Kalman Filter Lab

CS 470 Lab 4

Duane Johnson, Michael Ries and Jon Brandenburg

June 5, 2009

Abstract

This lab is focused on learning the basics of implementing a Kalman filter. The participants had to rely on “noisy” sensors which added normally distributed noise to each dimension independently. In order to evaluate the perfomance of the Kalman filter the participants had to implement an agent which would try to shoot another tank based on the filtered value of where it believed it was and where it believed the other tank was. Finally the participants implemented a set of “clay pigeons” with different, but well-defined behaviors to help them assess how the Kalman filter was behaving.

Contents

Chapter 1

Lab Context

1.1 Summary

The authors each developed and ran the software below on laptops. The laptops were unix-based systems; however, these results should be easily duplicated on a Windows machine.

1.2 Hardware

- Macbook Pro (32 bit) / 2.4 GHz Intel Core 2 Duo / 2 GB RAM
- Lenovo T400 (32 bit) / 2.2 GHz Intel Core 2 Duo / 2 GB RAM
- HP Pavilion (64 bit) / 2.5 GHz Intel Core 2 Duo / 4 GB RAM

1.3 Software

- Mac OS X 10.5
- Ubuntu 9.04
- bzFlag 2.0.7 with patched bzrobots client[1]
- Ruby Interpreter, version 1.8.7 (or 1.9.1)

[1]Supplied by CS470 in *Install Instructions*

- EventMachine Library[2] for Ruby
- PDF-Writer Library[3] for Ruby
- Rake Library[4] for Ruby Unit Tests
- NArray and NMatrix [5] for Ruby

[2]http://rubyeventmachine.com/
[3]http://ruby-pdf.rubyforge.org/pdf-writer/
[4]http://rake.rubyforge.org/
[5]http://narray.rubyforge.org/

Chapter 2

Experiences and Time Spent

2.1 Duane's Summary

My assignment for this project was primarily to implement the Kalman filter and to show its output in PDF form for the report. It was actually quite fun to turn the Kalman math into code using Ruby's NArray gem. Because Ruby supports operator overloading, the matrix math was nearly as easy to code up as any algebraic equation. We had to tweak the timing variable a bit so that our $t + 1$ matrices properly represented the "next iteration" of the cycle.

The "screenshots" in this report were produced using PDF output. Part of my contribution was to generate the non-Kalman and Kalman paths for this output.

I spent about 11 hours on this project, divided as follows:

- 5 hours implementing the Kalman Filter in Ruby
- 3 hours merging code with Michael and passing off with another team in the lab
- 2 hours writing new PDF output with kalman paths
- 1 hour taking screenshots of the kalman paths

2.2 Michael's Summary

This week I started off trying to debug why our agents from the last lab kept getting stuck after capturing the enemy flag. I spent part of Sunday and Monday night tracing the problem through our code and finally found it late Monday night. The problem turned out to be related to caching the search path and not re-calculating it when we had purposely cleared the cached search path.

Later in the week I got to work on fixing up our PD controller to be more accurate for following paths. This task - like the task above - was not directly related to the passof for this lab, but was a necessary step to making sure our team would be ready for the final lab.

Finally towards the latter part of the week I spent my time on the aiming/shooting code. I investigated 6 different methods of calculating the intersection of vectors. I tried several different methods of treating the vectors as parametric equations as well as solving the problem as a system of linear equations and even some trigonometry. In the end a simple iterative approach proved to be the fastest while maintaining the same level of accuracy as the others.

At the end of the week I had spent a total of 28 hours in the following areas:

- 4 hours fixing the bug from last lab.
- 4 hours working on the PD controller to get better path following.
- 6 hours researching vector intersection methods
- 8 hours implementing several aim/shoot ideas
- 4 hours meeting with the team to passoff the lab
- 2 hours compiling report materials

2.3 Jon's Summary

I spent roughly five hours writing and testing all of the clay pigeons.

Chapter 3

Details

3.1 Kalman Filter

Part of the challenge of implementing a Kalman filter for this lab was to appropriately tune the σ_x matrix. In its original form, we allowed an extremely large value for the x and y acceleration parameters—on the order of 100 or so. The result was a very loose guess that produced exaggerated acceleration and velocity vectors. Although the position vector was not too far off, the effect of a wildly inaccurate velocity vector was that our "hunter" tank overcompensated for the velocity on all counts. Next, we tried narrowing the values a bit, hoping that by reducing the parameters we would tighten the difference between observed values and filtered values. In this we were partially successful—by using 0.001 for the position vector, 0.01 for the velocity vector, and 0.1 for the acceleration vector, our Kalman filter seemed to train fairly well on the varying data. With a little room for error, and this tighter range, our hunter was able to shoot down all of the clay pigeons except for our "wild" pigeon, the one that moved in a non-gaussian manner.

One interesting observation was that it was actually ok to have a slightly inaccurate filtered position vector as long as the velocity was also only slightly inaccurate. Smoothing out the position vector by increasing its variance did not improve our odds. Therefore, it was tuning the velocity vector's accuracy that most helped us to achieve reasonable hunting results.

In all of the screenshots below the significance of coloration will follow this format:

- red lines - observed locations connected in chronological order

- green lines - filtered locations connected in chronological order
- yellow lines - normals representing levels of uncertainty around our “best-guess” of the location of the other tank

3.2 Clay Pigeons

As part of this lab we designed some simple “Clay Pigeon” agents to shoot at. They displayed various characteristics and served as a way to judge how effective our Kalman filter was at judging the position, velocity and acceleration of tanks on the field.

3.2.1 Conforming Pigeons

The confoming pigeons were pretty boring as far as implementation goes. Jon created the following conforming pigeons:

- Sitting duck - sat still waiting to be shot
- Constant Velocity - drove in a straight line at a constant velocity
- Constant Acceleration - Accelerated in a constant way
- Gaussian Acceleration - Accelerated in x and y according to a gaussian distribution.

The only thing worth nothing about these implementations is that they were progressively harder for our Kalman agent to shoot. Also we used two types of constant acceleration. One type kept a constant velocity and angular velocity and essentially drove in circles always accelerating towards the center. The other implementation actually changed its forward velocity and angular velocity in a consistent way. Both were somewhat difficult for our agent to hit.

3.2.2 Non-Conforming Pigeon

We designed the non-conforming pigeon to exploit the deficiencies in the Kalman filter. We did this by recognizing that by changing the behavior of the robot over relatively short periods of time (2.5 seconds or less) we could

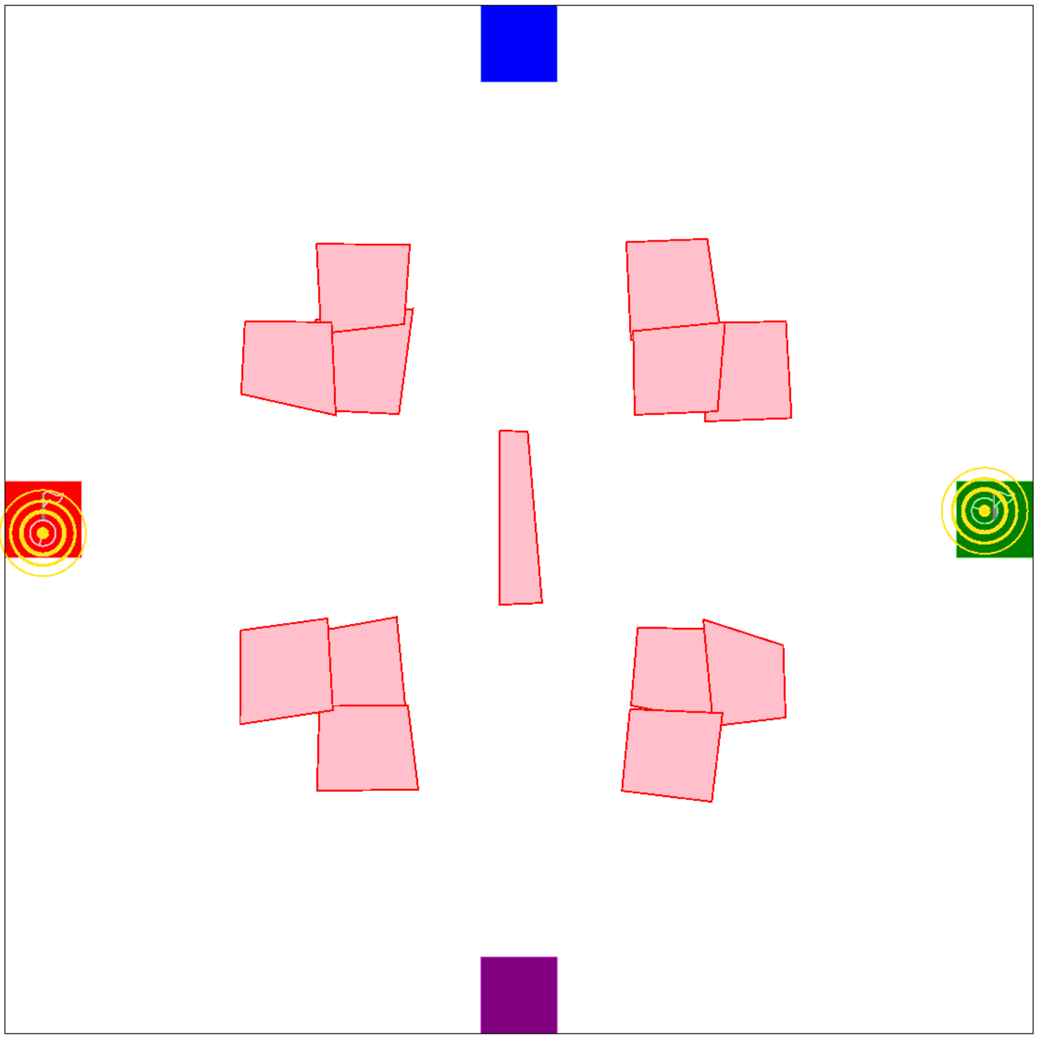

Figure 3.1: A view of the world as seen with "noisy" sensors

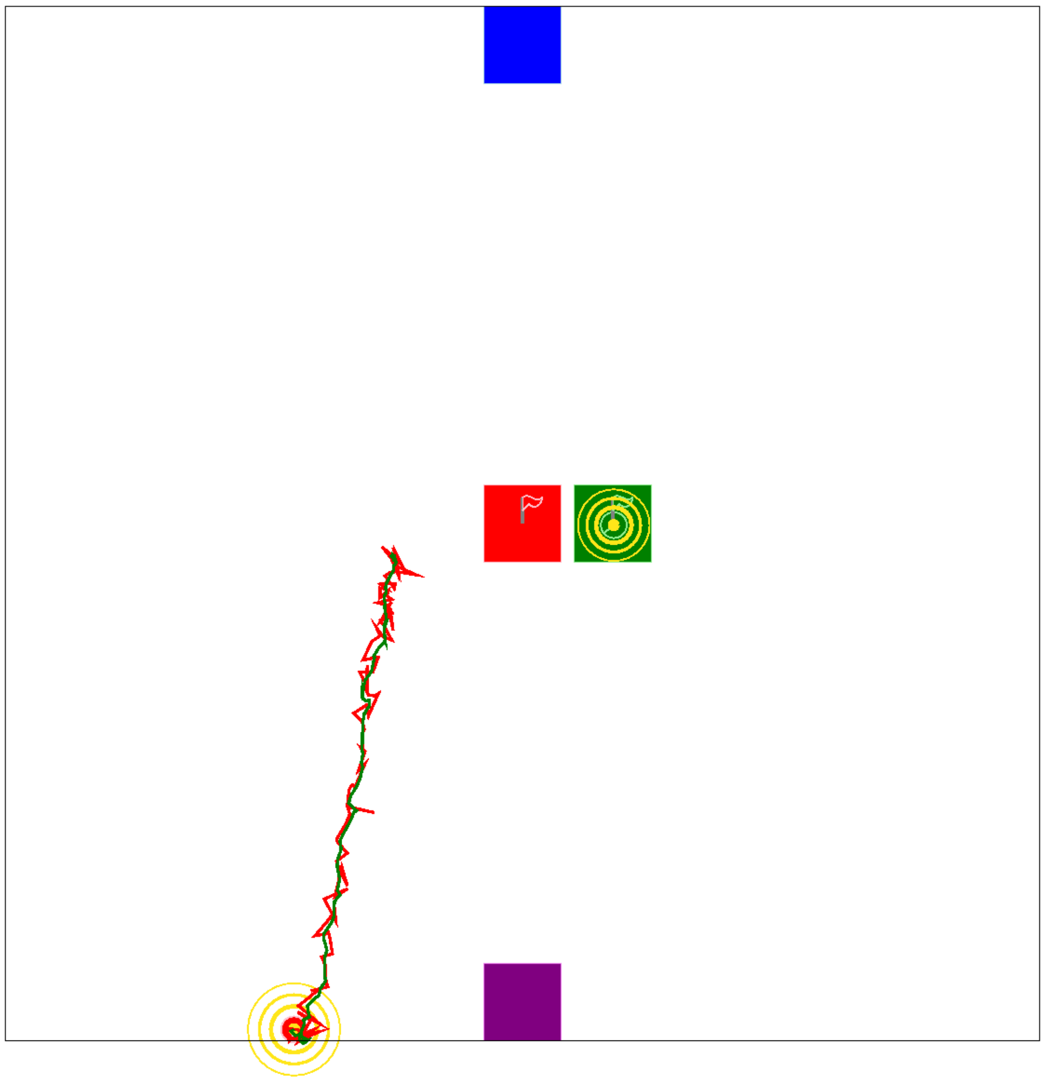

Figure 3.2: Our Constant Velocity Clay Pigeon

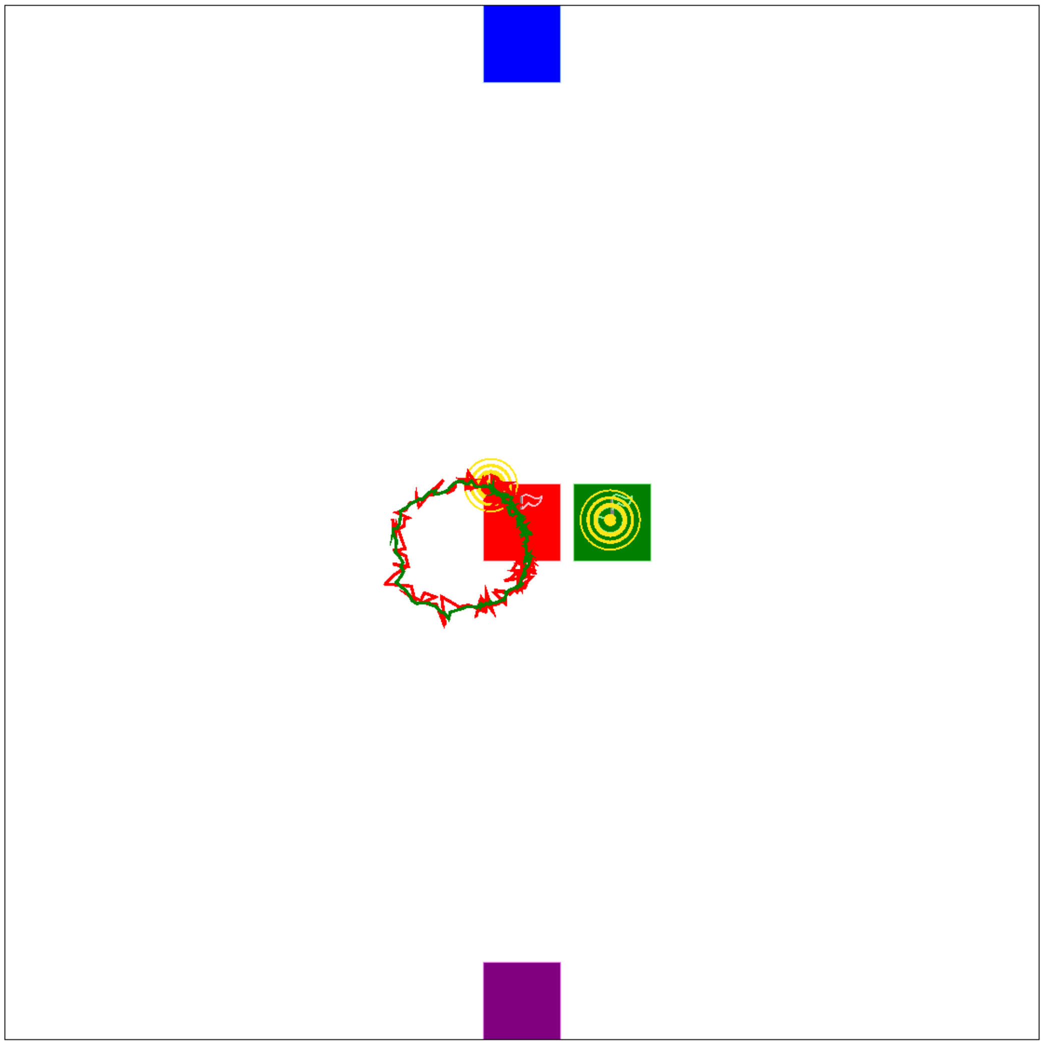

Figure 3.3: Our Constant Acceleration Clay Pigeon

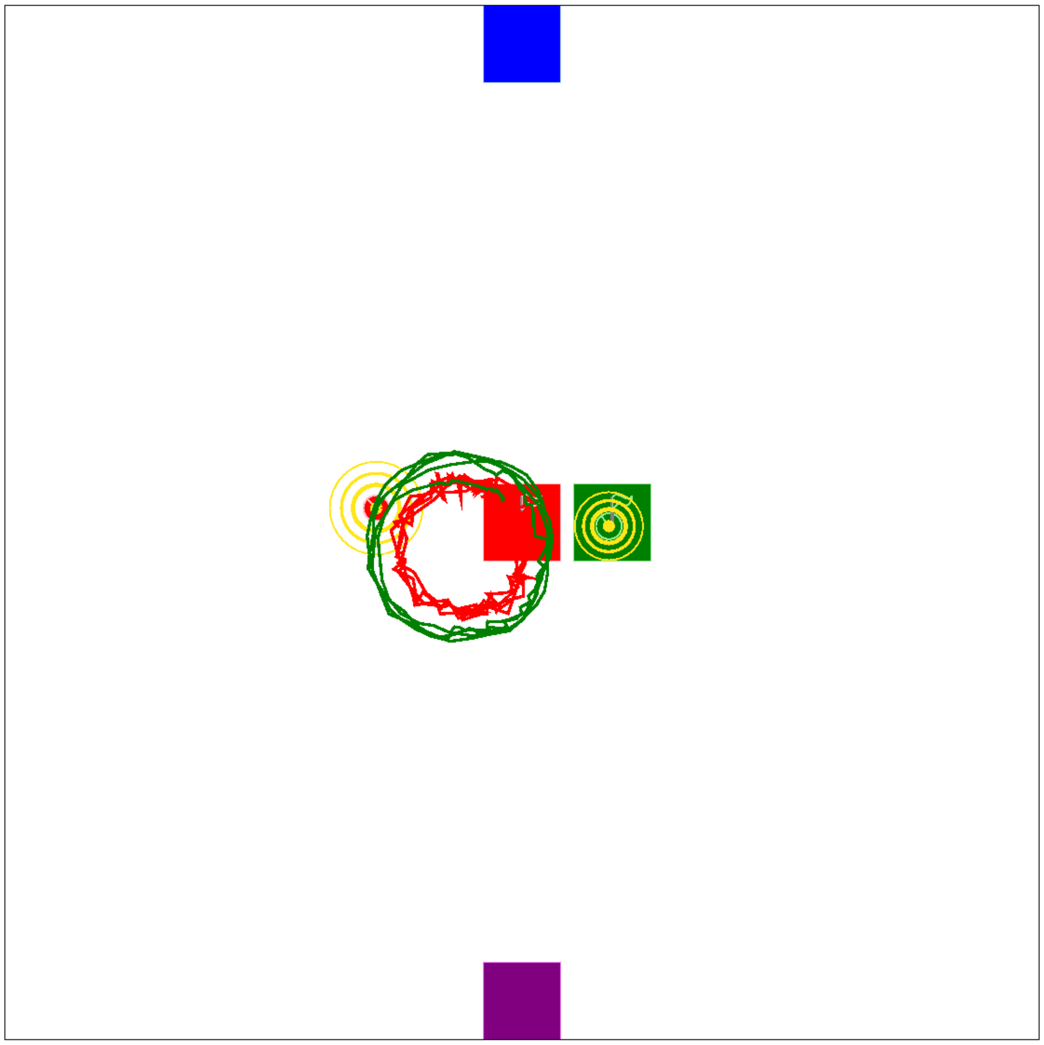

Figure 3.4: Our Constant Acceleration Clay Pigeon

deceive the filter since it relies on good readings over a reasonable length of time.

The further away the pigeon is from the targeting tank, the more effective the technique because the time in which the tank needs to calculate the position of the pigeon to both predict and make the shot are less than the amount of time it takes for the pigeon to change its speed or angular velocity.

We designed the pigeon to behave as follows. We decided to use two separate timers, one for the speed and the other for the angular velocity. Each timer was randomly set between 1 to 25 time steps, where each time step was defined as a tenth of a second.

Initially the time steps were set to between 1 to 100 for a maximum of 10 seconds, but we realized pretty quickly that this was far too long for it to deceive the filter.

At each iteration we decremented both timers and reset them when they hit zero. When the timers are reset we calculate and set the new speed and/or angular velocity.

In practice this technique proved to be effective since over a period of fifteen minutes the pigeon was hit only one time. We had some other ideas that we had thought about exploring if the original implementation did not perform as well as we would have liked, however the pigeon did not disappoint, thus no other algorithms were devised.

In planning for the tournament, we recognize that one easy way to negate the performance of the non-conforming pigeon is to move within a much closer shooting distance. By doing this the amount of time between change in behavior and the filters ability to accurately predict the range will be long enough for the tank to make accurate and deadly shots.

3.3 Tests with Another Group

3.3.1 Our Tests

Our tests were executd in a simple bzflag world with the red and green base near the center and the map was devoid of obstacles. This helped to keep us focused on the goals of this lab. In the last round of tests our agent was very accurate in killing the conforming clag pigeons. It took between three and five shots to kill each conforming clay pigeon. This was observed when killing both our clay pigeons and the other team's clay pigeons.

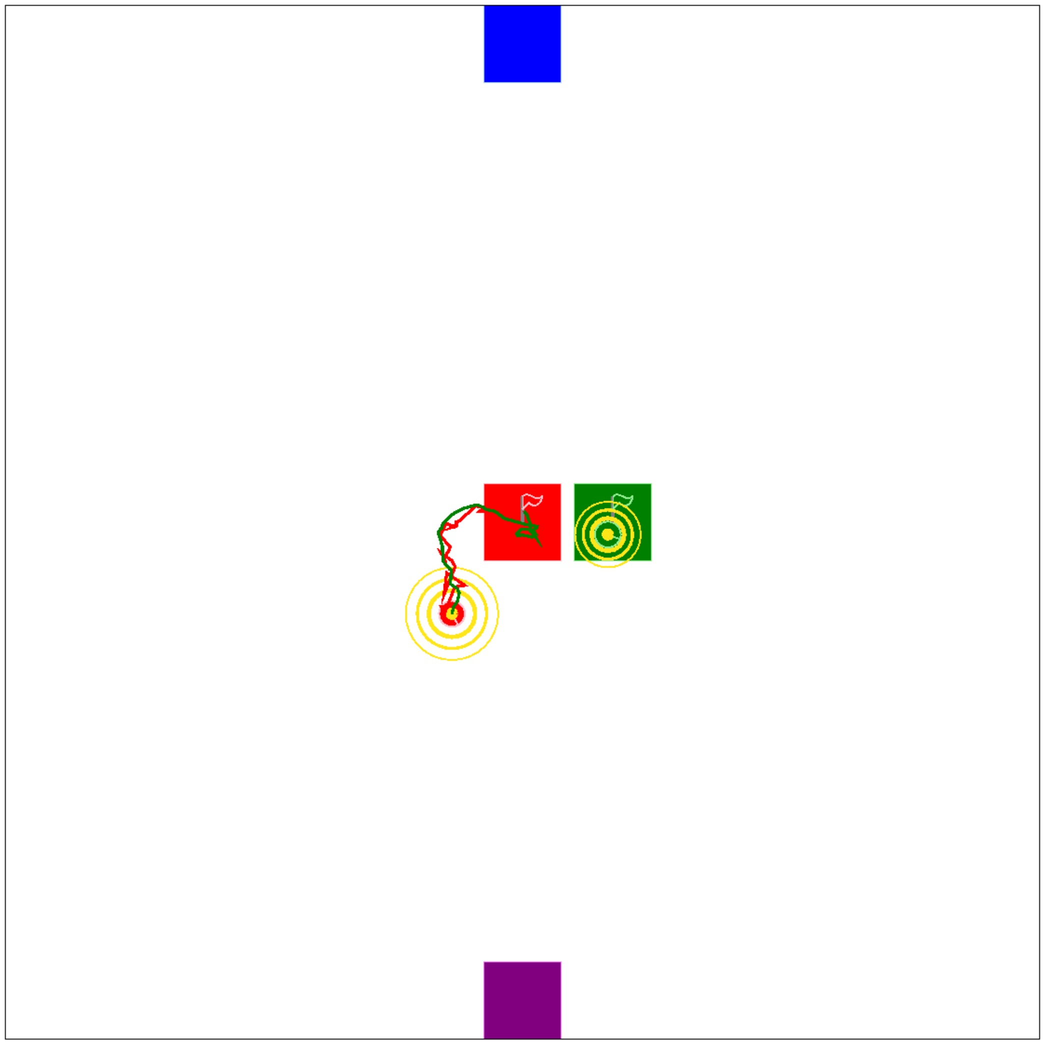

Figure 3.5: Our Wild Clay Pigeon

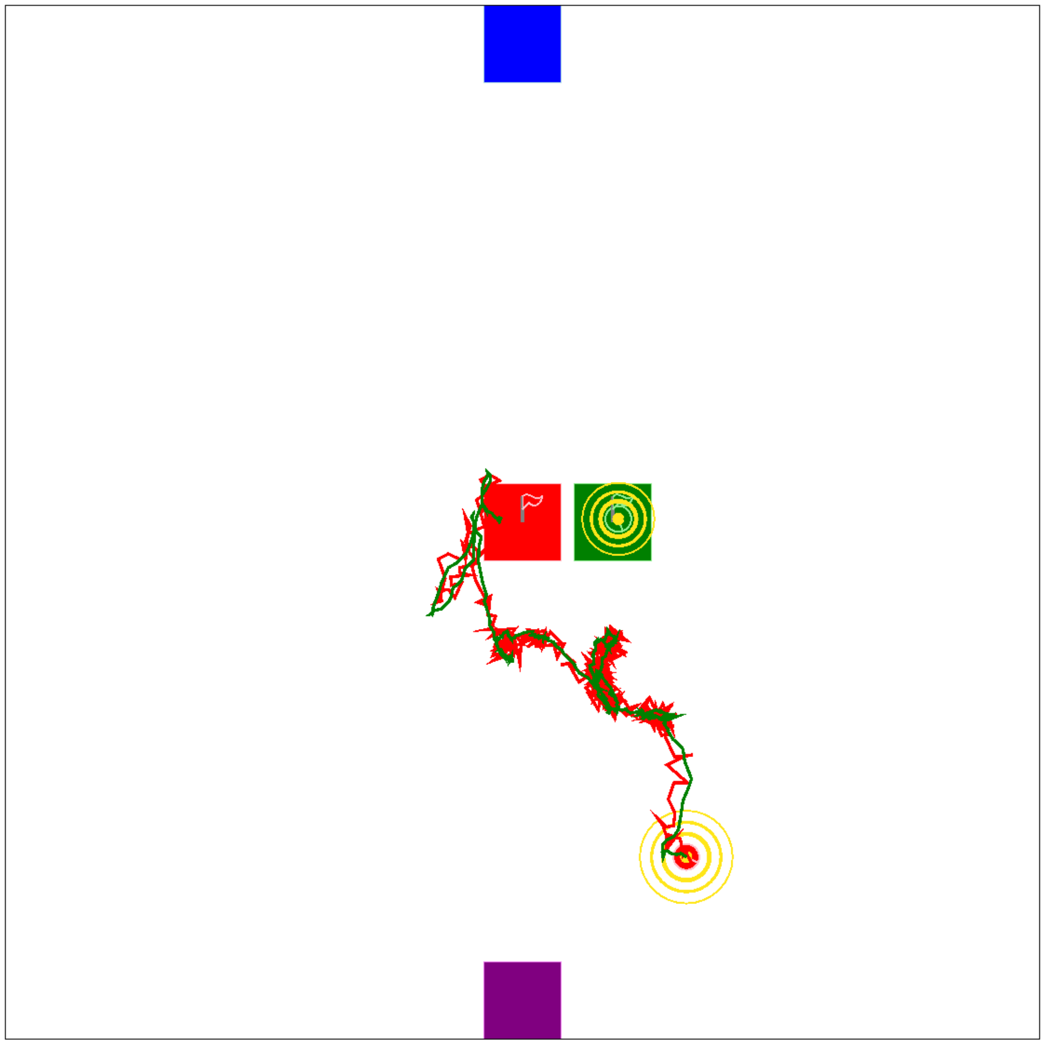

Figure 3.6: Our Constant Velocity Clay Pigeon

The wild pigeons proved to be very wild. Because every aspect of the wild pigeons behavior changed we were unable to find a good way to target it. We tried various techniques. Mostly we tried changing our Kalman filter to believe more strongly in the current observations than in its previously filtered values. This change helped and ultimately the wild pigeon wandered close enough and while making a turn the kalman agent destroyed it. A simliar experience with the other team's wild pigeon took place. When the wild pigeons make truly random changes to velocity and angle we found no good way to predict their location. Our inability to predict translated into a poor chance of shooting the wild pigeons.

3.3.2 The Other Team

We ran our battery of tests and observed the tests of two other groups: Group 1

- Dave Brinton
- Tracen Peterson

Group 2

- Matt Chou
- David Wilcox
- Kendell Clement

Watching the other teams agents, and passing off against their pigeons was interesting. All three teams seemed to be almost equally efficient in their ability to kill the clay pigeons, but the defects in each agent were different. Dave Brinton's group seemed to have some problems with turning to the right angle and so they missed about equally to the right or left of the pigeon. Our agent seemed to have difficulty with agents moving quickly. It would try to shoot too far in front of the pigeon and would always end up missing in front of the pigeon. This is probably because of some "fudge factors" that we built into the agent to give it a sense that turning was not instantaneous. This deficiency will have be addressed for the final lab. Matt Chou's group seemed to have the opposite problem as we did and they consistently shot too far behind the clay pigeons. They seemed to think that the PD controller was keeping them behind the desired angle as it slowed before reaching its optimal angle.

The pigeons from each group were very similiar. In fact the behaviour of all the conforming pigeons was nearly identical and only seemed different between the implementations of the gaussian acceleration pigeon. But even with the guassian acceleration pigeon the results were similar as to how quickly the pigeons were shot.

The wild pigeons were suprisingly similiar although each seemed to favor randomness in a different way. None of the teams seemed to have a good way to change their Kalman filters to adapt to the wild pigeons.

The code from the other teams looked good. Dave Brinton's group had code that looked a lot like ours because the syntax was the same. We noticed their code was full of printf statements used to debug strange behavior in their Kalman filter. Matt Chou's group had very organized code which was a bit more verbose because it was written in Java. All three teams used matrix libraries to solve the three central equations of the Kalman filter.

Appendix A

Ruby Code

See accompanying archive file (*lab4.tar.gz*) for a complete listing of this lab's Ruby and C code.