Autonomous RC Car Drifting

Samuel Henry and Andrew Perrault

Keywords: **Simulation, Apprenticeship Learning, Sensing.**
Robot: **Modified Drifting Mazda RX7 RC Car.**

***Abstract*—The goal of this project is to develop a controller for drifting an RC car through a pre-specified course. For the early stages of this problem, we restricted the course to 90-degree turns. The 4WD car has limited turning radius and low friction rear tires, so it must drift in order to turn around tight corners. The car is localized using sonar, wheel encoder and gyroscope, and it uses this localization data to access spatially-indexed control policies, both open- and closed-loop.**

I. Introduction

CONTROL algorithms in robotic cars often neglect slipping due to its unpredictable nature. Slipping depends on the road surface, and if speeds are kept low enough while turning, no slipping will occur. This project uses slipping for tight turns and handles it by switching between open- and closed-loop driving policies. The Stanford Autonomous Helicopter Project([1],[2]), copes with a similar problem by building a simulator, and actual odometry data to iteratively improve it (in the style of discovering the state space of an MDP). It uses demonstrations to estimate ideal trajectories, and improves this via local search. We hope to use a simulator to capture the general behavior of a slipping car, and use it as a tool for creating control regimes. A challenging aspect of this approach was to building a simulator that accurately depicts the essence of sliding behavior. Our access to odometry data was limited, so comparisons with reality were difficult. The method of simulating slipping was quite arbitrary, thus during this semester, we never got the simulator to a point where we could apply it to the real car. We did however greatly advance the state of the simulator and it should prove invaluable for future research. We developed the platform as we went, so unfortunately we made several breakthroughs in sensing and control that were too late to investigate thoroughly before the paper deadline.

For our non-simulator approach, we learned from expert demonstrations (as in [2]) and attempted to generalize it for a problem space with two main degrees of freedom: velocity going into a turn and battery level (to a limited extent). We used manual control on the car to make it drift 90 degrees at different speeds and different battery levels. Using these recorded runs we attempted to extract an ideal regime of policies. We mixed open-loop controllers for turns with simple closed-loop controllers for straights in order to achieve a level of autonomous driving. Towards the end of the development period, with improved sensing data, we developed what we believe to be a closed-loop controller for drifting – it is described, but we weren't able to fully test/implement it.

Four relevant prior results are [1],[2],[3] and [4]. [3] describes how to convert a time-indexed policy into a space-indexed one and how to deal with state transitions by linearizing the dynamics around them. This approach seems to require a higher degree of sophistication in odometry than we had access to, as well as a higher degree of control accuracy. [1] describes how creating the simulator and learning a trick were combined in an iterative test process, which also is odometry intensive and thus out of reach. [2] describes how imitation learning was used in conjunction with the simulator to learn from many expert performances. [4] describes a very similar problem to ours, but the algorithmic focus of the paper is in transitioning between open- and closed-loop controllers in a more sophisticated and detailed manner than our implementation.

Our simulation approach uses the Open Dynamics Engine, models the car simplistically, focuses on capturing the drift motion and roughly estimates other dynamics.

Samuel Henry is a Computer Science MEng student at Cornell University (netID: sth56).

Andrew Perrault is a junior Computer Science undergraduate student at Cornell University (netID: arp86).

II. Hardware

A. Base RC Car Specification

The car has a wheelbase of around 25 cm and an axle length of around 10 cm. The front wheels have rubber tires and thus grip much better than the smooth, plastic rear wheels (this is the primary cause of the drifting). The car is 4WD.

B. Sonar

The wide cone means the sonar is likely to detect closer objects that aren't directly in front of it, rather than farther ones directly ahead of it. This can be problematic in threshold detection because the sonar will sometimes pickup the edges of the side walls resulting in spikes in the data. Also, the sonar will not work at too much of an oblique angle to a surface – around 45 degrees is the maximum. The sonar has an update rate of 40 Hz and a maximum range of around 3.5 meters.

C. Batteries

We used three Tyco RC batteries. Deciding when to switch batteries was done pretty unscientifically. The batteries showed different levels of wear, which created problems with standardization between batteries.

D. Wheel Encoder

Hand-constructed wheel encoder with originally, 5 black, 5 white regions, and then updated to 10 black, 10 white regions for better resolution. The error in velocity measurement is the width of a region divided by the update time (30 ms for this sensor), in our case about .33 m/s. Could be reduced by slowing the update rate, but this sensor is definitely the least accurate for our purposes.

E. Communication

XBee wireless controller. We had some problems with dropped packets, which is a major problem in our original setup with 10 Hz control instructions. The car returned sensor data every 30 ms in a comma-separated values format. This was a serious problem in observing certain quantities, and so we created a precise controller where instructions occur at 1000 Hz on the microcontroller of the car itself. With this setup, we can send data updates at 200 Hz.

F. Gyroscope

+/- 300 degs/sec gyroscope. The gyroscope has a high level of standing noise due to its low resolution. However, the car turns 180 degrees in less than a second while drifting, so it is necessary.

III. Simulator

A. Reasons for Simulation

The use of a simulator allows for the complex physics of drifting to be modeled. This is particularly important for this project because the commands or rules necessary perform drifts are difficult and time consuming to construct. Multiple turns, especially in succession are near impossible to formulate due to the complex physics necessary of drifting.

The simulator's main purpose is to provide a learning environment for machine learning algorithms. Using a simulator is much faster than learning with the actual robot, and because of the complexities of drifting, mathematical models are difficult to formulate. It is essential that the simulation accurately reflect reality, since the physical robot will execute the rules or commands learned.

At the core of the simulator is a physics simulation. The physics engine used is pyODE, a python implementation of the Open Dynamics Engine, an open source physics engine (http://pyode.sourceforge.net/). pyODE provides the functionality for the primary physics equations and integrators used in simulation. It also provides methods for collision detection and response. Using pyODE at its core we built a framework that allows for minimal understanding of the underlying system and ease of use. This allows for future work to focus solely on learning algorithms without spending time understanding or modifying the simulator. To achieve this, the existing code-base was expanded, documented, and converted to an object oriented design.

B. Implementation

Previous work had been done with simulator and

laid the foundation for our simulator implementation. The rendering, track structure and methods, and a basic simulation loop were completed. The simulation had to be hardcoded into individual files in order to execute different commands or use different tracks. We created a new class structure and strived to create an object oriented and well-documented simulator that would be easy to use. Classes with constructors, state information, accessor and mutator methods have been created and the code has been well documented allowing for quick understanding of what methods are available and the functionality they provide. In order to aid improvements that may need to be made to the simulator global constants are used rather than magic numbers. The pre-existing code base was also modified to abide by these standards.

At the heart of the new design is the simulator class. This is the central class where the main simulation loop is executed. It allows for easy swapping between different tracks and sets of commands or command algorithms. In order to create a simulator, you simply specify the track and the command algorithm that you would like to use. To extend the functionality of the simulator several options, such as command or state logging, disabling rendering, and a few others may be specified.

Since algorithms that generate commands are vital to future progress the simulator allows them to be easily swapped by passing in different instance of the driver class. The driver class issues commands, and is the only class that should require modification for different planning and control algorithms. During each iteration simulation loop the simulator asks the driver for a command, and the driver issues the command. In order to make a decision, the driver has access to information regarding the global state of the system via the track and local state of the car, using these 2 objects the entire state of the system is defined, and intelligent planning and control systems can be implemented.

C. Car Movement

One of the major design decisions of the simulator was how to simulate the movement of the car. Several options were considered until we decided that rotation the wheels to simulate linear acceleration and turning the wheels to provide turning was the best option. Below are methods considered for motion and their consequences:

Perhaps the simplest method to simulate motion is to set the velocity of the car. This is allows for any state of the car to be easily manipulated. However setting speeds immediately is very unstable due to the way in which the simulation is integrated. By setting velocities, energy is added or taken away from the physics system. Since the system is based on physics equations that assume a conservation of energy, creating energy from nothing or destroying energy causes the equations to breakdown, and the simulation quickly degenerates. The same can be said for setting the orientation of the car instantly. Angular momentum follows similar laws as linear momentum, so instability quickly creeps in. Setting the orientation directly also fails to capture the physics of the drift and the interactions between the tires and the ground, which is the primary purpose of the simulation.

Another idea was to apply force to the car. Unlike setting velocities, applying a force does not cause instabilities in the physics system. Application of force worked well for linear acceleration, but by applying forces to turn the car the interactions between the surface and the tires is not modeled, so the system does not capture reality.

The method implemented, and the most reflective of reality is to have the tires move the car. Torques are applied to the tires and due to friction between the tire and the ground the car is propelled forward. Turning is achieved by turning the front tires, which consequently changes the orientation of the car. This system allows for intuitive use and manages to capture the physical properties of drifting. This also allows for actions to be more easily mapped to the actual robot itself. Actions learned in the simulator can be easily defined as ‘apply throttle’ and ‘turn the wheels’ and correspond directly to actions taken by the robot. This allow for minimal translation between

simulation and reality.

D. Drifting

The simulator has the ability to simulate drifting. This is achieved by using different friction coefficients for forces perpendicular to wheel rotation and forces parallel to rotation. Another key to simulating drift is setting different values of slip for the front and back tires. When the back tires can slip more easily drifting is easily achieved. We found that it was best to prevent the front tires from slipping at all, and allow the back to slip very easily. This is not entirely separate from reality, as the robot has rubber treads on the front tires, which provide a lot of grip, and the rear are smooth plastic which easily slip. Both of these conditions are applied during collision response in the simulator

E. Interfacing with the Robot

The robot can easily execute commands learned during simulation. As the simulation takes place, the commands being executed may be logged. These commands are saved to a text file and specify the throttle, direction (backward or forward) and the angle of the wheels. The commands are logged at the same pace as they are executed by the robot and allow the robot to parse the list as it runs.

While running the robot can collect and log sensor data. This data is easily incorporated into the simulator via file readers and interpreters. Data from front sonar or line detectors can be easily converted to linear acceleration using methods we created. More recently data collected from the robot has grown more robust and allows for velocities to be more directly determined from odometry on the wheels.

Both of these methods of interfacing with the robot are particularly useful when tuning simulation parameters to match reality.

F. Approximating Reality

The simulator must approximate reality in order to be useful. The most basic implementation of this is to match the physical characteristics of the car in the simulation. The car is approximated by a box and 4-cylinders. The box has the same scale dimensions (length, width, height) of the body of the physical car. Each cylinder corresponds to a wheel and the dimensions of the wheels (radius, width) also match the physical car. Furthermore different grips between tires is approximated by applying more torque to the front tires than the back tires. Perpendicular grip is approximated by allowing the back tires to slip and not allowing the front tires to slip (as explained in the drift section).

The acceleration of the car must match reality, and to achieve this, I use data collected from both the sensors on the robot and from the state of the car in simulation. Using gradient descent I adjust an acceleration parameter until the two data sets match to a specified degree of precision. This process begins by creating a set of commands, which easily can be captured by running the simulator and logging commands, or by manually coding commands in the correct format. Since the goal is to match linear acceleration, the commands consist of simply accelerating forward for 1 second using different levels of throttle. The throttle was divided into ten levels, ranging from 10 percent of throttle to 100 percent of throttle with 10 percent intervals separating each level. The result is 10 sets of commands each 1 second in length.

The physical car executes these commands while it collects data. That data is stored and linear velocity is approximated using either the line detectors or the front sonar. For these experiments front sonar was the primary sensor used. The sensor data is converted to velocity over time, and is used as the data to fit by the simulator.

The simulator then iteratively executes the same set of commands and adjusts, as a function of the error the parameter that controls linear acceleration. After many iterations the simulator matches the real data. This process is automated, and is implemented in the SimulatorAdjuster class.

G. The "Flying Off" Problem

Using the wheels to propel that car works well, but it presented a unique problem, the car tended to fly off the track. This behavior began as a front wheelie and continued as the car flipped wildly

through the air.

The cause of this is most likely numerical error created by the large forces of friction between the tires and the surface. This friction is so great that the car gains upward momentum and then due to imprecise integration the error explodes. The problem can be eliminated by lowering either the tire or track friction. Lowering friction is not a solution and causes a serious problem. Friction is the force responsible for acceleration and less friction therefore causes less acceleration. To ensure this problem would not occur, the friction must be lowered to the point where the car accelerates much slower than reality. In fact it accelerates so slowly that the simulation becomes unusable. Another idea was to increase the weight of the car. Since more weight requires more force to push the car upward, increased weight reduced the 'flying off' effect but did not solve the problem. More weight suffers from the same disadvantages of decreased friction; more weight also causes the car to accelerate slower. To ensure the car would never 'fly off' the weight needed to be increased to the point where acceleration was so slow it rendered the simulation useless. Increased weight also caused a problem with collision response. The car would occasionally sink through the floor. This is a result of numerical error during the calculation of collision forces between the tires and the ground.

Another attempt at fixing this effect was to lower the center of mass of the car. This did indeed reduce the occurrence of the effect, and also reduced the occurrence of the car flipping while turning. When performing tight turns at high speeds the centrifugal force can cause the car to flip. This is an acceptable reaction, but the original car model flipped too frequently. To create a lower center of mass I simply flattened the body of the car and lowered it with respect to the wheels.

When watching the car 'fly off' the car would first do a wheelie and then go in the air. This seemed to indicate that the back tires are doing the majority of the gripping causing the 'flying off' effect to occur. To solve this I lowered the friction of the back tires. This didn't seem to help, even when turning off any torque to the back tires or removing all friction from the back tires the effect continued. Different grips are still used in the simulation, and as explained in the drifting section, are essential to accomplishing drift.

Another possible cause of the ‘flying off’ effect was that the tires were colliding with the car body and the simulator was therefore applying collision forces to prevent interpenetration. Since the wheels are connected to the body any penalty forces would affect the body of the car as well as the wheels. Penalty forces tend to be very strong, and can cause instabilities in simulation, especially when time steps are large. After preventing these collisions by modifying the collision response code the ‘flying off’ effect continues and we ruled this out as the cause.

After trying the solutions explained above I switched my attention away from the car model towards the global simulation. During simulation, integration is approximated using time steps. If time step steps become infinitely small the integration is exact, but the approximation becomes increasingly inaccurate as the size of the time steps increase. pyODE uses a semi-implicit integrator which is more sensitive to this problem that other integration methods. By decreasing the time step size I hoped to eliminate the problem. Unfortunately even with very small time steps the ‘flying off’ effect continued. I believe that if time steps were small enough the flying off effect would not occur, but the simulation would become so slow that it would become unusable, and since the main purpose of the simulator is to train algorithms faster than the actual robot, speed is important and timesteps could not be reduced to an adequate size.

The last attempts prevent the car from moving in the y-direction. This can be achieved in two ways, pin the car the X-Z plane or sandwich the car in between two planes. Sandwiching the car between 2 planes is achieved by creating an invisible plane just above the car model. Any movement in the y-direction causes the car to collide with the overhead plane and therefore the movement will be stopped using collision forces. After implementing this, I found it to be just as

buggy and problematic as the original 'flying off' problem. The collision forces can quickly spiral out of control as the car bounces between the two planes and due to numerical error it would eventually pop through one of the planes and the simulation would therefore be broken.

The final attempt and current solution is to pin the car to the X-Z plane. To achieve this I set any angular or linear velocity in the positive y-direction to zero before integration occurs. I also set any forces pushing in the y-direction to 0. This solves the problem and the car can accelerate as fast as necessary. The car will sometimes still go in the y-direction, due to numerical error and collision response, but since acceleration and velocity is only stopped in the positive y-direction, the car quickly falls back to the ground. Also any motion in the y-direction tends to be fairly small, since the time steps are small. Pinning the car to the X-Z plane was the final solution to the 'flying off' effect and last component necessary to get the simulation in working order.

H. Reinforcement Learning

Since the main purpose of the simulator is for learning, particularly reinforcement learning, I created a rudimentary reinforcement learner. The learner works, and specifies how future teams might go about implementing a more robust, effective and efficient learner.

The reinforcement learner uses discrete states that consist of a four-tuple: x-position, y position, orientation and speed. Actions are defined as a tuples, consisting of throttle and the wheel angle. The actions correspond to actions that can be sent to the actual robot. Since the simulation is deterministic, transition probabilities are always 1, and do not need to be estimated. Transitions are instead calculated using a parallel simulator that determines the state of the car after an action is made.

The reinforcement learner uses the driver class to determine the best policy on the fly, so as the car moves around, the optimal path is determined based on its set of actions and learned state values. Rewards are always negative to encourage a speedy path to the goal. High penalties are given when the car drives off the edge, and the only positive reward is given at the goal state. Currently the goal state is just a position on the track, speed and orientation do not matter.

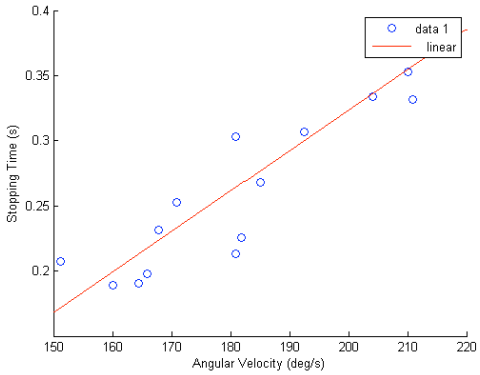
One of the problems that I came across was that because the transition probabilities are deterministic exploration was limited. Once a working path was found the car would stick to it and the learning would become stagnant. In order to encourage more exploration of the state space I added a random element, by causes a random action to occasionally occur.

IV. Algorithmic Approach

A. Apprenticeship Learning

Initially, our approach was to mix a closed-loop straightening controller with open-loop controllers for the turns. What ended up happening was that the open-loop controller would give non-deterministic results, even on a 1000 Hz controller. We attribute this to the wild fluctuations in battery level observed in the linear and angular velocities from run to run, as well as probably the unevenness of the carpet surface. So, we upgraded our hardware to allow for a closed-loop turn controller. The most reliable way we found to do this, after much experimentation, is to run a large number of open-loop trials with the same controller, and observe their behavior. Specifically, for a 90-degree right turn, we aim to perform the drift by turning the wheels right and setting the throttle high to initiate the drift and then reversing the wheels and killing the throttle to stop the drift. What we measured was the amount of time and angle covered by the spinning car when the throttle is killed and the wheels are reversed at different angular velocities. All of the trials of the same open-loop controller behave mostly the same way, so we can informally control for other factors – all except for the entry speed into the turn, which we had to break the policies into regimes for.

The relationship between angular velocity and stopping time is shown below for one linear velocity region. The relationship seemed linear to us, so we applied linear regression.



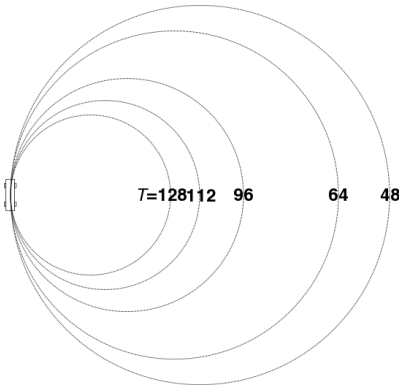
Since we applied different regressions to different regions of the data, we are performing local regression. If we hard-code a controller that estimates how far it will turn if it were to stop at each timestep, we can perform turns with low error (see our video, in which four turns are linked without straightening). Another approach we could use is to keep track of the number of degrees turned each time, and for a box-like course, we could then correct for error in a previous turn by setting a different target for the next one.

B. Straightening Controller

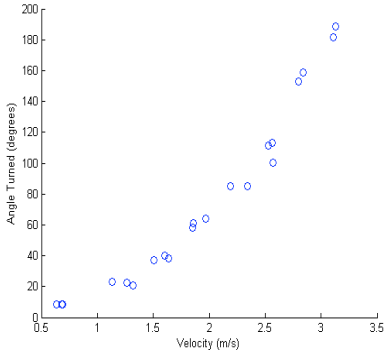
We created a very basic PID closed-loop straightening controller, using only sonar. This was meant to take control of the car between turns. Since turning proved so difficult, we spent most of our time working on it, rather than the straightening controller – which would have been a matter of applying known techniques and tuning them. For example, we could have written a Kalman filter for the car's angle with respect to the side wall, as measured by sonar, gyro and using the below control responses.

C. Data

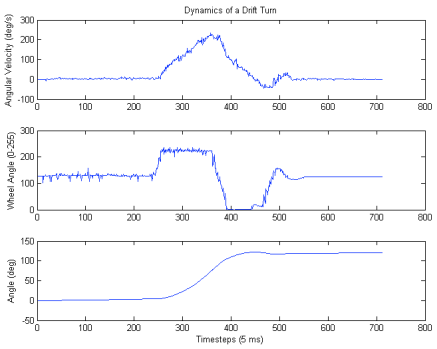
We did several successful experiments in measuring the behavior of the car. Below is the angular turning rate at different wheel angles in a non-slipping regime. This was used in the closed controller for straightening (reducing angle error to zero) and could be applied to a Kalman filter for angle or position.



We also measured the amount of energy required to drift by doing an experiment where we drive the car directly forward, and then turn with no throttle. It is not practical to drive the car on a course this way because it is too fast and uncontrollable, but it shows the energy threshold for drifting quite clearly.



When we gained access to data at a very good update rate, we were able to get much better idea of the physics of a drifting turn. During this time, we were using a strategy of turning the wheels, applying a throttle to spin the car, and then reversing the wheels to stop the drift. We hoped to determine the time required to go from any angular velocity to an angular velocity of zero. Thus creating a controller that calculates when to stop the car during a 90-degree turn based on the current angle and angular velocity.



V. Extensions

A. Improved Odometry and Sensing

Of the papers we surveyed, our robot was the weakest in terms of odometry. All others used GPS, often with high precision, to gain access to absolute position data. In our case, GPS would have greatly simplified the sensing problem. Additionally we could have used a greater degree of environmental specification or algorithm sophistication to get a similar result. For example, we could have implemented a particle filter (although it would not be easy because of the degree of isometry in our testing environment) or a Kalman filter that integrated knowledge of control response. There are probably other techniques that could be implemented as well, but any of them could have contributed to a more sophisticated apprenticeship learning approach (as in [4]) or used to train the simulator (as in [1]).

Odometry also limited our approaches to more sophisticated turning problems; for example, a slowly-curving turn would be difficult to run in our current setup because we would not be able to detect when the turn began, as we would need to see some wall perpendicular to our current heading to detect the onset of a turn.

These problems could also be addressed with improved sensing, with lasers for a high-power solution, or perhaps even a single camera. Sonar, in our current setup, suffers from large spikes – which have to be filtered out, causing a delay in threshold detection. Sonar also has a relatively low max range, around 3.5 meters. These two problems, compounded with a low instruction rate (see below), make it difficult to drive quickly and make quick decisions.

B. Robustness

We encountered several problems of robustness in our approach. One problem was the battery. We tried to avoid modeling to the battery because of the lack of interesting results to be observed in that area. **We used results that were battery independent, and also used the operation on a full battery as the canonical battery level.** As detailed in the battery section this had several undesirable consequences. There were differences in the degradation of our batteries as well. This made some theoretical methods more difficult to apply, and it also meant that the compromises we made because of the battery (a fair degree of hand tuning) made our open- and closed-loop controllers less robust.

Another area of difficulty was the slow instruction update rate (10 Hz). Most drift turns take less than a second, with the crucial information of when to stop the drift and let the car slide requiring high precision. Our slow update rate made adding a closed-loop component to our drift controller difficult because of the large amount of error incurred as a result of a bad reading or dropped packet. In order to eliminate this problem, we built software to run instructions at 1000 Hz on the microcontroller and send data back at 200 Hz.

C. True Expert Driver

Our ideal runs were performed through the keyboard controller by our team, none of which had experience drifting remote control cars. It would be interesting to see how someone who is a true expert could drive to minimize the error of the open-loop components. For example, we did little throttle management during the turn. It's possible that better control actions could allow for more error in input error.

D. Transitioning Between Open and Closed Loop Controllers

We took a very simplistic approach to transitioning between open- and closed-loop controllers – some kind of odometry condition usually. [4] presents an interesting method of probabilistic transitioning between open- and closed-loop controllers.

VI. Conclusion

We had to make many compromises in approaching our goals for the project, but we still believe that many of the problems we encounter are solvable. Our project is difficult in several of the main areas of robotics, but particularly, we did not expect the degree of the problems we encountered in sensing and communication. We were also building the platform as we went, which

was quite time-consuming. Although we invested a very large amount of time on the project during the semester, it wasn't until the end where we thought we might have some chance of solving the problem satisfactorily (due to the wheel encoder and gyroscope – new sensors, and the improved communication method).

ACKNOWLEDGMENT

Jonathan Diamond
Nan Rong

REFERENCES

[1] Learning for Control from Multiple Demonstrations, Adam Coates, Pieter Abbeel, and Andrew Y. Ng. ICML, 2008.

[2] Autonomous Autorotation of an RC Helicopter, Pieter Abbeel, Adam Coates, Timothy Hunter, and Andrew Y. Ng. In International Symposium on Robotics, 2008

[3] Space-indexed Dynamic Programming: Learning to Follow Trajectories. J. Zico Kolter, Adam Coates, Andrew Y. Ng, Yi Gu, and Charles DuHadway. ICML, 2008.

[4] A Probabilistic Approach to Mixed Open-loop and Closed-loop Control, with Application to Extreme Autonomous Driving. J. Zico Kolter, Christian Plagemann, David T. Jackson, Andrew Y. Ng, and Sebastian Thrun. To appear in *Proceedings of the International Conference on Robotics and Automation* (ICRA), 2010.