INDE599 - Project Report
Drone Collision Avoidance in low-altitude

Emilien Pilloud

July 19, 2018

Problem Statement

The past few years, we've witnessed the rise of personal small drones which are sold on-line and in many stores nowadays. Those drones are mainly created for low altitude recreational use and thus are prone to user mis-usage or lack of caution. Some of them even have the functionality to come back by themselves when their energy level is very low. In those situation, it is critical for the drone to avoid collision and in this project I tried to implement a non-trivial low-altitude collision avoidance policy when a drone is flying unmanned and undirected at a given altitude linear trajectory. Those objects can include poles, trees or houses and are lying on the ground. They also have various heights and shapes. The information to the drone intelligent systems come from sensors which, as we know, take noisy measurements of distances. The challenge is then to compose with this uncertainty in an approximately optimal way.

1 Problem Modeling

1.1 State space

My state space is based on the state space used in the paper [2] but I slightly modified it to better fit my problem and for simplicity. The environment in which the drone evolves is represented by a vertical plane. The state still consists of the perfect measurement of the object y-coordinate y as well as a series of n measurements which represents data from sensors. I will refer to these measurements as observations in the system. The x-speed will be fixed and be set to 1 for simplicity but its value should not change anything in theory.

1.1.1 Observations

The observations are provided by the sensors and I will model them differently. I will firstly have a `DeterministicSensor` which, as its name suggests, measures the distance from the drone to the closest obstacles deterministically without any error. It will help me both find a clear-state policy which will be useful later on and verify the correctness of such policy. Secondly, I defined a `StochasticSensor` with takes measurement in an additive Gaussian noise channel with standard deviation σ. I restricted the measurements to be positive as I believe it has more physical significance. Sensors take N, $N = 2k+1$, $k \in N^+$ independant observations at heights between $y - \lfloor \frac{N}{2} \rfloor$ and $y + \lfloor \frac{N}{2} \rfloor$. In the theory the sensors lines would all come from the drone but I modelled it like this for simplicity assuming you could recover this information from slant measurements. Sensors can also have a limited range r which model a realistic sensing constraint.

1.2 Action space

My action space is simply a discretized range of vertical speeds $[lower, upper]$. I decided not to use accelerations values both for simplicity and also because drones carry way less momentum than planes or bigger flying objects. However, the change would be easy to do but as the physical model is not the bulk of my project, I chose to keep it simple.

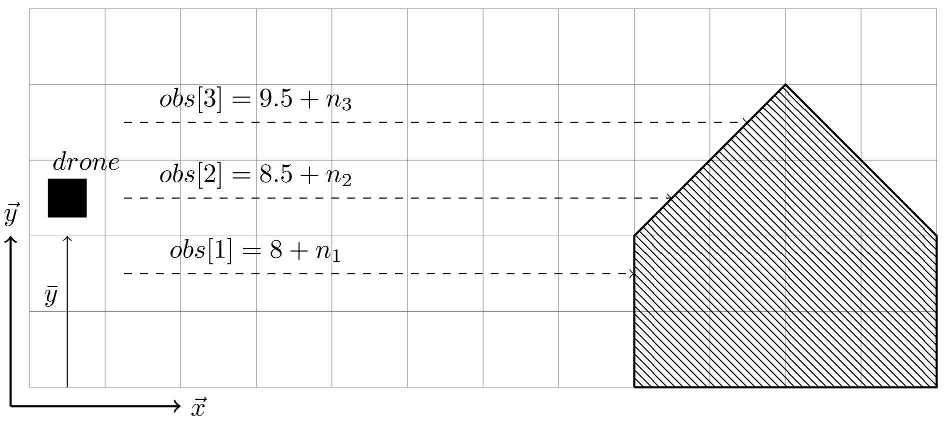


Figure 1: Representation of the discretized state space with $n = 3$

1.3 Transition Probabilities

Just as in [2], I considered deterministic transitions for the drone in both the y and x direction for simplicity. I also assumed that in practice, drones have a bigger freedom in vertical acceleration as their mass is so much lower than the masses of planes considered in [2].

1.4 Cost Model

I defined a cost model which penalizes taking three state variables into account:

- Is the model in a collision state? This is our worst case scenario and hence we want to penalize this event heavily.
- Is the drone position within a certain distance ϵ from an obstacle according to a metric $m(x, y)$? This make the agent avoid a certain proximity region around the obstacles.
- How far from its aimed linear trajectory y_{traj}? We want the drone the stay within its horizontal trajectory as much as the terrain allows. Indeed, we don't want to allow a too high altitude, especially if the drone is running low on power and it would also make the problem straightforward.

Hence, the cost function is defined as:

$$\begin{aligned} c(obs, \bar{y}) = \mathbf{C\_COLLISION} * \mathit{has\_collied}(obs, \bar{y}) + \\ \mathbf{C\_PROXIMITY} * \mathit{is\_within}(obs, \bar{y}, epsilon, m) + \\ \mathbf{C\_OUT\_OF\_LINE} * |Y - \bar{y}| \end{aligned} \tag{1}$$

I specifically chose for the rest of this project:

$$\begin{cases} \mathbf{C\_COLLISION} = -1000 \\ \mathbf{C\_PROXIMITY} = -100 \\ \mathbf{C\_OUT\_OF\_LINE} = -5 \end{cases} \tag{2}$$

2 Methodology

The state space in such problems typically blows up as we want to chose a small step of discretization to have precise results. Therefore, I decided to used q-learning to approximate the optimal policy of the MDP.

2.1 Q-Learning

Q-learning is a versatile and powerful reinforcement learning approximation technique which consists in updating the q-values online while learning through a certain number of episodes. The q-values are updated as follows [1]:

$$Q[(state, act)] = \begin{cases} cost(state, None) \quad \textit{if state is terminal} \\ Q[(state, act] + \alpha(cost(state, act) + \\ \qquad \gamma \max_{act'} Q[(state', act') - Q[(state, a)]) \; \textit{otherwise} \end{cases} \tag{3}$$

And the q-learning agent updates to the next state as follows:

$$state' \leftarrow successor(state, argmax_{act'} f(Q[state', act'], N[state', act']) \tag{4}$$

where f is an exploration function and N[s, a] a counter which counts the number of times we have used a (state, action) pair to evolve in the system.

2.1.1 Exploration function

In Q-Learning and many methods of approximation of MDPs optimal policy, one pitfall is to get stuck in a local minima and not to be able to get out of it. To avoid this phenomenon, I implemented as commonly used an exploration function. This function will tell the Q-learning agent what action to choose to transition to the next state using equation (4). A simple exploration function would consists in perturbing the action chosing process with random noise at a certain probability p and to decrease this probability with the number of iterations once the first third of iterations is reached. Another easy and effective exploration policy and which I used in combination with random noise in the project is defined as [1]:

$$f(u = Q[state, action], n = N[state, action]) = \begin{cases} R^{+} \quad \textit{if } n < N_e \\ u \quad \textit{otherwise} \end{cases} \tag{5}$$

This simply enforces that each reachable pair (state, action) per state is used at least N_e times before following the current best policy. It also has the nice property that it will push the q-learning agent to visit the border state of the state space.

2.2 POMDP

Using our `StochasticSensor`, we need to deal with the uncertainty of the measurement which makes our model partially observable (POMDP). We also want to make sure to use the temporal information of our subsequent measurements, i.e. if we sense 13, 12, 11, 2, 9 in a row, the 2 is very likely to be due to noise and therefore one should not act upon it. We can compute this likelihood using the Baye's formula and our MDP modeling assumption:

$$p(e_{t+1}|x_{t+1}, x_t) = P_{x_t, x_{t+1}}(act) * p(e_{t+1}|x_{t+1}) \tag{6}$$

and specifically with our discretized Gaussian sensor:

$$p(e_{t+1}|x_{t+1}) = \frac{\phi((\lceil e_{t+1} \rceil - x_{t+1})/\sigma) - \phi((\lfloor e_{t+1} \rfloor - x_{t+1})/\sigma)}{1 - \phi(-x_{t+1}/\sigma)} \tag{7}$$

where ϕ is the normal Gaussian distribution cdf. Note that we condition by $1 - \phi(-x_{t+1}/\sigma)$ because we enforced positive measurements from the `StochasticSensor`. This is motivated by the fact that even in a noisy channel, sensors are unlikely to predict negative distance values.

2.2.1 Particles Filters

Because it is very expensive to keep track of the exact probability of each possible state so far in our model, I decided to use particle filters to approximate the Bayesian inference. This consists in keeping a number M of particles which represents believes that we are in a certain state s, i.e. $\bar{p}$(current state is s) = #particles representing s / total #particles. We initialize the particles uniformly and as the model evolves, we update the particles as follows [1]:

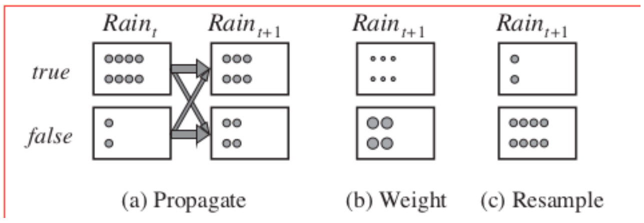


Figure 2: Particles update procedure

In our case, the propagation straight-forwardly consists in decreasing a particle value by X_SPEED. We then weight the particles by their likelihood (7) and resample the same number of particles using the posterior distribution.
In my model, I decided to keep particles for each height value of the grid and to update them every time we get a observation from this height. If the likelihoods get too low, I reinitialized my particle values as it we lost track of the object.

2.2.2 Clear-View approximation

The particle filters produce approximate Bayesian believes of the current state but we still need to act upon them. To do that, I act according to the clear view state policy. I use a percentile q of my particles to determine what clear view state I will consider and then act accordingly.

3 Implementation of the agents

I defined four different agents that uses the state space and various combination of techniques defined above. They fall into two categories which I named Deterministic and Probabilistic. The Deterministic agent handle the observations deterministically at each stage meaning they do not take previous observations into account whereas the probabilistic agent uses particles filters to keep track of the obstacles and estimate the Bayesian inference.

3.1 Deterministic agent without estimator

This is the simplest model. It acts upon a policy obtained by training on the simulation scenario with a DeterministicSensors. The observation are then individually rounded to fit into the discretized state space. Naturally, this is more tricky because the size of the state space increases significantly by the possible noise added into each sensing channel. However, in the theory, with enough iterations in the q-learning phase, it should eventually learn an estimate of the sensor's standard variation and act upon it. But this assumes it would converge quickly enough.

3.2 Deterministic agent with avg/min estimates

This model is also deterministic but instead of considering the n measurements independently, it uses the dependency assumption to compute an estimate. The *avg* model simply computes the statistical mean and the *min* model uses the mathematical minimum. Note that those estimators do not operate on the *No object in sight* value which is assumed correct and flawless.

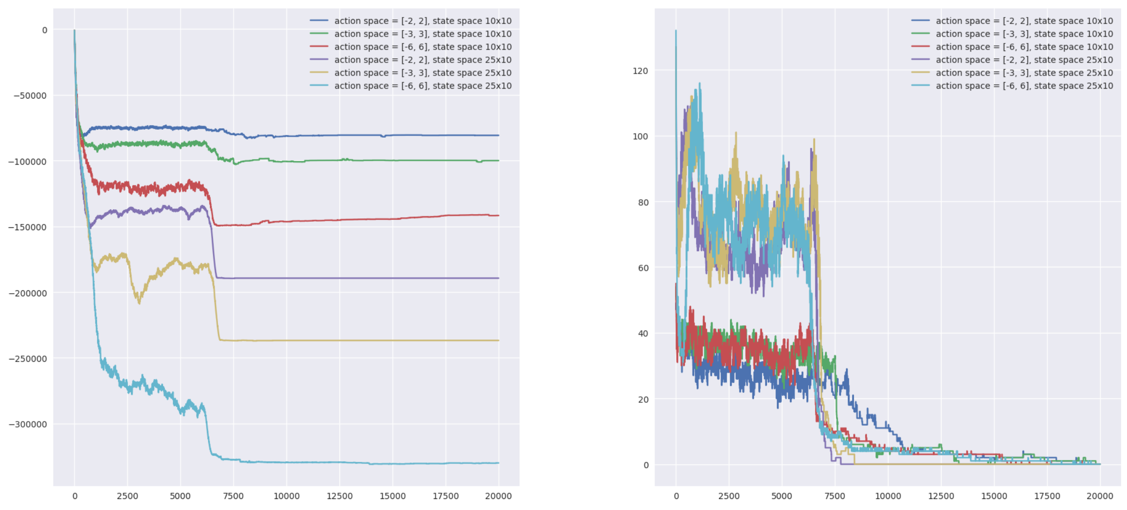
3.3 Probabilistic agent

This model is aware of the variance of the sensors (This is usually know or closely estimated in the industry) and uses a sequence of particles filters to keep a running estimate of how close the object is. There is one filter per vertical discretized space step. In this project, I will use 5000 particles per filter. As for the percentile q, I chose a value of 70 which seemed like a good trade-of between safety and performance.

4 Numerical results and Analysis

4.1 Convergence of Q-Learning

It is important to make sure that the Q-Learning was able to converge to an (sub)optimal policy. Here is what I get for different action spaces for my deterministic model with 20,000 iterations:



(a) Sum of the q-values per iteration (b) hamming distance btw i^{it} policy and final policy

Figure 3: Convergence of Q-Learning

Firstly, we can clearly see the effect of the random action noise added throughout the first third of the learning process. It slows down the converging process but it is good for exploring faster. We can also notice that the policy converges reasonably for all action and state spaces. However, we can see that the [-6, 6] in a 10X10 has not fully reached its plateau yet. The figure on the right is relative to the last got policy and is hence harder to analyze but we can nonetheless notice that policies vary very little from 15'000th iteration onward.

4.2 Optimal Policy

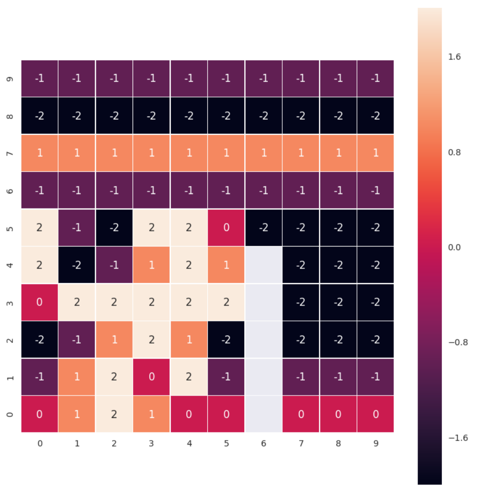
We also want to verify that our approximate optimal policy seems to fit the model constraints and assumption as well as the cost model. Here is the policy displayed for a vertical object of height of 5 in a 10x10 grid world with 3 observations side-by-side with a visualization of the cost model: This policy is nice because easy to read as I chose on purpose a small state and action space. We can see that the result seems reasonable as the drone approaches the obstacle. We see different behaviour regions:

- If still far from the obstacle: The drone tries to get back to its trajectory as the OUT_OF_TRAJECTORY cost occurs.
- At optimal distance, the drone starts to get over the obstacle to avoid the COLLISION_COST.
- If too close already, the drone avoids additional costs by staying on its trajectory until collision.

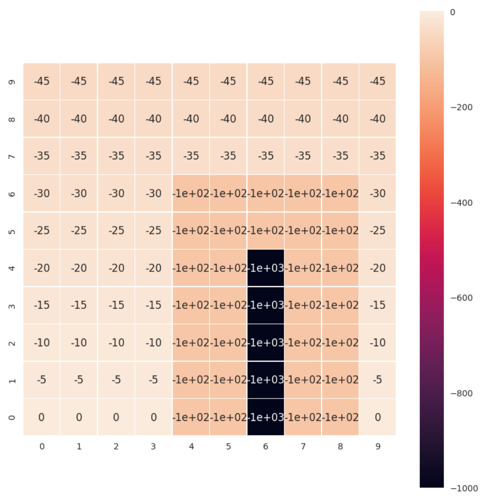
Hence, we see that it seems to fit well the cost model displayed on the right. We can also notice the orange stripe at $y = 7$ which seems out of place. However, it was expected as at this height, the basic agent has not way of knowing where it is located compared to the obstacle.

4.3 Simulated performances of the different agents

I simulated the performances of the different agents by running 10,000 scenarios of encounter of the agent with an obstacle of height 5 in the same 10x10 grid world. I recorded the number of collisions, the average time in risk, i.e. within $\epsilon = 2$ of the obstacle and the average cost incurred. See figure 5.



(a) Optimal Policy Map



(b) Model Cost Visualization

Figure 4: Optimal Policy

We can clearly see that the Deterministic model without estimator performs poorly. It was expected as it did not cope with the measurement noise at all. We can interestingly notice that the Deterministic agents with estimators perform better than the particle filter agent for small sigmas (0.5, 1). My hypothesis is that the particles filters can carry some mistakes over times and with such small sigmas, it is beneficial to sometimes forget about past observations to recover better. Also, avg and min are more effective in such scenarios as mistakes happens quite rarely. However, we can see that the particle filter agent starts performing much better with bigger sigmas. At $\sigma = 3$, it even does not experience any collision at all! It also all together gather the least average cost per run. It is then a very big improvement in the agent behaviour as expected. We can see that as sigma gets very big compared to the world, agents get hectic and their results are more random. I suspect it would converge for unreasonably big sigmas.

5 Conclusion and further work

The bulk of the project was to implement a way of approximating Bayesian inference using particle filter and it has revealed to be indubitably useful for the agent. As a side note, we should also remember that update all of the particles take some computation time and that not all drone would be able to carry such computations online. However, we found ways to model and to deal with the uncertainty of the sensors in an approximate manner.

5.1 Approximations

The motion control problem is taking place in a obviously continuous world and the drone would be subject to a lot of physical constraints that I decided not to consider. This implies that the underlying physical model is very approximate and should be augmented quite a lot for any practical implementation. A good amount of approximation also reside in the discretized Gaussian inference and one could come up with a better and more careful way of updating the particles likelihood.

5.2 Hyperparameters

Due to the constraint of time, I could not really perform an exhaustive hyperparmeters search and had to consider some as fixed and definite such as the percentile q of particles to consider, the number of particles per filters, etc... I also did not vary the height of the obstacles as I wanted to originally.

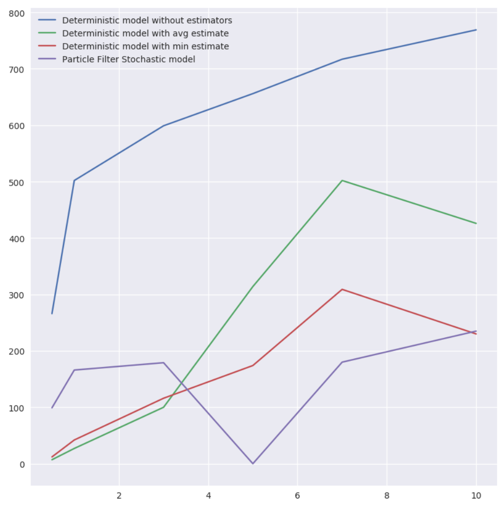
5.3 Vertical Likelihood Update

In this project particle filter implementation, I did not consider at all the correlation between observations at different heights. It would be very interesting to model and encode the shape of the object in the likelihood computation. For instance, if one measures a distance of 7 at height 0, there are great chances that the object is either at distance 6, 7 or 8 at height 1. This could be implemented using a right distribution obtained from the shape of encountered objects. This might be very useful although computationally needy.
All of the code can be found on my github:
http://github.com/Emilien-P/MDP-drone-collision-avoidance/

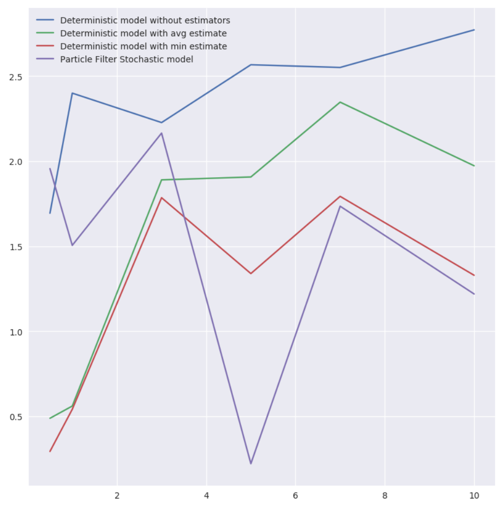
References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

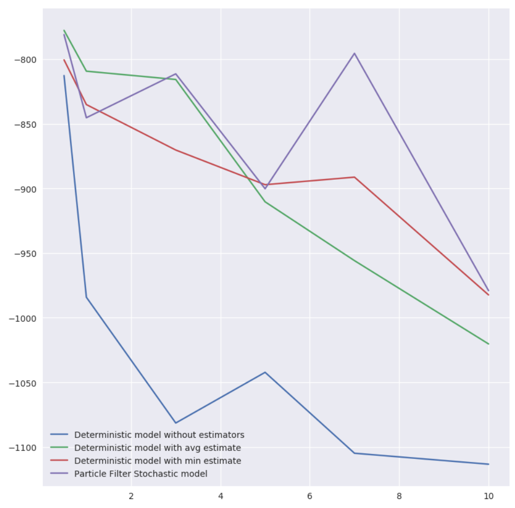
[2] Selim Temizer, Mykel J. Kochenderfer, Leslie P. Kaelbling, Toms Lozano-prez, and James K. Kuchar. Collision avoidance for unmanned aircraft using markov decision processes .



(a) Number of collisions per sigma



(b) Average time in risk per run per sigma



(c) Average cost per run per sigma

Figure 5: Performances of the different agents