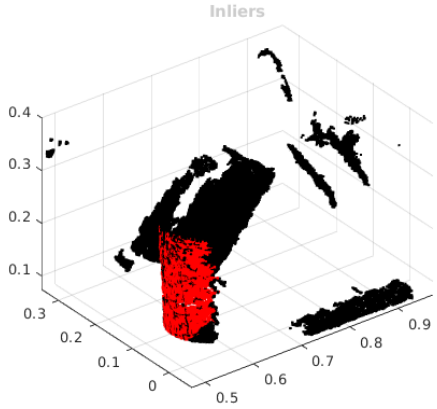
(d) Set of planes.



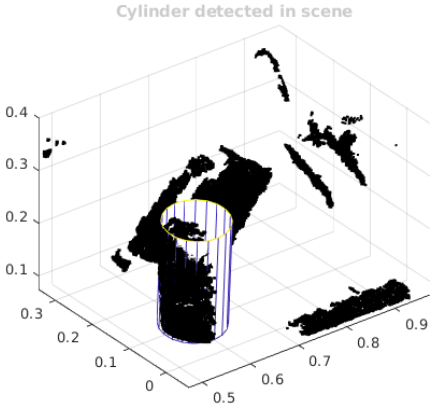
(a) Sphere inliers.



(b) Sphere geometry.



(a) Cylinder inliers.



(b) Cylinder geometry.