University of California, Merced

Controlling a Simple Inverted Pendulum with Markov Decision Processes

Manuel Meraz

EECS 270 Robot Aglorithms

Stefano Carpin

October 9, 2017

Objective

This project can be split up into three distinct components: The MDP controller, understanding the dynamics of the pendulum, and properly integrating the noise model into the whole system. Ideally, once the MDP controller has been finished, it should be able to generate an optimal policy to balance the pendulum for any given set point.

Design

My first step in designing the code for this project was to fully understand the dynamics of the pendulum. A positive *u* input into the dynamics model means that the torque is going clockwise, and a negative *u* means it's going counterclockwise. The angular velocity follows the same pattern. I used the ranges [-π, π] to model all of the possible angles, where *0* is the top portion where the pendulum is perfectly balanced.

The next step was figuring out how to integrate the noise into the MDP controller and calculate *p(s, a, s')*, or the probability of transitioning from a state *s* having committed to an action *a* and landing at state *s'* with a given noise model represented by a mean μ and a covariance matrix Σ. We do this for the current state *s* we are in, for all $a \in A$ and all $s' \in S$. If we are in state *s* and take action *a,* which in this case is represented by the *u* input into the dynamics model for the pendulum, then it will return the angular acceleration for the pendulum. Now that we have the position, velocity and acceleration of the pendulum, then we can use the following equations to get *s'*.

$$\theta = \theta_0 + \dot{\theta}\Delta t$$
$$\dot{\theta} = \dot{\theta}_0 + \ddot{\theta}\,\Delta t$$

With those values, before I discretize them, I input them into a function that gives me the probability spread based off of the noise parameters μ and Σ, where μ is around 0 for both dimensions and a variance of 0.1 for both dimensions. Due to this, we can simply say that μ is center around θ and $\dot{\theta}$. Matlab offers us a error function *erf()* that allowed me to compute the integral for a gaussian distribution, perfect for this noise model. With the bounds for failure [-π/4, π/4] and bounds for angular velocity *[-5, 5]* (determined after experimentation with the dynamics model) and a discrete number of states *n,* I was able to compute the transition probabilities. To do this I needed to determine a *dx* for each state, which was:

$$dx = \frac{\textit{Upper Bound} - \textit{Lower Bound}}{2n}$$

The lowest bounds probabilities was the integral from negative infinity to the value itself plus *dx*. The next value was the integral from negative infinity to the value itself minus the previous probability, and so on. With these values, for both the velocity and angle, we could use the bellman equations.

The Bellman equations were very straight forward, just computationally intensive. The most difficult part was debugging the recursion and determining whether the given policies were computed correctly, which of course they weren't at first. My reward model was fairly simple, the closer it was to its set point, the greater the reward. I added in a reward multiplier if the velocity for the state was moving towards the center and left it alone if it was close, but moving away.

Experimental Procedures

The code that I wrote wasn't the most efficient because my first intention was to simply get everything to work well together. After playing around with a lot of values, mainly different number of discrete states my controller was able to work with the following parameters

$$-\frac{\pi}{4} < \theta < -\frac{\pi}{4}$$

$$-5 < \dot{\theta} < 5$$

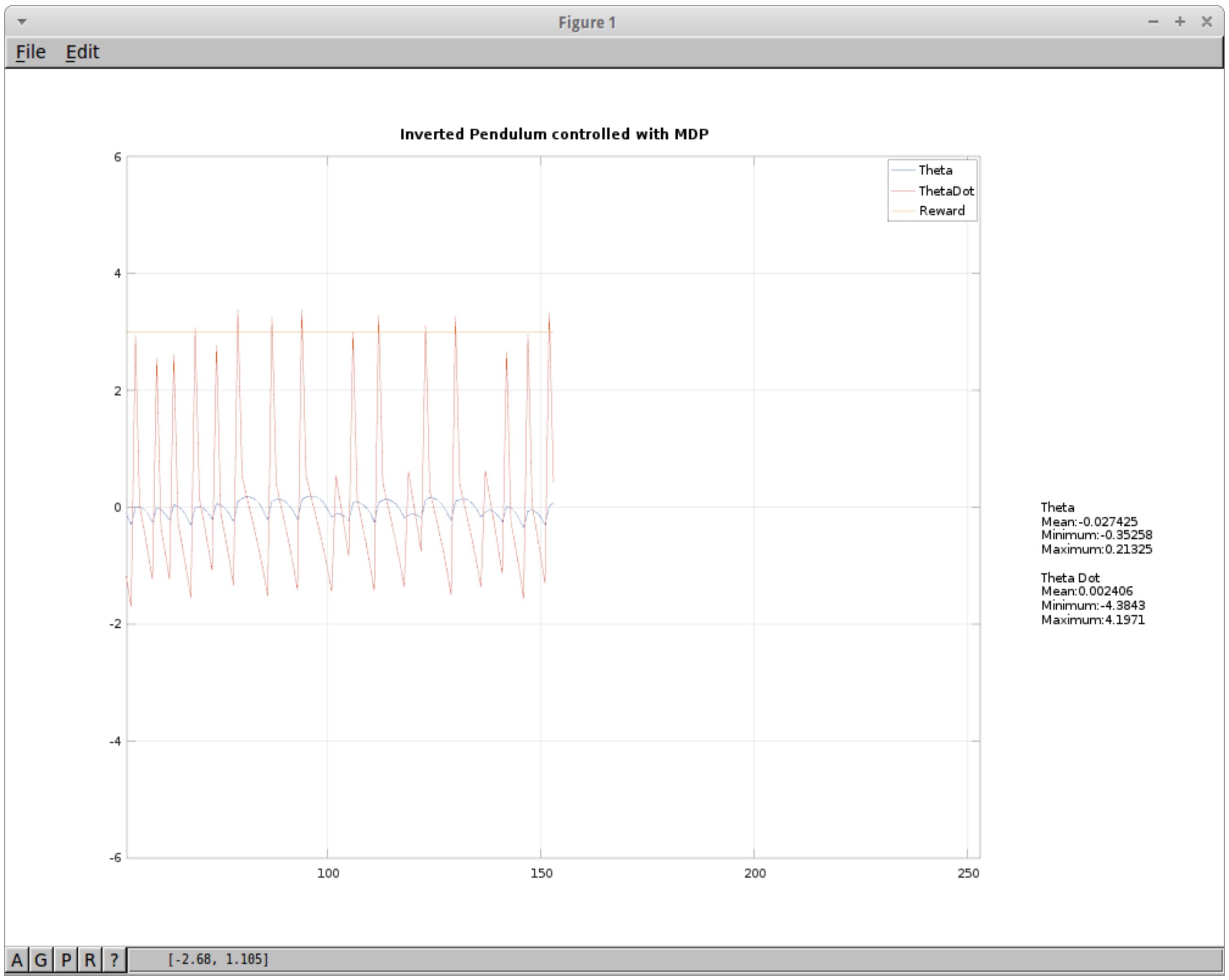
$$n >= 5$$

$$A = \{-200 : 200\}$$

I was able to actually generate a policy with $n = 3$ and have it work, but it quickly fell out of bounds and failed, eventually though regaining control and balancing. The deepest I went into the recursion tree was a depth of 2 as the time to compute the policy took a very long time since my code has not been optimized. The action set provided to us was not enough to control the pendulum, and I found that using a set of actions with a high resolution works best with -100 to 2100 working very effective. There was a limit though, increasing the force input any further would cause the velocity to change very drastically as the MDP controller would choose to use those quite effectively since my reward model did not account for it. Computing a policy with $n >= 10$ resulted in the computation taking hours, but I also have no optimized my code.

Once a policy is generated I store it into a .mat file and I reload the older policies to test. I did not account for other parameters besides different values for *n*.

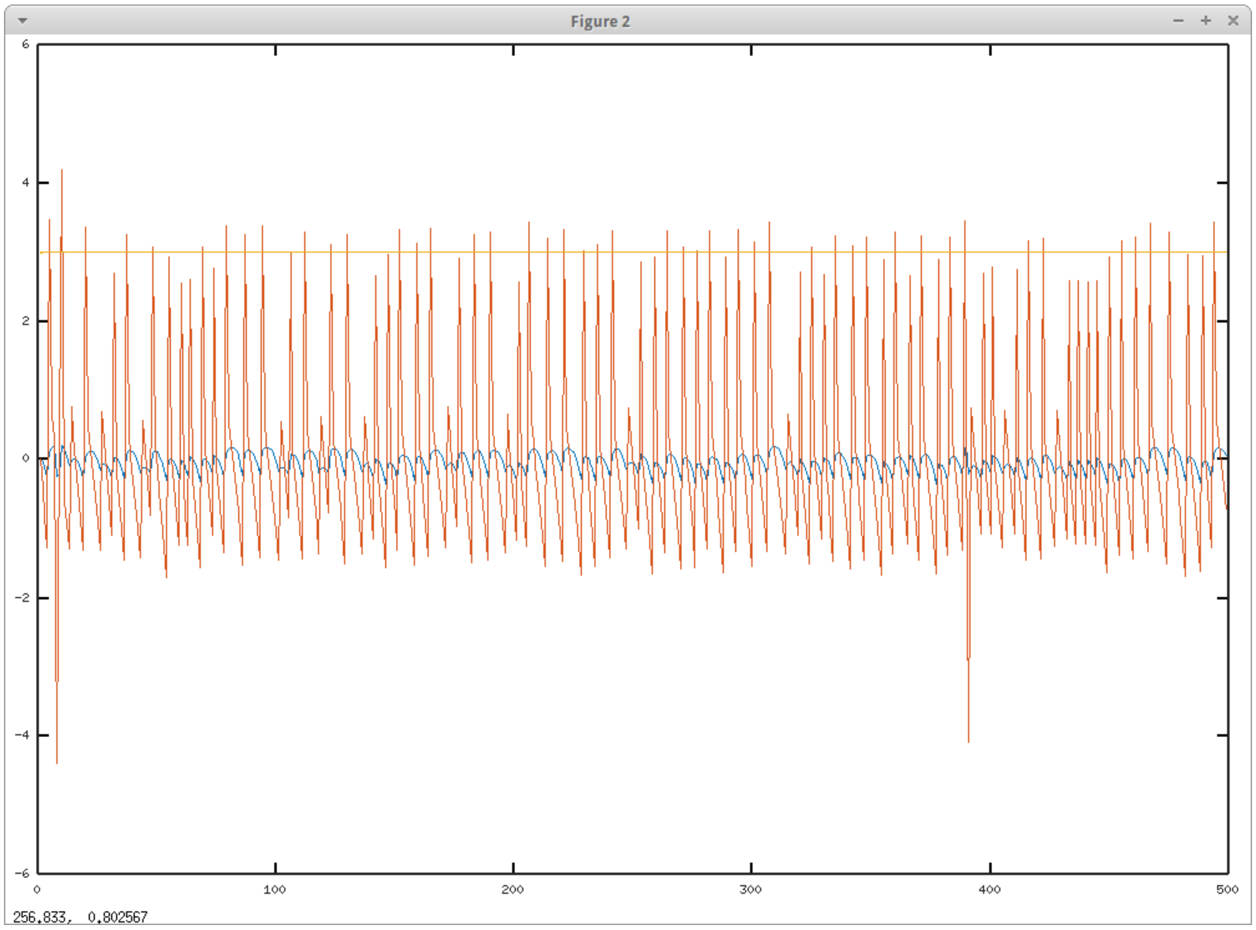
Results



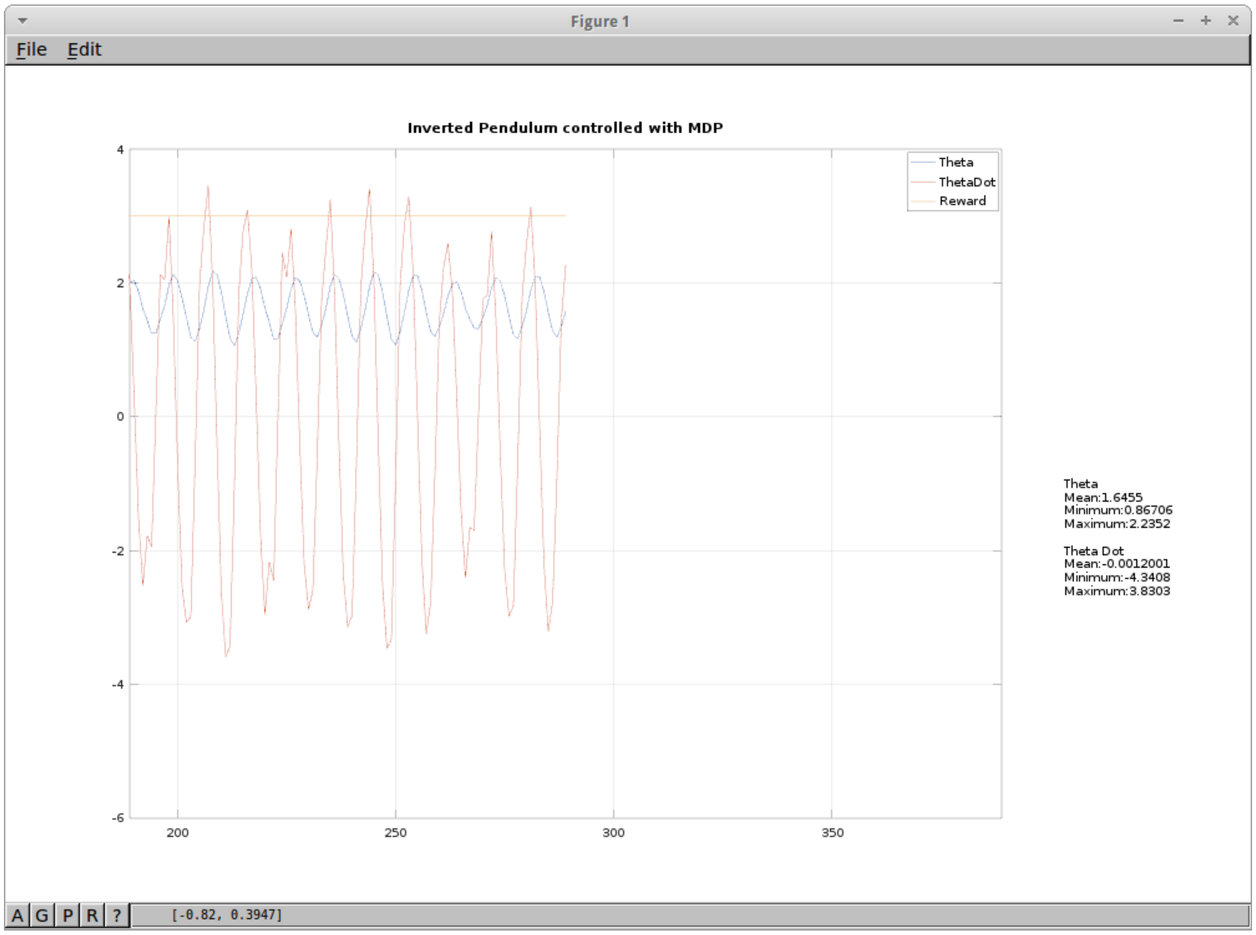
```
Theta
Mean:-0.027425
Minimum:-0.35258
Maximum:0.21325

Theta Dot
Mean:0.002406
Minimum:-4.3843
Maximum:4.1971
```

Here is the live view as the pendulum is balancing around 0 with 25 possible states, or 5 discrete states for theta and 5 for thetaDot. When the program is done running and hits 500 iterations it produces the total graph.

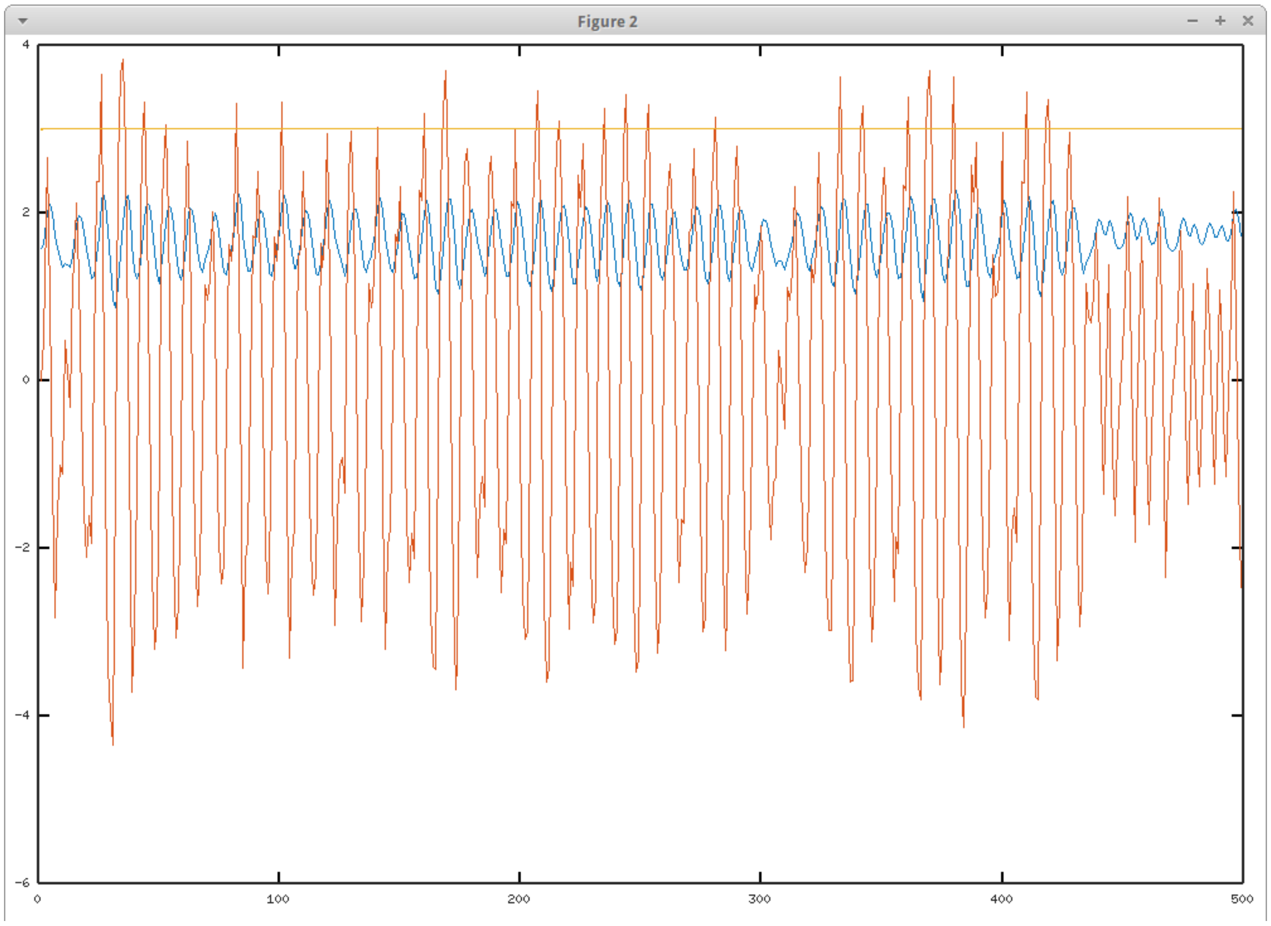


The next image is another view as the program is running showing the pendulum balancing around pi/2, or balancing parallel to the ground on the right side with the same number of states as the previous one.



Theta
Mean:1.6455
Minimum:0.86706
Maximum:2.2352

Theta Dot
Mean:-0.0012001
Minimum:-4.3408
Maximum:3.8303



Conclusions

After working on this project I have become really fascinated with MDP and have really learned the power of this type of controller. After every run you will see the output of the policy, with the given stats and a plot is generated by matlab. I actually coded everything in octave, and haven't fully tested it in matlab, but I believe they are backward compatible. I would have ideally enjoyed coding this more in C++, and plan to do so to make a ROS package and see if I can make a real inverted pendulum controlled by arduino.