Methods for Drone Obstacle Avoidance

Anthony Degleris
degleris@stanford.edu

Abhi Kulgod
akulgod@stanford.edu

Isaac Scheinfeld
ischein@stanford.edu

1 Introduction

Autonomous navigation is currently an active and diverse area of research. Self-driving cars are slowly being phased into production [1], robots are cooperating to move products around warehouses [2], and autonomous drones are being developed for applications ranging from package delivery [3] to search-and-rescue [4]. Our focus has been on the autonomous navigation of larger aircraft, for which we have developed and implemented various methods. While ground vehicles must often navigate unknown terrain, larger aircraft are primarily concerned with avoiding other aircraft and large land features, both of whose locations are usually known to the aircraft through maps and communications. For this reason, aircraft can navigate with close to perfect knowledge of their environment, focusing on planning the optimal path for their objective.

1.1 Problem

We have studied the problem of shortest-path planning in an idealized 2-dimensional space, containing stationary circular obstacles similar to the cross sections of land features and the restricted airspaces around other aircraft. As autopilot solutions well capable of following predefined paths are currently available, we abstracted away control and focused exclusively on path planning. While we developed methods for 2 dimensions and stationary obstacles, we have selected our techniques both with an eye towards extension beyond these simplifications and the incorporation of other real-world restrictions.

1.2 Related Work

While substantial research has been done on autonomous flight, much of it has focused on control rather than navigation. [5] Of the research on navigation, little of it is concerned specifically with finding a shortest path—most of it deals rather with finding any viable path. However, a few approaches to shortest-path navigation have received some attention.

Some approaches simplify the problem from continuous space to discrete space and fit a graph into the areas between obstacles before solve for a shortest path in that graph. This graph path is then somehow optimized or smoothed. [6] [7] While such methods are not necessarily guaranteed to find a shortest path, they almost always come close and can flexibly incorporate aircraft constraints through their path optimization step. However, such methods do not admit a natural extension to environments with moving obstacles, as they plan the entire path for a single obstacle configuration. Our first method follows this approach.

Multiple reinforcement learning based [8, 9] approaches have been used to implicitly learn a strategy for navigating around obstacles. Methods such as Deep Q-Learning have had success in simplified obstacle avoidance problems [10]. Since value based methods are unlikely to work for multiple environments, approximate policy methods are often used instead [11, 12].

One final common approach is the application of fuzzy logic [13]. We did not investigate this approach.

1.3 Flight Environment

We generated square flight environments with obstacles uniformly distributed with normally distributed radii, and our problem was to plan a path from one point to another, completely within the flight space and outside the obstacles. For our tests, our flight space was $(x, y) \in$

$[-50, 150] \times [-50, 150]$, our obstacles had radii $r \sim \mathcal{N}(25, 10)$, and our path was from $(0, 0)$ to $(100, 100)$. We rejected and resampled obstacles that covered either start or endpoints, but did not check that a navigable path existed. A path for us was a sequence of points with no two adjacent points more than 1 apart.

2 Methods

2.1 Graph Approximation and Optimization

2.1.1 Optimization

Our first method was developed in collaboration with StanfordAIR, based on conceptual work done over the previous summer [6][14]. At its core is a method of producing a locally optimal path by minimizing a sequence of cost functions that approximate the problem constraints increasingly well.

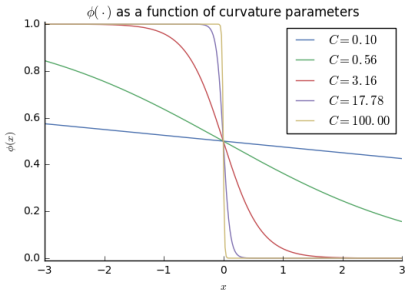


Figure 1: Boundary constraint

Let us begin by examining the nature of our obstacles. Since we can fly arbitrarily close to them but must not touch, our cost function should penalize the path only when it is inside the obstacle and then always by the same amount. However, this would add a hard wall or step term to our cost function and it would then no longer be differentiable, making gradient descent impossible. Therefore, in our cost function we approximate a hard wall constraint by $\phi(Cx)$ where $\phi(x) = \frac{1}{1+e^x}$. This allowed us to vary the hardness of the obstacle boundaries by varying C, as can be seen in Figure 1. Using just our boundary term, and for path points $x_i \in \mathbb{R}^2$ and obstacles with centers c_j and radii R_j, we can describe the boundary cost as

$$\mathcal{L}(x; c, R, C) = \sum_{ij} \phi\left(C\left(\frac{\|x_i - c_j\|_2^2}{R_j^2} - 1\right)\right)$$

Now, we want our path to do more than stay outside of obstacles—we want it to be as short as possible. To penalize lengthening the path, we add a term to our cost function giving

$$\mathcal{L}(x; c, R, C, \eta) = \sum_i \left[\sum_j \phi\left(C\left(\frac{\|x_i - c_j\|_2^2}{R_j^2} - 1\right)\right) + \eta\|x_i - x_{i+1}\|_2^2\right]$$

where η is a hyperparameter balancing the objectives of staying outside the obstacles and tightening the path. Using the squared norm also penalizes irregular distances between the points on the path, keeping them approximately uniformly spaced.

As our goal is to minimize the path length where the boundary constraint is hard, i.e. in the limit as we let $C \to \infty$, we let C_k be by some sequence such that $C_k \to \infty$ and take our final path to be $x^* = \lim_{k\to\infty} \min_x \mathcal{L}(x; c, R, C_k, \eta)$. The minimization for each C_k is done by gradient descent.

Note that since the cost function is not convex, we can only locally minimize x_k by gradient descent. To approximate x^*, we therefore minimize for $x^{(k)} = \min_x \mathcal{L}(x; c, R, C_{(k+1)}, \eta)$ at each step starting from $x^{(k)}$, the optimized path for the previous k.

Minimizing for each C_k separately is computationally expensive, so we apply two to techniques to speed up the process. First, we replace standard gradient descent with gradient descent plus momentum. Here, instead of our update rule being $x_{i+1} = x_i - \alpha\nabla\mathcal{L}$, we track a velocity term and our update rule becomes

$$v_{t+1} = -\gamma\nabla\mathcal{L}(x_t) + \beta v_t$$
$$x_{t+1} = x_t + v_{t+1}$$

where γ is the step size and β the momentum.

Second, instead of letting gradient descent converge once for each C_k, we send $C_k \to \infty$ as gradient descent is converging. That is, instead of minimizing for each C_k individually, we approximate minimizing for $C = \infty$ directly by letting $C \to \infty$ as our path converges.

Finally, all of these improvements bring with them hyperparameters in addition to η. We selected these manually at first and then used grid search to refine them.

2.1.2 Graph Approximation

While the optimization technique presented so far works nicely in many cases, it suffers from two difficulties. First, when the path starts inside an obstacle, the gradient pushing it outwards is not very steep and convergence can take a very long time. Second, since the method only finds a locally optimal path, the path's global optimality is very dependent on the initial path. For these reasons, instead of beginning the optimization with a straight path, we approximated a path outside the obstacles and close the the global optimum by filling the space between the obstacles with a graph and solving for the shortest graph path between our points. This graph path was then quantized into closely spaced points in $\mathbb{R}^2$, and as the initial path for the optimization step helped it converge more quickly and almost always to an approximation of the true shortest path.

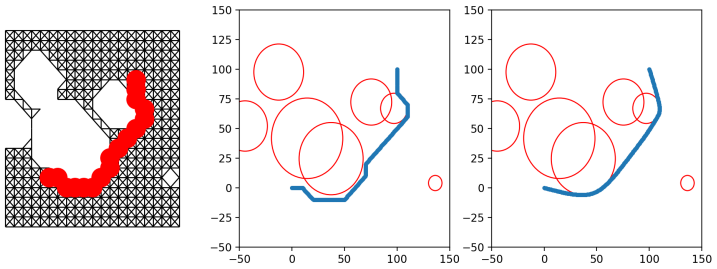


Figure 2: Graph and Optimization Steps

2.2 Approximate Policy Iteration

Although the previous method works well and comes close to guaranteeing a correct solution, a new solution must be computed every time the flight environment changes. Instead, we use reinforcement learning to predict the optimal action given any environment.

Unfortunately, our state space is both continuous and high dimensional. A value function that works in general (for all flight environments) would be highly complex and require sampling many states for each value update.

Instead, we choose to approximate the policy directly with some parameter θ, and use least squares policy iteration [15] to find some optimal policy. We represent the policy $\pi : S \to A$ as an inner product between the parameters and some feature mapping $\phi(s)$

$$\pi_\theta(s) = \theta^T \phi(s)$$

We would like to find the parameter θ that maximizes

$$J(\theta) = \sum_{s \in S} P(s) V^{\pi_\theta}(s)$$

Where $P : S \to \mathbb{R}$ is the probability of being in a state s and $V^\pi : S \to \mathbb{R}$ is the value of state s with respect to policy π. Since it is hard to explicitly differentiate our objective, we choose the following update rule for θ

$$\theta^{t+1}(s) = \arg\min_{\theta^t} V^{\pi^t}(T(s, \pi_{\theta^t}(s)))$$

Where $T : S \times A \to S$ is the transition from state s and action a to a new state s'. Intuitively, we are picking the policy at time $t + 1$ to be optimal with respect to the value function generated by the

policy at time t. In practice, since our space is continuous, we approximate this by sampling m states $s^{(1)}, ..., s^{(m)}$, finding the optimal actions $a^{(1)}, ..., a^{(m)}$ at each state, and minimizing

$$L(\theta) = \frac{1}{m} \sum_{i=1}^{m} \theta^T \phi(s^{(i)}) - a^{(i)}$$

Which can be solved with ordinary least squares.

2.3 Policy Function Improvements

In order to better model the complexity of the state space, we explored various methods to capture some of the higher order phenomena that may not have been captured by the approximate linear policy. First, we tried modifying our definition of the policy inputs – i.e. to find some optimal $\phi(s)$ that maximizes the policy's navigation ability. Consider the most basic definition of state at a given time step: a vector of the locations of the agent, target, and obstacles. Let $s_a \in \mathbb{R}^2$ be the location vector of the agent, $s_{obs^{(i)}}$ the location of the i-th obstacle, and $s_{rad^{(i)}}$ the radius of the i-th obstacle, for $i = 1, ..., m$. The state can be represented as:

$$s = \begin{bmatrix} s_a \\ s_{obs^{(1)}} \\ s_{rad^{(1)}} \\ \cdots \\ s_{obs^{(m)}} \\ s_{rad^{(m)}} \end{bmatrix}$$

We decided to construct non-linear features that were unlikely to be discovered by the regression model, since its output is a linear combination of its inputs. Now we calculate the feature for each obstacle as:

$$\phi(s_{obs^{(i)}}) = \frac{(s_a - s_{obs^{(i)}})}{(||(s_a - s_{obs^{(i)}})|| - s_{rad^{(i)}})^2}$$

In other words, the direction from the agent to a given obstacle was inversely weighted by the squared distance from the outer edge of the obstacle. For simplicity, if the agent's distance from the edge was greater than 3 times the radius, the feature for the obstacle was the 0 vector.

After developing the more robust feature mapping, we decided to explore more complex models to use for approximating the policy. We chose a neural network to fit the simulated data to optimal actions at each step of the policy iteration, so higher order combinations of the features could be used by the model. The neural network was designed to minimize the squared error between labeled actions and predicted actions, optimized using the Adam gradient descent solver from Sci-kit Learn. We tuned the size and dimension of the hidden layers based on performance on randomized obstacle environments and determined that a single hidden layer of 100 units provided optimal results.

3 Results

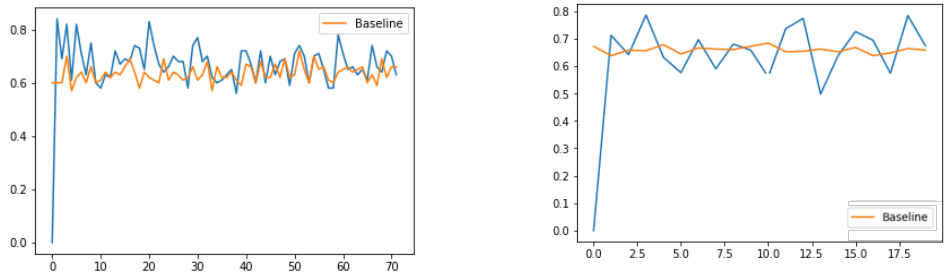
We trained the linear and neural network policy approximations for 80 and 20 iterations, respectively. Each iteration was approximated by randomly sampling 1000 possible environments. During training, we compared the algorithms' success rate to a baseline (following a straight line). Training was very unstable, so we chose models that significantly outperformed the baseline.

The algorithms were benchmarked using flight environments generated randomly after the models had been trained, measuring the success rate on 100 randomly generated environments.

We note the reinforcement learning models never saw these flight environments during training, and used the same parameters learned during training (i.e. no

Table 1: Algorithm Performance

Algorithm	Success Rate
Graph and Optimization	1.0
Linear Policy Approximation	0.78
Neural Network Policy Approximation	0.54



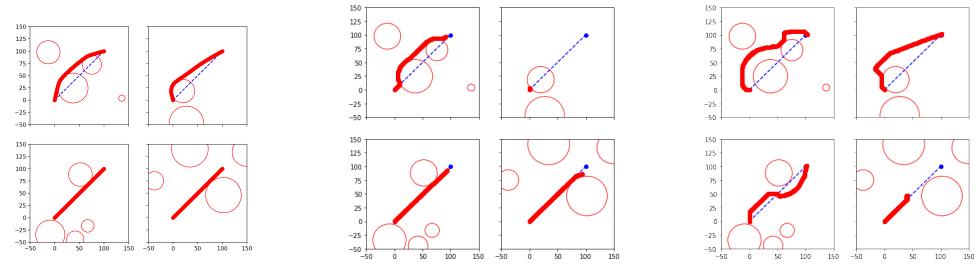
(a) Linear Learning Curve

(b) Neural Network Learning Curve

Figure 3: Learning Curves

additional learning was done during the test cycle). The linear model achieved a surprising success rate, reach the objective on most maps. Its most common failure was getting stuck when starting immediately next to an obstacle. The neural network achieved less consistent performance, sometimes performing well on complex environments but getting stuck on simple ones. We attribute this to a very simple neural network architecture with a short training period (20 iterations).

Distance metrics were omitted because algorithms that failed more frequently averaged shorter path lengths, since more of their successes came from straight line paths. The graph approximation and optimization technique always achieved an optimal path (up to a discretization). The linear policy approximation often came close to locally optimal paths, but was unable to regularly determine the globally optimal path. Finally, the neural network generated unusual paths, preferring to veer directly towards the finish and turn only when close to obstacles.



(a) Graph and Optimization

(b) Linear Policy

(c) Neural Network Policy

Figure 4: Sample Paths Generated on Test Environments

4 Conclusion and Future Work

While the Neural Network was outperformed by the linear model under the state space we defined, we believe it has the capacity to improve with more complex feature mappings. The trained agent was often unable to properly balance reacting to obstacles with moving towards the target – often succeeding in avoiding obstacles, but veering far away from the target. We plan to explore an optimization inspired by a previously developed DQN simulation for obstacle avoidance [10]. This method would provide the agent with a limited "vision radius", where input features would indicate whether an obstacle was occupying a given partition of that radius. In this mapping, the agent could likely find a more consistent method of responding to obstacles within a limited radius and direction.

In addition to these improvements, our models were chosen with the intension of extending them beyond 2 dimensions and to the case of moving obstacles. For the optimization method, this will require recalculating the path at each timestep and incorporating some form of path prediction. For the reinforcement learning methods, the state can be made more complex either by adding obstacle velocities or by passing in the state for multiple time steps, or the model itself can be given some form of memory.

References

[1] Guilbert Gates, Kevin Granville, John Markoff, Karl Russell, and Anjali Singhvi. The race for self-driving cars. *The New York Times*, June 6, 2017.

[2] Will Knight. Intelligent machines inside amazon's warehouse, human-robot symbiosis. *MIT Technology Review*, July 7, 2015.

[3] Nick Winigfield and Mark Scott. In major step for drone delivery, amazon flies package to customer in england. *The New York Times*, 2016.

[4] Cai Luo, Andre Possani Espinosa, Danu Pranantha, and Alessandro De Gloria. Multi-robot search and rescue team. In *Proceedings of the 2011 IEEE International Symposium on Safety,Security and Rescue Robotics*, November 2011.

[5] Haiyang Chao, Yongcan Cao, and YangQuan Chen. Autopilots for small fixed-wing unmanned air vehicles: A survey. In *Proceedings of the 2007 IEEE International Conference on Mechatronics and Automation*, 2007.

[6] Guillermo Angeris. Some thoughts on global path optimization (part 1/?).

[7] C. Goerzen, Z. Kong, and B. Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57:65–100, 2010.

[8] Michael C. Koval, Christopher R. Mansley, and Michael L. Littman. Autonomous quadrotor control with reinforcement learning. 2012.

[9] Loc Tran, Charles Cross, Gilbert Montague, Mark Motter, James Neilan, Garry Qualls, Paul Rothhaar, Anna Trujillo, and B Danette Allen. Reinforcement learning with autonomous small unmanned aerial vehicles in cluttered environments. In *AIAA Aviation Forum*, 2015.

[10] Andrej Karpathy. Reinforcejs: Waterworld deep q learning.

[11] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[12] David Silver. Policy gradient.

[13] K.M. Zemalache and H. Maaref. Controlling a drone: Comparison between a based model method and a fuzzy inference system. *Applied Soft Computing*, 9:553–562, 2009.

[14] Guillermo Angeris. Optimizers, momentum, and cooling schedules (part 2/?).

[15] Lihong Li, Michael L. Littman, and Christopher R. Mansley. Exploration in least-squares policy iteration. Technical Report Technical Report DCS-TR-641, Rutgers University, 2008.

Python 3.5.4

Numpy 1.13.3

Networkx 2.0

Scikit-Learn 0.19.1

Contributions

Anthony designed a module for approximate policy iteration, that can be used to test out different feature mappings and policy approximation functions. He tested this module with the linear policy using least squares update.

Abhi developed and tested the neural network model for the fitted policy iteration and applied the optimizations on the feature mappings for testing. He tested various frameworks for implementing the neural network before optimizing the final solution.

Isaac developed and tested the graph and optimization method, and has been involved with its conception from the start as StanfordAIR's engineering lead. However, it had not been implemented before he began doing so for this project. He was also responsible for developing the testing framework.