ROB 101: Project 3, Segway

Eva Mungai, Grant Gibson, Wami Dosunmu-Ogunbi, and Jessy Grizzle

Fall 2020

1 Project Introduction

Picture this: You are a new member of an up and coming competitive tag group called The Cassie Bots. It is your turn in the arena, and you are nervous because you do not want to let your teammates down. You observe your opponent intently, already formulating strategies of where and when to move to not get tagged. The buzzer rings, indicating that the round has begun. Your opponent lunges at you, and you evade. As she jumps over barriers and takes jabs at you, you duck, roll, and shimmy (yes shimmy because your moves are on point) out of the way. Your mind is racing, adjusting and readjusting strategies in accordance to how your opponent reacts. You start down one path, only to change course last minute when you see your worthy opponent make a move that would have ultimately cut you off.

Figure 1: Competitive tag

Besides being such a cool way to legitimize grown adults playing a childish game, competitive tag provides quite the helpful illustration to the next major topic that we would like to cover in this class: **Model Predictive Control** (MPC). (If you have never seen competitive tag before it's pretty intense! Check out a clip here if you are interested.)

The premise of MPC in the context of this example is this. You have information on your past (the previous moves that you made to evade your opponent as well as the moves that she has made to try to best you) and you know your current state (yours and your opponent's current position). Using this knowledge, you plan a path for you to take to evade your opponent's next move.

In other words, in MPC you use information on your past and present to predict what would happen in the future so that you can act appropriately now.

In this document, we will show you how to model a self-driving car driving on a straight path. Using what you have learned earlier in this semester, we will show you how to transform an MPC program to look like a quadratic program for this self-driving car example. Finally, we will introduce to you a simplified segway system where you will use what we teach from the self-driving car example to design your own segway controller.

2 Car on a straight path

Let's study a simple example that will help motivate and explain Model Predictive Control.

Imagine that you've landed your dream internship at a self-driving car company (lucky you!). Your first task on the job is to design an automatic throttle control system [1] for the car. As the car is a prototype, it is programmed to only drive in a straight line. For any other intern this might be a daunting task but not for you. Armed with your ROB 101 knowledge you will be ready! Let us take a look at how we can solve this problem. The first question at hand is how to describe the motion (or dynamics) of the self-driving car.

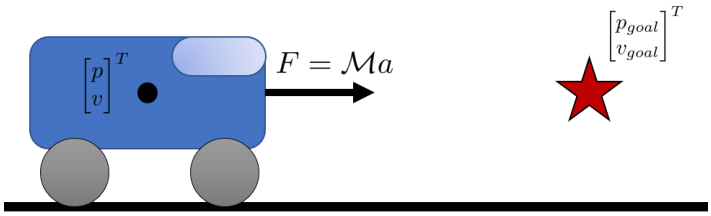


Figure 2: Self-driving car on flat ground.

2.1 Equations of Motion

Newton's second law tells us that the force (F) applied to an object with mass ($\mathcal{M}$) is equivalent to its mass times acceleration (a). In other words, $F = \mathcal{M}a$. Calculus tells us that acceleration is the derivative of velocity (v) with respect to time (t).

$$\frac{dv}{dt} = a$$

If we plot velocity vs time, the acceleration can be approximated by the slope of the plot

$$\frac{\Delta v}{\Delta t} \approx a \implies v_{k+1} - v_k \approx a\Delta t \implies v_{k+1} \approx v_k + a_k \Delta t \tag{1}$$

We now have an approximate **finite difference equation** that describes how the velocity changes over time as a function of acceleration. k represents the current discrete time index. The equivalent continuous time index can be derived by multiplying k by a prescribed time increment (Δt). From now on we will represent this finite difference equation with an '=' symbol because it converges to the continuous differential equation solution as Δt decreases.

Using this same method we can compute a finite difference equation that expresses how position changes

$$\frac{dp}{dt} = v \tag{2}$$

$$\frac{\Delta p}{\Delta t} \approx v \implies p_{k+1} - p_k \approx v\Delta t \implies p_{k+1} \approx p_k + v_k \Delta t \tag{3}$$

For more information see Appendix C of the ROB 101 Booklet.

2.2 What are States and what are Control Inputs?

Now that we have derived the equations of motion for our system, we can combine them into a single equation that will allow us to predict the position and velocity of the self-driving car at future moments in time as a function of the applied force. as a point of vocabulary, we use the position and velocity of the car are called the **state** of the car or **state variables** of the car, while the force as a function of time is called a **control input** because we are allowed to vary it as we wish.

We represent the state of the car at time index k by x_k, where $x_k \in \mathbb{R}^{n_x}$, and where n_x represents the number of state variables, which for us is two,

$$x_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix}. \tag{4}$$

[1] "In an internal combustion engine, the throttle is a means of controlling an engine's power by regulating the amount of fuel or air entering the engine. In a motor vehicle the control used by the driver to regulate power is sometimes called the throttle, accelerator, or gas pedal." https://en.wikipedia.org/wiki/Throttle

From (1) and (3) we can write

$$x_{k+1} = \begin{bmatrix} p_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{bmatrix} p_k + v_k \Delta t \\ v_k + a_k \Delta t \end{bmatrix} = \begin{bmatrix} p_k + v_k \Delta t \\ v_k + \frac{F_k}{\mathcal{M}} \Delta t \end{bmatrix} \tag{5}$$

Notice that the state vector at the next time instant x_{k+1} can be rewritten as a function of state vector at current time x_k and the throttle force F_k,

$$\begin{aligned} x_{k+1} &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_k \\ v_k \end{bmatrix} + \begin{bmatrix} 0 \\ \Delta t / \mathcal{M} \end{bmatrix} F_k \\ &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ \Delta t / \mathcal{M} \end{bmatrix} F_k \end{aligned} \tag{6}$$

F_k is the **control input** that is applied to the system at its current state (x_k) in order to drive the system to its next state (x_{k+1}). Equation (6) in control theory is referred to as the **discrete dynamics**. The discrete dynamics equation can be generalized to any system with finite difference equations. It is common practice to represent the matrix that multiplies the previous state as A and the matrix that multiplies the control input as B. The control input is also represented by the letter u with length n_u.

$$x_{k+1} = Ax_k + Bu_k. \tag{7}$$

We will use the notation shown in (7) for the remainder of this project. To be extra clear, in our case,

$$A := \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, B := \begin{bmatrix} 0 \\ \Delta t / \mathcal{M} \end{bmatrix}, x_k := \begin{bmatrix} p_k \\ v_k \end{bmatrix}, \text{ and } u_k := F_k.$$

2.3 State Trajectories

Now we have a way to compute the future state based on the current state and current input! A nice way to combine current and future state information is to store it inside of an object called a trajectory. A **state trajectory** is a set of all future states of a system given some initial state and control input. The set is described by (8) where N represents the final time step.

$$x^{traj} = \{x_0, x_1, \cdots x_N\} \tag{8}$$

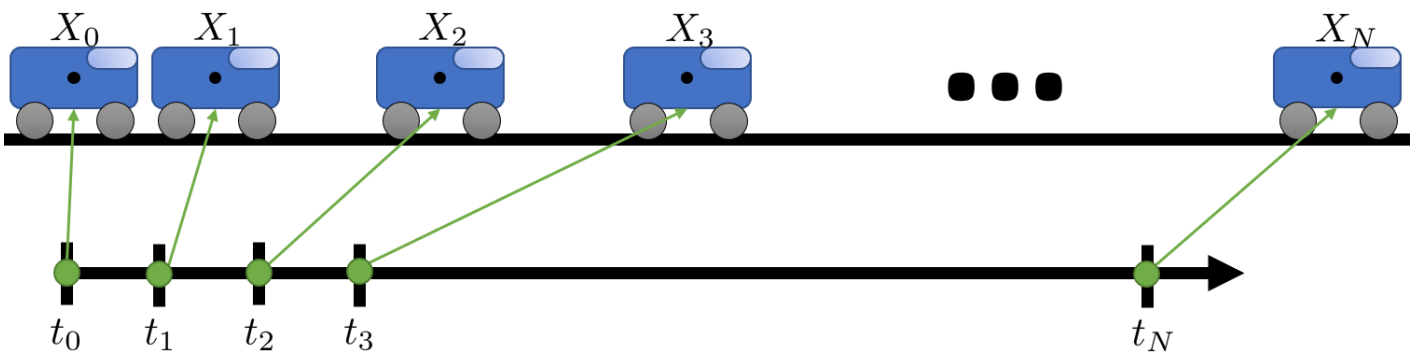


Figure 3: Car state trajectory

Let us now take a look at an example that uses (6) and the concept of state trajectories.

Example 1

Given an initial state $x_0 = \begin{bmatrix} p_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0\ m \\ 2\ m/s \end{bmatrix}$, a constant throttle force input $F = 3\ Newtons$ for all discrete time indices, a time increment $\Delta t = 0.1\ s$, and mass $\mathcal{M} = 1\ kg$.

1. Compute the state trajectory of the car for $k = 0, 1, 2, 3$
   Solution:
   Our goal is to derive the state trajectory of the car for $k = 0, 1, 2, 3$ with a time increment of $\Delta t = 0.1$. To do so, we apply the discrete dynamics equation (6) to the state at each time index

$$
\begin{aligned}
x_0 &= \begin{bmatrix} 0 \\ 2 \end{bmatrix} \\
x_1 &= \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x_0 + \begin{bmatrix} 0 \\ \Delta t/\mathcal{M} \end{bmatrix} F = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.1 \end{bmatrix} 3 = \begin{bmatrix} 0.2 \\ 2.3 \end{bmatrix} \\
x_2 &= \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x_1 + \begin{bmatrix} 0 \\ \Delta t/\mathcal{M} \end{bmatrix} F = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0.2 \\ 2.3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.1 \end{bmatrix} 3 = \begin{bmatrix} 0.43 \\ 2.6 \end{bmatrix} \\
x_3 &= \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x_2 + \begin{bmatrix} 0 \\ \Delta t/\mathcal{M} \end{bmatrix} F = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0.43 \\ 2.6 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.1 \end{bmatrix} 3 = \begin{bmatrix} 0.69 \\ 2.9 \end{bmatrix}
\end{aligned}
$$

   We have now computed the state trajectory of this system and can describe it as such

$$x^{traj} = \left\{ \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 0.2 \\ 2.3 \end{bmatrix}, \begin{bmatrix} 0.43 \\ 2.6 \end{bmatrix}, \begin{bmatrix} 0.69 \\ 2.9 \end{bmatrix} \right\} \tag{9}$$

   When stored as a matrix,

$$x^{traj} = \begin{bmatrix} 0.00 & 0.20 & 0.43 & 0.69 \\ 2.00 & 2.30 & 2.60 & 2.90 \end{bmatrix} \tag{10}$$

2. What is the time at $k = 3$
   Solution: This part is left as an exercise.

The MATLAB code used to solve the example above is given here

```
% Example 1
clear; clc;
x0 = [0;2];
u0 = 3;
dt = 0.1;
N = 3;
[X_traj,U_traj] = Discrete_Dynamics_Update(x0,u0,dt,N)
function [X_traj,U_traj] = Discrete_Dynamics_Update(x_init,u_init,delta_T,N)
A = [1, delta_T; 0 1];
B = [0; delta_T];
U_traj = zeros(length(u_init),N);
X_traj = zeros(length(x_init),N+1);
for i = 1:N+1
    if i == 1
        U_traj(:,i) = u_init;
        X_traj(:,i) = x_init;
    else
        U_traj(:,i) = u_init;
        X_traj(:,i) = A*X_traj(:,i-1) + B*U_traj(:,i-1);
    end
end
end
```

Problem 1

Now given an initial state $x_0 = \begin{bmatrix} p_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0\ m \\ 25\ m/s \end{bmatrix}$, time increment $\Delta t = 0.1\ s$, throttle force $F_{k+1} = (2v_k + 30)\ Newtons$ with $F_0 = 0\ Newtons$ and $k = 0, 1, 2, 3, 4, 5$ find the state trajectory. The answer should be a single matrix where each column vector corresponds to a state at a single time index.

2.4 State Transition Matrix

For predicting future states as is required for model predictive control, it is beneficial to reorganize the state trajectory into a single matrix, X, called the **state transition matrix**. In fact, this matrix is just one large vector with each state stacked on top of each other (11). The initial state condition (x_0) is given and so it usually omitted from this matrix.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \tag{11}$$

Remark: It is very important to note that each x_k is a vector! Moreover, X is a giant vector with all of the states of the system stacked up. It contains the entire trajectory of the system from time $k = 1$ to time $k = N$. If we go back to (9), then we have built X by taking the columns of x^{traj} and stacking them up to form a vector, like this

$$x^{traj} := \begin{bmatrix} 0.00 & 0.20 & 0.43 & 0.69 \\ 2.00 & 2.30 & 2.60 & 2.90 \end{bmatrix} \longrightarrow X := \begin{bmatrix} 0.00 \\ 2.00 \\ 0.20 \\ 2.30 \\ 0.43 \\ 2.60 \\ 0.69 \\ 2.90 \end{bmatrix}.$$

Both methods of keeping track of the trajectory are useful. It is all a question of bookkeeping! In the end, that is what matrices and vectors are all about.

Inspecting (7) we see that the next state is only a function of the current state and current input. Since we already know the initial state at $k = 0$, we can compute the state transition matrix as a function of only the initial state and sequential control inputs. Let's see how this works!

$$\begin{aligned} x_0 &= \text{ given} \\ x_1 &= Ax_0 + Bu_0 \\ x_2 &= Ax_1 + Bu_1 = A(Ax_0 + Bu_0) + Bu_1 = A^2x_0 + ABu_0 + Bu_1 \\ x_3 &= Ax_2 + Bu_2 = A(A^2x_0 + ABu_0 + Bu_1) + Bu_2 = A^3x_0 + A^2Bu_0 + ABu_1 + Bu_2 \\ &\vdots \\ x_N &= A^Nx_0 + A^{N-1}Bu_0 + A^{N-2}Bu_1 + \cdots Bu_{N-1} \end{aligned} \tag{12}$$

In general we can compute the state at any time index using the state transition formula ($x_k = A^kx_0 + \sum_{i=0}^{k-1} A^{k-1-i}Bu_i$). Let's define an additional vector called the **control trajectory matrix** that stacks the control inputs on top of each other.

$$U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix} \tag{13}$$

Using (12) and (13) we can rewrite the state transition matrix as a function of matrix multiplications and additions.

$$X = \underbrace{\begin{bmatrix} B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{N-1}B & A^{N-2}B & \cdots & B \end{bmatrix}}_{S} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix}}_{U} + \underbrace{\begin{bmatrix} A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}}_{M} x_0$$

$$X = SU + Mx_0 \tag{14}$$

(We introduce the matrices S and M for simplicity and compactness of notation.)

Problem 2

1. Derive the S and M matrices for Example 1.
2. Write a function that outputs S and M for any given A and B matrices. As a sanity check if you compute the state transition matrix from these matrices you should get back the state trajectory from Example 1. In this case, the states are stacked vertically in a single vector and the initial condition is omitted.

$$SU + Mx_0 = S \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} + M \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 2.3 \\ 0.43 \\ 2.6 \\ 0.69 \\ 2.9 \end{bmatrix}$$

2.5 From QP to MPC and Back Again!

Up until now, we have given you equations for computing the control input at each time step to get the car from point A (x_0) to point B (x_{goal}). In an ideal world, this would be sufficient. However, in reality we are constrained by factors such as the amount of fuel we have. We are also affected by external disturbances such as road conditions or weather. How can we account for such factors and conditions when designing our control inputs?

Let's look at fuel efficiency. To conserve the amount of fuel that you use as you drive you adjust the amount of pressure you apply on the gas pedal, thereby limiting the throttle force of the car. You are still trying to get from point A to point B, but you are trying to do it with the least amount of fuel. In other words, we want to minimize the force and the state error of the car throughout the entire state trajectory. The state error is defined as the difference between your current state (x_k) and your goal (x_{goal}). This sounds like an optimization problem!

In the past we have used **quadratic programs** (QP) to solve optimization problems. It turns out that with some modification, we can use a QP to solve this problem as well! Let's take a look at how we can do that!

Recall that a QP is written as follows:

$$z^* = \arg \min_{z \in \mathbb{R}^n} \frac{1}{2} z^T Q z + qz \tag{15}$$

subject to

$$A_{in} z \leq b_{in} \tag{16}$$

$$A_{eq} z = b_{eq} \tag{17}$$

$$lb \leq z \leq ub \tag{18}$$

where z is the optimization variable, (15) is the objective function, (16) and (17) are the linear inequality and equality constraints respectively, and lb and ub are the lower and upper bounds of z respectively. Note that for our example we will have to include the dynamic equations in our constraints. We do this because the dynamic equations describe the system that we want to apply the solution to. Another thing to note is that when we defined QP earlier in the semester, we used x in the objective function. However, here we use z to avoid confusion with our state variables. For more information see Section 11.6 in the ROB 101 Booklet.

To meet the goal of minimizing the force and the state error throughout the trajectory while respecting the dynamics of the self-driving car we write the following optimization problem:

$$z^* = \arg\min \sum_{k=0}^{N-1} \left[(x_k - x_{goal})^T Q_{state} (x_k - x_{goal}) + u_k^T Q_{control} u_k \right] + (x_N - x_{goal})^T Q_{state} (x_N - x_{goal})$$

subject to

$$x_{k+1} = Ax_k + Bu_k$$

$$x_{min} \leq x_k \leq x_{max}$$

$$u_{min} \leq u_k \leq u_{max} \tag{19}$$

For reference, $(x_k - x_{goal})^T Q_{state}(x_k - x_{goal})$ corresponds to the weighted square error between each state of the trajectory and the goal state, $u_k^T Q_{control} u_k$ corresponds to the weighted square magnitude of the control input, Q_{state} is the **state error penalty**, and $Q_{control}$ is the **control magnitude penalty**. Both of these matrices, Q_{state} and $Q_{control}$, can be tuned for user preference. Note that the penalty matrices are also referred to as cost matrices. This optimization problem, however, does not resemble a QP. Here are a few differences you might notice:

- the objective function has a summation,
- the optimization variable z is not explicitly used in the objective function, and
- the constraints are not defined in terms of A_{in} and b_{in} or A_{eq} and b_{eq}.

However not all is lost, thanks to matrix manipulation we can modify our optimization problem to resemble a QP! Let's start by taking a look at how we can modify the objective function.

The first step is to understand the purpose of the summation. The summation ensures that we are minimizing the state error and the input throughout the trajectory. It is left up to you to figure out why the final state error is not included in the summation (hint: take a look at (11) and (13)) .

To explicitly show the dependency of the objective function on the state and the input we rewrite it as follows:

$$z^* = \arg\min \sum_{k=0}^{N-1} \left[(x_k - x_{goal})^T Q_{state}(x_k - x_{goal})\right] + (x_N - x_{goal})^T Q_{state}(x_N - x_{goal}) + \sum_{k=0}^{N-1} \left[u_k^T Q_{control} u_k\right]$$

Given (11) and (13) we can derive

$$\sum_{k=0}^{N-1} \left[(x_k - x_{goal})^T Q_{state}(x_k - x_{goal})\right] + x_N^T Q_{state} x_N = (X - X_{goal})^T \bar{Q}_{state}(X - X_{goal})^T + (x_0 - x_{goal})^T Q_{state}(x_0 - x_{goal}) \tag{20}$$

$$\sum_{k=0}^{N-1} u_k^T Q_{control} u_k = U^T \bar{Q}_{control} U \tag{21}$$

where $\bar{Q}_{state}$ and $\bar{Q}_{control}$ are block diagonal matrices with the associated penalty matrices included along the diagonal.

$$\bar{Q}_{state} = \begin{bmatrix} Q_{state} & & 0 \\ & \ddots & \\ 0 & & Q_{state} \end{bmatrix} \in \mathbb{R}^{N \cdot n_x \times N \cdot n_x} \tag{22}$$

$$\bar{Q}_{control} = \begin{bmatrix} Q_{control} & & 0 \\ & \ddots & \\ 0 & & Q_{control} \end{bmatrix} \in \mathbb{R}^{N \cdot n_u \times N \cdot n_u} \tag{23}$$

We can now finally write our objective function as an algebraic matrix expression sans summation!

$$z^* = \arg\, min (X - X_{goal})^T \bar{Q}_{state}(X - X_{goal})^T + (x_0 - x_{goal})^T Q_{state}(x_0 - x_{goal}) + U^T \bar{Q}_{control} U$$

Notice that since x_0 is given, $(x_0 - x_{goal})^T Q_{state}(x_0 - x_{goal})$ is a constant. Therefore, we can exclude it from our objective function. We can do this because adding a constant to an objective function does not change the optimal answer. Our objective function then becomes

$$z^* = \arg\, min (X - X_{goal})^T \bar{Q}_{state}(X - X_{goal})^T + U^T \bar{Q}_{control} U$$

The optimization variables in the objective function are X and U. There are a couple of ways to proceed from here. One is to define $z = \begin{bmatrix} X \\ U \end{bmatrix}$ and rewrite the objective function to resemble a QP by using matrix manipulation. The other is to use (14) to write X in terms of U. We will use the second method. After some algebra, see Appendix A, we arrive at the following objective function:

$$U^* = \arg\min_U U^T H U + qU \tag{24}$$

where
$H = S^T \bar{Q}_{state} S + \bar{Q}_{control}$
$q = 2(Mx_0 - X_{goal})^T \bar{Q}_{state} S$

Now that our objective function resembles the one for the QP, let's turn our attention to the constraints. Since our objective function now only depends on U and x_0 (which is given), we need to figure out a way to rewrite our constraints in terms of U. We start by stacking the state bounds for each time step on top of each other in the following vectors

$$X_{min} = \begin{bmatrix} x_{min} \\ x_{min} \\ \vdots \\ x_{min} \end{bmatrix} \ , \ X_{max} = \begin{bmatrix} x_{max} \\ x_{max} \\ \vdots \\ x_{max} \end{bmatrix} \tag{25}$$

$X_{min}, X_{max} \in \mathbb{R}^{N \cdot n_x}$.

We can modify the state bounds accordingly,

$$X_{min} \leq X = SU + Mx_0 \leq X_{max} \tag{26}$$

$$\Downarrow$$

$$\begin{bmatrix} SU \\ -SU \end{bmatrix} \leq \begin{bmatrix} X_{max} - Mx_0 \\ -X_{min} + Mx_0 \end{bmatrix} \tag{27}$$

$$\Downarrow$$

$$\begin{bmatrix} S \\ -S \end{bmatrix} U \leq \begin{bmatrix} X_{max} - Mx_0 \\ -X_{min} + Mx_0 \end{bmatrix} \tag{28}$$

The control bounds can be reorganized in a similar manner,

$$U_{min} \leq U \leq U_{max} \tag{29}$$

$$\begin{bmatrix} I \\ -I \end{bmatrix} U \leq \begin{bmatrix} U_{max} \\ -U_{min} \end{bmatrix} \tag{30}$$

Combining (28) and (30) we can derive the matrix inequality that's needed for the QP.

$$A_{in} U \leq b_{in} \tag{31}$$

$$\boxed{\begin{bmatrix} S \\ -S \\ I \\ -I \end{bmatrix} \begin{bmatrix} u_0 \\ \vdots \\ u_{N-1} \end{bmatrix} \leq \begin{bmatrix} X_{max} - Mx_0 \\ -X_{min} + Mx_0 \\ U_{max} \\ -U_{min} \end{bmatrix}} \tag{32}$$

2.5.1 QP optimization

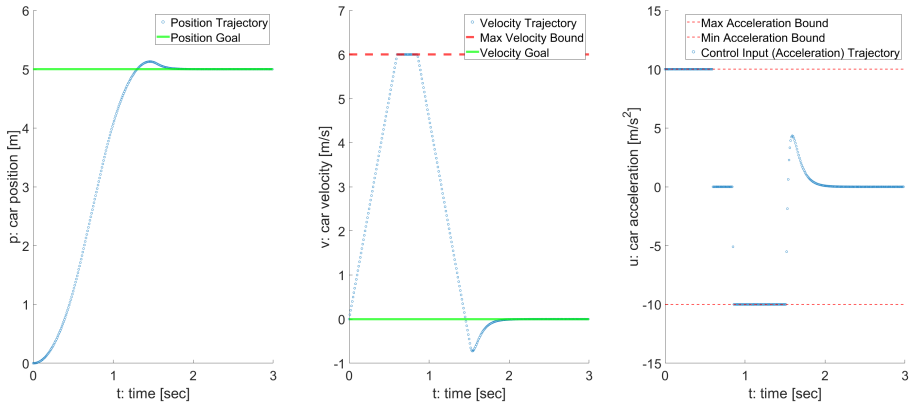
After all of our hard work, (19) can now be simply rewritten as a QP, which we already know how to solve!

$$\begin{aligned} U^* = \arg\min_U \ & U^T H U + qU \\ & \textbf{subject to} \\ & A_{in} U \leq b_{in} \end{aligned} \tag{33}$$

Using the knowledge you have just acquired from this scroll of wisdom, solve Problem 3 below.

Problem 3

1. Given an initial state $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, time increment $\Delta t = 0.01\ s$, mass $\mathcal{M} = 1\ kg$, $N = 300$ time steps, $A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ \Delta t/\mathcal{M} \end{bmatrix}$. Use (33) to compute the optimal input forces for the car to arrive at $x_{goal} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$ with state and control penalties $Q_{state} = \begin{bmatrix} 100 & 0 \\ 0 & 1 \end{bmatrix}, Q_{control} = 0.001$. The magnitude of the velocity cannot exceed 6 m/s $\left(\begin{bmatrix} -\infty \\ -6 \end{bmatrix} \leq x_k \leq \begin{bmatrix} \infty \\ 6 \end{bmatrix} \right)$ and the magnitude of the force cannot exceed 10 $Newtons$ $\left(-10 \leq u_k \leq 10 \right)$. You need to compute $X_{min}, X_{max}, U_{min}, U_{max}, X_{goal}, H, q, A_{in}$ and b_{in}.
The quadratic program solver will take H, q, A_{in}, and b_{in} as inputs and will return N control inputs. If we apply the state transition matrix equation (14) to obtain the state trajectory we can see that the state goal is reached!



(34)

Functions to fill out

```
function [Qstate_bar,Qctrl_bar] = Compute_Penalty_Matrices(Qstate,Qctrl,N)
Qstate_bar = [];
Qctrl_bar = [];
for i = 1:N
    Qstate_bar = blkdiag(Qstate_bar,Qstate);
    Qctrl_bar = blkdiag(Qctrl_bar,Qctrl);
end
end
```

```
function [X_goal,X_max,X_min,U_max,U_min] = Compute_Stacked_Constraints_Goals_Matrices(x_goal,x_max,x_min,
    u_max,u_min,N)
X_goal = repmat(x_goal,N,1);
X_max = repmat(x_max,N,1);
X_min = repmat(x_min,N,1);
U_max = repmat(u_max,N,1);
U_min = repmat(u_min,N,1);
end
```

```
function [H,q,Ain,bin] = Compute_QP_Matrices(params)
% Extract params
N = params.N;
X_goal = params.X_goal;
X_max = params.X_max;
X_min = params.X_min;
U_max = params.U_max;
U_min = params.U_min;
S = params.S;
M = params.M;
Qstate_all = params.Qstate_all;
```

```
Qctrl_all = params.Qctrl_all;
x_init = params.x_init;
dim_u = params.dim_u;

% compute QP matrices
H = S'*Qstate_all*S + Qctrl_all;
q = (2*(M*x_init - X_goal)'*Qstate_all*S)';
Ain = [S;
    -S;
    eye(N*dim_u);
    -eye(N*dim_u)];
bin = [X_max - M*x_init;
    -X_min + M*x_init;
    U_max;
    -U_min];
end
```

2.6 Transitioning to MPC

Now that we have rewritten our optimization problem as a QP, let's take a look at a new scenario. Imagine that the self-driving car has been deployed as a taxi and is on its way to pick up a customer. Given our newly rewritten optimization problem, would the car successfully pick up the passenger in the following situations?

- A change in the speed limit. Note that this change corresponds to a change in the bound, x_{max}.
- A change in the customer's pick-up location. What does this change correspond to?
- A change in the slope of the road which results in the car driving on an incline.

Using our current optimization problem, it's not readily apparent how we would incorporate these changes to the model. This is where model predictive control (MPC) comes in! With MPC, we can update the state bounds, and state goals as the the car is moving! We can even guarantee, depending on how far ahead we look into the future (prediction horizon), that at each time step the car will continue moving towards the goal. There are two questions at hand now: (a) How does MPC address all these challenges? and (b) How do we modify our optimization problem (33) so that it's an MPC problem? Luckily for us, we have done most of the work!

In MPC, our goal is to find the optimal control input at each time step that respects the discrete dynamics as well as the state and input constraints. We do so by evaluating how the states and inputs evolve $N - steps$ in the future. Therefore, to modify our optimization problem to an MPC problem, we just need to continuously recompute it for $U = \begin{bmatrix} u_k \\ u_{k+1} \\ \vdots \\ u_{k+N-1} \end{bmatrix}$ where x_k is the current state we're in, u_k is the control input that will get us to x_{k+1}, and N is the prediction horizon. Let us see how this works by looking at the scenario where the incline of the ground is different from what we expect.

In the original scenario we developed discrete dynamics which relied on the fact that the ground was flat. In our new scenario the acceleration of the car is affected by a constant force due to gravity and the angle of the incline (α). A simple schematic highlighting the relation between both forces is shown in Figure 5. The acceleration of the car along the direction of the slope is shown in (35).

$$\begin{aligned} \mathcal{M}a &= F - F_g \\ &\Downarrow \\ a &= \frac{F}{\mathcal{M}} - g\sin(\alpha) \end{aligned} \tag{35}$$

We now need to modify (1) to include the new definition for a derived in (35).

$$v_{k+1} = v_k + a_k \Delta t = v_k + \left(\frac{F_k}{m} - g\sin(\alpha) \right) \Delta t \tag{36}$$

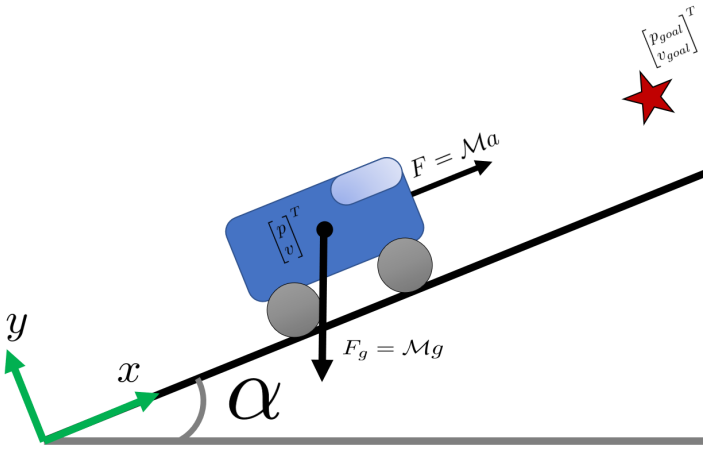


Figure 4: Car on Inclined Plane

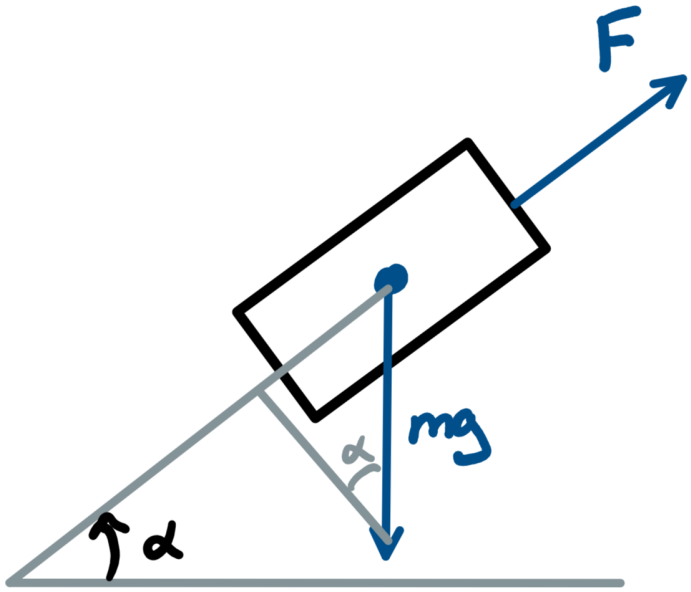


Figure 5: Incline force diagram

The resulting discrete dynamics equations are

$$x_{k+1}^{incline} = Ax_k + Bu_k + \begin{bmatrix} 0 \\ -g\sin(\alpha) \end{bmatrix} \tag{37}$$

The resulting state transition matrix can be computed using a similar method to (14)

$$X^{incline} = SU + Mx_0 + C_{incline} \tag{38}$$

However, without updating the discrete dynamics from (7) to (37) in the QP formulation, the controller will not know that the road conditions have changed. Therefore, assuming that the self-driving car is still trying to achieve the same goal state x_{goal}, the optimal control trajectory computed by the QP would be the same as the ones computed in Problem 3a. When these control inputs are applied to the new scenario, the self-driving car fails to reach x_{goal}. The resulting state trajectories computed from (38) using the control inputs from Problem 3a are plotted in Figure 6.

Instead of computing the throttle forces a single time, let us see if we can get a better result by recomputing the force inputs after

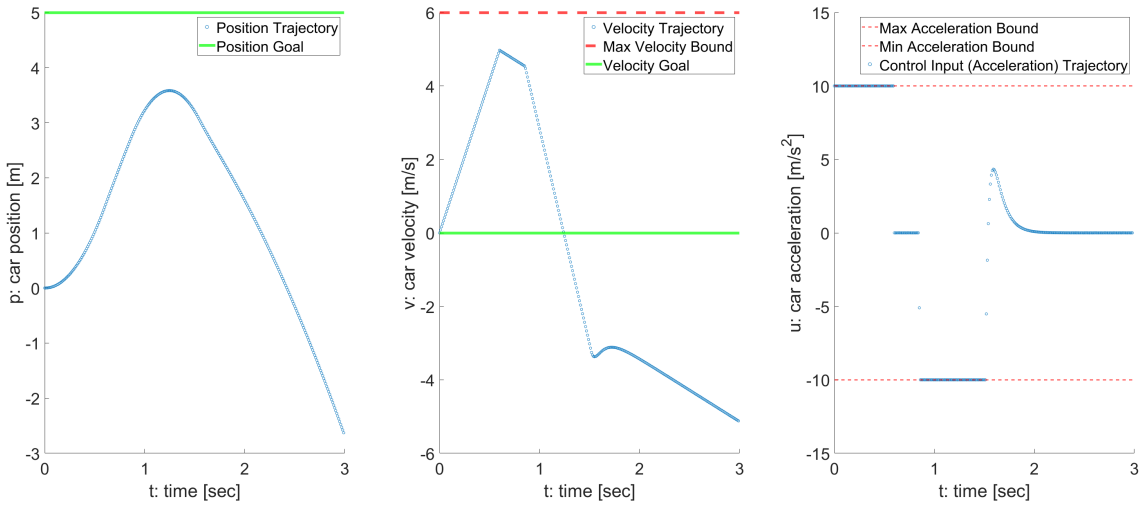


Figure 6: Problem 3a Solution applied to car on incline plane

$1.5s$ and applying these new inputs. Note that our control trajectory now becomes

$$U = \begin{bmatrix} \begin{Bmatrix} u_0 \\ \vdots \\ u_{k_{1.5s}-1} \end{Bmatrix} \text{control inputs from Problem 3a} \\ \\ \begin{Bmatrix} u_{k_{1.5s}} \\ \vdots \\ u_{N-1} \end{Bmatrix} \text{new control inputs recomputed after } 1.5s \end{bmatrix}$$

Note that we will have to recalculate the QP matrices (H, q, A_{in}, b_{in}) since they are dependent on the current state (x_k). **Reminder: the discrete dynamics used to formulate the QP are still defined by** (7)**. In other words, the QP is unaware of the new inclined plane dynamics!** Figure 7 displays the resulting state trajectories. Comparing the position plot of Figure 6 to Figure 7, we notice that at 1.5s (when we recomputed the control inputs) the car tries to recover from the downwards slope by choosing maximum force inputs. This recovery is directly related to the state error term in the objective function.

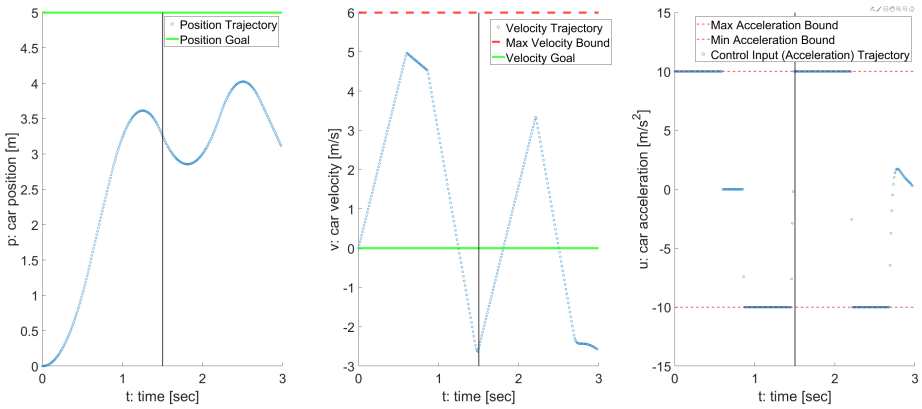


Figure 7: Car on incline plane with a QP re-computation at 1.5s

As we recompute the control inputs more frequently, the state trajectory converges to the desired goal state (see Figure 8).

If we recompute the QP optimization after each time step we notice that we are able to converge to the goal state.

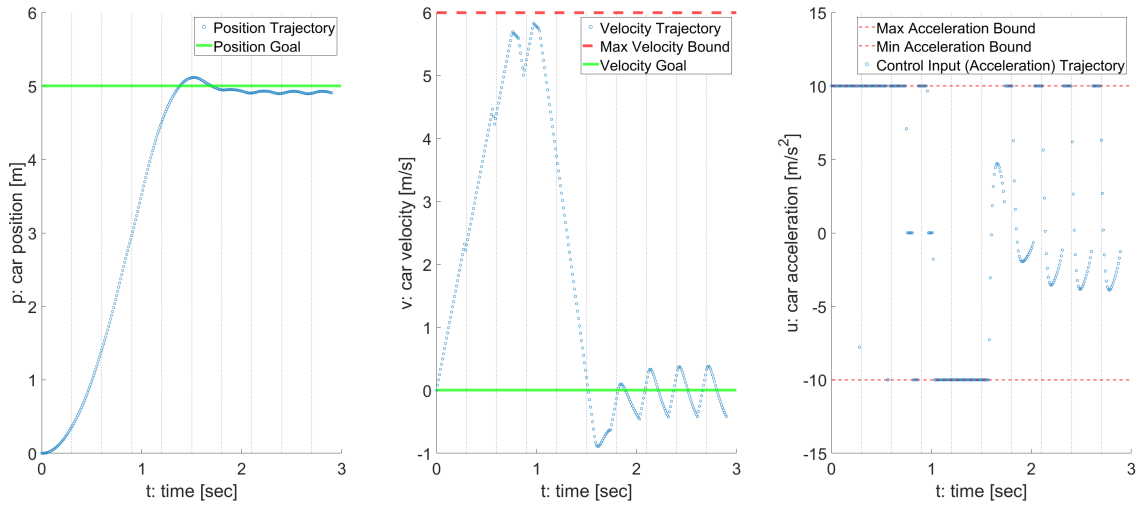


Figure 8: Trajectories for inclined car with re-computations every 0.3 seconds.

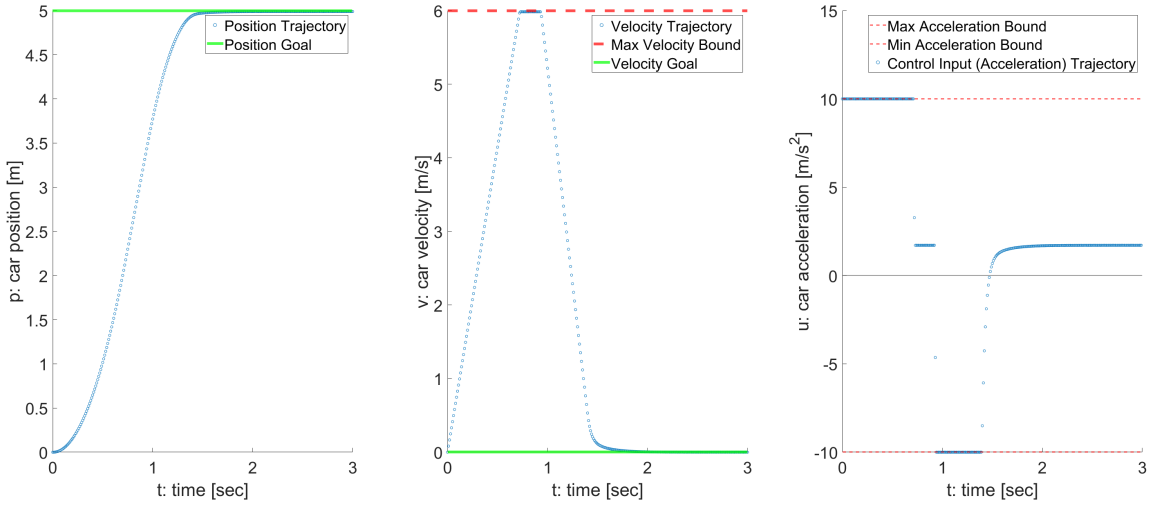


Figure 9: Trajectories for inclined car with re-computations every time step (0.01 seconds)

Did you notice what we just did? Somehow we were able to control the car even though the real world model (inclined plane) was different from our simplified model (flat ground). This is the power of Model Predictive Control!

By continuously recomputing the QP problem at each time step (k) and only applying the first control input (u_k) we can derive control trajectories that help us reach desired state goals in real time in lieu of model mismatch, or other factors that push us away from our desired goal. Therefore, our optimal MPC controller givel (33) is

$$U^*_{mpc} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}_{1 \times N-1} U^* \tag{39}$$

Armed with all this knowledge you are now ready to take on the segway project!

3 Segway Intro

3.1 How Segways Work

Figure 10: In this project, you will learn a method to keep a Segway upright! Left photo found here. Right photo found here

A segway is a personal transportation vehicle that balances on two wheels. Invented by the American engineer Dean Kamen, segways stirred up quite the buzz when they were brought to market in 2001.

One might be rightly perplexed by how such an odd-looking device manages to stay upright on its own–especially when a human is standing on it! Intuitively, one would deduce that a person who stands on the device will immediately fall forward flat on their face (or backward on their backs!) as the platform that the person stands on rotates in one direction or the other taking its human companion with it. If this is what you are thinking, do not fret because your intuition is exactly correct! If the segway consisted of only two wheels and a platform to stand on, a human would very likely fall soon after trying to stand on the device.

But humans do not fall immediately after mounting a segway–otherwise I am sure that we would see many lawsuits–and that is because segways are not just two wheels and a platform. Segways also have motors that spin each of its two wheels individually, sensors that track the tilt, speed, and direction of the segway, and microprocessors that use the information that the sensors give it to determine how fast each motor should spin to keep the segway from falling.

Let me pose a question to you. If you were standing on top of a segway and started to fall forward (thus causing the tilt sensors on the segway to sense a forward tilt), which direction do you think the microprocessors will tell the wheels of the segway to spin to maintain its balance: forward or backward?

If you answered forward, you would be correct! If you were to lean forward on a segway, gravity would want to make you continue to fall forward, but the wheels of the segway would spin forward faster to keep the segway platform level beneath your feet. Thus, when you wish to move forward on a segway, you lean your body forward to get the wheels on the segway to spin forward. The more you lean, the faster the wheels will spin because ultimately the wheels are trying to spin fast enough such that the platform beneath your feet stays leveled. Similarly, if you were to lean backward on a segway, the microprocessors would tell the wheels to spin faster backward to keep the platform beneath you.

This logic still applies when you wish to stand stationary on a segway. When standing "still" on a segway, you are really falling forward one moment and backward in the next (unless you have amazing balancing skills that allows you to easily balance on a platform supported by two wheels). Thus the wheels will also move forward one moment and backward the next to keep the platform level.

If you are still having trouble understanding how the logic of segways works, it may help to draw a comparison with the human body. If you stood upright on solid ground and suddenly started to lean forward such that you became out of balance, you probably would not fall flat on your face. Your brain would know that you have lost balance because the fluid in your inner ear would shift in a manner such that it prompts your brain to tell your foot to step forward to prevent you from falling. If you continued to fall forward, your brain would have you put another foot forward, and another foot forward (causing you to walk–or run–forward), until it senses that you are no longer falling.

This is exactly what the segway is doing, except instead of legs, it has wheels; instead of muscles, it has motors; instead of inner ear fluid, it has a set of sophisticated sensors; and instead of a brain, it has a series of microprocessors.

Here is a cool link which describes segways in more detail if you are interested!

3.2 The Hardware on Our Segway

Now that we understand how segways work, let us take a look at the actual hardware components on a segway. Segways can vary between manufacturers and product lines and so may have varied hardware components. However, there are certain components that must exist on every segway in one form or another for the segway to function appropriately. In our implementation, we will focus on these bare minimum components. These components are as follows:

- Sensors: position encoders, velocity sensors
- Microprocessor: controller
- Actuators: wheel motor
- Physical components: platform, wheel, shaft

The sensors track the position and velocities of the shaft angle and wheel angle. Note the singular "wheel" here. We will be working with one wheel because we will be using a simplified 2D model. The microprocessor–which will be the controller that we develop–will take the data from the sensors, interpret them, and provide torque outputs to the actuator. The actuator here is a single motor that powers the wheel. Finally, the physical components of our simple system (the platform, wheel, and shaft) will respond accordingly to the output of the actuator. The sensors pick up the new position and velocities, and the cycle continues until we have reached a desired state.

If it helps, you may wish to refer to Figure 11 to understand the logic flow.

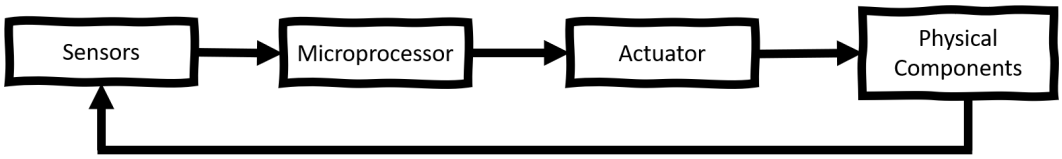


Figure 11: You may wish to refer to this diagram to help you visualize the logical flow of how the various components of our segway interact with one another.

3.3 Your Tasks

And now let us segue into what you actually need to do for this project (you knew that we had to use that pun at least once). The main task for this project is for you to design a controller to get our simplified 2D segway to a certain set of states, given an initial set of states. You will use the strategies that you learned from the car on a straight path example described in Section 1 of this scroll of wisdom.

We have selected four states for this segway problem: the shaft angle (ϕ), wheel angle (θ), shaft velocity ($\dot{\phi}$), and wheel velocity ($\dot{\theta}$). Refer to Figure 12 to see how these angles are defined.

$$x_k = \begin{bmatrix} \text{wheel angle} \\ \text{shaft angle} \\ \text{wheel angular velocity} \\ \text{shaft angular velocity} \end{bmatrix} = \begin{bmatrix} \theta \\ \phi \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} \tag{40}$$

When formulating the optimization problem, it is important to to set torque constraints on your motors. Do you know why?

If not, try thinking about what would happen if you let your microprocessor (controller) tell your motor to output an infinite amount of torque. That sounds impossible–because it is impossible! Sure, our controller is being applied to a virtual model of a segway and so the imaginary motors here would gladly output as much torque as you see fit.

But here at Michigan Robotics, we pride ourselves in creating models that can be applied to real life systems. If we want to design a segway controller that could be feasibly applied to an actual living, breathing segway (segways are alive, right?), then a step in that

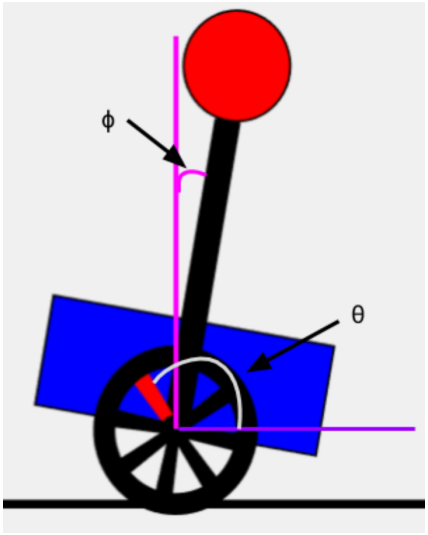


Figure 12: Segway angle definitions

direction is to ensure that our controller does not yield torque outputs that are infeasible for actual real life motors! So you better put bounds on those torques!

Note that the segway torque is analogous to what we called the throttle force in the car example. In other words, it is the control input for the segway.

We mentioned earlier that we will be using a simplified 2D model. This resulting planar model is a nonlinear model, see Appendix C in the ROB 101 Booklet, which make it difficult to implement our MPC controller in real-time. Therefore, we further simplify this model by linearizing it. In the end, the linearized discrete dynamics for the segway are:

$$x_{k+1} = \begin{bmatrix} 1.0129 & 0 & 0.10043 & 0 \\ -0.025154 & 1 & -0.00083774 & 0.1 \\ 0.2579 & 0 & 1.0129 & 0 \\ -0.50415 & 0 & -0.025154 & 1 \end{bmatrix} x_k + \begin{bmatrix} -0.0035937 \\ 0.008387 \\ -0.072027 \\ 0.16804 \end{bmatrix} u_k$$

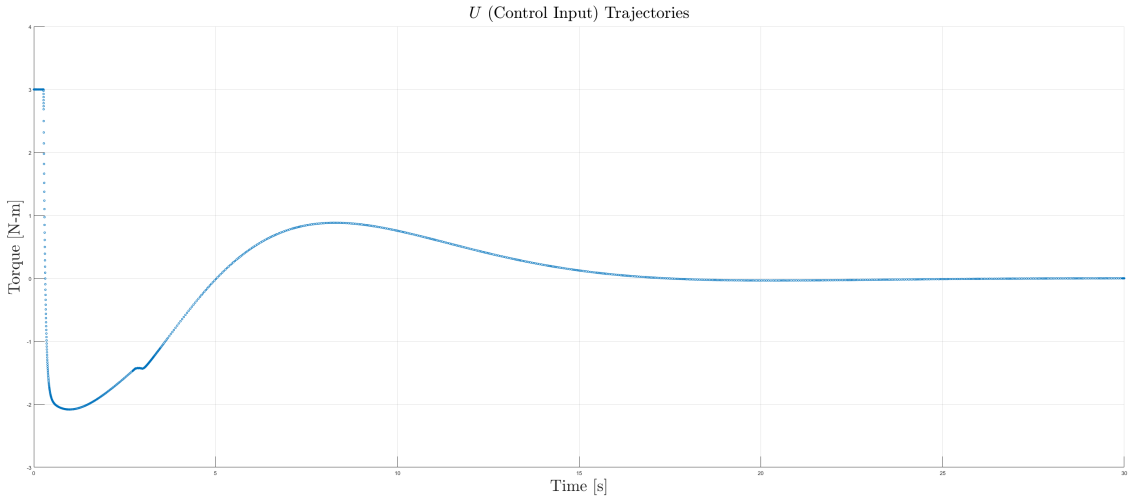
$$x_{k+1} = Ax_k + Bu_k \tag{41}$$

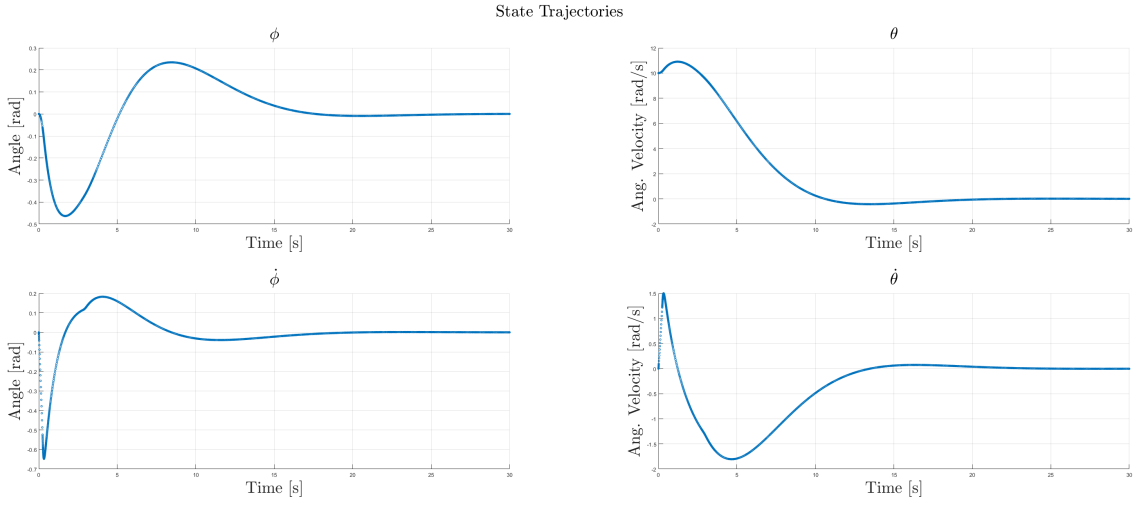
You will use these linearized dynamics to solve problems similar to the ones presented in the car example.

3.3.1 Stabilize vertically at single position

Problem 4a

1. Given an initial state $x_0 = \begin{bmatrix} 0 \\ 10 \\ 0 \\ 0 \end{bmatrix}$, time increment $\Delta t = 0.01\ s$, prediction horizon $N = 5$, A and B from (41) . Use (33) and (39) to compute the optimal input forces for the car to arrive at $x_{goal} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ with state and control penalties $Q_{state} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, Q_{control} = 0.01$. The magnitude of the torque cannot exceed 3 N-m $\left(-3 \leq u_k \leq 3 \right)$. You need to compute H, q, A_{in} and b_{in}.



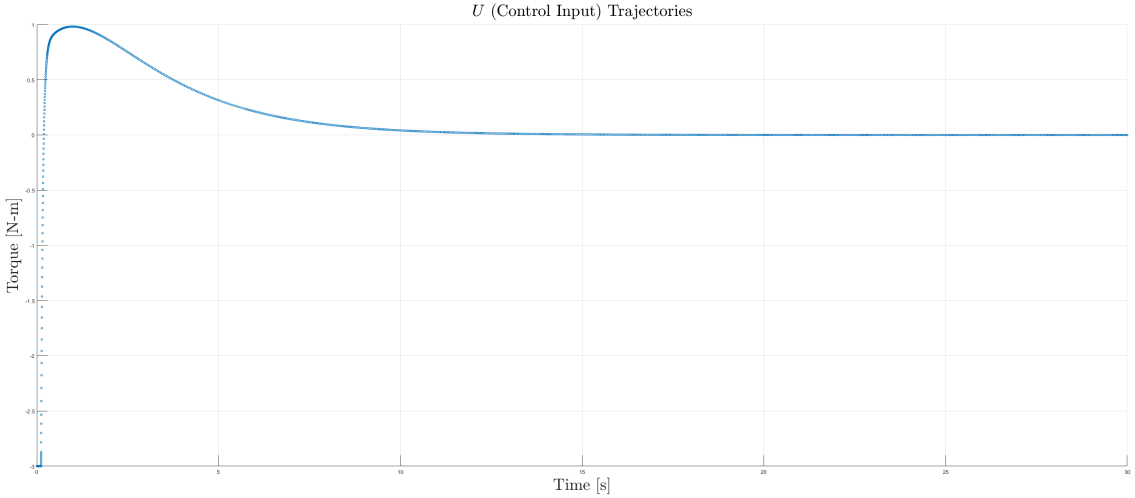


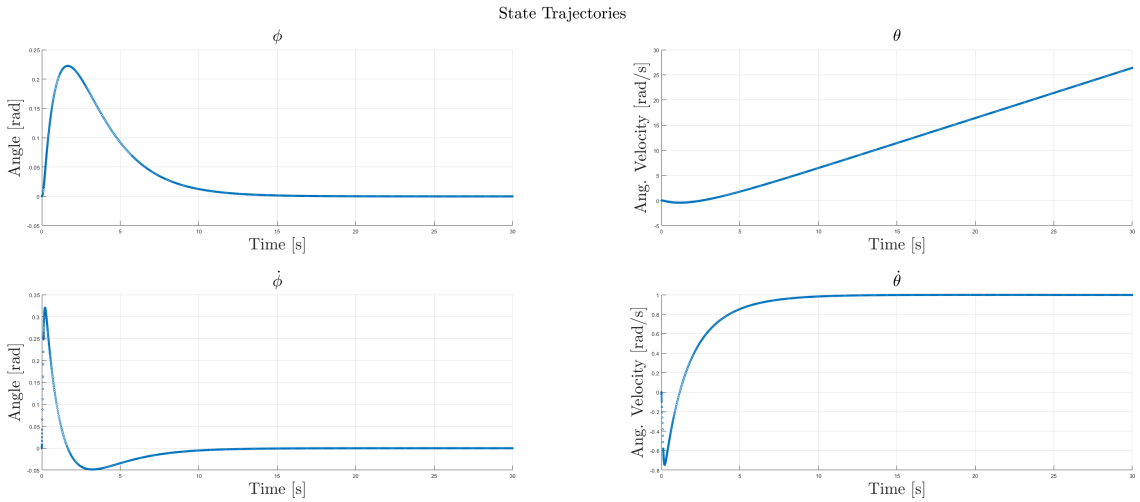
3.3.2 Stabilize about constant wheel angular velocity

Problem 4b

1. Given an initial state $x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, time increment $\Delta t = 0.01\ s$, prediction horizon $N = 5$, A and B from (41) . Use (33) and (39) to compute the optimal input forces for the car to arrive at $x_{goal} = \begin{bmatrix} \text{arbitrary} \\ \text{arbitrary} \\ 0 \\ 1 \end{bmatrix}$ with state and control penalties $Q_{state} = \begin{bmatrix} 0.0001 & 0 & 0 & 0 \\ 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 10000 \end{bmatrix}$, $Q_{control} = 0.01$. The magnitude of the torque cannot exceed 3 N-m $\left(-3 \leq u_k \leq 3 \right)$. You need to compute H, q, A_{in} and b_{in}.

   Note that for this problem the desired shaft and wheel angles do not matter because we our goal is to have the segway move with constant angular velocity. In other words, the a constant desired position goal does not make sense. As a result, we you can choose any values for the angles. This phenomenon is a direct result of our choice of Q_{state}, can you figure out why?





As mentioned earlier the real world dynamics of the segway are described by nonlinear dynamics. The dynamics in (41) are in fact the linearized version of the actual dynamics (42). Let's see if we can use our MPC controller to handle the model mismatch between the linear and nonlinear dynamics. Note that this is a very similar problem to Problem 3 in the car example. So if you find yourself lost in the sauce please refer back to Problem 3.

Your task here is to redo Problem 4a and Problem 4b with the QP formulated using the linear discrete dynamics but with the control inputs applied to the nonlinear dynamics. Will the segway still be able to reach the desired state goal? Do you think that a longer prediction will help?

$$x_{k+1}^{nonlinear} = f(x_k, u_k) \tag{42}$$

3.3.3 Stabilize vertically at single position with nonlinear dynamics

Problem 5a

1. Compute control inputs for the goal in Problem 4a using the nonlinear dynamics

3.3.4 Stabilize about constant wheel angular velocity with nonlinear dynamics

Problem 5b

1. Compute control inputs for the goal in Problem 4b using the nonlinear dynamics

3.3.5 Challenge

1. Try find the minimum control input bounds that satisfy Problem 5a (to the nearest tenth).
2. Try to find out how different choices of the penalty matrices will affect your results in the previous sections.
3. Change x_{goal} and see if the segway can reach the desired goal state. What if the desired goal state changes in the middle of the trajectory?
4. Create your own x_{goal} and modify Q_{state}, Q_{ctrl}

A Appendix: QP Cost Formulation

Substitution of State Transition Matrix into objective function.

$$
\begin{aligned}
U^* &= \arg\min_U (X - X_{goal})^T \bar{Q}_{state} (X - X_{goal})^T + U^T \bar{Q}_{control} U + (x_0 - x_{goal})^T Q_{state} (x_0 - x_{goal}) \\
&= \arg\min_U (SU + Mx_0 - X_{goal})^T \bar{Q}_{state} (SU + Mx_0 - X_{goal})^T + U^T \bar{Q}_{control} U + (x_0 - x_{goal})^T Q_{state} (x_0 - x_{goal}) \\
&= \arg\min_U U^T (S^T \bar{Q}_{state} S + \bar{Q}_{control}) U + 2x_0^T M^T \bar{Q}_{state} SU - 2X_{goal}^T \bar{Q}_{state} SU + X_{goal}^T \bar{Q}_{state} X_{goal} + \cdots \\
&\quad + x_0^T M^T \bar{Q}_{state} M x_0 + (x_0 - x_{goal})^T Q_{state} (x_0 - x_{goal}) \\
&= \arg\min_U U^T H U + qU + c
\end{aligned}
$$

where
$H = S^T \bar{Q}_{state} S + \bar{Q}_{control}$
$q = 2(Mx_0 - X_{goal})^T \bar{Q}_{state} S$
$c = X_{goal}^T \bar{Q}_{state} X_{goal} + x_0^T M^T \bar{Q}_{state} M x_0 + (x_0 - x_{goal})^T Q_{state} (x_0 - x_{goal})$

Here c is a constant and can be removed because it has no dependence on the optimization variables, leaving us with a cost function that can be used in a QP.