Automated Aircraft Touchdown

Rahul Sarkar, Amy Shoemaker, Sagar Vare

Abstract

In this paper, we explore the challenging and important problem of automated aircraft landing. We created a simulator to be able to perform online experiments in various conditions including stormy weather and starting from off-trajectory starting points. We experiment with a wide range of techniques to learn the optimal flight policy including reinforcement learning, Monte Carlo based dynamic programming and direct policy search; and through our experiments we are able to analyze the efficiency of these strategies with respect to the problem at hand. We also discuss our use of parallelization to speed up our computations in cases applicable, and the tradeoff between accuracy and run time. You can find the code for each of the learning algorithms and the simulator at : `https://github.com/sagarvare/CS238-Automated-Aircraft-Landing-Reinforcement-Learning.git`

1 Introduction

Automated flight control systems are installed in all military and commercial aircraft today due to their superior collision avoidance capabilities, their ability to auto-pilot the aircraft under mild weather conditions, and their ability to act as a check against piloting errors. In recent years, reinforcement learning (RL) has been used to design controllers for unmanned aircraft vehicles [1].

While these systems have tremendously increased the safety of air travel by reducing accidents, particular attention is now being paid to take-off and landing, as these are two of the most critical phases during a flight's journey from the point of view of air safety. Hence, there exist automated landing systems (ALS) designed to land aircrafts with high reliability. RL techniques have also been applied to this particular task of landing aerial vehicles. Vision based algorithms have been shown to be capable of helicopter landing and of performing automatic detection of landing sites [4]. RL-based algorithms have even been used to demonstrate how in the future one could perform efficient and accurate spacecraft landings on a planet like Mars, without the need for an offline trajectory [2].

A major issue, however, is that the ALS disengages when the flight conditions are beyond the preset parameter range of the ALS system. This frequently happens in turbulent weather conditions, thereby forcing the pilots to perform a manual landing, which presents a great risk especially for pilots with limited experience and due to limited visibility. Thus there is a need to increase the operational range of the ALS systems. This problem has been examined using recurrent neural networks and genetic algorithms, guiding automated controllers to successfully overcome wind disturbance during the landing stage of an aircraft [3]. We are interested in further investigating the potential for safe, RL-based ALS in the face of high and variable winds.

2 Problem Definition

We implemented an RL agent that guides an aircraft in its final descent phase, during potentially windy weather. In modeling, we considered two simplifications of this four dimensional problem (one time dimension, and three spatial dimensions). First, we considered a model with three dimensions, of which the RL agent needs to control two dimensions (y and z), namely the lateral velocity v_y and the vertical velocity v_z of the aircraft (motion along the x-direction is predefined). Next we considered a model with only two dimension, in which the RL agent must learn to control only the lateral velocity v_y. The unpredictability of the crosswind, especially during stormy weather, leads to sudden lateral changes in the aircraft's position and velocity that alter the planned trajectory of the landing. At each time step, the aircraft has a set of possible actions that it can take, which changes its lateral (and vertical) velocities, taking into account the effect of the wind on its planned trajectory.

Success of the descent is determined by safe landing and by the level of passenger comfort throughout the journey. In considering safety, we require that our aircraft lands near the middle of the runway at a

very minimal horizontal and vertical velocity. To minimize passenger discomfort, we look to minimize jerk and extreme acceleration throughout the entire journey.

MDP Formulation. In tackling this problem, we decided to model the dynamics of the airplane motion as a finite state MDP. In order to achieve this, we were faced with the challenge of translating a continuous state space into to a discrete one. (Note that this discretization does lead to a loss of the Markov property.) In the 2D version of our problem, we consider time t, positions y and z, velocities v_y and v_z, and wind speed v_w as state variables (t, y, z, v_y, v_z, v_w). The actions are defined as $(\Delta v_y, \Delta v_z)$. We create a predefined number of discrete bins for each state and action variable, and assign the actual state or action to the center of the bin it belongs to. In this way, discretization leads to a lack of precision, which can be interpreted as noise associated with sensor measurements of the state and action spaces.

The 2D version of our problem is similar, with the state space represented by (t, y, v_y, v_w) and action space by Δv_y, both discretized and bounded as described above. In this case, we assume that we know the policy to control the motion along the vertical direction and hence do not need to solve for it. In this sense, we have broken the problem down into two sub-problems, one in the z-direction and one in the y-direction, with the distinct goals of achieving a smooth descent and of managing the wind, respectively.

At each step we have a penalty based upon the jerk and the acceleration of the aircraft in any particular direction, which model the level of discomfort the passengers feel. Our model penalizes extreme acceleration disproportionately in the y or z directions, which happens for high values of $|\Delta v_y|$ and $|\Delta v_z|$. To ensure safety, we also have a quadratic penalty if the position of the aircraft strays far from the center of the runway in the lateral direction. The aircraft receives a final penalty based upon its landing position, which is a positive reward if the aircraft lands in the center on the runway and steeply decreases away from the center.

Wind acts as an adversarial phenomenon for our RL agent, as there is a stochastic component to it. To account for this, we model wind as a Markov process with a normal Gaussian distribution for its transition probabilities from one time step to the next. If the wind velocity is v_{w_t} km/hr at time t, then $v_{w_{t+1}} = \mathcal{N}(v_{w_t}, \sigma)$ for some given parameter, σ. Setting a large initial v_w and/or a large σ can help simulate stormy weather. We experiment with the value of σ, to understand the behavior of our aircraft to high wind.

Simulator. All the learning algorithms studied in this project require us to be able to model the dynamics of aircraft motion in the presence of wind and evaluate the utility function for each episode. This is carried out with the help of a simulator. The simulator takes an initial configuration of the continuous state variables (except t which is always discrete) and updates them using a first order integrator for the action received during each time step. Time t is always decremented by 1 second after taking each action. After updating the continuous states of the aircraft, the simulator returns the discrete state associated with the new state of the aircraft. The simulator also returns the reward associated with the action at each time step, including the terminal rewards when the simulator reaches $t = 0$, or if an end state is reached.

3 Experiments and Results

In this section, we describe the various experiments we performed using different algorithms and the results obtained.

3.1 Q-learning over the 3D MDP

In our first experiment, we considered by the y and z dimensions over time, with a fixed speed v_x that allowed us scale our problem to consider t decreasing by one time step with each action.

Algorithm. We ran a simple Q-learning algorithm on the entire 3D problem, with state (t, y, z, v_y, v_z, v_w), and action $(\Delta v_y, \Delta v_z)$, using an ϵ-greedy strategy for exploration. Our discretized state space is $276 \times 101 \times 101 \times 101 \times 41 \times 51$ and action space is 101×101, which makes standard Q-learning unfeasible without function approximation. We began by employing an identity feature extractor that simply used each of the six state variables as features for each action, yielding ~ 6.8 million features. We ran this algorithm through several experiments, considering different values of σ, the wind variance, and different starting states.

Results. Due to the high probability of the aircraft reaching an end state before reaching the runway (via flying out of radar in one of three possible directions ($z_{max}, y_{min}, y_{max}$) or via crash-landing before reaching the runway), the learning rate was very slow. As such, we were not able to reach convergence. The algorithm able to withstand neither wind variability $\sigma > 2$, nor starting states with strong wind. The learning ability also showed an unfortunate susceptibility to the lateral start state variable y_0; as the distance between the plane's start state and the edge of the runway increased, the time before crashing or flying out of radar decreased drastically.

Starting at $t = 275$ (a fixed 5km from the runway), we performed several trials of running our algorithm for 2,000 iterations and for 5,000 iterations. We found that, on average, the plane was able to travel about 50 time steps if trained for 2,000 iterations, and about 100 time steps if trained for 5,000 iterations, before it reached an end state (usually flying out of radar). These results were not much better than random flight (since we found that the best run of the plane did not necessarily occur in later iterations). The slow learning rate combined with the large computational complexity of the problem made performing these experiments intractable. The time it took to run a single iteration scaled linearly with the number of time steps the plane was able to reach. Running our algorithm for 2,000 iterations (where it traveled on average 20 time steps and a maximum of ~ 50) took about 3hrs; Running our algorithm for 5,000 iterations took over 6hrs. Due to the high cost of running this 3D experiment, and the fact that our learning rate appeared to be sub-par, we decided to narrow the problem to two dimensions, focusing on the plane's lateral movement over time.

3.2 Q-learning over the 2D MDP

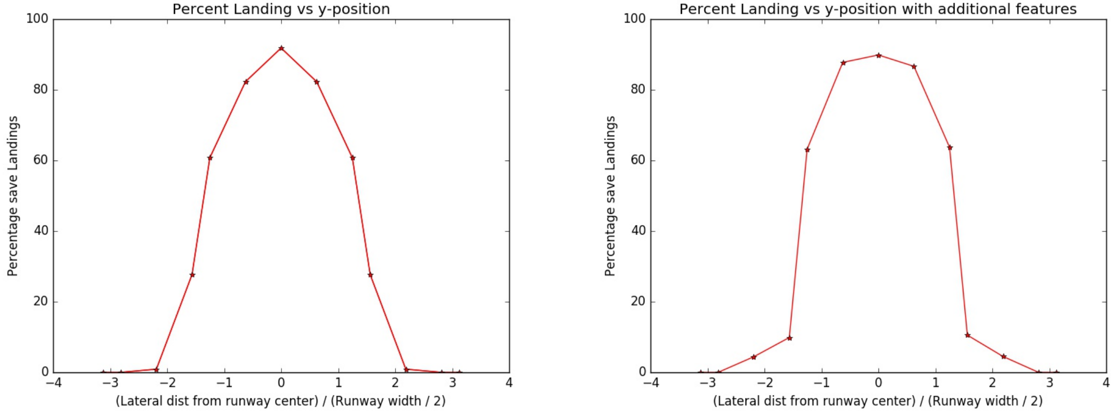
In our next experiment, we reduced the problem to focus on the y-dimension problem independently from the z-dimension. We concerned ourselves only with the y-dimension, as this was the only dimension being affected by wind in our model. Since the y and z directions are independent in our model, in theory, the 2D (t, z) dimension problem could be solved on its own (quite easily, as the z dimension is deterministic) and combined with our y-dimension results.

Algorithms. To solve this 2D problem, we used the same Q-learning algorithm as in the 3D case. First we ran the algorithm with the same identity feature extractor over the individual components of the state: (t, y, v_y, v_w). Next, we decided to add some more intelligently designed features to help the agent estimate the utility from a $(state, action)$ more accurately. Specifically, we added a feature, $\theta = \arctan \frac{y}{t}$, which represents the angle of a direct path from the current location to the center of the runway at $t = 0$, and a feature, $\frac{y}{t} - v_y$, which serves as a measure of the appropriate amount of correction. The motivation for including these features is that they capture the interaction between y and t, to offer the agent some information on how urgently it should correct its position.

Results. The metric used to measure these experiments was to test how many times can the RL agent successfully land the aircraft from different y-positions over a time period of 10 units (seconds). The plots for the two results are shown in Figure 3.1. In both these simulations we see that the aircraft is able to fly on its own and make small corrections to its flying patterns, and hence it has a high success rate even if it is slightly off the runway in these last 10 time steps. But, in both cases it is unable to fly accurately when its distance from the edge of the runway is large.

Surprisingly, the simple identity features works slightly better than designed features (which includes θ and the correction value $\frac{y}{t} - v_w$) together with the identity features, as shown in Figure 3.1. The figures show that identity features produce slightly better results, within the central region. This is probably because the designed features are more informative when the aircraft is off the runway. We also believe that the number of simulations that we ran (15,000 per initial lateral position) were insufficient for convergence, and these designed features were not directly very helpful. Unfortunately parallelization for speed-up is not an option for this algorithm. We also experimented by removing some of the original identity features, but in every case, we got a lower value for the final percentage of correct landings than our original trial, and thus we stuck with the simple identity features.

Other limitations of this algorithm include its robustness against weather. When we varied wind to extreme values, it was unable to pick up on the wind conditions very accurately, in both the feature cases. Using a higher standard deviation σ for the wind leads to a drop in performance if we compare



(a) Percent Safe Landings using identity features (b) Percent Safe Landings using designed features

Figure 3.1: Results of Monte Carlo based dynamic programming, optimal utility function plots

over 15,000 simulations, as expected, because the aircraft needs to learn to fly accurately in windy and calm conditions, which is a very difficult task given the simple identity features. We believe our designed features are not capturing the interaction between the different states accurately enough. On the plus side, the RL agent succeeded in quickly learning to fly quite accurately from the central region in moderately adverse wind conditions. Another advantage of this experiment is the low compute time required for the aircraft to achieve these decent results. However, this comes at the expense of accuracy around the periphery. As this is clearly not a satisfying trade-off, we next discuss a method that overcomes this model's insufficiencies, but does incur a larger computational cost.

3.3 Monte-Carlo Based Dynamic Programming

Our final experiment combines ideas based on Monte-Carlo simulations together with dynamic programming when searching for the optimal policy for the aircraft, for the 2D problem. The basic idea is motivated by the fact that our state space is completely acyclic. This is because the time evolution always decreases t by units of 1 second, and the episode ends either when $t = 0$ or an end state is reached. The algorithm is presented below.

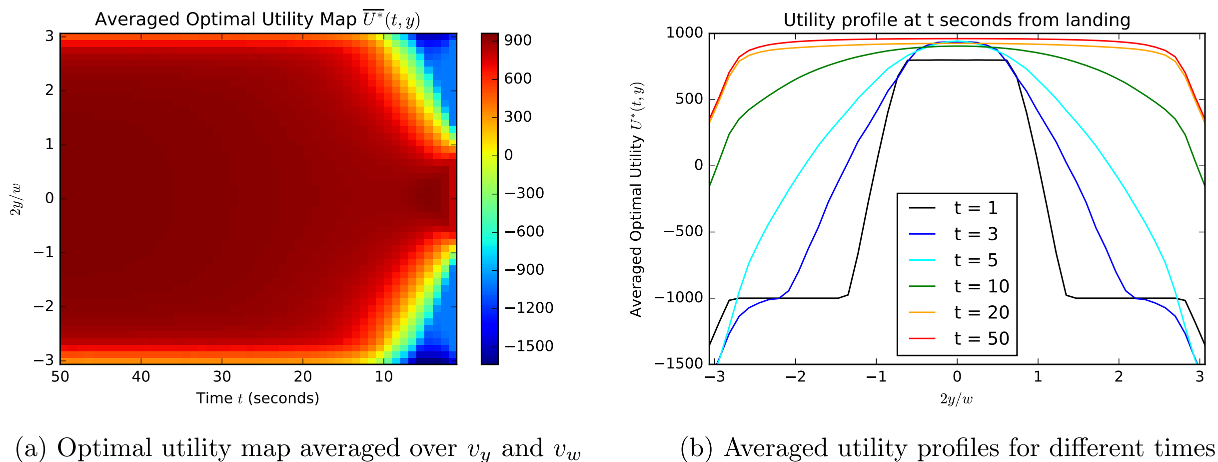
Algorithm. We start from the terminal states $\{s_0 : t = 0 \text{ for state } s_0\}$. For all these states we know that $U^\star(s_0) = 0$ and $\pi^\star(s_0)$ can be any action. Now let us consider all the states $\{s_1 : t = 1 \text{ for state } s_1\}$. The goal is to be able to evaluate $U^\star(s_1)$ and $\pi^\star(s_1)$, from the knowledge of $U^\star(s_0)$. To do this we first note that $Q^\star(s_1, a) = \sum_{s'} T(s_1, a, s')[r(s_1, a, s') + U^\star(s')]$, where $s' = Succ(s_1, a)$ denotes the successor state reached after taking action a from state s_1. But $Succ(s_1, a)$ is equal to some state s_0 for which we already know $U^\star(s_0)$. Hence, if we can evaluate $Q^\star(s_1, a)$, we van evaluate $U^\star(s_1) = \max_a Q^\star(s_1, a)$, and $\pi^\star(s_1) = \arg\max_a Q^\star(s_1, a)$. Clearly this process can be applied recursively so that at each time $t + 1$, we can compute U^* and π^*, from the values of the optimal utility function at time t, which are then stored on disk to a file. This is the dynamic programming idea, and the implementation is very efficient as it only requires us to store the U^* values at time t for the update at time $t + 1$.

We now explain how we can calculate the quantity $Q^\star(s_1, a)$ using Monte-Carlo simulations, for a fixed value of s_1 and a. The idea is to create random initial states that correspond to the discrete state s_1. This is done by sampling each state variable (except t) using a uniform distribution from the interval specified by the min/max values of the bin corresponding to the discrete state s_1. Next, for the ith random initialization we take the action a and compute an estimate of $Q_i^\star(s_1, a) = r(s_1, a, s_i') + U^\star(s_i')$, where s_i' is the next state reached. We repeat this process n times and finally calculate $Q^\star(s_1, a) = \frac{1}{n} \sum_{i=1}^{n} Q_i^\star(s_1, a)$. In our experiment, we choose $n = 5$. Repeating this for every state s_1 and every action a, we get the desired result needed to compute $U^\star(s_1)$.

We noticed that the computation described above is highly parallelizable. The computation of $U^\star$ for each state at any time t can be done independently of all other states. We therefore decided to parallelize our code, and ran it on a 140-core cluster provided by the Institute of Computational Mathematical

and Engineering, Stanford University. By parallelizing the work done at each time t, we achieved a speed up factor of roughly 100x. Instead of requiring 25 minutes per time step, the same computation could be done in under 15 seconds. Note that the specific number of bins for the different state variables and the actions mentioned in this project report were chosen in order to make the Monte-Carlo based dynamic programming computation finish in a reasonable amount of time. Initially, we had a much finer approximation, with about twice the number of bins for each variable, but the computation turned out to be too expensive. While this compromise led to a slight loss of granularity, we believe it is justified by the exponential reduction in compute time. To replicate this experiment with the original fineness of discretization, we would need to run the code on a cluster with either larger number of cores or for much longer time.

Results. Recursively running this Monte-Carlo based dynamic programming algorithm backwards up to time $t = 50$, we found impressive results. In order to verify the correctness of the results, we came up with a novel strategy. First, we checked the optimal utility values $U^\star(s_t)$, where s_t represents all states for time t. However, this plot is hard to visualize as even for fixed t, as we have three axes y, v_y and v_w over which $U^\star$ is defined. Hence, for ease of analysis, we decided to compute an average optimal utility function that only depends on t and y, $\overline{U^\star}(t,y) = \frac{1}{|v_y|\times|v_w|}\sum_{v_y,v_w} U^\star(t,y,v_y,v_w)$, where $|v_y|$ and $|v_w|$ represent the number of bins for v_y and v_w respectively.



(a) Optimal utility map averaged over v_y and v_w

(b) Averaged utility profiles for different times

Figure 3.2: Results of Monte Carlo based dynamic programming, plots of $\overline{U^\star}(t,y)$

.

$\overline{U^\star}(t,y)$ is plotted in Figure 3.2a as a color map, with t along the horizontal axis and $\frac{2y}{w}$ along the vertical axis, where w is the width of the runway. The location of the runway is to the right end of the figure. The quantity $\frac{2y}{w}$ functions as a scaled distance along the y direction, such that the center of the runway has value 0 and the edges of the runway have values 1 or -1. We can notice from this plot that when we are very close to the runway, i.e. $t \to 0$, we have very large negative values of average optimal utility for larger values of $|\frac{2y}{w}|$, indicating that the plane crashed under these conditions. This is expected, as the plane cannot land safely when it does not have enough time to correct its position into the landing strip. However, we also see that as time increases, the rewards are always positive indicating that successful landing is possible using an optimal policy. One can also note that the plot is symmetric about the line $\frac{2y}{w} = 0$, which is expected from the symmetry of the problem. These observations are a good validation for our computations.

In Figure 3.2b, we plot slices of the quantity $\overline{U^\star}(t,y)$ for $t = 1, 3, 5, 10, 20, 50$. It is interesting to note that for the curves corresponding to $t = 1$ and $t = 3$, we see that the agent lands the aircraft safely when it is initialized close to the runway, but the rewards get more and more negative as the plane is initialized farther away from the center of the runway, indicated more failures. Then comes a flat region of negative utilities for these $t = 1, 3$ curves, which corresponds to the fact that even if the airplane took the largest possible action to move towards the runway, it is incapable reaching the runway center, and hence the optimal policy under these circumstances is not to do anything and simply crash, as taking a drastic

action will incur a jerk penalty in addition to the inevitable crash penalty. Now, if we look at the curves for $t = 5$ and $t = 10$, we see that the flat region of negative utilities disappeared, which means that the aircraft has an improved chance of landing from any position along y. Finally, if we look at the curves for $t = 20$ and $t = 50$ we see that the curves are nearly the same and are nearly flat, with positive utilities everywhere. This suggests that unless the aircraft is exactly at the edge (which is an end state), it will always be able to land safely with the optimal policy.

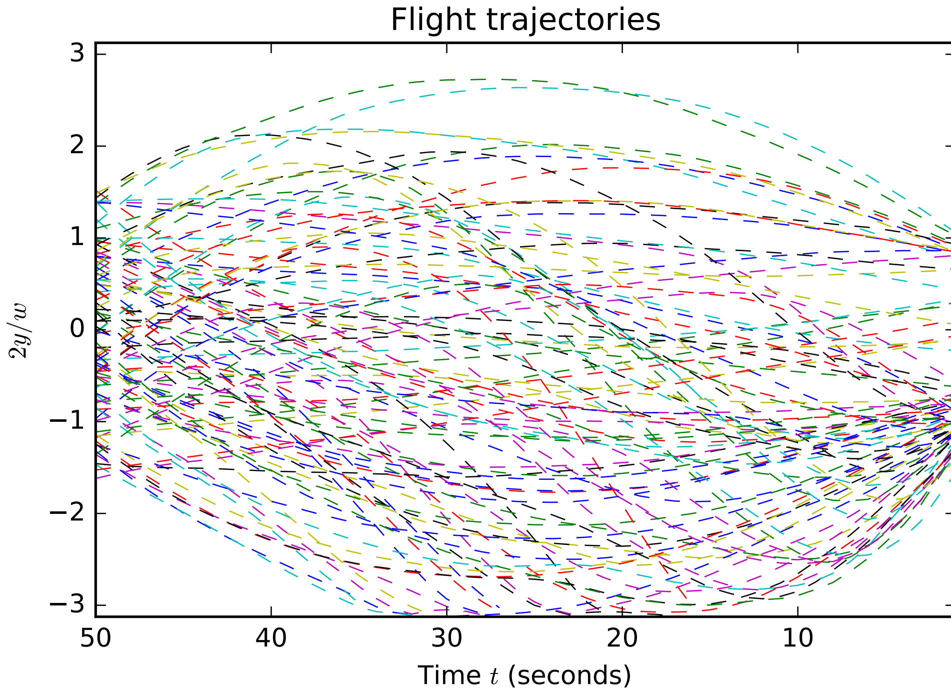


Figure 3.3: Flight trajectories over time for 100 different episodes, using the optimal policies learned from Monte Carlo based dynamic programming.

In Figure 3.3 we plot the aircraft trajectories resulting from running the simulation with an agent that follows the optimal policy learned through Monte Carlo based dynamic programming. 100 different episodes are plotted in the figure. We initialize the agent with a lateral position chosen randomly from within a reasonable range, and with $t = 50$ seconds before the aircraft must land. At each time step, based upon the state of the aircraft, the agent takes an action that it learned, and we plot its path. We see that it learns to correct its path smoothly, which indicates an accurate effect of the quadratic penalty for jerk. We see that almost all of the trajectories reach the runway successfully, showing that the agent learned to take optimal actions for landing the aircraft even in adverse wind conditions.

3.4 Directed Policy Search

Algorithm. For each state (t, y, v_y, v_w), we compute the feature vector $\vec{\phi} = (1, \frac{y}{t}, \frac{y}{t}^2, v_y, v_y^2, v_y \cdot \frac{y}{t}, v_w)$. Using some predefined weight vector $\vec{W}$, we compute a mapping from these features to the action space as $\vec{W} \cdot \vec{\phi} \mapsto action$. This is a parameterization of the policy. Given a weight vector $\vec{W}$ we can run multiple simulations with this policy and approximate the expected utility with an average over the different runs: $\mathbb{E}[U^\pi] = \frac{1}{n} \sum_i^n U_i(\vec{W})$, where U_i is the utility of the ith rollout of π. The next step would be to optimize the weight vector so as to improve the expected utility, using a method like cross-entropy. Our theoretical assessment is that this is an interesting idea and seems to be a promising path to follow, but due to the lack of time we could not pursue it. The computation of the estimated utility requires multiple simulations which can all be parallelized making this method feasible with enough computational resources ! We believe that the policy space might be much easier to search instead of computing the utility over the entire state space, and we would like to consider further developing and pursuing this idea in the future.

4 Conclusion

In tackling the problem of automated aircraft landing, we found the original problem in 3D was too large, so we broke it into two sub-problems, and worked on the more interesting sub-problem along the y dimension. We implemented a RL agent, and compared the successes of the two function approximations in landing the plane. Interestingly, we found that the results did not improve with additional features, beyond the straightforward identity features. We saw that the method quickly learns to land correctly from the central region in moderately adverse wind conditions, but has several limitations in terms of initial conditions. The RL agent performs poorly on the periphery of the runway and when wind is highly variable, possibly due to insufficient number of trial runs. Successful use of reinforcement learning techniques requires good feature definitions that capture the interaction of the different state variables in the computation of the utility function. We showed that the current feature definitions learn, but slowly.

Exploiting the acyclic structure of our problem, we also implemented the interesting method of Monte Carlo based dynamic programming with parallelization. Through graphical visualizations of our results, we displayed the superior performance of this method as compared to the simple RL agent, albeit at the cost of higher computation time.

Future Work. We are particularly interested in exploring the direct policy search algorithm laid out in Section 3.4. We believe that with parallelization, this method shows some promise and might work well even in the larger 3D MDP setting. We would also like to experiment more with the different features in the 2D MDP case, to see if we can get some good features that make the learning task for the values of $Q(s, a)$ much faster for the RL agent, and by extension accurately capture the interactive term between the states. We also want to test out the Monte Carlo based dynamic programming on the larger 3D MDP, in hopes of getting results comparable to the 2D case. Finally, we would like to more thoroughly and systematically test the parameters associated with wind, to better understand the ways in which the RL agent learns under different conditions.

5 Acknowledgments

We would like to thank Mykel Kochenderfer for piquing our interest in this material, for being so supportive and caring, and for being such a great professor. We'd also like to thank the TAs of CS238 for all of their time and effort.

Division of Labor. All theoretical ideas were discussed and planned out as a group. In terms of coding, Amy and Rahul worked on the simulator, Sagar wrote the first draft of Q-learning, Amy wrote the second draft of Q-learning, Rahul implemented Monte Carlo based dynamic programming and Sagar wrote the parallelization code for it.

References

[1] Haitham Bou-Ammar, Holger Voos, and Wolfgang Ertel. Controller design for quadrotor uavs using reinforcement learning. In *2010 IEEE International Conference on Control Applications*, pages 2130–2135. IEEE, 2010.

[2] B. Gaudet and R. Furfaro. Adaptive pinpoint and fuel efficient mars landing using reinforcement learning. *IEEE/CAA Journal of Automatica Sinica*, 1(4):397–411, Oct 2014.

[3] Jih-Gau Juang, Hou-Kai Chiou, and Li-Hsiang Chien. Analysis and comparison of aircraft landing control using recurrent neural networks and genetic algorithms approaches. *Neurocomputing*, 71(16âĂŞ18):3224 – 3238, 2008.

[4] S. Saripalli, J. F. Montgomery, and G. S. Sukhatme. Visually guided landing of an unmanned aerial vehicle. *IEEE Transactions on Robotics and Automation*, 19(3):371–380, June 2003.