ASE 479W Laboratory 4 Report
Path Planning

Harrison Qiu Jin

April 5, 2022

1 Introduction

Autonomous drones have a variety of applications, from agriculture to military to surveying. At the core of every application, the drone must be able to fly from some starting point to some ending point. In many cases, it is desirable for a drone to be able to plan this path itself, without any input from a human operator. This leads to the essential problem of path planning for autonomous drones.

There are a number of different approaches to autonomous path planning, such as optimal control or graph traversal. This paper will explore an approach centered around graph traversal. For the purposes of this paper, it is assumed that the domain over which the drone can fly has already been mapped and discretized into an occupancy grid. An occupancy grid can be represented as a graph, where the nodes are the grid cells and each neighboring grid cell is connected by an edge. A graph search algorithm will then be used to generate a set of waypoints for the quadrotor drone to follow. Finally, a piecewise polynomial function will be fit to those waypoints such that a simulated quadrotor can follow the trajectory described by the polynomial function.

2 Theoretical Analysis

The path planning algorithms explored in this report using the grid shown in Figure 1. The grid can be represented by a graph where the nodes are the grid cells and the cost of each edge is the distance between each grid cell. In this grid, the starting node is the top left grid cell, and the target node is the bottom right grid cell. Red grid cells denote blocked cells, i.e., they are nodes with no edges. The grid is assumed to be homogeneous such that the distance between each cell is the Euclidean distance, and adjacent (non-diagonal) cells are 1 unit apart. Diagonally adjacent cells are considered neighbors. The cost function for a given path is the total length of that path.

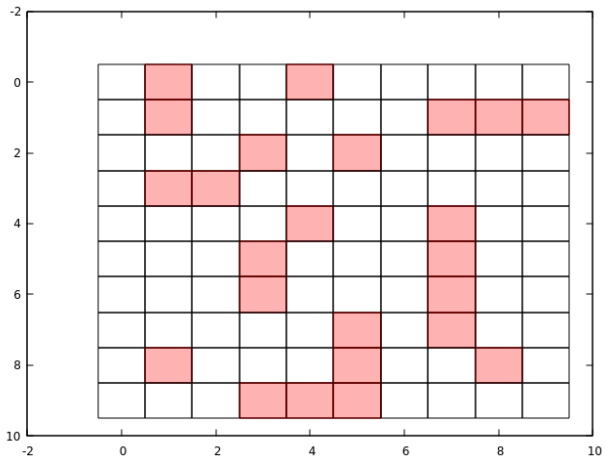


Figure 1: Example grid to test path planning algorithms.

2.1 Depth-First Search

A depth-first search (DFS) searches for the target node by traversing a given path and all branches off that path as far as possible before searching a long a different path. If one or more path to the target node exists, DFS will successfully discover that path. However, it is not guaranteed to be the shortest or lowest cost path. Running DFS on the test grid results in the following path. The

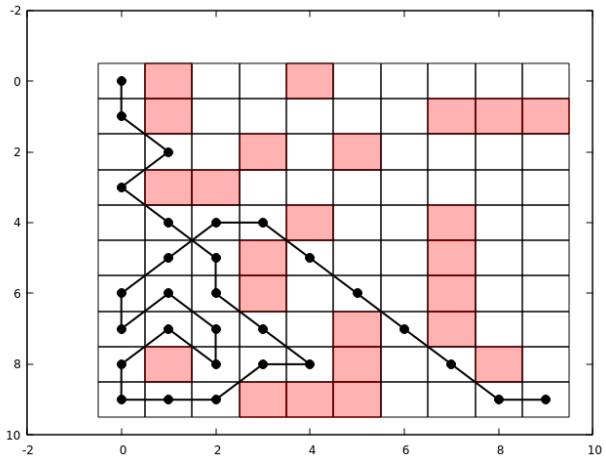


Figure 2: The path returned by the depth-first search algorithm.

path returned by DFS has the following statistics:

```
Number of Nodes Explored: 30
Number of Nodes in Path: 29
Cost of path: 35.4558
Run Time: 63 microseconds
```

It is clear from Figure 2 that the path returned by DFS is not optimal. The length of the optimal path is 13.8995, which can be found using Dijkstra's algorithm.

2.2 Dijkstra's Algorithm

Rather than traversing a path to its conclusion as is done in DFS, Dijkstra's algorithm gives preference to exploring nodes that have the lowest cost. This search strategy means that Dijkstra's algorithm is guaranteed to find the optimal path between two nodes. Running Dijkstra's algorithm on the test grid results in the path shown below in Figure 3. The path returned by Dijkstra's

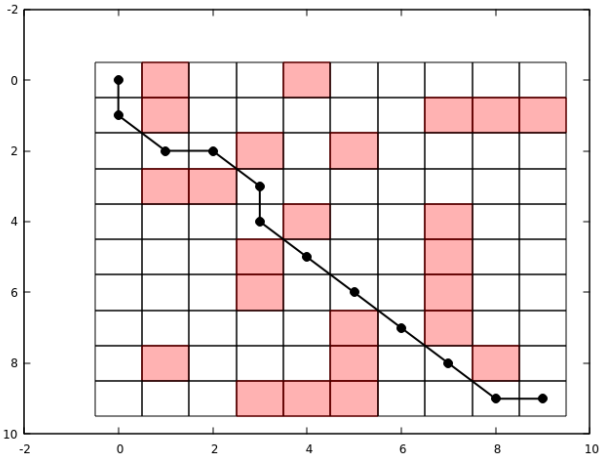


Figure 3: The path returned by Dijkstra's search algorithm.

algorithm has the following statistics:

```
Number of Nodes Explored: 74
Number of Nodes in Path: 12
Cost of path: 13.8995
Run Time: 136 microseconds
```

This path is clearly more optimal when compared to the path found by DFS. However, Dijkstra's algorithm took significantly longer to find the path as compared to DFS. The extra computational cost needed by Dijkstra's algorithm can be seen by observing that both the number of nodes explored and the total run time of Dijkstra's algorithm is higher than that of DFS. Compared to DFS, Dijkstra's algorithm has the additional computational cost of sorting the nodes by cost, which is another contributing factor to the longer runtime. A quadrotor drone is already limited in the compute power it can carry onboard, so this additional runtime is undesirable in this use case. Therefore, Dijkstra's algorithm is not necessarily always the best algorithm to use in practice, even if it is guaranteed to find the optimal path.

2.3 A* Algorithm

The A* algorithm is a refinement of Dijkstra's algorithm. A* uses the same underlying principles, but it adds an additional heuristic to the cost evaluation that estimates the remaining cost to the end node. This can be seen by the fact that running A* with a zero heuristic function yields nearly identical results to running Dijkstra's algorithm.

```
Number of Nodes Explored: 74
Number of Nodes in Path: 12
Cost of path: 13.8995
Run Time: 101 microseconds
```

The negligible difference in run time can be attributed to minor variations of the physical computer. In order to reduce the computational cost of the search algorithm, the heuristic function must not overestimate the true cost, while also providing more information to the algorithm about which node should be explored next. If the heuristic function does overestimate the true cost, A* is not guaranteed to find the optimal path. This can be seen below in Figure 4, which shows the path output by A* using a heuristic function defined by the sum of 1 and the Manhattan distance between the current node and the end node. Since the Manhattan distance is the longest possible path to the end node, this heuristic always overestimates the true cost.

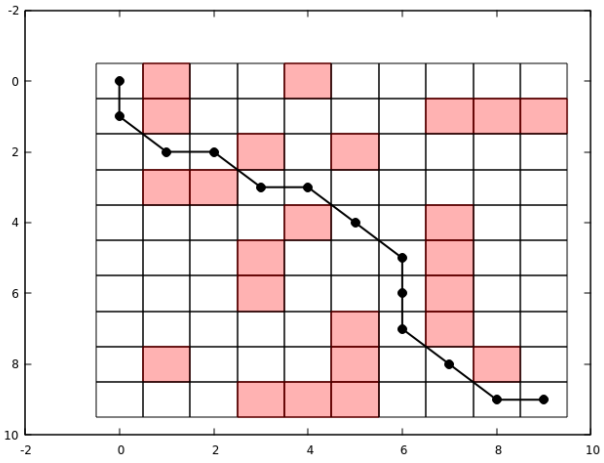


Figure 4: The path returned by the A* search algorithm using an overestimate heuristic function.

```
Number of Nodes Explored: 12
Number of Nodes in Path: 13
Cost of path: 14.4853
Run Time: 20 microseconds
```

The cost of the path is 14.4853, which is more than the optimal path's cost of 13.8995. One possible heuristic that does not overestimate is the difference between the row number of the current node and the row number of the end node. This heuristic causes A* to favor nodes that are on rows

closer to the end node's row. Using this heuristic again yields the optimal path with the following statistics:

```
Number of Nodes Explored: 51
Number of Nodes in Path: 12
Cost of path: 13.8995
Run Time: 73 microseconds
```

The number of nodes explored decreased as compared to a zero heuristic or Dijkstra's algorithm. However, the heuristic function can be optimized even farther. The closer the heuristic is to estimating the true cost of proceeding to the end node, the more efficient the search algorithm can be on average. The Euclidean distance between the node being explored and the end node is one such heuristic. This heuristic works because there is no path between two nodes that could be shorter than a straight line between those nodes, meaning that the true cost will never be overestimated. Using this heuristic for A* on the test grid yields the path shown below in Figure 5. This path has the following statistics:

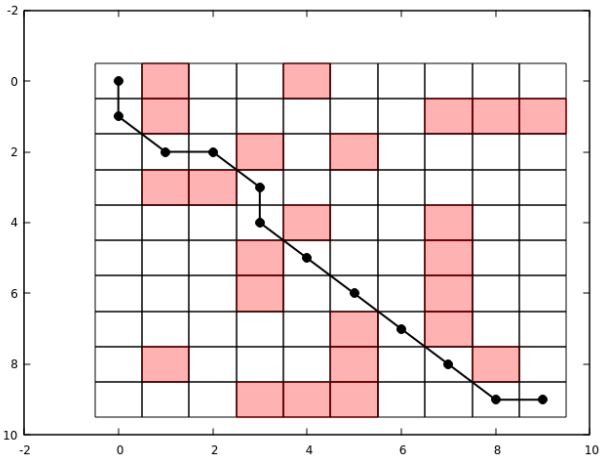


Figure 5: The path returned by the A* search algorithm using Euclidean distance as the heuristic.

```
Number of Nodes Explored: 20
Number of Nodes in Path: 12
Cost of path: 13.8995
Run Time: 40 microseconds
```

As shown in Figure 5, A* finds a path identical to the path produced by Dijkstra's algorithm. This is the expected behavior, as A* is simply an extension of Dijkstra's algorithm. However, the number of nodes explored and the run time of A* is significantly shorter than both DFS and Dijkstra's algorithm. While this is not true in every case, A* has a lower computational cost on average than Dijkstra's algorithm, making it the preferred search algorithm.

All of the search algorithms described above output the positions of the waypoints. However, a quadrotor drone must be provided positions, velocities, and accelerations all as a function of time.

To do that with an algorithm like A*, the path of the drone would need to be interpolated with the positions of the waypoints. Next, the position of the drone could be expressed as a function of time by assigning a time value to each waypoint. Finally, the velocity and acceleration functions could be found by simply differentiating the position function with respect to time. However, assigning time values to each waypoint is non-trivial. The time values determine the net acceleration (and therefore net force) required to act on the drone. An algorithm like A* is agnostic to what the path it generates is used for, so it has no knowledge of the dynamics of the drone or any other physical constraints. Therefore, generating position, velocity, and acceleration as functions of time is not performed within A* in practice.

3 Implementation

All search algorithms described in this paper were implemented using C++. Any data structures referenced in this section refer to the Standard Template Library (STL).

3.1 Depth-First Search

The DFS is implemented using the `stack`, a Last-In First-Out (LIFO) data structure. Exploring the start node first, the current node's neighbors are all added to the `stack`. The next node to be explored is the first node on the `stack`. Because the `stack` is LIFO, the nodes are explored in an order such that an entire neighbor's paths are searched before the next neighbor's paths are searched, hence the name Depth-First Search. A list of explored nodes is also kept to ensure that the same node is not explored multiple times. Once the end node is found, the search is terminated. The DFS algorithm does not account for the cost of the path. To keep track of all nodes that are part of the path, each node that is placed onto the stack is assigned a "parent" node that points to the node that came before. Once the end node is reached, this linked list is traversed to store the entire path that was taken.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is implemented in the same way as DFS, with one main difference. Instead of using a `stack` to store the nodes left to be explored, a minimum priority queue, implemented using C++'s `priority_queue`, is used. This allows the search algorithm to choose which node to explore next using the total cost to reach that node rather than simply choosing the last node that was added. The cost to reach a node's neighbor can be defined as the sum of the cost to reach that node and the cost of the edge between the node and its neighbor.

3.3 A* Algorithm

As an extension of Dijkstra's algorithm, much of the A* algorithm is identical. The only addition is that of the heuristic function. The value of the heuristic function for each node is tracked in the same way that the cost to reach that node is tracked, and the minimum priority queue sorts the nodes by the sum of the heuristic function and cost rather than just the cost. For more information about the heuristic function, see Section 2.3.

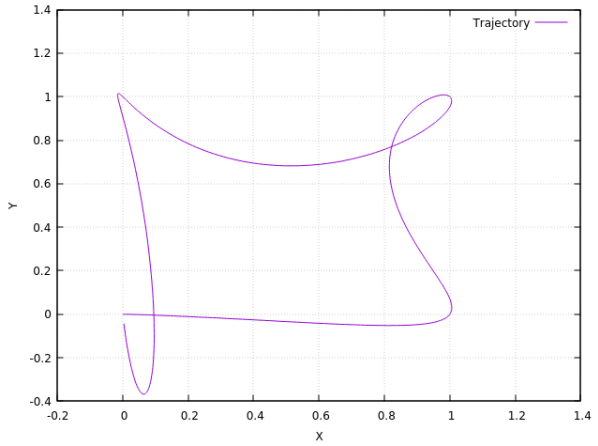
4 Results and Analysis

4.1 Polynomial Smoothing

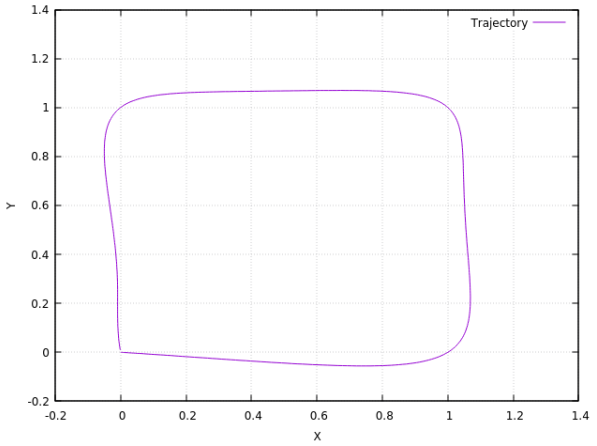
The P4 library was used for polynomial smoothing for the purposes of this lab. Polynomial smoothing is used to generate a position function that is (at least) twice-differentiable so that velocity and acceleration functions can be generated as well. This position function, or path, can also be optimized by minimizing the squared norm of the r^{th} derivative from initial time t_0 to final time t_f.

$$\min \int_{t_0}^{t_f} \frac{d^r}{dt^r} \|p(t)\|^2 \, dt$$

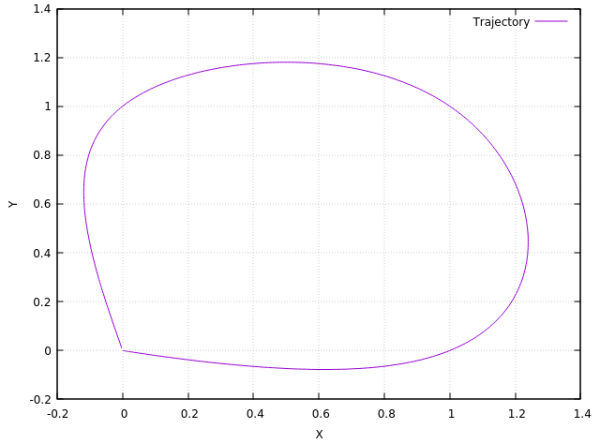
The results of this optimization problem for a square trajectory are shown below in Figure 6. The trajectory follows the points in the order $(0,0), (1,0), (1,1), (0,1)$. The initial velocity and acceleration are set to 0.



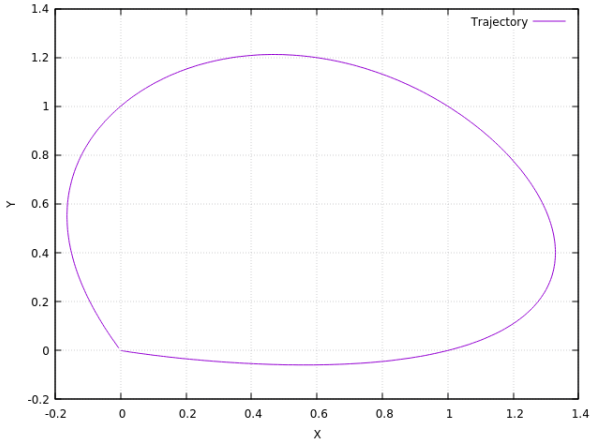
(a) Result of polynomial smoothing while minimizing position.



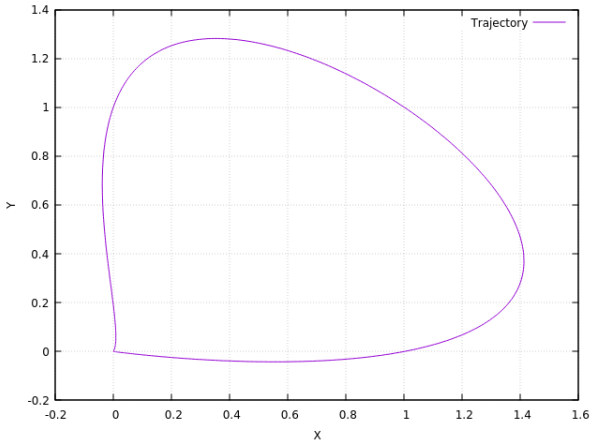
(b) Result of polynomial smoothing while minimizing velocity.



(c) Result of polynomial smoothing while minimizing acceleration.



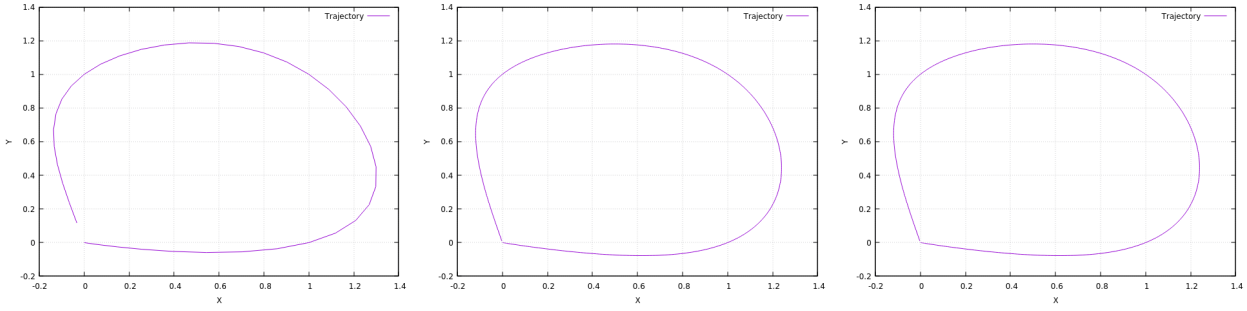
(d) Result of polynomial smoothing while minimizing jerk.



(e) Result of polynomial smoothing while minimizing snap.

Figure 6: Results from optimizing a square trajectory for various orders of position derivatives.

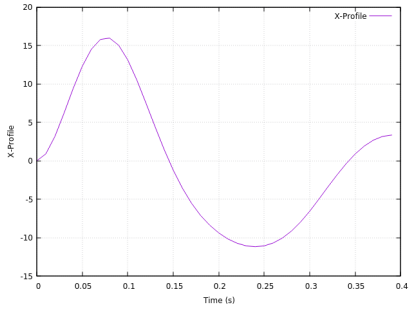
The trajectory in Figure 6a seeks to minimize the magnitude of the position, so the position tends towards the origin as much as possible. Figure 6b shows a rounded square, which is expected given that this shape has the lowest overall velocity. As the order of the derivative that is minimized increases, the shape of the trajectory tends to get more and more rounded. These more rounded trajectories are definitely the easiest for a quadrotor drone to follow. For the purposes of this paper, acceleration, and therefore net force, will be minimized. The next factor to consider when fitting polynomials as a function of time is just how much time it takes to fly the trajectory, otherwise known as the arrival time. Minimizing acceleration, the trajectories for three different arrival times are shown in Figure 7. The arrival times Δt are defined as the time allotted between each waypoint in the square trajectory.



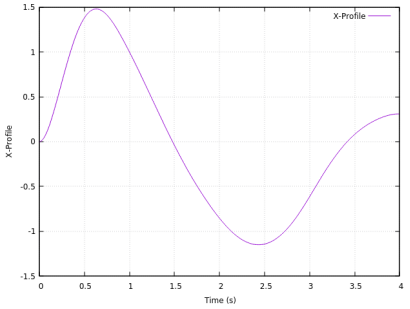
(a) Square trajectory for $\Delta t = 0.1s$. (b) Square trajectory for $\Delta t = 1s$. (c) Square trajectory for $\Delta t = 10s$.

Figure 7: Square trajectories generated for various arrival times while minimizing acceleration

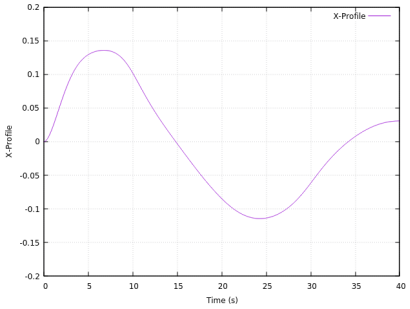
As shown in Figure 7, the arrival time has little effect on the shape of the overall trajectory. The noticeable differences for $\Delta t = 0.1s$ can likely be attributed to the size of the time step used by P4. However, changing the arrival time does have a noticeable impact on the velocity and acceleration functions, as shown in Figure 8.



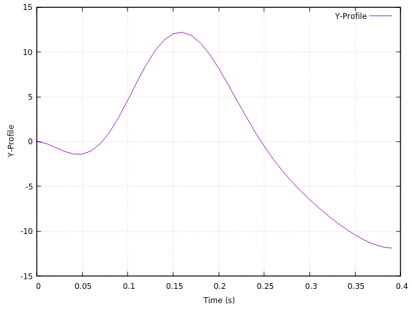
(a) Velocity profile in the x direction for $\Delta t = 0.1s$



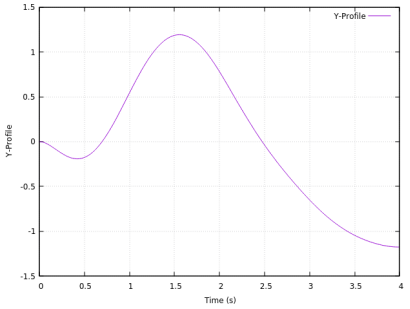
(b) Velocity profile in the x direction for $\Delta t = 1s$



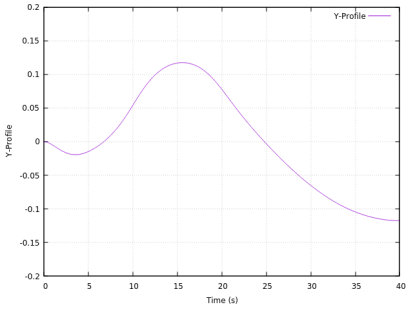
(c) Velocity profile in the x direction for $\Delta t = 10s$



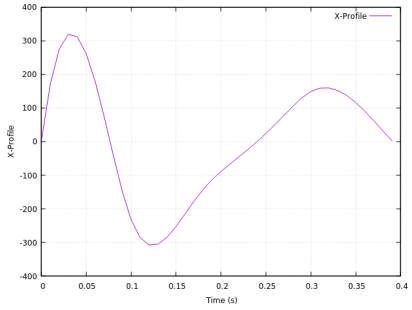
(d) Velocity profile in the y direction for $\Delta t = 0.1s$



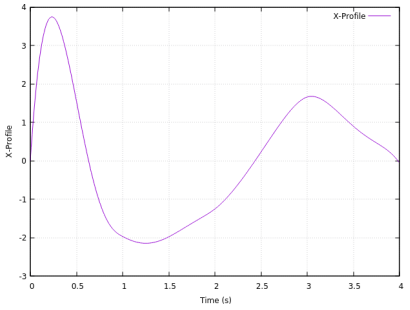
(e) Velocity profile in the y direction for $\Delta t = 1s$



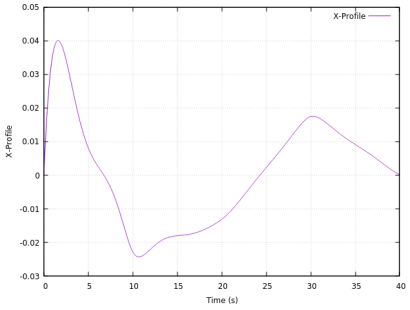
(f) Velocity profile in the y direction for $\Delta t = 10s$



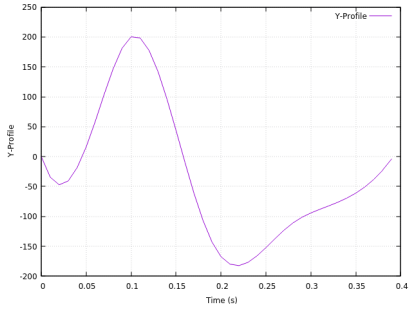
(g) Acceleration profile in the x direction for $\Delta t = 0.1s$



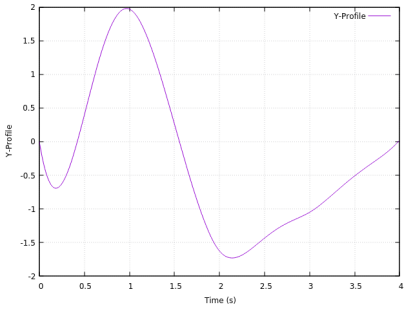
(h) Acceleration profile in the x direction for $\Delta t = 1s$



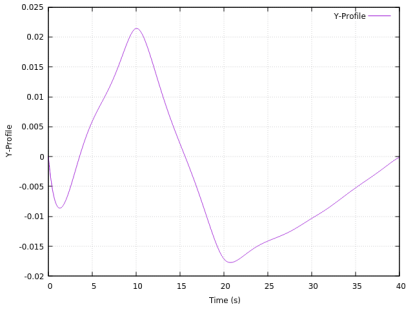
(i) Acceleration profile in the x direction for $\Delta t = 10s$



(j) Acceleration profile in the y direction for $\Delta t = 0.1s$



(k) Acceleration profile in the y direction for $\Delta t = 1s$

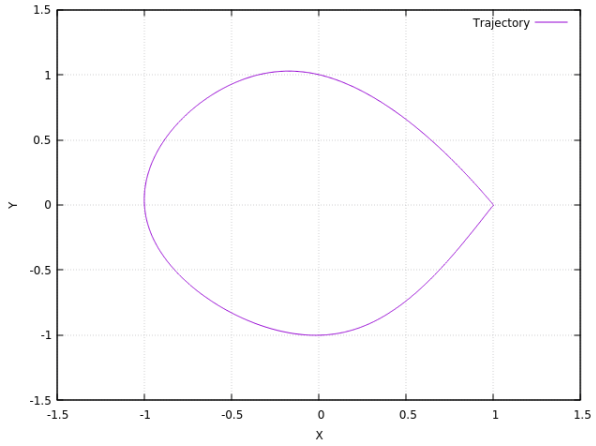


(l) Acceleration profile in the y direction for $\Delta t = 10s$

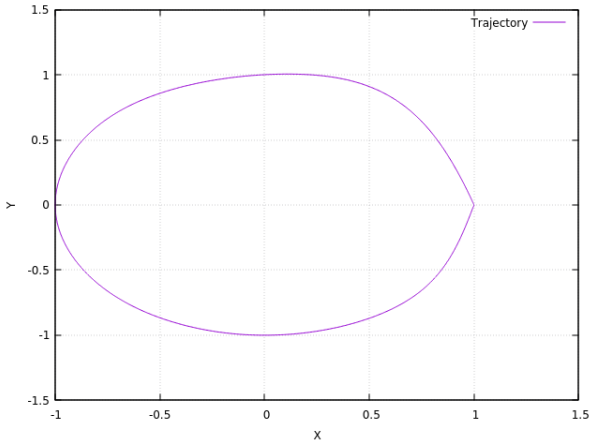
Figure 8: Velocity and acceleration profiles for various arrival times.

The overall shape of the velocity and acceleration profiles do not change significantly with arrival time, which is expected as the trajectory itself does not change much with arrival time. However, noting the scale of the axes in each graph shows that the lower the arrival time is, the more acceleration and velocity is needed to reach the waypoint "on time." In other words, lowering the arrival time increases the amount of force needed to follow the desired trajectory. This could be a limiting factor since quadrotor drones are limited by the properties of its motors, propellers, and other physical components.

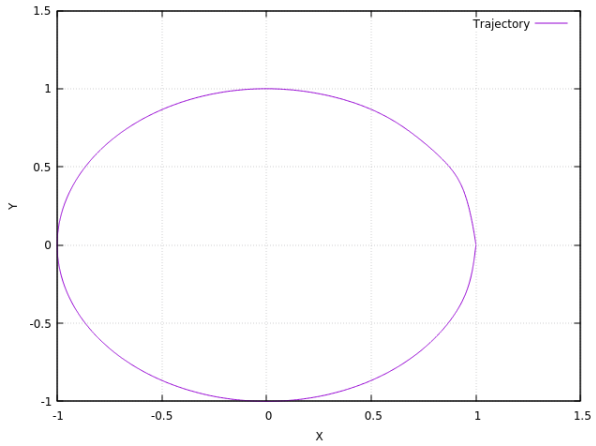
Another factor to consider when fitting a polynomial to a path is how many waypoints to use. For example, a circular path may not be very circular if too few waypoints are used. This is shown below in Figure 9. The starting point for each trajectory is $(1, 0)$, and the initial velocity and acceleration are set to 0. The total time to complete the trajectory is held constant at 10 seconds.



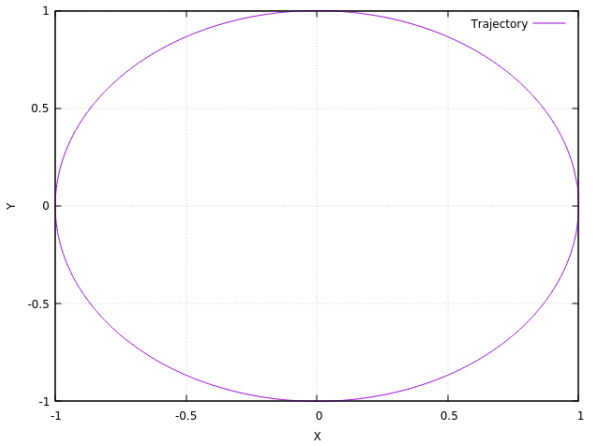
(a) Circular trajectory with 4 waypoints.



(b) Circular trajectory with 8 waypoints.



(c) Circular trajectory with 20 waypoints.



(d) Circular trajectory with 100 waypoints.

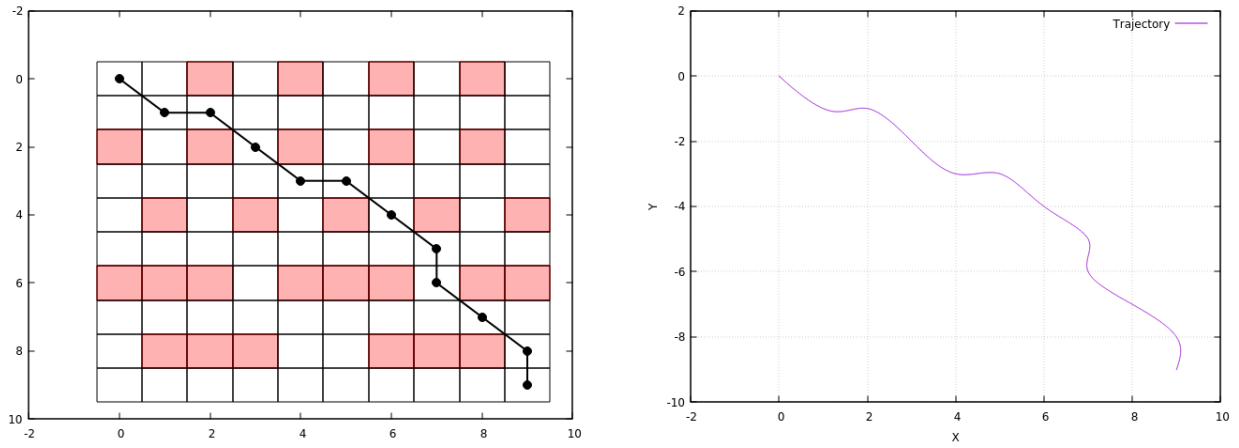
Figure 9: Circular trajectory minimizing acceleration using a varying number of waypoints.

As shown in Figure 9, the more waypoints there are, the more accurate the final trajectory is. This makes sense because providing more waypoints to P4 gives it more information about what the desired trajectory actually is. This effect is especially noticeable at the beginning of the

trajectory in Figures 9a and 9b, where the first portion is pretty much a straight line. Because the initial velocity and acceleration are 0, the trajectory simply travels in a nearly straight line to the first waypoint. Providing more waypoints minimizes this effect.

4.2 Experimentation

After generating a set of waypoints using A* and feeding those waypoints into P4, position, velocity, and acceleration time histories can be extracted to input into the quadrotor simulator. For this experiment, the following path was used. Note that in Figure 10b, the y-component of the trajectory has been inverted to match the coordinate system of the occupancy grid.

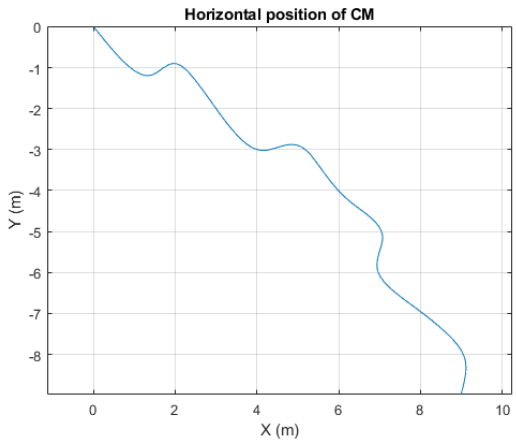


(a) Path planned by A* algorithm.

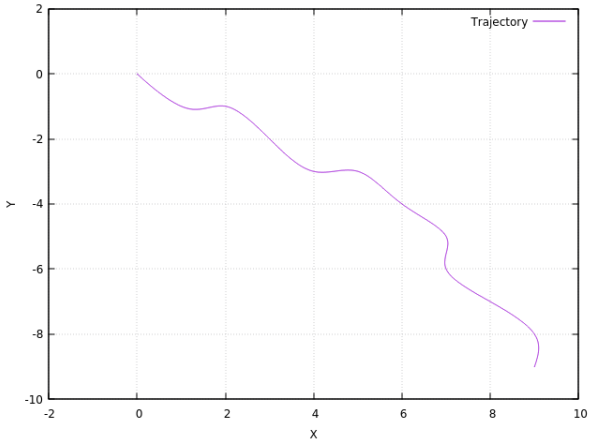
(b) Polynomial trajectory output by P4.

Figure 10: Test trajectory for use in quadrotor simulator.

The position, velocity, and acceleration functions generated by P4 were sampled at 200 hz to input into the quadrotor simulator. All desired z-components were set to 0. The quadrotor simulator was then run using estimation and controllers as described in previous labs. The position of the simulated quadrotor is shown below in Figure 11. Note that the y-component of the trajectory has been inverted to match the coordinate system of the occupancy grid.



(a) Trajectory followed by simulated quadrotor.



(b) Desired trajectory output by P4.

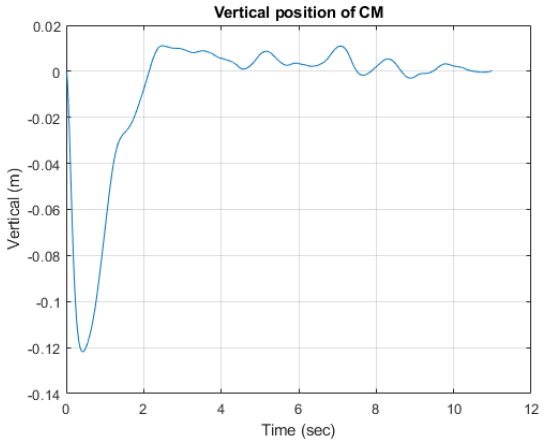


Figure 11: Trajectory followed by simulated quadrotor compared to desired trajectory.

Figure 11 shows that the path planning pipeline described in this paper can indeed be used to plan a trajectory for a simulated quadrotor drone. The momentary dip in height of the drone is expected because the simulated quadrotor drone begins at 0 initial velocity and acceleration.

5 Conclusion

Given an occupancy grid and a starting and ending point, it is possible to generate a trajectory for a quadrotor drone using the pipeline described in this paper. The occupancy grid was first searched using the A* algorithm, which produced a set of waypoints that were used to fit a piecewise polynomial function. This function described a trajectory that was sampled to produce input for the quadrotor simulator. Finally, it was shown that the quadrotor simulator was successfully able to follow the desired trajectory.

The next problem to explore is how to produce an occupancy grid in a way that is both efficient and an accurate representation of the physical world. It is also important to consider the physical constraints that come with physical drones and the real world, and how those constraints can be

input as parameters into this path planning pipeline. These considerations should be explored in future papers.