Model Predictive Control of an Inverted Pendulum

Manish Patel

October 12, 2021

1 Introduction

The goal of this project was to implement Model Predictive Control (MPC) and demonstrate its effectiveness on an example system - in this case, an inverted pendulum. For the classic stabilization problem, where the only goal is to have the pendulum remain upright, simpler schemes such as Proportional-Integral-Derivative (PID) Control or the Linear Quadratic Regulator (LQR) are sufficient. In order to show what MPC is truly capable of, the design goal for the controller was to have it not only stabilize the pendulum, but also simultaneously make the cart position follow a time-varying reference trajectory.

2 System Model

The system model for an inverted pendulum is derived in [1]. As derived there, the two equations of motion for the cart are:

$$(M+m)\ddot{x} + b\dot{x} + mL\ddot{\theta}cos(\theta) - mL\dot{\theta}^2 sin(\theta) = u$$
$$(I+mL^2)\ddot{\theta} + mgLsin(\theta) = -mL\ddot{x}cos(\theta)$$

Now, if we perform the same substitution, $\theta = \phi + \pi$, that they do in [1], we obtain the following equations:

$$(M+m)\ddot{x} + b\dot{x} - mL\ddot{\phi}cos(\phi) + mL\dot{\phi}^2 sin(\phi) = u$$
$$(I+mL^2)\ddot{\phi} - mgLsin(\phi) = mL\ddot{x}cos(\phi)$$

At this point, we could make the small angle approximation and simplify the equations in a similar manner as the original derivation. Because the approximation is fairly accurate within the regime in which the controller is expected to operate, this approximation is likely sufficient. However, to make things more interesting the system was instead linearized using the standard Jacobian method which is more accurate; [2] is an excellent source describing this procedure. Because the algebra is incredibly tedious and nearly impossible to do by hand for a system this complicated, the SymPy symbolic algebra library was used to perform the calculations automatically. Please refer to the source code defining the model [3], specifically the **derive_model_symbolically** function, to see how this was done. After linearization, the model has the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + v(t)$$

$$y = Cx(t) + w(t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} x + w(t)$$

where the state vector is given by:

$$x = \begin{bmatrix} q \\ \dot{q} \\ \phi \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} cart\_position \\ cart\_velocity \\ pendulum\_angle \\ pendulum\_angular\_velocity \end{bmatrix}$$

and the output matrix, C, is chosen so that the outputs are the cart position and pendulum angle, which can be measured using sensors in a physical system. $v(t)$ and $w(t)$ are assumed to be zero-mean Gaussian noise terms, with covariance matrices V and W, respectively.

For many control algorithms, we often work with a discretized version of the system. One common method for discretizing a system is known as zero-order-hold, which was used here. Zero-order-hold assumes that the input is a step function, and for a small enough sampling rate T_s this assumption can be good enough. Discretizing a system using zero-order-hold involves some fairly complex computation [5], which we won't reproduce here. After discretization, the system model is given by:

$$x[k+1] = A_d x[k] + B_d u[k] + v_d(k)$$
$$y[k] = C_d x[k] + w_d[k] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} x_d[k] + w_d[k]$$

In practice, SciPy was used to determine values of A_d and B_d given the original A and B matrices. To obtain the discretized versions of the noise covariance matrices, V_d and W_d, the following approximation specified in [5] was used instead of the full calculation:

$$V_d = VT_s$$
$$W_d = W/T_s$$

3 Extended Kalman Filter

In an ideal world, the nonlinear model in the previous section would perfectly capture the real world system dynamics, and sensors would be able to perfectly measure the full system state. However, there are always inaccuracies in the model and noise that will affect the system, which makes it necessary to implement a state estimator. For this simulation, the Continuous-Discrete Extended Kalman Filter was implemented as the state estimator. Section 3.7 of [4] provides an excellent description of this variant of the standard EKF and is what this implementation was based upon. The derivation won't be reproduced here, but we will mention a few important implementation details. First, in order to solve the differential equations in the prediction step, Euler's method was used. A possible future improvement would be to use the fourth order Runge-Kutta method instead, which is more accurate. Second, it is important to remember that in the Continuous-Discrete EKF formulation, the prediction step happens in continuous time, so the continuous time version of the process noise matrix, V, must be used. On the other hand, the update step happens in discrete time, which means that we must use the discretized version of the output noise matrix, W_d.

4 Model Predictive Control

4.1 Quadratic Programming Formulation

Assume that we have a discrete time system with the form described in Section 1 and the current state is $x[k]$. Given a prediction horizon of L time steps, and a state reference trajectory that we want the system to follow, $x_{ref}[k], ..., x_{ref}[k+L]$, the goal is to determine the optimal input sequence $u[k], ..., u[k+L]$ that will make the system follow the reference trajectory. In order for the word optimal to mean anything, we need a cost function to optimize, which we will choose to be the following:

$$J = (x[k+L] - x_{ref}[k+L])^T Q_f (x[k+L] - x_{ref}[k+L]) + \sum_{i=k}^{k+L-1} (x[i] - x_{ref}[i])^T Q (x[i] - x_{ref}[i]) + u[i]^T R u[i]$$

where Q_f is the terminal state cost matrix, Q is the non-terminal state cost matrix, and R is the input weight matrix. The cost function is quadratic and balances following the reference trajectory with the input effort. (Why we have a separate terminal cost matrix will become clear later). Q_f, Q, and R must be symmetric matrices, Q_f and Q must be positive semidefinite, and R must be positive definite.

(Note that we have not specified a u_{ref} signal that will drive the system to follow x_{ref}. This can lead to steady state error in the controller, but this can be addressed by adding integral action to compensate for the error [6]. This will be discussed in a later section).

With some tedious algebra, it is possible to show that the cost function reduces to a quadratic form, $J = r^T H r + 2 f^T r$, at which point any off-the-shelf quadratic programming solver can be used to minimize J. The derivation here mostly follows the one found in [7], but without skipping some important steps that aren't immediately obvious.

First, observe that the state of the system evolves as follows:

$$\begin{aligned}
x[k+1] &= Ax[k] + Bu[k] \\
x[k+2] &= A^2 x[k] + ABu[k] + Bu[k+1] \\
&\vdots \\
x[k+L] &= A^L x[k] + A^{L-1} Bu[k] + A^{L-2} Bu[k+2] + ... + ABu[k+L-2] + Bu[k+L-1]
\end{aligned}$$

Let's define new "state", "input", and "reference" vectors, $\hat{X}$, $\hat{U}$, and $\hat{X}_{ref}$ that look like:

$$\hat{X} = \begin{bmatrix} x[k+1] \\ x[k+2] \\ ... \\ x[k+L] \end{bmatrix}, \hat{U} = \begin{bmatrix} u[k] \\ u[k+1] \\ ... \\ u[k+L-1] \end{bmatrix}, \hat{X}_{ref} = \begin{bmatrix} x_{ref}[k+1] \\ x_{ref}[k+2] \\ ... \\ x_{ref}[k+L] \end{bmatrix}$$

Following the pattern that we see above in the state evolution equations, we can write the equation for $\hat{X}$ in matrix form:

$$\hat{X} = \begin{bmatrix} A \\ A^2 \\ A^3 \\ ... \\ A^L \end{bmatrix} x[k] + \begin{bmatrix} B & 0 & 0 & ... & 0 \\ AB & B & 0 & ... & 0 \\ A^2B & AB & B & ... & 0 \\ ... & ... & ... & ... & 0 \\ A^{L-1}B & A^{L-2}B & A^{L-3}B & ... & B \end{bmatrix} \hat{U}$$

$$\hat{X} = \hat{A}x[k] + \hat{B}\hat{U}$$

We'll similarly define new weight matrices $\hat{Q}$ and $\hat{R}$:

$$\hat{Q} = \begin{bmatrix} Q & 0 & 0 & ... & 0 \\ 0 & Q & 0 & ... & 0 \\ 0 & 0 & Q & ... & 0 \\ 0 & 0 & 0 & ... & Q_f \end{bmatrix}, \hat{R} = \begin{bmatrix} R & 0 & 0 & ... & 0 \\ 0 & R & 0 & ... & 0 \\ 0 & 0 & R & ... & 0 \\ 0 & 0 & 0 & ... & R \end{bmatrix}$$

Using these new matrices, you can convince yourself that we can re-write and simplify the original cost function as follows:

$$J = (\hat{A}x[k] + \hat{B}\hat{U} - \hat{X}_{ref})^T \hat{Q}(\hat{A}x[k] + \hat{B}\hat{U} - \hat{X}_{ref}) + \hat{U}\hat{R}\hat{U}$$

$$J = (x[k]^T \hat{A}^T \hat{Q} + \hat{U}^T \hat{B}^T \hat{Q} - \hat{X}_{ref}^T \hat{Q})(\hat{A}x[k] + \hat{B}\hat{U} - \hat{X}_{ref}) + \hat{U}\hat{R}\hat{U}$$

$$\begin{aligned} J =& x[k]^T \hat{A}^T \hat{Q} \hat{A} x[k] + x[k]^T \hat{A}^T \hat{Q} \hat{B} \hat{U} - x[k]^T \hat{A}^T \hat{Q} \hat{X}_{ref} + \\ & \hat{U}^T \hat{B}^T \hat{Q} \hat{A} x[k] + \hat{U}^T \hat{B}^T \hat{Q} \hat{B} \hat{U} - \hat{U}^T \hat{B}^T \hat{Q} \hat{X}_{ref} - \\ & \hat{X}_{ref}^T \hat{Q} \hat{A} x[k] - \hat{X}_{ref}^T \hat{Q} \hat{B} \hat{U} + \hat{X}_{ref}^T \hat{Q} \hat{X}_{ref} + \hat{U}\hat{R}\hat{U} \end{aligned}$$

Recall that the goal is to find the value of $\hat{U}$ the minimizes J, and not the minimum value of J itself. As a result, we can omit all constant terms; that is, all that do not depend on $\hat{U}$.

$$J = x[k]^T \hat{A}^T \hat{Q} \hat{B} \hat{U} + \hat{U}^T \hat{B}^T \hat{Q} \hat{A} x[k] + \hat{U}^T \hat{B}^T \hat{Q} \hat{B} \hat{U} - \hat{U}^T \hat{B}^T \hat{Q} \hat{X}_{ref} - \hat{X}_{ref}^T \hat{Q} \hat{B} \hat{U} + \hat{U}\hat{R}\hat{U}$$

Our next trick is to realize that since J is a scalar function, each term in the above equation is a scalar, so we can freely transpose any of the terms without changing the expression. Let's transpose all of the non-quadratic terms starting with $\hat{U}^T$.

$$J = x[k]^T \hat{A}^T \hat{Q} \hat{B} \hat{U} + x[k]^T \hat{A}^T \hat{Q}^T B U + \hat{U}^T \hat{B}^T \hat{Q} \hat{B} \hat{U} - \hat{X}_{ref}^T \hat{Q}^T \hat{B} \hat{U} - \hat{X}_{ref}^T \hat{Q} \hat{B} \hat{U} + \hat{U}\hat{R}\hat{U}$$

And for our final trick, recall that we require $\hat{Q}$ to be symmetric, so we can replace $\hat{Q}^T$ with $\hat{Q}$.

$$J = x[k]^T \hat{A}^T \hat{Q} \hat{B} \hat{U} + x[k]^T \hat{A}^T \hat{Q} B U + \hat{U}^T \hat{B}^T \hat{Q} \hat{B} \hat{U} - \hat{X}_{ref}^T \hat{Q} \hat{B} \hat{U} - \hat{X}_{ref}^T \hat{Q} \hat{B} \hat{U} + \hat{U}\hat{R}\hat{U}$$

$$J = \hat{U}^T (\hat{B}^T \hat{Q} \hat{B} + \hat{R}) \hat{U} + 2(x[k]^T \hat{A}^T \hat{Q} \hat{B} - \hat{X}_{ref}^T \hat{Q} \hat{B}) \hat{U}$$

The form of the final, simplified cost equation clearly has the aforementioned quadratic form. QP solvers also allow adding equality and inequality constraints, which are useful for constraining the state and input variables to their physical limits. The current implementation doesn't allow specifying such constraints, but that is something that will be added soon.

4.2 Terminal Cost Matrices

In the unconstrained case, it is possible to guarantee the stability of MPC by using a terminal state cost matrix, Q_f, that is the solution to the Discrete Algebraic Riccati Equation, as demonstrated in [8]. Based on actually trying to run simulations without choosing the correct terminal cost matrix, we confirmed that the controller is indeed unstable without performing this important step.

4.3 Integral Action

As mentioned previously, by not explicitly specifying a reference input u_{ref} to follow the desired trajectory, we force the cost function to trade off between following the reference trajectory and not using any control effort. A way to address this is to transform our original system such that it penalizes the difference in input between time steps rather than the input itself. As shown in [6], we can augment the system as follows to achieve this new formulation:

$$\begin{bmatrix} x[k+1] \\ u[k] \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & I \end{bmatrix} \begin{bmatrix} x[k] \\ u[k-1] \end{bmatrix} + \begin{bmatrix} 0 \\ I \end{bmatrix} \Delta u[k]$$

$$y = \begin{bmatrix} C & 0 \end{bmatrix} \begin{bmatrix} x[k] \\ u[k-1] \end{bmatrix}$$

where $\Delta u[k] = u[k] - u[k-1]$. By rewriting the system in this form, the new input is now effectively the difference in input effort between successive time steps, and the extra state variables are essentially integrating this difference. This allows us to penalize the difference in input effort instead of the input itself. Intuitively, this approach makes sense - assume that over a small horizon the reference trajectory is relatively constant, and the input needed to follow the reference is also relatively constant. Then, the MPC formulation with integral action will only penalize the input from the first time step, but not the later ones, which is what is desired.

5 Simulation Results

The EKF and MPC algorithms were tested using the inverted pendulum model. The model parameters were the same as the ones found in [1]. The full nonlinear model was simulated with noise to determine the "true" state of the system. The following parameters were also used for the simulation:

$$T_s = 0.01$$

$$L = 20$$

$$V = \begin{bmatrix} T_s^4/3 & T_s^3/2 & 0 & 0 \\ T_s^3/2 & T_s^2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$W = \begin{bmatrix} 0.0001 & 0 \\ 0 & 0.0001 \end{bmatrix}$$

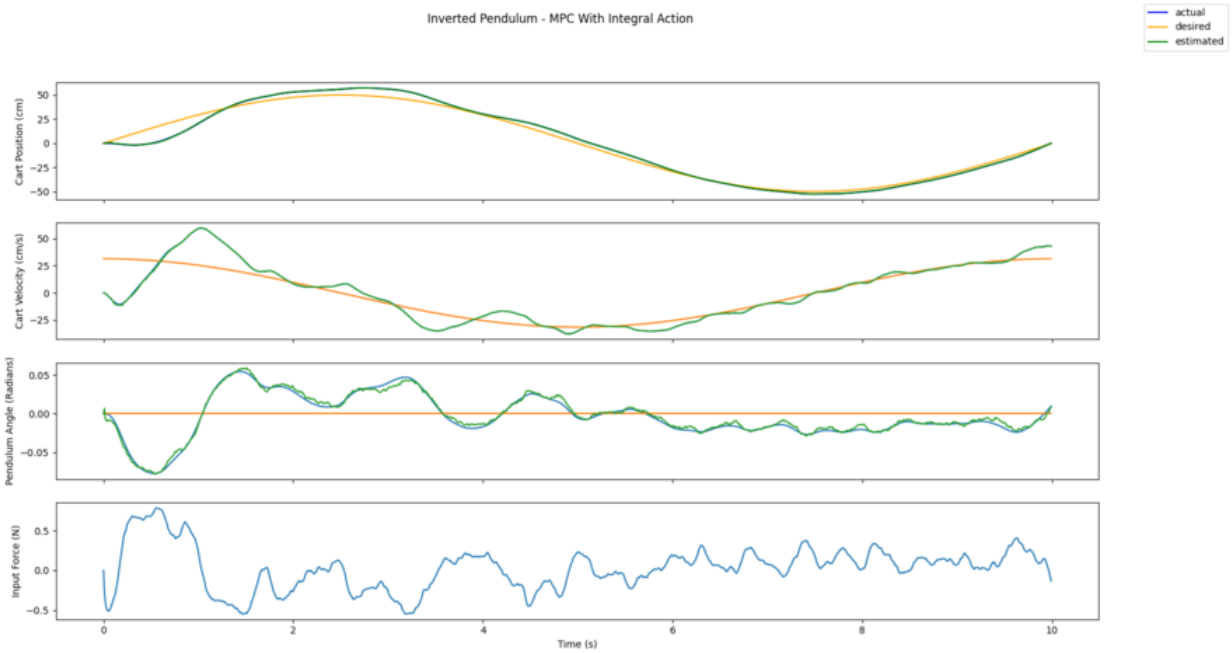
$$Q = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

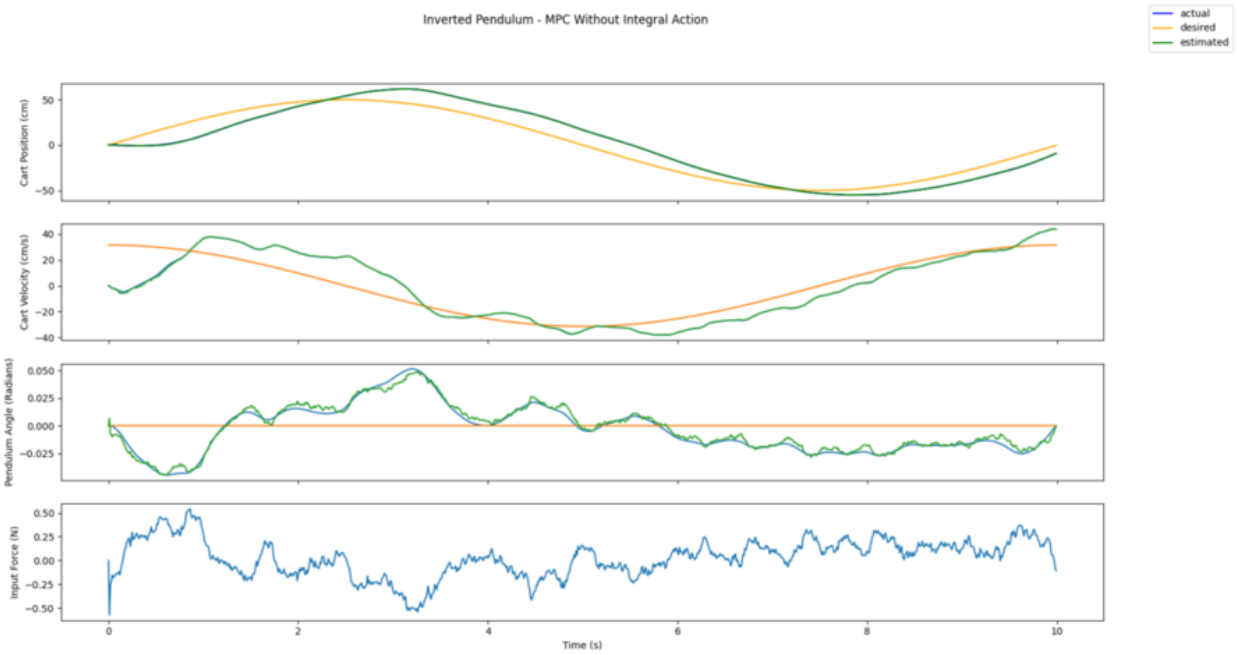
$$R = \begin{bmatrix} 10 \end{bmatrix}$$

$$x_{ref}[k] = \begin{bmatrix} 0.5sin(\pi T_s k/5) \\ 0.1\pi cos(\pi T_s k/5) \\ 0 \\ 0 \end{bmatrix}$$

$$x_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T$$

The V matrix was chosen based on the random acceleration model for noise, see [9]. The Q matrix shown above is for the augmented integral action version of the system model - note that we do not weight the input itself, which is part of the state. For the original model, the Q matrix is just the upper left $4x4$ section. The reference trajectory is sinusoidally varying in the cart position and velocity, while also requiring the pendulum remain upright. The following plots show the results of the controller both with and without integral action included:





As can be seen in the above plots, the version with Integral Action performed much better. To be fair, one could likely adjust the weight on the R matrix to make the non-Integral Action version have better reference tracking, but it would probably increase the overall input effort. Also, notice that the input signal is much smoother for the version with integral action - this is a desirable quality for real life systems.

References

[1] https://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling

[2] https://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/pph02-ch19-23.pdf

[3] https://github.com/redwall9/controls_simulations/blob/main/models/InvertedPendulumModel.py

[4] https://lewisgroup.uta.edu/ee5322/lectures/ContDiscKalmanFilter.pdf

[5] https://natanaso.github.io/ece276a2017/ref/DiscretizingContinuousSystems.pdf

[6] https://www.kth.se/social/upload/5194b53af276547cb18f4624/lec13_mpc2_4up.pdf

[7] https://arxiv.org/pdf/2109.11986.pdf

[8] https://www.mathworks.com/help/mpc/ug/terminal-weights-and-constraints.html

[9] https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7289336