ASE 389 Laboratory 4 Report

Path Planning

Justin Hart

March 23, 2023

1 Introduction

In previous labs, a full quadrotor simulator was created and a desired path was fed into it. In practice, however, it is unlikely a known, desired trajectory to include velocity and acceleration time histories already exists. It is more likely that a desired trajectory will need to be calculated based on a known map of the environment as well as start and end locations. One path planning algorithm that can be used to do this is A*, as described in [2].

In this laboratory assignment, three different path planning algorithms will be implemented in C++. These are the Depth-First Search, Dijkstra's Algorithm, and the A* Algorithm. Methods of smoothing the trajectories produced will be explored as well.

2 Theoretical Analysis

2.1 Question 6 - A* Time History Generation

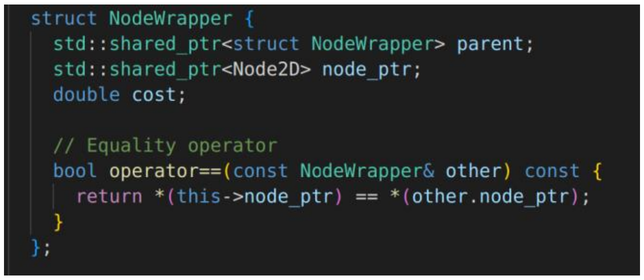
The A* algorithm that is implemented in this lab solves for the positions of waypoints for quadrotors, but the MATLAB simulation requires position, velocity, acceleration, and yaw time histories. The 2D position-only A* implementation could be modified in order to generate the position, velocity, and acceleration time histories as follows. Since the position of each waypoint is known, and the time to reach each waypoint is known, the velocity and acceleration can be calculated by taking the difference in position over time and difference in velocity over time respectively.

This method is not used in practice because the velocity and acceleration would need to be calculated at each waypoint. This would be computationally demanding as finer and finer grids with a significantly larger number of waypoints are needed in practice. By using a polynomial, only the function of the position's information needs to be stored (not a computational burden) and the velocity and acceleration can be found by taking the derivative of this function (which is quite simple for polynomials).

3 Implementation

The implementation of the code for this lab is done using C++, located in the game-engine repository within the Aerial Robotics course virtual machine. The three path planning algorithms are implemented using helper files that are called by the main function, path_planning.cc (see Appendix A for all code).

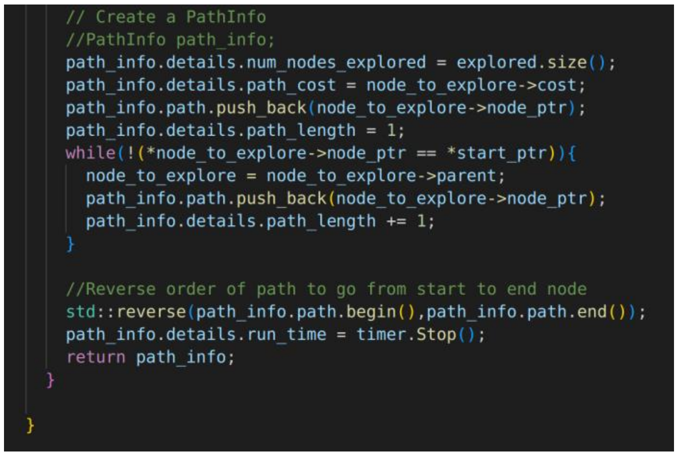
The depth first search initializes a stack of nodes to explore, and a list of explored nodes. While the nodes to explore stack is not empty, the top node from the stack of nodes is explored (unless it has already been explored or is the end node). This means all its neighbors are added to the stack of nodes to explore. This node is then added to the list of explored nodes. A node wrapper structure is used to keep track of each node’s parent as well as the cumulative cost and node ID:



```
struct NodeWrapper {
  std::shared_ptr<struct NodeWrapper> parent;
  std::shared_ptr<Node2D> node_ptr;
  double cost;

  // Equality operator
  bool operator==(const NodeWrapper& other) const {
    return *(this->node_ptr) == *(other.node_ptr);
  }
};
```

This is used to help back out the found path after the while loop is complete. A PathInfo structure is used to store the path information:

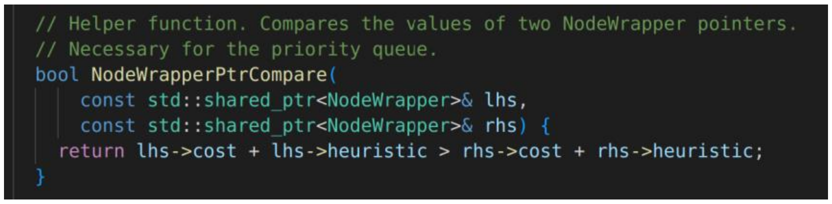


```
      // Create a PathInfo
      //PathInfo path_info;
      path_info.details.num_nodes_explored = explored.size();
      path_info.details.path_cost = node_to_explore->cost;
      path_info.path.push_back(node_to_explore->node_ptr);
      path_info.details.path_length = 1;
      while(!(*node_to_explore->node_ptr == *start_ptr)){
        node_to_explore = node_to_explore->parent;
        path_info.path.push_back(node_to_explore->node_ptr);
        path_info.details.path_length += 1;
      }

      //Reverse order of path to go from start to end node
      std::reverse(path_info.path.begin(),path_info.path.end());
      path_info.details.run_time = timer.Stop();
      return path_info;
    }

  }
```

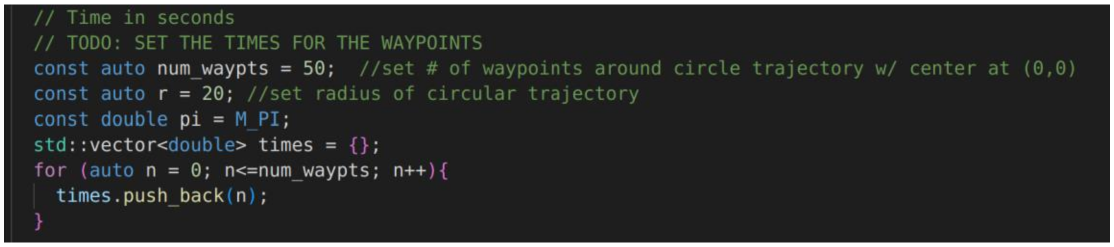
The Dijkstra’s Algorithm implementation is much the same as the depth first search, except that a min priority queue is used for the nodes to explore. This means the next node to explore is chosen based on the node with the least cost (if it hasn’t already been explored or is the end node). The NodeWrapperPtrCompare function is used to compare NodeWrapper pointers for the priority queue:

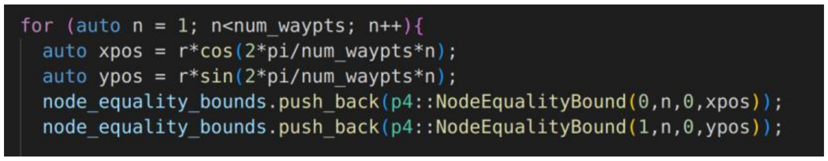


```
// Helper function. Compares the values of two NodeWrapper pointers.
// Necessary for the priority queue.
bool NodeWrapperPtrCompare(
    const std::shared_ptr<NodeWrapper>& lhs,
    const std::shared_ptr<NodeWrapper>& rhs) {
  return lhs->cost + lhs->heuristic > rhs->cost + rhs->heuristic;
}
```

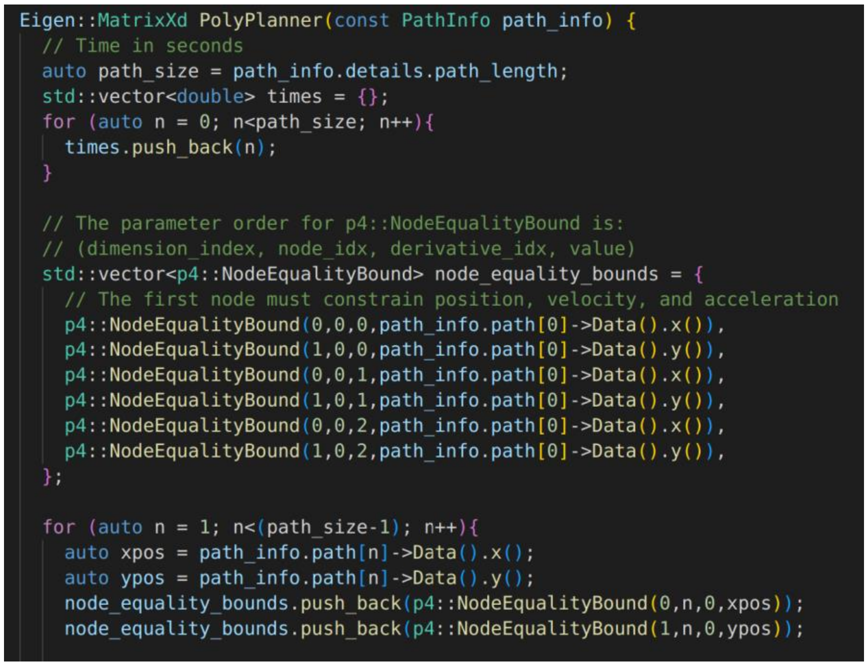
The A* algorithm implementation adds a heuristic function from Dijkstra’s implementation that must be initialized and is added to the cost when determining the order of nodes to explore in the min priority queue.

The polynomial_planning.cc file is used to test the polynomial solver implementation. For the number of waypoints experiment, a for loop was used to set the arrival times as well as waypoints based on the number of waypoints desired on the circular trajectory:


```
// Time in seconds
// TODO: SET THE TIMES FOR THE WAYPOINTS
const auto num_waypts = 50;  //set # of waypoints around circle trajectory w/ center at (0,0)
const auto r = 20; //set radius of circular trajectory
const double pi = M_PI;
std::vector<double> times = {};
for (auto n = 0; n<=num_waypts; n++){
  times.push_back(n);
}
```


```
for (auto n = 1; n<num_waypts; n++){
  auto xpos = r*cos(2*pi/num_waypts*n);
  auto ypos = r*sin(2*pi/num_waypts*n);
  node_equality_bounds.push_back(p4::NodeEqualityBound(0,n,0,xpos));
  node_equality_bounds.push_back(p4::NodeEqualityBound(1,n,0,ypos));
```

The full_stack_planning.cc file is used to combine the A* path planner with the polynomial solver in order to export position, velocity, and acceleration time histories for a planned trajectory. First with the main function, RunAStar is executed, outputting a path_info object. The RunAStar function was copied from the path_planning.cc file and is altered to return the path info. Next, within the main function, a polynomial solver, PolyPlanner, is executed, with the path info used as the input. And Eigen matrix is the output of this function call. The polynomial solver was copied from the polynomial_planning.cc file and altered to set input times and waypoints based on the path info size and chosen path:


```
Eigen::MatrixXd PolyPlanner(const PathInfo path_info) {
  // Time in seconds
  auto path_size = path_info.details.path_length;
  std::vector<double> times = {};
  for (auto n = 0; n<path_size; n++){
    times.push_back(n);
  }

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,0,path_info.path[0]->Data().y()),
    p4::NodeEqualityBound(0,0,1,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,1,path_info.path[0]->Data().y()),
    p4::NodeEqualityBound(0,0,2,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,2,path_info.path[0]->Data().y()),
  };

  for (auto n = 1; n<(path_size-1); n++){
    auto xpos = path_info.path[n]->Data().x();
    auto ypos = path_info.path[n]->Data().y();
    node_equality_bounds.push_back(p4::NodeEqualityBound(0,n,0,xpos));
    node_equality_bounds.push_back(p4::NodeEqualityBound(1,n,0,ypos));
```

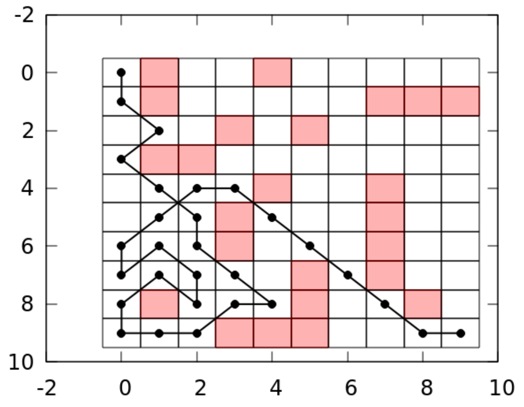
The sampling and plotting sections within the polynomial solver are repeated in order to sample the position, velocity, and acceleration. These sampled histories are placed into an Eigen matrix which is the output of the PolyPlanner function call. Then, this matrix is formatted into a CSV file and output to the current directory. This is done using the provided link to stack overflow in the lab 4 pdf file.

This CSV file is placed within the correct MATLAB directory in order to be used by the topSimulateControl.m script to read in the desired trajectory data. This is done using the built in ‘readmatrix’ function to convert the CSV file to a matrix, which can then be manipulated like any MATLAB matrix.

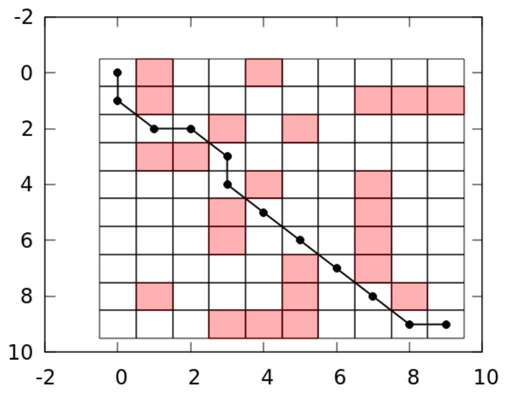
4 Results and Analysis

4.1 Question 1 – Depth First Search Algorithm

The first path planning algorithm considered was the Depth First Search Algorithm (DFS). This is not guaranteed to find the shortest path. This can be shown by looking at Fig. 1, which is the test_grid_medium provided in the lab4 data folder on the virtual machine with start and end points at [0,0] and [9,9] respectively.



(a)



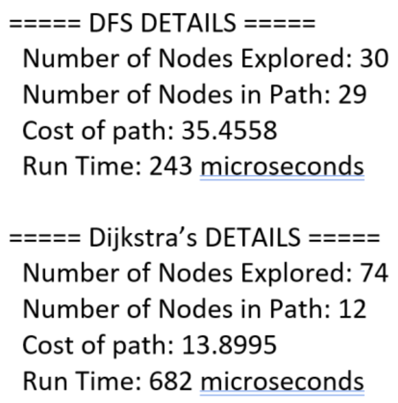
(b)

Figure 1: (a) Depth First Search Algorithm results on test_grid_medium (b) Optimal path using Dijkstra's Algorithm on the same test_grid_medium

The DFS algorithm found a path with a length of 29 nodes, whereas the optimal path was 12 nodes.

4.2 Question 2 – Dijkstra's Algorithm

The second path planning algorithm tested was Dijkstra's Algorithm. This was tested using the test_grid_medium as well and the result is shown in Fig. 1 (b). When compared to the DFS implementation result in Fig. 1 (a), the DFS Algorithm explored fewer nodes (30 compared to 74) and was also faster (243ms run time vs 682ms). Dijkstra's Algorithm found the shortest path at a length of 12 nodes vs. 29 nodes in the path found by the DFS Algorithm. The results of the comparison are shown in Fig. 2 below:



```
===== DFS DETAILS =====
  Number of Nodes Explored: 30
  Number of Nodes in Path: 29
  Cost of path: 35.4558
  Run Time: 243 microseconds

===== Dijkstra’s DETAILS =====
  Number of Nodes Explored: 74
  Number of Nodes in Path: 12
  Cost of path: 13.8995
  Run Time: 682 microseconds
```

Figure 2: DFS Algorithm vs. Dijkstra’s Algorithm results when run using the test_grid_medium.

4.3 Questions 3-5 – A* Algorithm

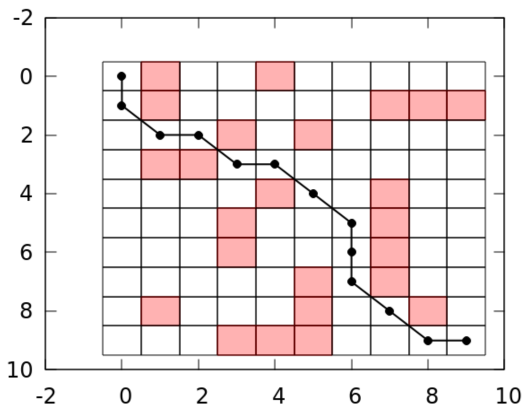
The last path planning algorithm considered was the A* Algorithm. The first heuristic cost function implemented overestimates the true cost of proceeding from a given node to the end node, making it inadmissible. This cost function is listed in (1):

$$h(current, end) = abs(\Delta x) + abs(\Delta y) \tag{1}$$

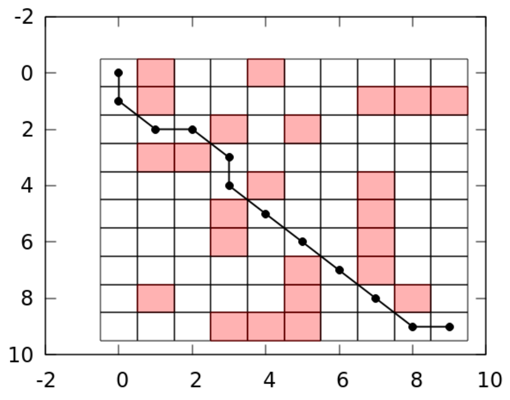
Where the deltas are the differences in x and y positions respectively for the current and end nodes. The performance of this cost function was compared to an admissible cost function, the Euclidean distance, which underestimates the true cost. This equation is shown in (2):

$$h(current, end) = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{2}$$

The A* Algorithm was run on the test_grid_medium provided once using the overestimate cost function and once using the Euclidean distance cost function. The comparison is shown in Figs. 3 and 4.

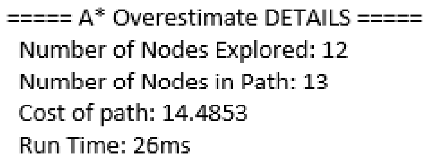


(a)

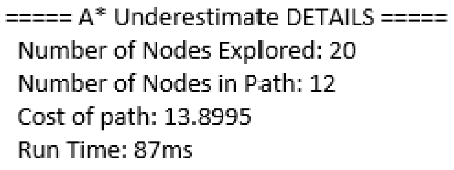


(b)

Figure 3: (a) path generated using A* Algorithm with overestimate cost function. (b) path generated using A* Algorithm with Euclidean distance cost function.



```
===== A* Overestimate DETAILS =====
 Number of Nodes Explored: 12
 Number of Nodes in Path: 13
 Cost of path: 14.4853
 Run Time: 26ms
```



```
===== A* Underestimate DETAILS =====
 Number of Nodes Explored: 20
 Number of Nodes in Path: 12
 Cost of path: 13.8995
 Run Time: 87ms
```

(a) (b)

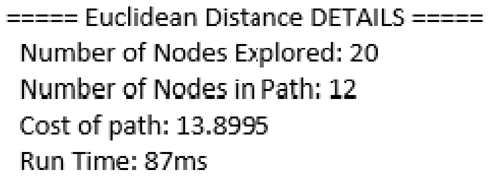
Figure 4: (a) path details using A* Algorithm with overestimate cost function. (b) path details using A* Algorithm with Euclidean distance cost function.

The overestimate cost function explores less nodes, giving it a faster run time. The underestimate cost function finds the shortest and least cost path in this case.

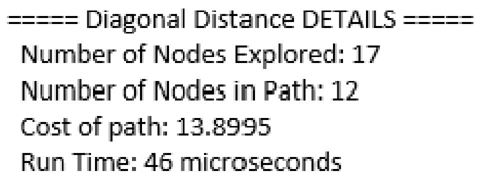
Next, a second admissible heuristic function was designed and compared with the Euclidean distance heuristic in (2). The heuristic chosen was a diagonal distance heuristic as described in [3], used when movement is allowed in eight directions on a grid. The equation for the heuristic is shown in (3):

$$h(current, end) = D * (\Delta x + \Delta y) + (D_2 - 2 * D) * \min(\Delta x, \Delta y) \tag{3}$$

Where D is the length of each grid square (1 in our case), and D_2 is the diagonal distance between each adjacent node ($\sqrt{2}$). This heuristic function was compared to the Euclidean distance heuristic by running the A* Algorithm with each heuristic on the test_grid_medium. Both resulted in the same path found (shown in Fig. 3(b)) and the path details for each are shown in Fig. 5:



```
===== Euclidean Distance DETAILS =====
 Number of Nodes Explored: 20
 Number of Nodes in Path: 12
 Cost of path: 13.8995
 Run Time: 87ms
```



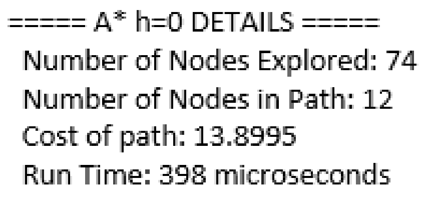
```
===== Diagonal Distance DETAILS =====
 Number of Nodes Explored: 17
 Number of Nodes in Path: 12
 Cost of path: 13.8995
 Run Time: 46 microseconds
```

(a) (b)

Figure 5: (a) path details using A* Algorithm with Euclidean distance cost function. (b) path details using A* Algorithm with diagonal distance cost function.

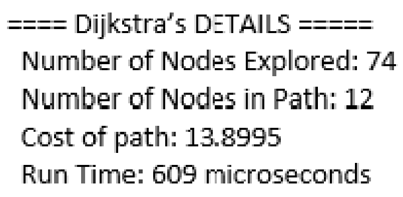
The diagonal distance heuristic function finds the same path as the Euclidean distance heuristic, but explores less nodes and has a slightly faster run time.

Lastly, using a heuristic cost function equal to zero for the A* Algorithm was compared with Dijkstra's Algorithm results when run on test_grid_medium. Both methods determined the same grid path as shown in Fig. 3(b). The respective path details when run using test_grid_medium are shown in Fig. 6 below:



```
===== A* h=0 DETAILS =====
  Number of Nodes Explored: 74
  Number of Nodes in Path: 12
  Cost of path: 13.8995
  Run Time: 398 microseconds
```

(a)



```
==== Dijkstra's DETAILS =====
  Number of Nodes Explored: 74
  Number of Nodes in Path: 12
  Cost of path: 13.8995
  Run Time: 609 microseconds
```

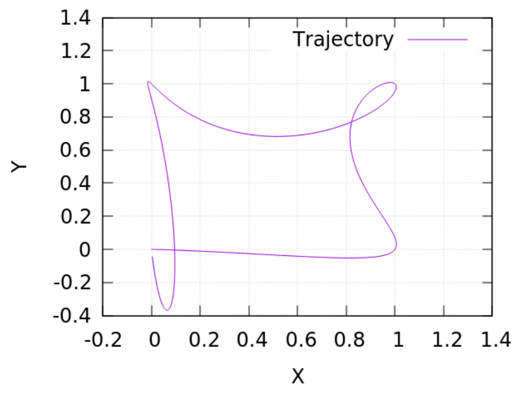
(b)

Figure 6: (a) path details using A* Algorithm with zero cost function. (b) path details using Dijkstra's Algorithm.

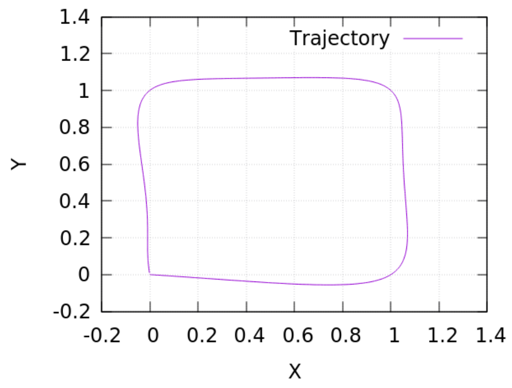
When comparing these methods, the same path was found with the same cost, and the same number of nodes was explored. This makes sense as the A* Algorithm with a cost function equal to zero is Dijkstra's Algorithm.

4.4 Questions 7-9 – Polynomial Smoothing

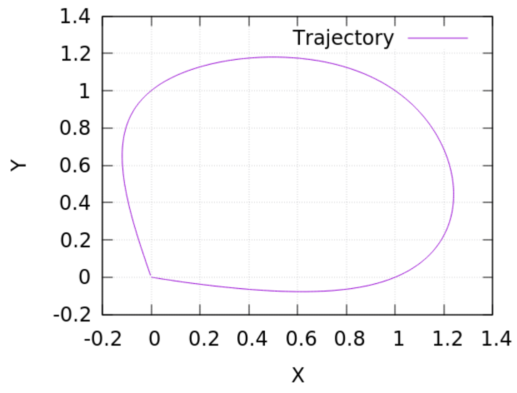
In order to test the polynomial smoothing procedure, first a 2D square layout of waypoints was created using the points (0,0), (1,0), (1,1), and (0,1). A one second time interval was used for each waypoint, and the polynomial solver results were compared when minimizing derivatives 0 through 4. These results are shown in Fig. 7:



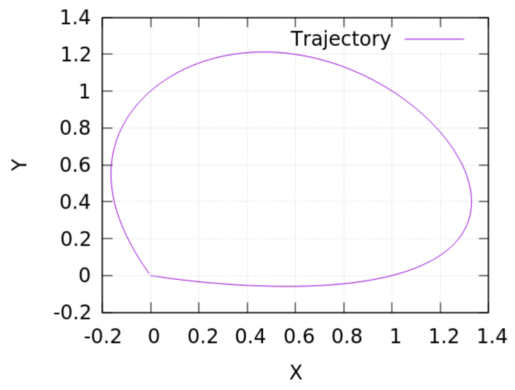
(a)



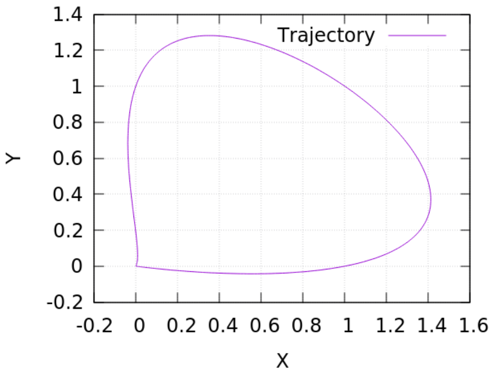
(b)



(c)



(d)

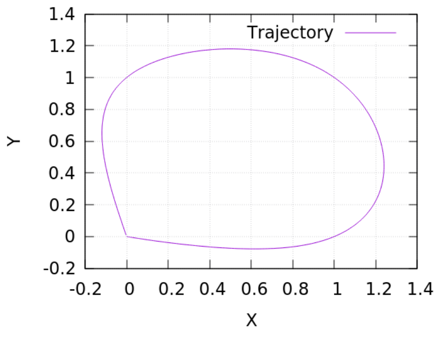


(e)

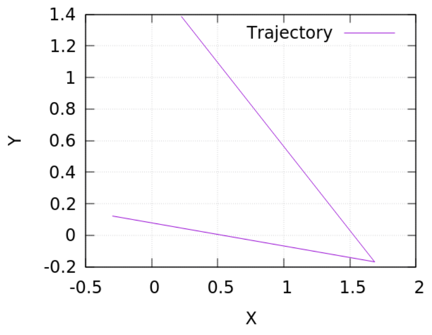
Figure 7: (a) min. 0 (b) min. 1 (c) min. 2 (d) min. 3 (e) min. 4

As the derivative to be minimized increased, the trajectory became more rounded and the overshoot from the desired waypoints increased.

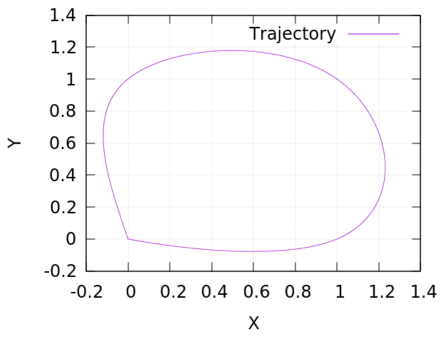
Next, the same square 2D waypoint layout was used, and acceleration (derivative 2) was minimized. The time interval between each waypoint was varied between reasonable, unreasonably short, and unreasonably long. The reasonable arrival time chosen was 1s for each waypoint. The unreasonably short time chosen was .009s. This value was chosen as this would mean the quad would have to travel faster than the fastest existing quad (257mph). The unreasonably long arrival time chosen was 60s as this is two orders of magnitude slower than the speed of a turtle on land of 3mph (.03mph). The comparison of the trajectories in each scenario is shown in Fig. 8:



(a)



(b)



(c)

Figure 8: (a) reasonable arrival time trajectory (b) unreasonably short arrival time trajectory (c) unreasonably long arrival time trajectory

The trajectory produced by the reasonable and unreasonably long arrival times are very close to the same, whereas the unreasonably short arrival time trajectory is choppy and incomplete. A comparison of the velocity and acceleration profiles between each arrival time implementation are shown in Figs. 9 and 10 respectively.

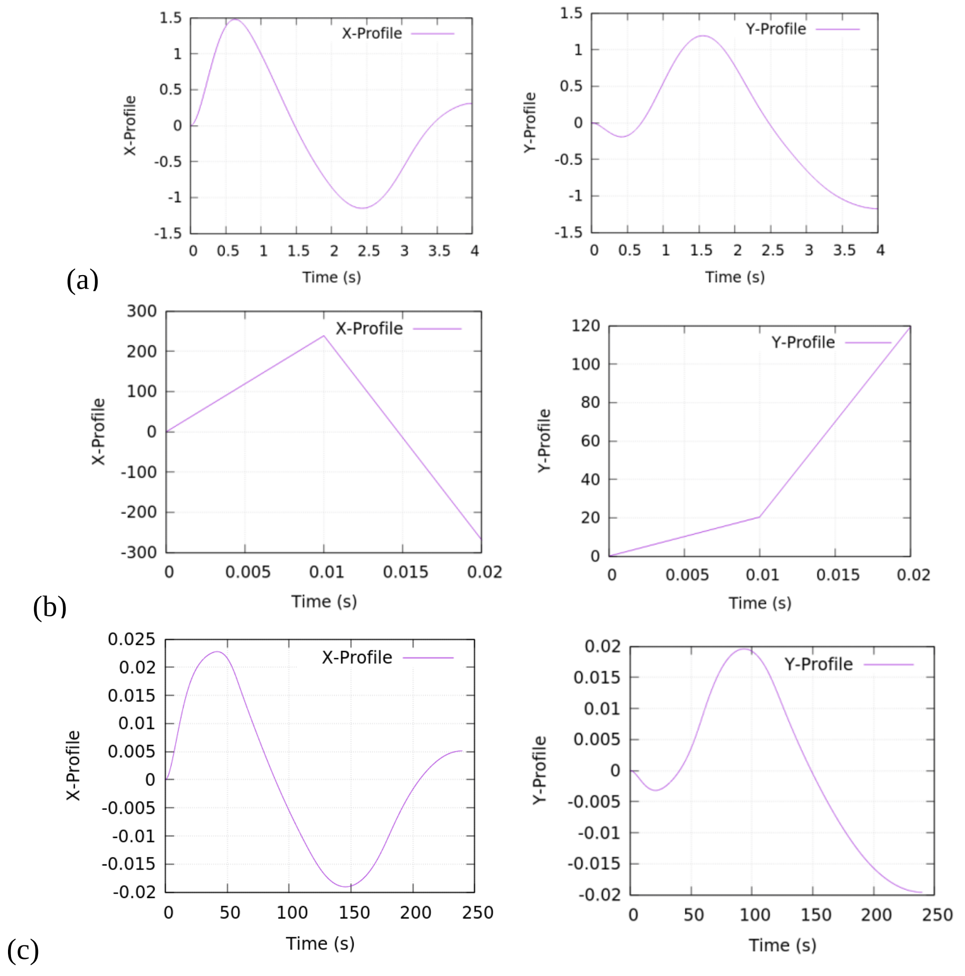


Figure 9: (a) reasonable arrival time velocity profile (b) unreasonably short arrival time velocity profile (c) unreasonably long arrival time velocity profile

For the unreasonably short arrival time, the velocities required are extremely high and are not achievable in reality. The acceleration profile of the unreasonably long arrival time is similar in shape to the reasonable arrival time profiles but hovers around 0.

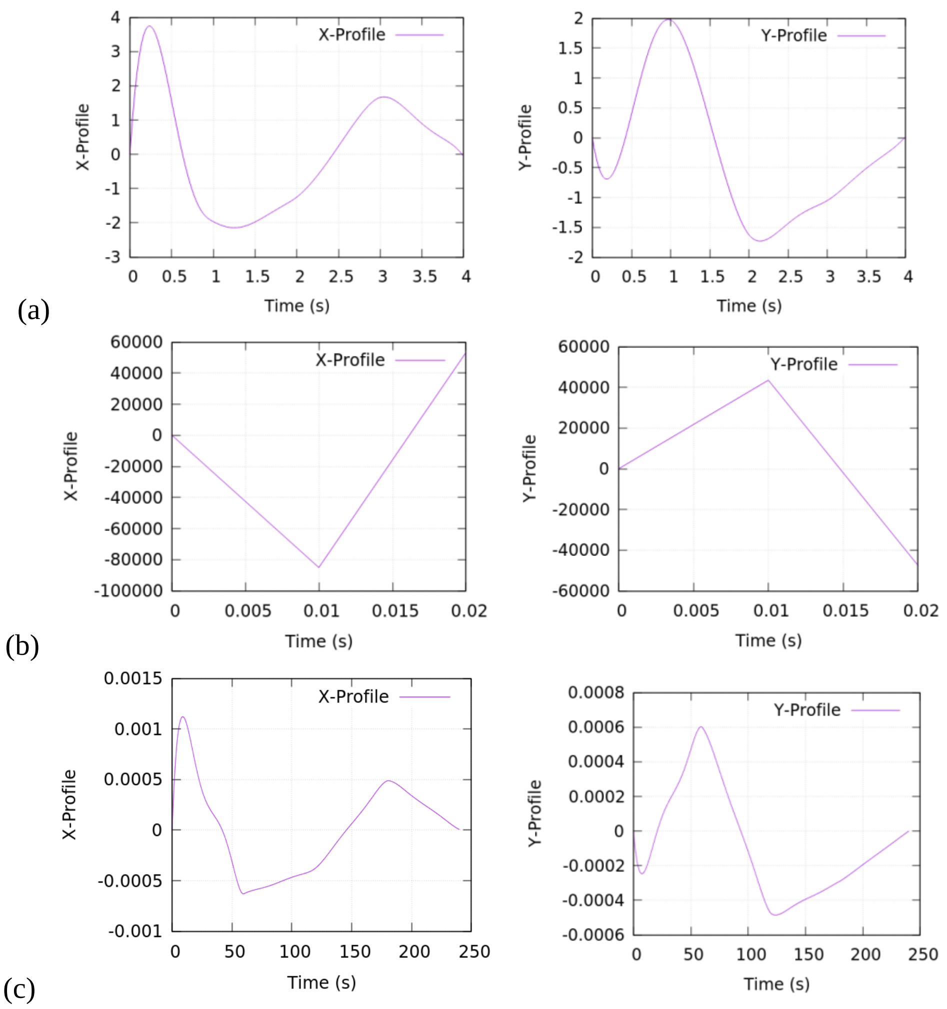


Figure 10: (a) reasonable arrival time acceleration profile (b) unreasonably short arrival time acceleration profile (c) unreasonably long arrival time acceleration profile

For the unreasonably short arrival time, the accelerations required are impossible to reach in reality as they are much higher than the thrust to weight ratio of the quad (~1.7) would allow. The acceleration profile of the unreasonably long arrival time basically goes to 0 over time.

Lastly, a 2D trajectory was created by placing a varying number of waypoints uniformly around a circle. The polynomial solver was implemented over the waypoints, minimizing acceleration. This was repeated with 5, 50, and 500 waypoints around a circle with a radius of 10, centered at (0,0).

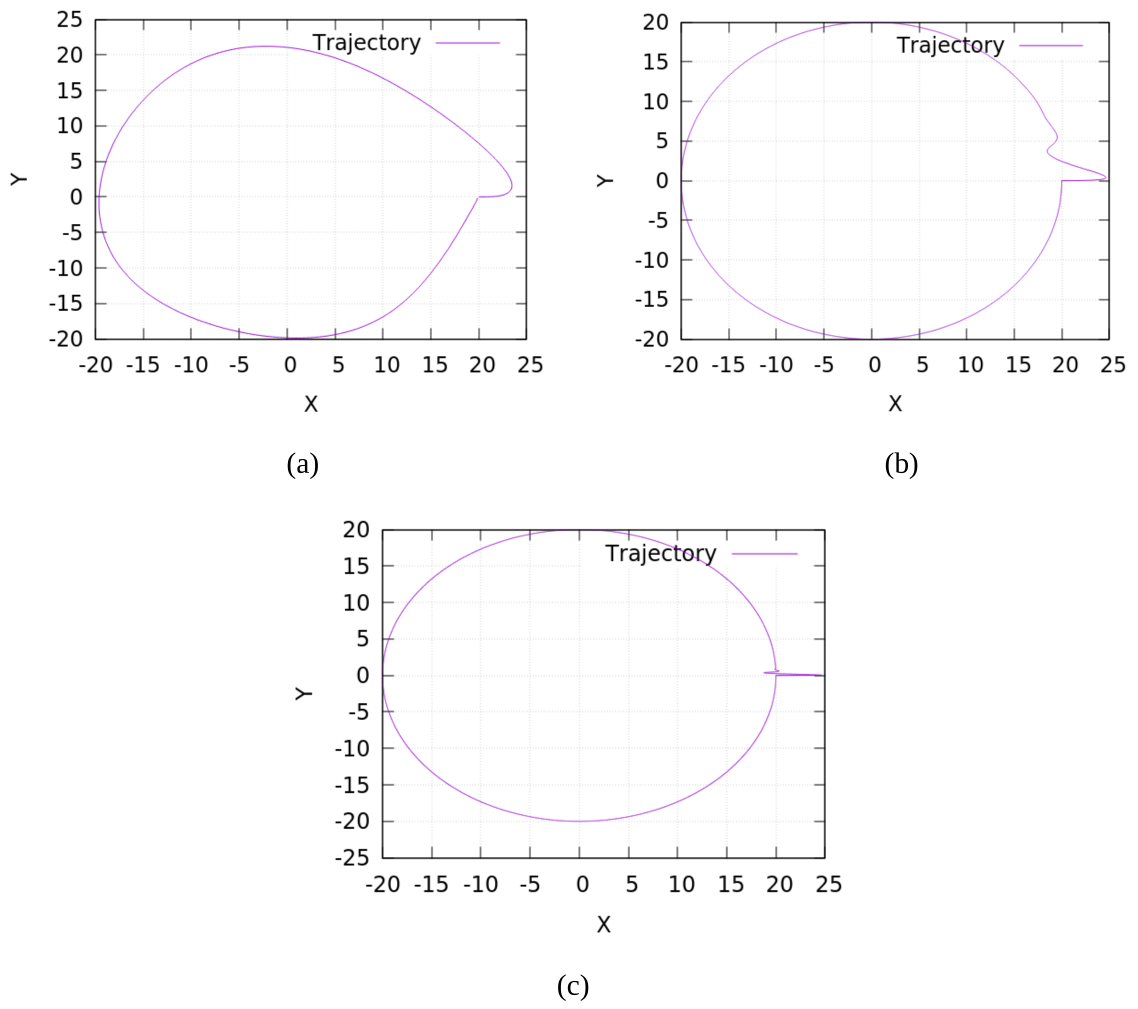
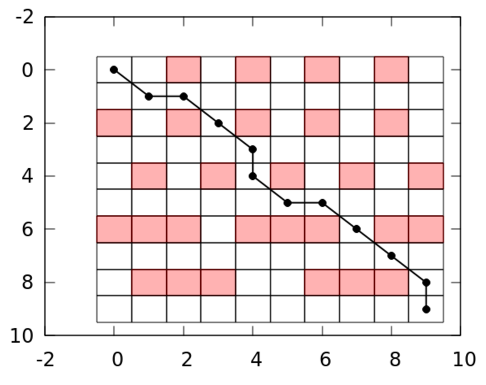


Figure 11: (a) circular trajectory with 5 waypoints (b) circular trajectory with 50 waypoints (c) circular trajectory with 500 waypoints

Increasing the number of waypoints on the edge of the circle gets the trajectory closer and closer to being an actual circle and minimizes the initial path correction. This is because the polynomial solver can solve for a polynomial over a smaller and smaller interval as the number of waypoints increases.

4.5 Question 10 – Full Implementation

In order to test the full implementation of the path planner along with the polynomial solver, the A* solver was run over the full_stack_grid starting and ending at points (0,0) and (9,9) respectively. The waypoints generated were fed into the polynomial solver, and an arrival time to each waypoint was set to 1s while minimizing acceleration. Position, velocity, and acceleration of the trajectory were sampled at 200Hz and this data was input into the MATLAB simulator (assuming a z position of 0 for all time). The path found by the A* solver and the trajectory calculated by the polynomial solver are shown in Fig. 12.



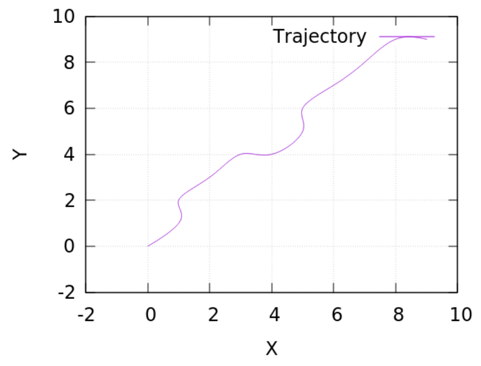
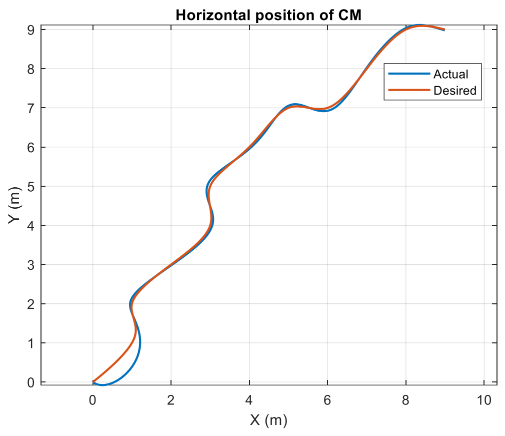


Figure 12: (a) A* solver path over full_stack_grid (b) polynomial solver trajectory using A* solver path

The results of the MATLAB simulator using the position, velocity, and acceleration data sampled from the output trajectory are shown in Fig. 13. The MATLAB simulator used in this case did not implement the estimator.



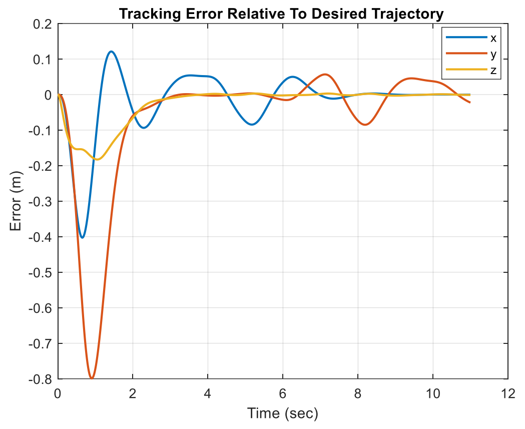


Figure 13: (a) Desired vs. Actual path of the quad's CM in the horizontal, X-Y plane. (b) Tracking error of the quad's CM relative to the desired trajectory in the X, Y, and Z directions

As shown in Figure 13, the quadrotor is able to follow the trajectory reasonably well. The largest offset occurs within the first couple of seconds but the quad settles onto the desired trajectory very well after that.

5 Conclusion

Three different path planning algorithms; Depth-First Search, Dijkstra's, and A*, were implemented in C++. A polynomial solver was tested in order to smooth the trajectories produced by the path planners. The A* algorithm was joined with the polynomial smoother to develop a desired trajectory based on the algorithm's output waypoints, and the trajectory's position, velocity, and acceleration were sampled. This information was fed into the MATLAB quadrotor simulator, and it was shown the quadrotor was able to follow the trajectory.

References

[1] Humphreys, T 2023, *Aerial Robotics Course Notes for ASE 479W/ASE 389*, lecture notes, Aerial Robotics ASE 389, University of Texas at Austin, delivered 10 January 2023.

[2] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136

[3] GeeksforGeeks. (2023, March 8). *A* search algorithm*. GeeksforGeeks. Retrieved March 23, 2023, from https://www.geeksforgeeks.org/a-search-algorithm/

Appendix A: Code

A.1 topSimulateControl.m

```
% Top-level script for calling simulateQuadrotorControl or
% simulateQuadrotorEstimationAndControl

% 'clear all' is needed to clear out persistent variables from run to
run
clear all; clc;
% Seed Matlab's random number: this allows you to simulate with the
same noise
% every time (by setting a nonnegative integer seed as argument to
rng) or
% simulate with a different noise realization every time (by setting
% 'shuffle' as argument to rng).
rng('shuffle');
%rng(1234);
% Assert this flag to call the full estimation and control simulator;
% otherwise, only the control simulator is called
estimationFlag = 0;
% Total simulation time, in seconds
Tsim = 10;
% Update interval, in seconds
delt = 0.005;
% Time vector, in seconds
N = floor(Tsim/delt);
tVec=[0:N-1]'*delt;
% Angular rate of orbit, in rad/sec
n = 2*pi/10;
% Radius of circle, in meters
r = 2;
% Populate reference trajectory
cpp_in = readmatrix('cpptraj.csv');
R.tVec = cpp_in(:,1);
R.rIstar = [cpp_in(:,2),cpp_in(:,3),zeros(length(cpp_in),1)];
R.vIstar = [cpp_in(:,4),cpp_in(:,5),zeros(length(cpp_in),1)];
R.aIstar = [cpp_in(:,6),cpp_in(:,7),zeros(length(cpp_in),1)];
% The desired xI points toward the origin. The code below also
normalizes
% each row in R.xIstar.
R.xIstar = diag(1./vecnorm(R.rIstar'))*(-R.rIstar);
% Matrix of disturbance forces acting on the body, in Newtons,
expressed in I
S.distMat = 0*randn(length(cpp_in),3);
% Initial position in m
S.state0.r = [cpp_in(1,2) cpp_in(1,3) 0]';
% Initial attitude expressed as Euler angles, in radians
S.state0.e = [0 0 pi/4]';
% Initial velocity of body with respect to I, expressed in I, in m/s
S.state0.v = [0 0 0]';
```

```
% Initial angular rate of body with respect to I, expressed in B, in
rad/s
S.state0.omegaB = [0 0 0]';
% Oversampling factor
S.oversampFact = 2;
% Feature locations in the I frame
S.rXIMat = [0,0,0; 0,0,0.7];
%S.rXIMat = [];
% Quadrotor parameters and constants
quadParamsScript;
constantsScript;
sensorParamsScript;
P.quadParams = quadParams;
P.constants = constants;
P.sensorParams = sensorParams;

if(estimationFlag)
  Q = simulateQuadrotorEstimationAndControl(R,S,P);
else
  Q = simulateQuadrotorControl(R,S,P);
end

S2.tVec = Q.tVec;
S2.rMat = Q.state.rMat;
S2.eMat = Q.state.eMat;
S2.plotFrequency = 20;
S2.makeGifFlag = false;
S2.gifFileName = 'testGif.gif';
S2.bounds=2.5*[-1 1 -1 1 -0.1 1];
%visualizeQuad(S2);

%get vector from CM in direction of xI projected onto XY plane
lg = length(Q.state.eMat);
xIvec = [];
for mm = 1:lg
    RBIk = euler2dcm(Q.state.eMat(mm,:)');
    xk = RBIk'*[1;0;0];
    xIvec = [xIvec;xk'];
end

figure(1);clf;
plot(Q.tVec,Q.state.rMat(:,3),'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Vertical (m)');
title('Vertical position of CM');

figure(3);clf;
psiError = unwrap(n*Q.tVec + pi - Q.state.eMat(:,3));
meanPsiErrorInt = round(mean(psiError)/2/pi);
plot(Q.tVec,psiError - meanPsiErrorInt*2*pi,'LineWidth',1.5);
grid on;
xlabel('Time (sec)');
```

```
ylabel('\Delta \psi (rad)');
title('Yaw angle error');

figure(2);clf;
plot(Q.state.rMat(:,1), Q.state.rMat(:,2),'LineWidth',1.5);
axis equal; grid on;hold on;
xlabel('X (m)');
ylabel('Y (m)');
title('Horizontal position of CM');
plot(R.rIstar(:,1),R.rIstar(:,2),'LineWidth',1.5);
legend('Actual','Desired');
%quiver(Q.state.rMat(:,1), Q.state.rMat(:,2),xIvec(:,1),xIvec(:,2));

%calculate performance
xerror = Q.state.rMat(1:2:end,1)-R.rIstar(:,1);
meanxerr = round(mean(xerror));
yerror = Q.state.rMat(1:2:end,2)-R.rIstar(:,2);
meanyerr = round(mean(yerror));
zerror = Q.state.rMat(1:2:end,3)-R.rIstar(:,3);
meanzerr = round(mean(zerror));
figure(4);clf;
plot(R.tVec,xerror-meanxerr,R.tVec,yerror-meanyerr,R.tVec,zerror-
meanzerr,'LineWidth',1.5); grid on;
xlabel('Time (sec)');
ylabel('Error (m)');
title('Tracking Error Relative To Desired Trajectory');
legend('x','y','z');
```

A.2 a_star2d.cc

```
#include <queue>
#include <cmath>
#include <algorithm>

#include "a_star2d.h"

namespace game_engine {
 // Anonymous namespace. Put any file-local functions or variables in here
 namespace {
  // Helper struct that functions as a linked list with data. The linked
  // list represents a path. Data members are a node, a cost to reach that
  // node, and a heuristic cost from the current node to the destination.
  struct NodeWrapper {
   std::shared_ptr<struct NodeWrapper> parent;
   std::shared_ptr<Node2D> node_ptr;

   // True cost to this node
   double cost;

   // Heuristic to end node
   double heuristic;

   // Equality operator
   bool operator==(const NodeWrapper& other) const {
     return *(this->node_ptr) == *(other.node_ptr);
   }
  };

  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;
  //Returns true if the NodeWrapper pointed to by nwPtr is found among
  //those pointed to by the elements of nwPtrVec; otherwise returns false
  bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec){
   for(auto n : nwPtrVec){
     if(*n == *nwPtr)
       return true;
   }
   return false;
  }

  // Helper function. Compares the values of two NodeWrapper pointers.
  // Necessary for the priority queue.
  bool NodeWrapperPtrCompare(
     const std::shared_ptr<NodeWrapper>& lhs,
     const std::shared_ptr<NodeWrapper>& rhs) {
   return lhs->cost + lhs->heuristic > rhs->cost + rhs->heuristic;
  }

  ///////////////////////////////////////////////////////////////////
  // EXAMPLE HEURISTIC FUNCTION
  // YOU WILL NEED TO MODIFY THIS OR WRITE YOUR OWN FUNCTION
  ///////////////////////////////////////////////////////////////////
  double Heuristic(
     const std::shared_ptr<Node2D>& current_ptr,
     const std::shared_ptr<Node2D>& end_ptr) {
     double dx = abs(end_ptr->Data().x()-current_ptr->Data().x());
     double dy = abs(end_ptr->Data().y()-current_ptr->Data().y());
   //return sqrt(pow(dx,2)+pow(dy,2)); //admissible 1, euclidean distance
   //return dx + dy; //overestimate
   return (dx+dy) + (sqrt(2)-2)*std::min(dx,dy); //admissible 2, diagonal distance
   //return 0; //zero heuristic
  }

}

PathInfo AStar2D::Run(
   const Graph2D& graph,
   const std::shared_ptr<Node2D> start_ptr,
   const std::shared_ptr<Node2D> end_ptr) {
  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

  ///////////////////////////////////////////////////////////////////
  // SETUP
  // DO NOT MODIFY THIS
  ///////////////////////////////////////////////////////////////////
  Timer timer;
  timer.Start();

  // Use these data structures
  std::priority_queue<
   NodeWrapperPtr,
   std::vector<NodeWrapperPtr>,
   std::function<bool(
       const NodeWrapperPtr&,
       const NodeWrapperPtr& )>>
     to_explore(NodeWrapperPtrCompare);

  std::vector<NodeWrapperPtr> explored;

  ///////////////////////////////////////////////////////////////////
  // YOUR WORK GOES HERE
  // SOME EXAMPLE CODE INCLUDED BELOW
  ///////////////////////////////////////////////////////////////////
  NodeWrapperPtr node_to_explore;
  PathInfo path_info;

  // Create a NodeWrapperPtr
  NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
  nw_ptr->parent = nullptr;
  nw_ptr->node_ptr = start_ptr;
  nw_ptr->cost = 0;
  nw_ptr->heuristic = Heuristic(start_ptr, end_ptr);
  to_explore.push(nw_ptr);

  while(true){
   if(to_explore.empty()){
     std::cout << " No path to destination found." <<std::endl;
     return path_info;
   }
   node_to_explore = to_explore.top();to_explore.pop();
   if(is_present(node_to_explore,explored)){
     continue;
   }
   if(*node_to_explore->node_ptr == *end_ptr){
     break;
   }
```

```
      else{
        explored.push_back(node_to_explore);
        auto edges = graph.Edges(node_to_explore->node_ptr);
        for(auto edge : edges){
          auto& neighbor = edge.Sink();
          NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
          nw_ptr->parent = node_to_explore;
          nw_ptr->node_ptr = neighbor;
          nw_ptr->cost = node_to_explore->cost + edge.Cost();
          nw_ptr->heuristic = Heuristic(neighbor,end_ptr);
          to_explore.push(nw_ptr);
        }
      }
    }

    // Create a PathInfo
    path_info.details.num_nodes_explored = explored.size();
    path_info.details.path_cost = node_to_explore->cost;
    path_info.path.push_back(node_to_explore->node_ptr);
    path_info.details.path_length = 1;
    while(!(*node_to_explore->node_ptr == *start_ptr)){
      node_to_explore = node_to_explore->parent;
      path_info.path.push_back(node_to_explore->node_ptr);
      path_info.details.path_length += 1;
    }

    std::reverse(path_info.path.begin(),path_info.path.end());
    path_info.details.run_time = timer.Stop();

    // You must return a PathInfo
    return path_info;
  }

}
```

A.3 depth_first_search2d.cc

```
#include <stack>

#include "depth_first_search2d.h"

namespace game_engine {
  // Anonymous namespace. Put any file-local functions or variables in here
  namespace {
    // Helper struct that functions as a linked list with data. The linked
    // list represents a path. Data members are a node and a cost to reach
    // that node.
    struct NodeWrapper {
      std::shared_ptr<struct NodeWrapper> parent;
    std::shared_ptr<Node2D> node_ptr;
    double cost;

    // Equality operator
    bool operator==(const NodeWrapper& other) const {
      return *(this->node_ptr) == *(other.node_ptr);
    }
  };

  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;
  //Returns true if the NodeWrapper pointed to by nwPtr is found among
  //those pointed to by the elements of nwPtrVec; otherwise returns false
  bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec){
    for(auto n : nwPtrVec){
      if(*n == *nwPtr)
        return true;
    }
    return false;
  }
}

PathInfo DepthFirstSearch2D::Run(
    const Graph2D& graph,
    const std::shared_ptr<Node2D> start_ptr,
    const std::shared_ptr<Node2D> end_ptr) {
  using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

  ///////////////////////////////////////////////////////////////////
  // SETUP
  // DO NOT MODIFY THIS
  ///////////////////////////////////////////////////////////////////
  Timer timer;
  timer.Start();

  // Use these data structures
  std::stack<NodeWrapperPtr> to_explore;
  std::vector<NodeWrapperPtr> explored;

  ///////////////////////////////////////////////////////////////////
  // YOUR WORK GOES HERE
  // SOME EXAMPLE CODE INCLUDED BELOW
  ///////////////////////////////////////////////////////////////////
  NodeWrapperPtr node_to_explore;
  PathInfo path_info;

  // Create a NodeWrapperPtr
  NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
```

```
nw_ptr->parent = nullptr;
nw_ptr->node_ptr = start_ptr;
nw_ptr->cost = 0;
to_explore.push(nw_ptr);


while(true){
  if(to_explore.empty()){
    std::cout << " No path to destination found." <<std::endl;
    return path_info;
  }
  node_to_explore = to_explore.top();to_explore.pop();
  if(is_present(node_to_explore,explored)){
    continue;
  }
  if(*node_to_explore->node_ptr == *end_ptr){
    break;
  }
  else{
    explored.push_back(node_to_explore);
    auto edges = graph.Edges(node_to_explore->node_ptr);
    for(auto edge : edges){
      auto& neighbor = edge.Sink();
      NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
      nw_ptr->parent = node_to_explore;
      nw_ptr->node_ptr = neighbor;
      nw_ptr->cost = node_to_explore->cost + edge.Cost();
      to_explore.push(nw_ptr);
    }
  }
}

// Create a PathInfo
//PathInfo path_info;
path_info.details.num_nodes_explored = explored.size();
path_info.details.path_cost = node_to_explore->cost;
path_info.path.push_back(node_to_explore->node_ptr);
path_info.details.path_length = 1;
while(!(*node_to_explore->node_ptr == *start_ptr)){
  node_to_explore = node_to_explore->parent;
  path_info.path.push_back(node_to_explore->node_ptr);
  path_info.details.path_length += 1;
}

//Reverse order of path to go from start to end node
std::reverse(path_info.path.begin(),path_info.path.end());
path_info.details.run_time = timer.Stop();
return path_info;
```

```
 }

}
```

A.4 dijkstra2d.cc

```
#include <queue>

#include "dijkstra2d.h"

namespace game_engine {
  // Anonymous namespace. Put any file-local functions or variables in here
  namespace {
    // Helper struct that functions as a linked list with data. The linked
    // list represents a path. Data members are a node and a cost to reach
    // that node.
    struct NodeWrapper {
      std::shared_ptr<struct NodeWrapper> parent;
      std::shared_ptr<Node2D> node_ptr;
      double cost;

      // Equality operator
      bool operator==(const NodeWrapper& other) const {
        return *(this->node_ptr) == *(other.node_ptr);
      }
    };

    using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;
    //Returns true if the NodeWrapper pointed to by nwPtr is found among
    //those pointed to by the elements of nwPtrVec; otherwise returns false
    bool is_present(const NodeWrapperPtr nwPtr,
                const std::vector<NodeWrapperPtr>& nwPtrVec){
      for(auto n : nwPtrVec){
        if(*n == *nwPtr)
          return true;
      }
      return false;
    }

    // Helper function. Compares the values of two NodeWrapper pointers.
    // Necessary for the priority queue.
    bool NodeWrapperPtrCompare(
        const std::shared_ptr<NodeWrapper>& lhs,
        const std::shared_ptr<NodeWrapper>& rhs) {
      return lhs->cost > rhs->cost;
    }
  }

  PathInfo Dijkstra2D::Run(
  const Graph2D& graph,
  const std::shared_ptr<Node2D> start_ptr,
  const std::shared_ptr<Node2D> end_ptr) {
using NodeWrapperPtr = std::shared_ptr<NodeWrapper>;

///////////////////////////////////////////////////////////////////
// SETUP
// DO NOT MODIFY THIS
///////////////////////////////////////////////////////////////////
Timer timer;
timer.Start();

// Use these data structures
std::priority_queue<
  NodeWrapperPtr,
  std::vector<NodeWrapperPtr>,
  std::function<bool(
      const NodeWrapperPtr&,
      const NodeWrapperPtr& )>>
    to_explore(NodeWrapperPtrCompare);

std::vector<NodeWrapperPtr> explored;

///////////////////////////////////////////////////////////////////
// YOUR WORK GOES HERE
// SOME EXAMPLE CODE INCLUDED BELOW
///////////////////////////////////////////////////////////////////
NodeWrapperPtr node_to_explore;
PathInfo path_info;

// Create a NodeWrapperPtr
NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
nw_ptr->parent = nullptr;
nw_ptr->node_ptr = start_ptr;
nw_ptr->cost = 0;
to_explore.push(nw_ptr);

while(true){
  if(to_explore.empty()){
    std::cout << " No path to destination found." <<std::endl;
    return path_info;
  }
  node_to_explore = to_explore.top();to_explore.pop();
  if(is_present(node_to_explore,explored)){
    continue;
  }
  if(*node_to_explore->node_ptr == *end_ptr){
    break;
```

```
      }
      else{
        explored.push_back(node_to_explore);
        auto edges = graph.Edges(node_to_explore->node_ptr);
        for(auto edge : edges){
          auto& neighbor = edge.Sink();
          NodeWrapperPtr nw_ptr = std::make_shared<NodeWrapper>();
          nw_ptr->parent = node_to_explore;
          nw_ptr->node_ptr = neighbor;
          nw_ptr->cost = node_to_explore->cost + edge.Cost();
          to_explore.push(nw_ptr);
        }
      }
    }

    // Create a PathInfo
    path_info.details.num_nodes_explored = explored.size();
    path_info.details.path_cost = node_to_explore->cost;
    path_info.path.push_back(node_to_explore->node_ptr);
    path_info.details.path_length = 1;
    while(!(*node_to_explore->node_ptr == *start_ptr)){
      node_to_explore = node_to_explore->parent;
      path_info.path.push_back(node_to_explore->node_ptr);
      path_info.details.path_length += 1;
    }

    std::reverse(path_info.path.begin(),path_info.path.end());
    path_info.details.run_time = timer.Stop();

    // You must return a PathInfo
    return path_info;

  }

}
```

A.5 full_stack_planning.cc

```
#include <cstdlib>
#include <vector>
#include <cmath>
#include <string>

#include "a_star2d.h"
#include "occupancy_grid2d.h"
#include "path_info.h"
#include "polynomial_solver.h"
#include "polynomial_sampler.h"
#include "gnuplot-iostream.h"
```

```
#include "gui2d.h"

using namespace game_engine;

///////////////////////////////////////////////////////////////////
//Function Prototypes
///////////////////////////////////////////////////////////////////
PathInfo RunAStar(
   const Graph2D& graph,
   const OccupancyGrid2D* occupancy_grid,
   const std::shared_ptr<Node2D>& start_ptr,
   const std::shared_ptr<Node2D>& end_ptr);

Eigen::MatrixXd PolyPlanner(const PathInfo path_info);

void writeToCSVfile(std::string name, Eigen::MatrixXd matrix);
///////////////////////////////////////////////////////////////////

int main(int argc, char** argv) {
  if(argc != 6) {
    std::cerr << "Usage: ./full_stack_planning occupancy_grid_file row1 col1 row2 col2" << std::endl;
    return EXIT_FAILURE;
  }

  // Parsing input
  const std::string occupancy_grid_file = argv[1];
  const std::shared_ptr<Node2D> start_ptr = std::make_shared<Node2D>(
      Eigen::Vector2d(std::stoi(argv[2]),std::stoi(argv[3])));
  const std::shared_ptr<Node2D> end_ptr = std::make_shared<Node2D>(
      Eigen::Vector2d(std::stoi(argv[4]),std::stoi(argv[5])));

  // Load an occupancy grid from a file
  OccupancyGrid2D occupancy_grid;
  occupancy_grid.LoadFromFile(occupancy_grid_file);

  // Transform an occupancy grid into a graph
  const Graph2D graph = occupancy_grid.AsGraph();

  ///////////////////////////////////////////////////////////////////////////
  // RUN A STAR
  // TODO: Run your A* implementation over the graph and nodes defined above.
  //      This section is intended to be more free-form. Using previous
  //      problems and examples, determine the correct commands to complete
  //      this problem. You may want to take advantage of some of the plotting
  //      and graphing utilities in previous problems to check your solution on
  //      the way.
  ///////////////////////////////////////////////////////////////////////////
  PathInfo path_info = RunAStar(graph, &occupancy_grid, start_ptr, end_ptr);

  ///////////////////////////////////////////////////////////////////////////
  // RUN THE POLYNOMIAL PLANNER
  // TODO: Convert the A* solution to a problem the polynomial solver can
  //      solve. Solve the polynomial problem, sample the solution, figure out
  //      a way to export it to Matlab.
  ///////////////////////////////////////////////////////////////////////////

  Eigen::MatrixXd trajectory = PolyPlanner(path_info);

  writeToCSVfile("trajectory_info.csv",trajectory);

  return EXIT_SUCCESS;
}

///////////////////////////////////////////////////////////////////////////
//Helper Functions
///////////////////////////////////////////////////////////////////////////
PathInfo RunAStar(
    const Graph2D& graph,
    const OccupancyGrid2D* occupancy_grid,
    const std::shared_ptr<Node2D>& start_ptr,
    const std::shared_ptr<Node2D>& end_ptr) {

  std::cout << "============================================" << std::endl;
  std::cout << "=============  RUNNING A Star  =============" << std::endl;
  std::cout << "============================================" << std::endl;

  // Run A*
  AStar2D a_star;
  PathInfo path_info = a_star.Run(graph, start_ptr, end_ptr);

  // Display the solution
  Gui2D gui;
  gui.LoadOccupancyGrid(occupancy_grid);
  gui.LoadPath(path_info.path);
  gui.Display();

  // Print the solution
  path_info.details.Print();

  std::cout << "=====  PATH   =====" << std::endl;
  for(const std::shared_ptr<Node2D>& node: path_info.path) {
    std::cout << "[" << node->Data().transpose() << "]" << std::endl;
  }

  std::cout << std::endl;
  return path_info;

}

Eigen::MatrixXd PolyPlanner(const PathInfo path_info) {
  // Time in seconds
  auto path_size = path_info.details.path_length;
  std::vector<double> times = {};
  for (auto n = 0; n<path_size; n++){
    times.push_back(n);
  }

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,0,path_info.path[0]->Data().y()),
    p4::NodeEqualityBound(0,0,1,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,1,path_info.path[0]->Data().y()),
    p4::NodeEqualityBound(0,0,2,path_info.path[0]->Data().x()),
    p4::NodeEqualityBound(1,0,2,path_info.path[0]->Data().y()),
  };

  for (auto n = 1; n<(path_size-1); n++){
    auto xpos = path_info.path[n]->Data().x();
    auto ypos = path_info.path[n]->Data().y();
    node_equality_bounds.push_back(p4::NodeEqualityBound(0,n,0,xpos));
    node_equality_bounds.push_back(p4::NodeEqualityBound(1,n,0,ypos));

    /*
    std::cout<< "n = " << n <<std::endl;
    std::cout<< "time = " << times[n] <<std::endl;
    std::cout<< "xpos = " << xpos <<std::endl;
    std::cout<< "ypos = " << ypos <<std::endl;
    */
  }

  // The final node constrains position
  node_equality_bounds.push_back(p4::NodeEqualityBound(0,path_size-1,0,path_info.path[path_size-
1]->Data().x()));
  node_equality_bounds.push_back(p4::NodeEqualityBound(1,path_size-1,0,path_info.path[path_size-
1]->Data().y()));

  // std::cout<< node_equality_bounds.size()<<std::endl;

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
```

```
solver_options.num_dimensions = 2;     // 2D
solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
solver_options.continuity_order = 4;   // Require continuity to the 4th order
solver_options.derivative_order = 2;   // Minimize the 2nd order (acceleration)

osqp_set_default_settings(&solver_options.osqp_settings);
solver_options.osqp_settings.polish = true;       // Polish the solution, getting the best answer possible
solver_options.osqp_settings.verbose = false;     // Suppress the printout

// Use p4::PolynomialSolver object to solve for polynomial trajectories
p4::PolynomialSolver solver(solver_options);
const p4::PolynomialSolver::Solution path
  = solver.Run(
      times,
      node_equality_bounds,
      {},
      {});

// Sampling and Plotting
{ // Plot 2D position
  // Options to configure the polynomial sampler with
  p4::PolynomialSampler::Options sampler_options;
  sampler_options.frequency = 200;             // Number of samples per second
  sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

  // Use this object to sample a trajectory
  p4::PolynomialSampler sampler(sampler_options);
  Eigen::MatrixXd samples = sampler.Run(times, path);

  // Plotting tool requires vectors
  std::vector<double> t_hist, x_hist, y_hist;
  for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
    t_hist.push_back(samples(0,time_idx));
    x_hist.push_back(samples(1,time_idx));
    y_hist.push_back(samples(2,time_idx));
  }

  // gnu-iostream plotting library
  // Utilizes gnuplot commands with a nice stream interface
  {
    Gnuplot gp;
    gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
    gp.send1d(boost::make_tuple(x_hist, y_hist));
    gp << "set grid" << std::endl;
    gp << "set xlabel 'X'" << std::endl;
    gp << "set ylabel 'Y'" << std::endl;
    gp << "replot" << std::endl;
  }
```

```
 {
   Gnuplot gp;
   gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
   gp.send1d(boost::make_tuple(t_hist, x_hist));
   gp << "set grid" << std::endl;
   gp << "set xlabel 'Time (s)'" << std::endl;
   gp << "set ylabel 'X-Profile'" << std::endl;
   gp << "replot" << std::endl;
 }
 {
   Gnuplot gp;
   gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
   gp.send1d(boost::make_tuple(t_hist, y_hist));
   gp << "set grid" << std::endl;
   gp << "set xlabel 'Time (s)'" << std::endl;
   gp << "set ylabel 'Y-Profile'" << std::endl;
   gp << "replot" << std::endl;
 }

// Velocity
 // Options to configure the polynomial sampler with
 p4::PolynomialSampler::Options sampler_options_v;
 sampler_options_v.frequency = 200;            // Number of samples per second
 sampler_options_v.derivative_order = 1;       // Derivative to sample (1 = vel)

 // Use this object to sample a trajectory
 p4::PolynomialSampler sampler_v(sampler_options_v);
 Eigen::MatrixXd samples_v = sampler_v.Run(times, path);

 // Plotting tool requires vectors
 std::vector<double> xdot_hist, ydot_hist;
 for(size_t time_idx = 0; time_idx < samples_v.cols(); ++time_idx) {
   xdot_hist.push_back(samples_v(1,time_idx));
   ydot_hist.push_back(samples_v(2,time_idx));
 }

// Acceleration
 // Options to configure the polynomial sampler with
 p4::PolynomialSampler::Options sampler_options_a;
 sampler_options_a.frequency = 200;            // Number of samples per second
 sampler_options_a.derivative_order = 2;       // Derivative to sample (2 = acc)

 // Use this object to sample a trajectory
 p4::PolynomialSampler sampler_a(sampler_options_a);
 Eigen::MatrixXd samples_a = sampler_a.Run(times, path);

 // Plotting tool requires vectors
 std::vector<double> xdotdot_hist, ydotdot_hist;
```

```
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      xdotdot_hist.push_back(samples_a(1,time_idx));
      ydotdot_hist.push_back(samples_a(2,time_idx));
    }

  //Matrix of all histories
    Eigen::MatrixXd time = Eigen::Map<Eigen::MatrixXd>(t_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd xpos = Eigen::Map<Eigen::MatrixXd>(x_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd ypos = Eigen::Map<Eigen::MatrixXd>(y_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd xvel = Eigen::Map<Eigen::MatrixXd>(xdot_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd yvel = Eigen::Map<Eigen::MatrixXd>(ydot_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd xacc = Eigen::Map<Eigen::MatrixXd>(xdotdot_hist.data(),t_hist.size(),1);
    Eigen::MatrixXd yacc = Eigen::Map<Eigen::MatrixXd>(ydotdot_hist.data(),t_hist.size(),1);

    Eigen::MatrixXd
trajectory(time.rows(),time.cols()+xpos.cols()+ypos.cols()+xvel.cols()+yvel.cols()+xacc.cols()+yacc.cols());
    trajectory << time, xpos, ypos, xvel, yvel, xacc, yacc;
    return trajectory;
  }

  //return trajectory;

}


const static Eigen::IOFormat CSVFormat(Eigen::StreamPrecision, Eigen::DontAlignCols, ", ", "\n");
void writeToCSVfile(std::string name, Eigen::MatrixXd matrix){
  std::ofstream file(name.c_str());
  file << matrix.format(CSVFormat);
}
```

A.6 polynomial_planning.cc

```
#include <cstdlib>
#include <vector>
#include <cmath>

#include "polynomial_solver.h"
#include "polynomial_sampler.h"
#include "gnuplot-iostream.h"

///////////////////////////////////////////////////////////////////
// FUNCTION PROTOTYPES
// DO NOT MODIFY
///////////////////////////////////////////////////////////////////
void Example();
void DerivativeExperiments();
void ArrivalTimeExperiments();
void NumWaypointExperiments();
```

```
////////////////////////////////////////////////////////////////////
// MAIN FUNCTION
// TODO: UNCOMMENT THE FUNCTIONS YOU WANT TO RUN
////////////////////////////////////////////////////////////////////
int main(int argc, char** argv) {
  // Example();
  DerivativeExperiments();
  // ArrivalTimeExperiments();
  // NumWaypointExperiments();

  return EXIT_SUCCESS;
}

// Example function that demonstrates how to use the polynomial solver. This
// example creates waypoints in a triangle: (0,0) -- (1,0) -- (1,1) -- (0,0)
void Example() {
  // Time in seconds
  const std::vector<double> times = {0,1,2,3};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constrains position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constrains position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The fourth node constrains position
    p4::NodeEqualityBound(0,3,0,0),
    p4::NodeEqualityBound(1,3,0,0),
  };

  // Options to configure the polynomial solver with
  p4::PolynomialSolver::Options solver_options;
  solver_options.num_dimensions = 2;     // 2D
  solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
  solver_options.continuity_order = 4;   // Require continuity to the 4th order
  solver_options.derivative_order = 2;   // Minimize the 2nd order (acceleration)

  osqp_set_default_settings(&solver_options.osqp_settings);
  solver_options.osqp_settings.polish = true;       // Polish the solution, getting the best answer possible
  solver_options.osqp_settings.verbose = true;     // Suppress the printout

  // Use p4::PolynomialSolver object to solve for polynomial trajectories
  p4::PolynomialSolver solver(solver_options);
  const p4::PolynomialSolver::Solution path
    = solver.Run(
        times,
        node_equality_bounds,
        {},
        {});

  // Sampling and Plotting
  { // Plot 2D position
    // Options to configure the polynomial sampler with
    p4::PolynomialSampler::Options sampler_options;
    sampler_options.frequency = 100;             // Number of samples per second
    sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

    // Use this object to sample a trajectory
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
```

```
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void DerivativeExperiments() {
  // Time in seconds
  // TODO: SET THE TIMES FOR THE WAYPOINTS
  const std::vector<double> times = {0,1,2,3,4};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ///////////////////////////////////////////////////////////////////
    // TODO: CREATE A SQUARE TRAJECTORY
    ///////////////////////////////////////////////////////////////////
    // creates waypoints in a square: (0,0) -- (1,0) -- (1,1) -- (0,1) -- (0,0)
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
    p4::NodeEqualityBound(0,0,2,0),
    p4::NodeEqualityBound(1,0,2,0),

    // The second node constrains position
    p4::NodeEqualityBound(0,1,0,1),
    p4::NodeEqualityBound(1,1,0,0),

    // The third node constrains position
    p4::NodeEqualityBound(0,2,0,1),
    p4::NodeEqualityBound(1,2,0,1),

    // The fourth node constrains position
  p4::NodeEqualityBound(0,3,0,0),
  p4::NodeEqualityBound(1,3,0,1),

  // The fifth node constrains position
  p4::NodeEqualityBound(0,4,0,0),
  p4::NodeEqualityBound(1,4,0,0),
};

// Options to configure the polynomial solver with
p4::PolynomialSolver::Options solver_options;
solver_options.num_dimensions = 2;     // 2D
solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
solver_options.continuity_order = 4;   // Require continuity to the 4th order
solver_options.derivative_order = 0;   // TODO: VARY THE DERIVATIVE ORDER

osqp_set_default_settings(&solver_options.osqp_settings);
solver_options.osqp_settings.polish = true;       // Polish the solution, getting the best answer possible
solver_options.osqp_settings.verbose = false;     // Suppress the printout

// Use p4::PolynomialSolver object to solve for polynomial trajectories
p4::PolynomialSolver solver(solver_options);
const p4::PolynomialSolver::Solution path
  = solver.Run(
      times,
      node_equality_bounds,
      {},
      {});

// Sampling and Plotting
{ // Plot 2D position
  // Options to configure the polynomial sampler with
  p4::PolynomialSampler::Options sampler_options;
  sampler_options.frequency = 100;            // Number of samples per second
  sampler_options.derivative_order = 0;       // Derivative to sample (0 = pos)

  // Use this object to sample a trajectory
  p4::PolynomialSampler sampler(sampler_options);
  Eigen::MatrixXd samples = sampler.Run(times, path);

  // Plotting tool requires vectors
  std::vector<double> t_hist, x_hist, y_hist;
  for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
    t_hist.push_back(samples(0,time_idx));
    x_hist.push_back(samples(1,time_idx));
    y_hist.push_back(samples(2,time_idx));
  }

  // gnu-iostream plotting library
```

```
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void ArrivalTimeExperiments() {
  // Time in seconds
  // TODO: SET THE TIMES FOR THE WAYPOINTS
  const std::vector<double> times = {0,1,2,3,4};

  // The parameter order for p4::NodeEqualityBound is:
  // (dimension_index, node_idx, derivative_idx, value)
  const std::vector<p4::NodeEqualityBound> node_equality_bounds = {
    ///////////////////////////////////////////////////////////////////
    // TODO: CREATE A SQUARE TRAJECTORY
    ///////////////////////////////////////////////////////////////////
    // creates waypoints in a square: (0,0) -- (1,0) -- (1,1) -- (0,1) -- (0,0)
    // The first node must constrain position, velocity, and acceleration
    p4::NodeEqualityBound(0,0,0,0),
    p4::NodeEqualityBound(1,0,0,0),
    p4::NodeEqualityBound(0,0,1,0),
    p4::NodeEqualityBound(1,0,1,0),
  p4::NodeEqualityBound(0,0,2,0),
  p4::NodeEqualityBound(1,0,2,0),

  // The second node constrains position
  p4::NodeEqualityBound(0,1,0,1),
  p4::NodeEqualityBound(1,1,0,0),

  // The third node constrains position
  p4::NodeEqualityBound(0,2,0,1),
  p4::NodeEqualityBound(1,2,0,1),

  // The fourth node constrains position
  p4::NodeEqualityBound(0,3,0,0),
  p4::NodeEqualityBound(1,3,0,1),

  // The fifth node constrains position
  p4::NodeEqualityBound(0,4,0,0),
  p4::NodeEqualityBound(1,4,0,0),
};

// Options to configure the polynomial solver with
p4::PolynomialSolver::Options solver_options;
solver_options.num_dimensions = 2;     // 2D
solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
solver_options.continuity_order = 4;   // Require continuity to the 4th order
solver_options.derivative_order = 2;   // Minimize ACCELERATION

osqp_set_default_settings(&solver_options.osqp_settings);
solver_options.osqp_settings.polish = true;       // Polish the solution, getting the best answer possible
solver_options.osqp_settings.verbose = false;     // Suppress the printout

// Use p4::PolynomialSolver object to solve for polynomial trajectories
p4::PolynomialSolver solver(solver_options);
const p4::PolynomialSolver::Solution path
  = solver.Run(
      times,
      node_equality_bounds,
      {},
      {});

// Sampling and Plotting
{ // Plot 2D position
  // Options to configure the polynomial sampler with
  p4::PolynomialSampler::Options sampler_options;
  sampler_options.frequency = 100;             // Number of samples per second
  sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

  // Use this object to sample a trajectory
```

```
    p4::PolynomialSampler sampler(sampler_options);
    Eigen::MatrixXd samples = sampler.Run(times, path);

    // Plotting tool requires vectors
    std::vector<double> t_hist, x_hist, y_hist;
    for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}

void NumWaypointExperiments() {
  // Time in seconds
  // TODO: SET THE TIMES FOR THE WAYPOINTS
  const auto num_waypts = 50;  //set # of waypoints around circle trajectory w/ center at (0,0)
  const auto r = 20; //set radius of circular trajectory
const double pi = M_PI;
std::vector<double> times = {};
for (auto n = 0; n<=num_waypts; n++){
  times.push_back(n);
}

//std::vector<double> times = {0,1,2,3,4,5,6};

// The parameter order for p4::NodeEqualityBound is:
// (dimension_index, node_idx, derivative_idx, value)
std::vector<p4::NodeEqualityBound> node_equality_bounds = {
  ///////////////////////////////////////////////////////////////////
  // TODO: CREATE A CIRCLE TRAJECTORY
  ///////////////////////////////////////////////////////////////////
  // creates waypoints in a circle:
  // The first node must constrain position, velocity, and acceleration
  p4::NodeEqualityBound(0,0,0,r),
  p4::NodeEqualityBound(1,0,0,0),
  p4::NodeEqualityBound(0,0,1,r),
  p4::NodeEqualityBound(1,0,1,0),
  p4::NodeEqualityBound(0,0,2,r),
  p4::NodeEqualityBound(1,0,2,0),
};

for (auto n = 1; n<num_waypts; n++){
  auto xpos = r*cos(2*pi/num_waypts*n);
  auto ypos = r*sin(2*pi/num_waypts*n);
  node_equality_bounds.push_back(p4::NodeEqualityBound(0,n,0,xpos));
  node_equality_bounds.push_back(p4::NodeEqualityBound(1,n,0,ypos));

  /*
  std::cout<< "n = " << n <<std::endl;
  std::cout<< "time = " << times[n] <<std::endl;
  //std::cout<< "num_waypts = " << num_waypts <<std::endl;
  std::cout<< "xpos = " << xpos <<std::endl;
  std::cout<< "ypos = " << ypos <<std::endl;
  */
}

/*
node_equality_bounds.push_back(p4::NodeEqualityBound(0,1,0,10*cos(2*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,1,0,10*sin(2*pi/6)));

node_equality_bounds.push_back(p4::NodeEqualityBound(0,2,0,10*cos(4*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,2,0,10*sin(4*pi/6)));

node_equality_bounds.push_back(p4::NodeEqualityBound(0,3,0,10*cos(6*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,3,0,10*sin(6*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(0,4,0,10*cos(8*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,4,0,10*sin(8*pi/6)));

node_equality_bounds.push_back(p4::NodeEqualityBound(0,5,0,10*cos(10*pi/6)));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,5,0,10*sin(10*pi/6)));
*/

// The last node constrains position
node_equality_bounds.push_back(p4::NodeEqualityBound(0,num_waypts,0,r));
node_equality_bounds.push_back(p4::NodeEqualityBound(1,num_waypts,0,0));

// std::cout<< node_equality_bounds.size()<<std::endl;

// Options to configure the polynomial solver with
p4::PolynomialSolver::Options solver_options;
solver_options.num_dimensions = 2;     // 2D
solver_options.polynomial_order = 8;   // Fit an 8th-order polynomial
solver_options.continuity_order = 4;   // Require continuity to the 4th order
solver_options.derivative_order = 2;   // Minimize ACCELERATION

osqp_set_default_settings(&solver_options.osqp_settings);
solver_options.osqp_settings.polish = true;       // Polish the solution, getting the best answer possible
solver_options.osqp_settings.verbose = false;     // Suppress the printout

// Use p4::PolynomialSolver object to solve for polynomial trajectories
p4::PolynomialSolver solver(solver_options);
const p4::PolynomialSolver::Solution path
  = solver.Run(
     times,
     node_equality_bounds,
     {},
     {});

// Sampling and Plotting
{ // Plot 2D position
  // Options to configure the polynomial sampler with
  p4::PolynomialSampler::Options sampler_options;
  sampler_options.frequency = 100;             // Number of samples per second
  sampler_options.derivative_order = 0;        // Derivative to sample (0 = pos)

  // Use this object to sample a trajectory
  p4::PolynomialSampler sampler(sampler_options);
  Eigen::MatrixXd samples = sampler.Run(times, path);

  // Plotting tool requires vectors
  std::vector<double> t_hist, x_hist, y_hist;
  for(size_t time_idx = 0; time_idx < samples.cols(); ++time_idx) {
      t_hist.push_back(samples(0,time_idx));
      x_hist.push_back(samples(1,time_idx));
      y_hist.push_back(samples(2,time_idx));
    }

    // gnu-iostream plotting library
    // Utilizes gnuplot commands with a nice stream interface
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Trajectory'" << std::endl;
      gp.send1d(boost::make_tuple(x_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'X'" << std::endl;
      gp << "set ylabel 'Y'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'X-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, x_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'X-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
    {
      Gnuplot gp;
      gp << "plot '-' using 1:2 with lines title 'Y-Profile'" << std::endl;
      gp.send1d(boost::make_tuple(t_hist, y_hist));
      gp << "set grid" << std::endl;
      gp << "set xlabel 'Time (s)'" << std::endl;
      gp << "set ylabel 'Y-Profile'" << std::endl;
      gp << "replot" << std::endl;
    }
  }
}
```