The Drone that Flies Itself

A project where I challenge myself to understand how drones navigate and control themselves. This is the highest level summary of what I’ve done and how I’ve done it.

# The General Approach

## This is a summary of the purpose of this section. For example, this section will illustrate the general approach I took to solving this problem.

1. Identify the problem
2. Brainstorm solutions
3. Build the best solution
4. Test it
5. Optimize it
6. Explain it

# The Planning Problem

## Understanding the Problem and the requirements and constraints for a solution.

### This is a summary of the purpose of this section. For example, this is a summary of the instructions I’ve received and the reasoning that’s gone alone with the instructions.

Drones must be capable of planning their own flight trajectories while navigating an ever-changing environment. Conclusion.

Our drone software must be capable of defining a search space and conducting a search within that space to determine a viable flight plan. One constraint when it comes to the planning problem is computational resources. Therefore, a reasonable plan determined within a reasonable amount of time is preferred over a perfect plan. Representing flight plans as continuous curves would be too complicated. Instead, we should break the world up into discrete states. In fact, we should represent the planning problem using these key elements: a state space, an action space, a start state, and an end state, and a set of costs that are assigned to each action. In other words, the planning problem should be represented as a search space.

The following is an example of developing a simple search space using a 2d grid of cells, fly-zones and no-fly-zones, simple actions, (up, down, left, right), and costs associated with each, in this case, uniformly 1 unit of cost. However, with such a simple search space representation, there are numerous drawbacks. Ultimately, the designer must consider the pros vs. the cons for his or her desired search space representation.

Search works like this. First, the vehicle is considered to be in its start state. Then, all possible actions or partial plans are considered. After, one partial plan is selected, and the process repeats again. It’s important that a search algorithm doesn’t revisit already visited states and doesn’t hit obstacles.The way the algorithm makes the decision depends. In breadth first search, the plan selects the “shortest” path. It will find the shortest path to the goal before other algorithms would. However, it’s costly in terms of computation and storage. In depth first search, a particular action is selected and then repeatedly applied. It doesn’t always find a path from start to goal state.

#### BFS EXERCISE

What I did. How I did it. Why it could help in finding a solution to our main problem.

Here’s an attachment to my thought processes.

#### COST EXERCISE

An algorithm called A\* search provides a happy medium. It’s particularly useful in the field of aerial robotics because some map data is usually available. Breadth first search can be extended to include a cost function. For example, the cost function could include the euclidean distance between two states. But the notion of cost could be extended to include what’s called a heuristic.

#### A STAR EXERCISE

These are like distance estimates and must be underestimates of the true cost of reaching the goal from that candidate state. The heuristic could then be used by the search algorithm when making decisions. For example, the algorithm should add not only the cost of reaching the candidate state but also the estimated cost remaining to traverse from the candidate state to the final goal state. Heuristics must be admissible, which means they must be underestimates of the true cost of reaching the destination. If they weren’t, the algorithm could fail to find the shortest cost plan from start state to goal state. The heuristic function must obey the triangle inequality theorem (the heuristic cost estimate involving the traversal of node A to node B to node C must be at least the heuristic cost estimate involving the traversal of going from node to A to C directly). A\* will find the lowest cost plan first.

In this exercise, I’m going to build a grid and fiend the lowest cost plan using A\* search. I’m going to build this in Python. I’m going to use Priority queue data structure. I’m going to use the straight line distance as a heuristic. I’m going to define the cost of a straight line translation to 1 and a diagnoal line translation as sqrt(2). This will serve as a prototype program for future implementations. There is started code provided by my institution. Then, I will be adding the necessary parts.

[Insert exercise summary between here]

There's location. There’s rotation angles. This topic is on rotation angles. Connecting points in a grid doesn’t account for the physical size of the vehicle and actually orientation of the vehicle. There are 3 cord frames that can be used. Quaternions will simplify things. There’s something called a configuration space, which is an extension of the planning problem, incorporating physical size and shape into the representation of the environment.

The coordinate frame is how we represent the vehicle. The latitude and longitude frame is called the Geodetic frame. We need altitude, too. (altitude, longitude, latitude). Convert to cartesian coordinate frame, ECEF. The origin is an actual location on earth. Regular spacial dimensions. X north. Y east. Z down. It’s possible to convert from a Geodetic frame to a ECEF frame. There’s also the body frame. The origin might be the center of mass. Solves the problem of keeping track of relative distances between sensors, for example. Important for sensor fusion. Best for control inputs. Euler angles are common ways of representing 3d orientation. Order matters. Gimbal lock. It’s a consequence of order mattering. Could create a situation where one rotation can’t be used in math. Gimbal lock has this mathematical limitation. With rotation matrices, just apply rotation matrices using matrix multiplication on the original euler angles. Sometimes, you will run into gimbal lock. There’s also storage implications. Pros are intuitive. But, we should use quaternions. It’ll be better for the software.

#### GEODETIC TO NED EXERCISE

#### EULER ROTATIONS EXERCISE

Quaterions. Scalar plus imaginary i, j, k vector component. It’s the same as a composition of three rotation matrices. If the quaterian is basically magnitude of 1 (unit quaternion) then the rotation is unique. So, quaterions can be used to represent rotations. To take the inverse just change the signs of the vector part. To transform a point by rotation using quaterions, just premultiply the point coordinates by q and post multiply by inverse q. There are other properties. Avoids gimbal lock and overparameterization. Always good though, to go back to euler angles. There are equations to do so directly. Motion is translation to the reference point and a pre-post multiply of the orientation by the quaternion and inverse quaternion. You can use this idea to create a plan.

#### QUATERNIONS EXERCISE

The configuration space is a way of taking into consideration the size of the vehicle and the obstacles. Using all orientation angles about a fixed point. Flying car is 6 dimension configuration space. What this helps us do is represent the robot as a point within the configuration space.

#### CONFIGURATION SPACE EXERCISE

Grids to graphs

Grids are excessive. The autopilot stops at each grid cell. Why? The flight computer is sending the autopilot one waypoint at a time. We don’t want that. We want key waypoints, not a sequence of grid cells. There’s a test that converts a grid-based plan into a more succinct waypoint plan. Removing intermediate grid cells that approximates the original plan while still keeping the vehicle safe and collision free. One method is removing colinear waypoints. Keep just the start and end point of a colinear subsequence of locations. Compute the triangle area defined by three points and compare it to zero. Use the determinant. Can be applied in 2d or 3d. Consider subsequences by threes. Discretize world as grid. Search from star to goal with A\*. Test for colinearity. Remove cells. The colinearity test alone, isn’t good enough for an optimal result, though. We want to consider plans that cut corners when appropriate (free space). With ray tracing, consider the grid cells between one state and another on the straight line between them. If you check each state that exists on the line, then you can determine if you can fly from one point to another. One drawback though is floating point operations, which have historically been computationally expensive. To avoid this cost, use Bresenham algorithm. It contains calculations to integer arithmetic. At this point, I should be able to convert a grid representation of the environment to a graph, and then run A\* search to find a path. After that, I use colinearity or Besenham to eliminate unnecessary waypoints. Graph nodes are like states. Edges get the vehicle from node to node. A\* traverses node to node via edges. The road network can be converted to a graph. There are tradeoffs. For one, it won’t necessarily lead to an optimal or complete solution. Grids represent geometry, limited by resolution. But grids are computationally expensive. There’re non-metric graphs that could be complete, could be optimal, but are not expressive. Metric graphs are a happy medium. A voronoid graph is one way generating a graph to extract waypoints that will be a good distance away from obstacles. One the one hand, they are sort of like the safest path. On the other hand, real obstacles aren’t points, their polygons. Instead, we should consider the medial axis theorem and medial axis transform. Start with a grid of red and green. Then start highlighting freespace adjacent to obstacles. Then iterate away from obstacles, until you find cells that are equidistant from two different obstacles. Then use bresenham to take those grid cells and convert them to a graph of waypoints and edges. How do we not stop at each waypoint? Mavlink autopilot has an acceptance radius. The waypoints seem to move since GPS errors and noise. It’s called a deadband. Once in a deadband, then autopilot starts working on the next waypoint. You can make intermediate waypoints larger, or you could base them on velocity. All of this will be with 2d grids and graphs.

#### COLLINEARITY EXERCISE

#### BRESENHAM EXERCISE

#### PUTTING IT TOGETHER EXERCISE

#### MEDIAL AXIS EXERCISE

#### VORONOI GRAPH EXERCISE

Moving into 3d

Voxel maps can be made. More memory intensive. Instead, use a 2.5d map. 2d array with values taking on minimum allowable height. Can’t represent free space under that. Shouldn’t use medial axis transform because that would involve creating a 3d grid, which is what we are trying to avoid. Use random sampling to generate candidate states. Ignore those with obstacles. Then generate a plan. But checking if conflict with obstacles is kind of hard. But can check based on x,y,z. There’s actually an algorithm called probabilistic roadmap. Sample states at random first. THen discard states that collide with obstacles. Finally, build a graph. Then run search. Use bresenham to draw lines that aren’t through obstacles. Not perfect, but good. Discretize edges and test for obstacles OR testing for the intersection of line segments with obstacles. Which is better depends. THIS IS A DESIGN DECISION. Edge method could be asymptotically complete or optimal. COnsider using KD tree structure to make this more efficient. There’s a cost to build the model. There’s a cost to searching. But the environment changes all the time in the real world. So that means you use multiple query planning. Search will dominate the cost of the search. Reduce the branching factor. Or use a 2d grid at some altitude. Or a combination of 3d in local region and 2d in general. That’s called receding horizon planning. Flight computer will determine the local horizon. Since wind and other disturbances are always there, the vehicle must be able to determine if a path from it’s location to the destination is still viable. The state machine must be able to stop and think if it needs to stay safe and replan if needed.

#### VOXEL MAP EXERCISE

#### RANDOM SAMPLING EXERCISE

#### PROBABILISTIC ROADMAP EXERCISE

#### RECEDING HORIZON EXERCISE

Real world considerations

THIS IS WHERE THE SOLUTION IS OPTIMIZED.

Not true assumptions. Not perfect map. Not static environment. Not perfect controller. Existing robots have holonomic constraints. For example, going left requires roll. There’s a coupling between one state variable and another. There are kinematic constraints not the result of force or mass, rather geometry. For example, a car is constrained by its steering angle. They’re constrains on what actions are actually possible. This is the derivative of the state. There are constraints on how fast quodrotors can accelerate or decelerate. Quasiholonomic is what we’ve been dealing with. But vehicles moving quickly can’t be considered that. Constraints from force and mass are called dynamic constraints. For example, strong wind in one direction means change of direction in opposite direction is not possible. Vehicle model. A set of differential equations can be used for constraints. Requires integration. That can be complicated. Could use linear integration scheme.

#### MODELING DYNAMICS EXERCISE

Sgfdgsdfg

#### DUBINS CAR EXERCISE

Sdfg

#### STEERING EXERCISE

Sdfg

#### RRT EXERCISE

Sdfg

Defining the problem

Given a “map” of SF, create a plan from A to B. Given a 3D environment. Take methods and algorithms. Combine them. Figure out how to combine it. Build up the baseline with help. Take it to the next level with advanced techniques.

[Consider including a problem solution table to summarize five key problems and how they can be solved]

Writeup

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

Explain the Starter Code

Test that motion\_planning.py is a modified version of backyard\_flyer\_solution.py for simple path planning. Verify that both scripts work. Then, compare them side by side and describe in words how each of the modifications implemented in motion\_planning.py is functioning.

Implementing Your Path Planning Algorithm

In the starter code, we assume that the home position is where the drone first initializes, but in reality you need to be able to start planning from anywhere. Modify your code to read the global home location from the first line of the colliders.csv file and set that position as global home (self.set\_home\_position())

In the starter code, we assume the drone takes off from map center, but you'll need to be able to takeoff from anywhere. Retrieve your current position in geodetic coordinates from self.\_latitude, self.\_longitude and self.\_altitude. Then use the utility function global\_to\_local() to convert to local position (using self.global\_home as well, which you just set)

In the starter code, the start point for planning is hardcoded as map center. Change this to be your current local position.

In the starter code, the goal position is hardcoded as some location 10 m north and 10 m east of map center. Modify this to be set as some arbitrary position on the grid given any geodetic coordinates (latitude, longitude).

Write your search algorithm. Minimum requirement here is to add diagonal motions to the A\* implementation provided, and assign them a cost of sqrt(2). However, you're encouraged to get creative and try other methods from the lessons and beyond!

Cull waypoints from the path you determine using search.

Executing the flight

This is simply a check on whether it all worked. Send the waypoints and the autopilot should fly you from start to goal!

For a standout submission, consider using some of the more advanced techniques presented in the lessons, like the probabilistic roadmap, receding horizon planning and automatic replanning. Play around with a dynamical model for the vehicle, deadzones to allow smooth transitions through waypoints and even a potential field modification to your planner!

## Implementing and testing a Barebones Solution

### This is a summary of the purpose of this section. For example, I summarize the end result of the solution–what it was able to accomplish. Then, I detail the major steps I took to achieve those results. For each step, explain how the step relates back to the core instruction or theory, and evaluate the performance.

The exercises would actually go here. Then they would build the system prototype.

1. Load map
2. Chunk environment
3. Define start and goal
4. Perform search
5. Remove unnecessary waypoints
6. Return waypoints

## Optimizing the Solution and thinking about future directions

### This is a summary of the purpose of this section. For example, in this section, I summarize what I decided to optimize and what the results the optimized solution was able to accomplish. Then, I detail the major steps I took to achieve these results.

1. Try more complex trajectories
2. Modify deadbands to be a function of velocity

## Explaining what I really learned and why this project was meaningful for me.

### Computer Science, which is building on…

### Problem-solution mindset, which is building on…

### Mathematics on Khan Academy and IXL, which is building on…

### My points of weakness in college.

# The Controls Problem

## Understanding the problem

## Implementing a barebones solution

1. Implement the body rate roll/pitch controller
2. Implement the position/velocity and yaw angle controller
3. Explore nonidealities and robustness
4. Track the trajectories

## Optimizing the solution

1. Generate a trajectory that incorporates velocity information rather than only position information
2. Figure out a way to provide more info for better tracking
3. Figure out a way to fly the trajectory faster

# The Estimation Problem

## Understanding the problem

## Implementing a barebones solution

1. Add sensor noise to simulation
2. Estimate attitude
3. Implement a prediction step
4. Incorporate magnetometer update
5. Incorporate GPS
6. Revamp the controller for the noisy simulation

## Optimizing the solution