

# A Tutorial on CLF, CBF / CBF-CLF-Helper Manual

Library: <https://github.com/HybridRobotics/CBF-CLF-Helper>

Authors: Jason Jangho Choi ([jason.choi@berkeley.edu](mailto:jason.choi@berkeley.edu))

[Hybrid Robotics Lab](#)

# Introduction

- Control Lyapunov Function and Control Barrier Function based methods are effective in many safety-critical control problems. Each deal with safety in terms of stability and set invariance, respectively.
- CBF-CLF-Helper is designed to let users easily implement safety-controller based on CBFs and CLFs with Matlab. We provide:
  - An easy interface for construction and simulation of a control-affine nonlinear system.
  - Safety controller including CLF-QP, CBF-QP, and CBF-CLF-QP as built-in functions.
  - Demonstrations on toy examples.
- In this tutorial, the followings are provided.
  - Summary of fundamentals for the Control Lyapunov Function (CLF), Control Barrier Function (CBF), and relevant safety-controllers—CLF-QP and CBF-CLF-QP.
  - Illustration of steps for designing a CBF-CLF-QP controller using the CBF-CLF-Helper.

# Contents

- Backgrounds
  - Dynamics – Control Affine System
  - Control Lyapunov Function (CLF) and CLF-QP
  - Control Barrier Function (CBF) and CBF-CLF-QP
- Design Steps (Toy example – Adaptive Cruise Control)
- How to code up? (Toy example – Adaptive Cruise Control)
- Remarks
  - Effects of hyperparameters – CLF and CBF rates, slack cost weight.
  - Relative Degree
  - Deadlock Under Symmetry
- Other toy examples
  - Multiple control inputs & High relative degree - 2D Double Integrator
  - Dubins Car
  - Inverted Pendulum

# Backgrounds

# Dynamics – Control Affine System

Expression for dynamics of a general nonlinear controlled system:

$$\dot{x} = F(t, x, u)$$

where  $x \in \mathbb{R}^n$  is the system state,  $u \in \mathbb{R}^m$  is the control input.

If  $F$  is Lipschitz continuous in  $x$ , continuous in  $u$ , and piecewise continuous in  $t$ , and if  $u(\cdot)$  is piecewise continuous in  $t$ , we are guaranteed that given an initial state  $x(t_0) = x_0$ , the trajectory of the dynamics  $x(t)$  exists and it is unique\*.

In this library, we mainly deal with a specific type of a nonlinear system, a time-invariant **control affine system**:

$$\dot{x} = f(x) + g(x)u$$

where  $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$  and  $g: \mathbb{R}^n \rightarrow \mathbb{R}^{n \times m}$  are Lipschitz continuous in  $x$ .

We assume that  $x_e \equiv \mathbf{0}$  is an equilibrium point.

# Control Lyapunov Function (CLF)

Let  $V(x): \mathbb{R}^n \rightarrow \mathbb{R}$  be a continuously differentiable function.

If there exists a constant  $c > 0$  such that

- 1)  $\Omega_c := \{x \in \mathbb{R}^n : V(x) \leq c\}$ , a sublevel set of  $V(x)$  is bounded,
- 2)  $V(x) > 0$  for all  $s \in \mathbb{R}^n \setminus \{x_e\}$ ,  $V(x_e) = 0$ ,
- 3)  $\inf_{u \in U} \dot{V}(x, u) < 0$  for all  $x \in \Omega_c \setminus \{x_e\}$ ,

Then  $V(x)$  is a local **Control Lyapunov Function** and  $\Omega_c$  is a region of attraction (ROA), i.e. **every state in  $\Omega_c$  is asymptotically stabilizable to  $x_e$ .**

Derivative of  $V(x)$  along the dynamics:

$$\begin{aligned}\dot{V}(x, u) &= \nabla V(x) \cdot \dot{x} \\ &= \nabla V(x) \cdot f(x) + \nabla V(x) \cdot g(x)u \\ &= \mathbf{L}_f V(x) + \mathbf{L}_g V(x) \mathbf{u} \quad (L_p q(x) := \nabla q(x) \cdot p(x) \text{ is a Lie derivative operator}\end{aligned}$$

is affine in  $u$ .

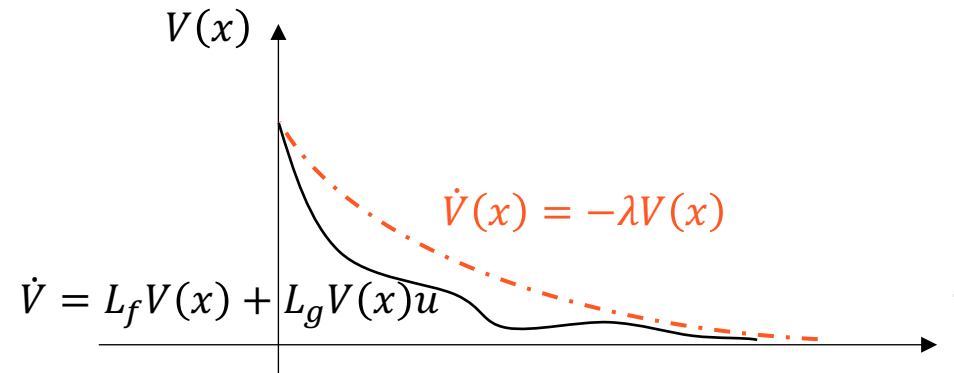
used to make formulas concise.)

# Exponentially Stabilizing Control Lyapunov Function (CLF):

Let  $V(x): \mathbb{R}^n \rightarrow \mathbb{R}$  be a continuously differentiable, positive definite, and radially unbounded function. If there exists some constant  $\lambda > 0$  such that

$$\inf_{u \in U} \dot{V}(x, u) + \lambda V(x) \leq 0,$$

then  $V(x)$  is an **exponentially stabilizing CLF (ES-CLF)** and any  $x \in \mathbb{R}^n$  is exponentially stabilizable to  $x_e^*$ .  $\lambda$  serves as a decay rate of an upper bound of  $V(x(t))$ .



$$\dot{V}(x, u) + \lambda V(x) \leq 0, \text{ for } \exists u \in U, \lambda > 0$$

# CLF-QP

- The CLF constraint is linear in  $u$ , so using it as a constraint in a min-norm controller results in a Quadratic Program (QP) formulation.

$$\begin{array}{ll} \underset{\substack{u: \text{ control input} \\ \delta: \text{ slack variable}}}{\text{argmin}} & (u - u_{ref})^T H(u - u_{ref}) + p\delta^2 \\ \\ \text{subject to: } & L_f V(x) + L_g V(x)u + \lambda V(x) \leq \delta \quad \text{CLF Constraint} \\ & u \in U \quad \text{Input Constraints} \end{array}$$

- This QP, which is a convex optimization problem, can be solved fast enough for real-time applications\*.
- Input constraints should be linear for the QP formulation.
- The CLF constraint is often relaxed with a slack variable to guarantee feasibility of the problem. If the feasibility is guaranteed without the relaxation, the controller will exponentially stabilize the system to  $x_e$ .



# Control Barrier Function (CBF)

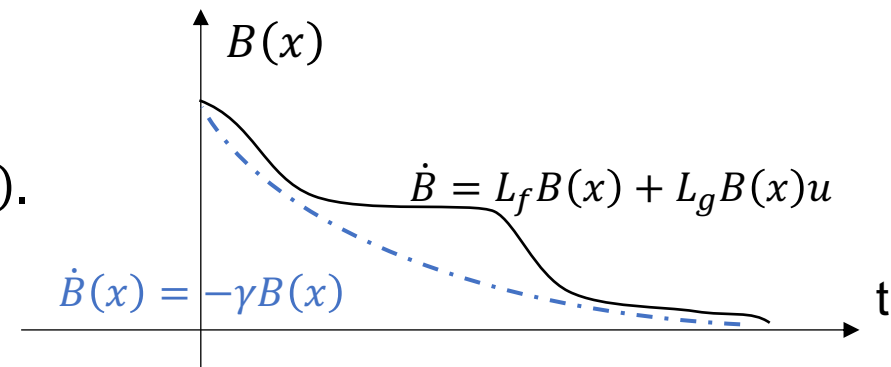
Let  $B(x): \mathbb{R}^n \rightarrow \mathbb{R}$  a continuously differentiable function whose zero-superlevel set is  $\mathcal{C}$  and  $\nabla B(x) \neq 0$  for all  $x \in \partial\mathcal{C}$ .

If there exists an extended class  $\mathcal{K}_\infty$  function  $\alpha$  and a set  $\mathcal{D} \subset \mathbb{R}^n$  such that  $\mathcal{C} \subset \mathcal{D}$  and

$$\sup_{u \in U} [L_f B(x) + L_g B(x)u] + \alpha(B(x)) \geq 0$$

for all  $x \in \mathcal{D}$ , then  $B(x)$  is a **Control Barrier Function** and any Lipschitz continuous control law that satisfies the above constraint will render the set  $\mathcal{C}$  safe (i.e. **control invariant**)\*.

In practice, a linear function with positive coefficient  $\gamma$  is often used as  $\alpha(\cdot)$ ;  $\alpha(B(x)) = \gamma B(x)$ . Then,  $\gamma$  serves as a decay rate of a lower bound of  $B(x(t))$ .



$$\dot{B}(x, u) + \gamma B(x) \geq 0, \text{ for } \exists u \in U, \gamma > 0$$

# CBF-CLF-QP

$$\begin{aligned} & \underset{\substack{u: \text{control input} \\ \delta: \text{slack variable}}}{\text{argmin}} \quad (u - u_{ref})^T H(u - u_{ref}) + p\delta^2 \\ & \text{subject to: } L_f V(x) + L_g V(x)u + \lambda V(x) \leq \delta \quad \text{CLF Constraint} \\ & \quad \quad L_f B(x) + L_g B(x)u + \gamma B(x) \geq 0 \quad \text{CBF Constraint} \\ & \quad \quad u \in U \quad \text{Input Constraint} \end{aligned}$$

- If  $B(x)$  is a valid CBF under the input constraints, the QP is always feasible.
- When  $u$  that satisfies both CLF and CBF constraint exists, the slack variable of the solution is 0.
- Obviously, we can formulate the QP without the CLF constraint, called CBF-QP.

# Remarks

- Generally, Control Lyapunov Functions are designed for reaching a target state (or set) and Control Barrier Functions are designed for avoiding an unsafe set.
- A general method for designing valid CLFs and CBFs amounts to a research problem.

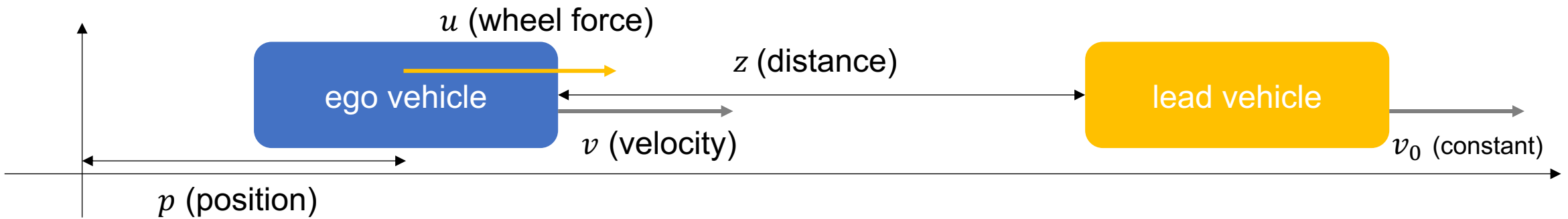
# Design Steps

Explained with Toy example:

CBF-CLF-QP for Adaptive Cruise Control (ACC)\*

\*This example is excerpted from A. D. Ames, J. W. Grizzle, and P. Tabuada, “Control barrier function based quadratic programs with application to adaptive cruise control,” CDC 2014, with a slight modification.

# Step 1. Define your problem: Dynamics & Control Objectives.



State

$$x = [p \ v \ z]^T \in \mathbb{R}^3$$

Control Input

$$u \in \mathbb{R}^1$$

Dynamics

$$\dot{s} = \underbrace{\begin{bmatrix} v \\ -\frac{1}{m}F_r(v) \\ v_0 - v \end{bmatrix}}_{f(s)} + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{m} \\ 0 \end{bmatrix}}_{g(s)} u$$

$F_r(v) = f_0 + f_1 v + f_2 v^2$  is a rolling resistance.

Input constraints

$$-mc_d g \leq u \leq mc_a g$$

Stability objective

$$v \rightarrow v_d \quad (v_d: \text{desired velocity})$$

Safety objective

$$z \geq T_h v \quad (T_h: \text{lookahead time})$$

## Step 2. Design a CLF and evaluate.

Stability objective  $v \rightarrow v_d$  ( $v_d$ : desired velocity)

A Lyapunov function should be 0 at  $x_e( [\cdot \ v_d \ \cdot]^T )$  and positive everywhere else.

Let's try the most intuitive one:  $V(x) = (v - v_d)^2$

$$\nabla V(x) = [0 \ 2(v - v_d) \ 0]$$

$$L_f V(x) = -\frac{2}{m} F_r(v)(v - v_d), \quad L_g V(x) = \frac{2}{m} (v - v_d)$$

$$\text{Constraint: } \dot{V}(x, u) + \lambda V(x) = L_f V(x) + L_g V(x)u + \lambda V(x) = (v - v_d) \left\{ \frac{2}{m} (u - F_r) + \lambda (v - v_d) \right\} \leq 0$$

The value of rolling resistance  $F_r$  is very small.

If  $v < v_d$ , if we accelerate sufficiently large,  $\left\{ \frac{2}{m} (u - F_r) + \lambda (v - v_d) \right\}$  will be positive and we can make the constraint satisfied. Similarly, the constraint is satisfied for the opposite case ( $v > v_d$ ) if we decelerate enough. Therefore, we conclude that this is a valid CLF.

Remark: If the value of  $\lambda$  becomes too large, depending on the input constraints the CLF can be invalid.

# Step 3, Design a CBF and evaluate.

Safety objective  $z \geq T_h v$  ( $T_h$ : lookahead time)

- Again, let's start with an intuitive choice for the CBF— $B(x) = z - T_h v$ .

$$\nabla B(x) = [0 \quad -T_h \quad 1]$$

$$L_f B(x) = \frac{T_h}{m} F_r(v) + (v_0 - v), \quad L_g B(x) = -\frac{T_h}{m}$$

$$\text{Constraint: } \dot{B}(x, u) + \gamma B(x) = \frac{T_h}{m} (F_r(v) - u) + (v_0 - v) + \gamma(z - T_h v) \leq 0$$

Neglecting the effect of  $F_r$ , if we apply the maximum deceleration  $u = -c_d m g$ ,

$$\dot{B}(x, u) + \gamma B(x) = T_h c_d g + v_0 + \gamma z - (1 + T_h \gamma) v$$

When the value of  $v$  is big compared to the positive terms (determined by  $c_d$  and  $v_0$ ), the constraint might still not be satisfied.

To remedy this, we modify a CBF to include a term regarding a minimum braking distance required to decelerate from  $v$  to  $v_0$ :

$$B(x) = z - T_h v - \frac{1}{2} \frac{(v - v_0)^2}{c_d g}$$

Then we get  $\dot{B}(x, u) = \frac{1}{m} (T_h + \frac{v-v_0}{c_d g}) (F_r(v) - u) + (v_0 - v)$

Under maximum deceleration ( $u = -c_d m g$ ),  $\dot{B}(x, u) = \frac{1}{m} T_h F_r(v) + T_h c_d g > 0$

Therefore, the constraint is always feasible at any state, so  $B(s)$  is now a valid CBF.

## Step 4. Implement and tune the parameters.

- Once you got valid CLF and CBF, implement your controller and run the simulation. The details are explained in the next section.
- As a remark in Step 2 indicates, depending on your hyperparameters  $\lambda$  and  $\gamma$ , the feasible space of the CLF constraint and CBF constraint varies. (It might vanish under wrong parameters.) Also, the outcome will depend a lot on the parameters. Therefore, be sure to tune these values according to the desired goals. (More details in the Remark section.)



# How to code up?

Explained with Toy example:

CBF-CLF-QP for Adaptive Cruise Control (ACC)

# Steps

1. Create a class that inherit `CtrlAffineSys`.
2. Create a class function `defineSystem` and define your dynamics using the symbolic toolbox.
3. Create class functions `defineClf` and `defineCbf` and define your CLF and CBF in each function respectively using the same symbolic expressions.
4. To run the simulation or run the controller, create a class instance with parameters specified as a Matlab structure array, and use the built-in functions—`dynamics` and other controllers such as `ctrlCbfClfQp`, `ctrlClfQp`, etc.

# Step 1. Create a class that inherit `CtrlAffineSys`.

dynsys/@ACC/ACC.m

```
classdef ACC < CtrlAffineSys
    methods
        % Constructor
        function obj = ACC(params)
            obj = obj@CtrlAffineSys(params);
            %% Add your own code here.
        end
        % Custom function (rolling resistance)
        function Fr = getFr(obj, x)
            v = x(2);
            Fr = obj.params.f0 + obj.params.f1 * v + obj.params.f2 * v^2;
        end
    end
end
```

- Create a class directory (e.g. '@ACC' and define your class. To use the built-in functions of the library It should inherit the CtrlAffineSys class.
- The constructor is not necessary unless you need to add your own code.
- **params** is a structure array that contains all values of the model & control parameters.

# Step 2. Create a class function `defineSystem` and define your dynamics.

dynsys/@ACC/defineSystem.m

```
% Use the same input and output argument structure.
```

```
function [x, f, g] = defineSystem(~, params)
```

```
    syms p v z % states
```

```
    x = [p; v; z];
```

*state  $x$*

```
    f0 = params.f0;
```

```
    f1 = params.f1;
```

```
    f2 = params.f2;
```

```
    v0 = params.v0;
```

```
    m = params.m;
```

```
    Fr = f0 + f1 * v + f2 * v^2;
```

```
% Dynamics
```

```
    f = [v; -Fr/m; v0-v];
```

*$f(x)$*

```
    g = [0; 1/m; 0];
```


*$g(x)$*

```
end
```

- **params** is the same structure array which is in your class constructor input argument.
- Use symbolic expression to write the dynamics.
- Make sure to define the state as a column vector.
- Create each vector fields  $f(x)$  and  $g(x)$  separately.
- This function will allow your class instance to hold function\_handle `obj.f(x)` and `obj.g(x)` which are generated from these symbolic expressions.

# Step 3.1. Create class functions `defineClf` and define your CLF.

dynsys/@ACC/defineClf.m

```
% Use the same input and output argument structure.  
function clf = defineClf(obj, params, symbolic_state)  
    v = symbolic_state(2);  
    vd = params.vd; % desired velocity.  
  
    clf = (v - vd)^2;   $V(x)$   
end
```

- Consider `symbolic_state` as the same vector  $s$  in symbolic expression that you defined in your `defineSystem`.
- Define your CLF using this vector and parameters.
- After setting up your `defineClf`, the class instance will hold function handles `obj.clf(x)`, `obj.lf_clf(x)`, and `obj.lg_clf(x)` which are generated from these symbolic expressions.
- If you are not using CLF for your controller, this step is optional.

## Step 3.2. Create class functions `defineCbf` and define your CBF.

dynsys/@ACC/defineCbf.m

```
% Use the same input and output argument structure.  
function cbf = defineCbf(obj, params, symbolic_state)  
    v = symbolic_state(2);  
    z = symbolic_state(3);  
  
    v0 = params.v0;  
    T = params.T;  
    cd = params.cd;  
  
    cbf = z - T * v - 0.5 * (v0-v)^2 / (cd * params.g);  $\leftarrow B(x)$   
end
```

- Consider `symbolic_state` as the same vector  $s$  in symbolic expression that you defined in your `defineSystem`.
- Define your CBF using this vector and parameters.
- After setting up your `defineCbf`, the class instance will hold function\_handle `obj.cbf(x)`, `obj.lf_cbf(x)`, and `obj.lg_cbf(x)` which are generated from these symbolic expressions.
- If you are not using CBF for your controller, this step is optional.

# Step 4.1. Create a class instance with parameters specified as a Matlab structure array.

demos/run\_cbf\_clf\_simulation\_acc.m

```
dt = 0.02; sim_t = 20; % sampling rate and terminal time of the simulation.  
x0 = [0; 20; 100]; % Initial state
```

```
%% Parameters are from  
% Aaron Ames et al. Control Barrier Function based Quadratic Programs  
% with Application to Adaptive Cruise Control, CDC 2014, Table 1.
```

```
params.v0 = 14; % lead vehicle velocity.
```

```
params.vd = 24; % desired velocity.
```

```
% model parameters
```

```
params.m = 1650; params.g = 9.81;
```

```
params.f0 = 0.1; params.f1 = 5; params.f2 = 0.25;
```

```
params.ca = 0.3; params.cd = 0.3;
```

```
params.T = 1.8;
```

```
% input constraints
```

```
params.u_max = params.ca * params.m * params.g;
```

```
params.u_min = -params.cd * params.m * params.g;
```

```
% clf & cbf constraint parameters
```

```
params.clf.rate = 5; ←  $\lambda$ 
```

```
params.cbf.rate = 5; ←  $\gamma$ 
```

```
% weight parameters
```

```
params.weight.input = 2/params.m^2; ←  $H$ 
```

```
params.weight.slack = 2e-2; ←  $p$ 
```

```
%% Create class instance.
```

```
accSys = ACC(params);
```

- **params** should contain all necessary values for the model and the controller.
- The three essential values **params** should contain are
  - **params.clf.rate** ( $\lambda$  in CLF constraint)
  - **params.cbf.rate** ( $\gamma$  in CBF constraint)
  - **params.weight.slack** (weight for slack variable in the QP problem.)

Step 4.2. Use the built-in functions—dynamics and other controllers such as `ctrlCbfClfQp`, `ctrlClfQp` to simulate the system.

demos/run\_cbf\_clf\_simulation\_acc.m (continued)

```
odeFun = @accSys.dynamics;
controller = @accSys.ctrlCbfClfQp;
odeSolver = @ode45;

total_k = ceil(sim_t / dt);
x = x0; t = 0;
% initialize traces.
xs = zeros(total_k, 3); ts = zeros(total_k, 1);
us = zeros(total_k-1, 1); hs = zeros(total_k-1, 1); Vs = zeros(total_k-1, 1);
xs(1, :) = x0'; ts(1) = t;
% Run simulation.
for k = 1:total_k-1
    % Determine control input.
    Fr = accSys.getFr(x);
    [u, slack, h, V] = controller(x, Fr);  $u_{ref} = F_r$ 
    us(k, :) = u'; hs(k) = h; Vs(k) = V;

    % Run one time step propagation.
    [ts_temp, xs_temp] = odeSolver(@(t, s) odeFun(t, s, u), [t t+dt], x);
    x = xs_temp(end, :);

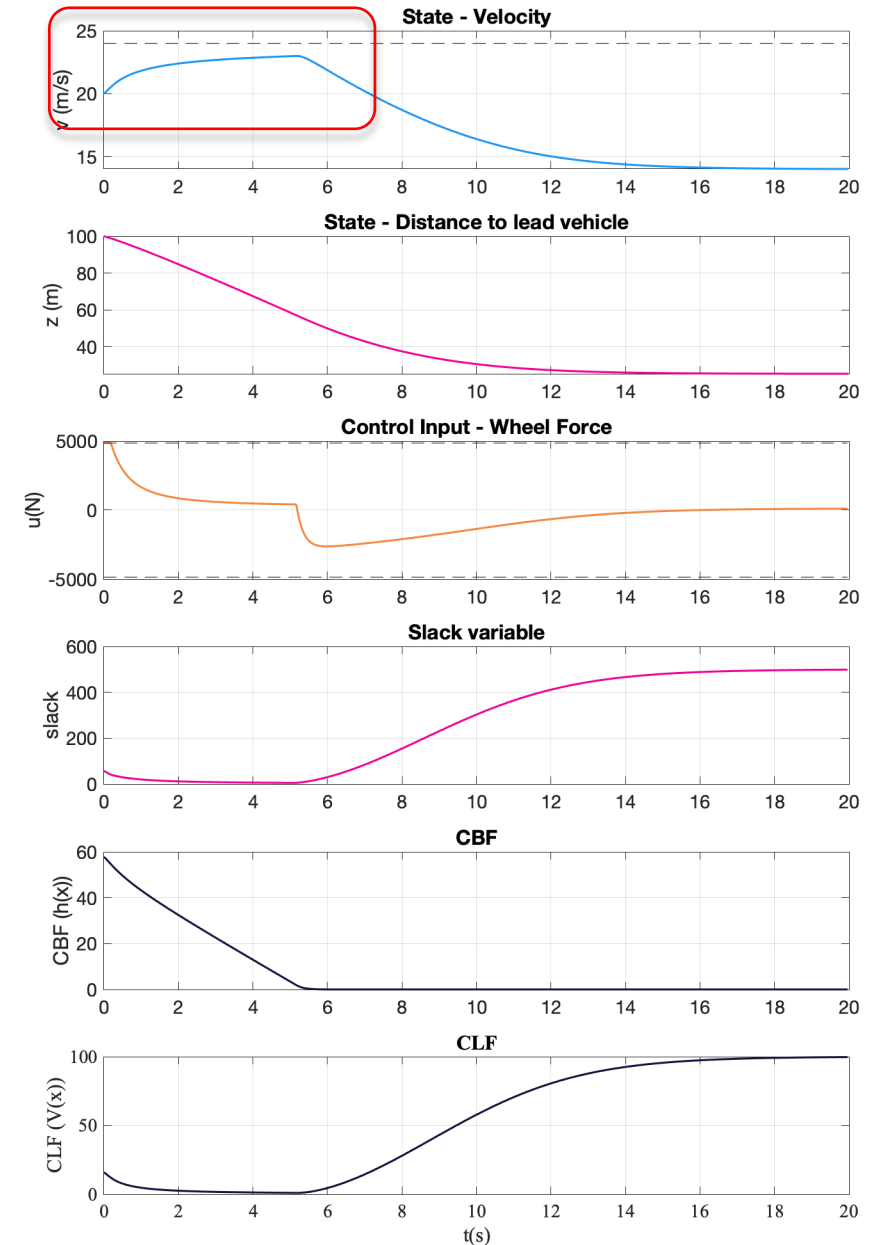
    xs(k+1, :) = x';
    ts(k+1) = ts_temp(end);
    t = t + dt;
end
```

- **dynamics** takes time, state, and control input of the system as input arguments and returns the value of  $\dot{x}$ , which can be used in the matlab ode functions for simulation.
- **ctrlCbfClfQp** and other built-in controller functions take state as input arguments, and outputs the feedback control. It can also take  $u_{ref}$  as the second input argument.



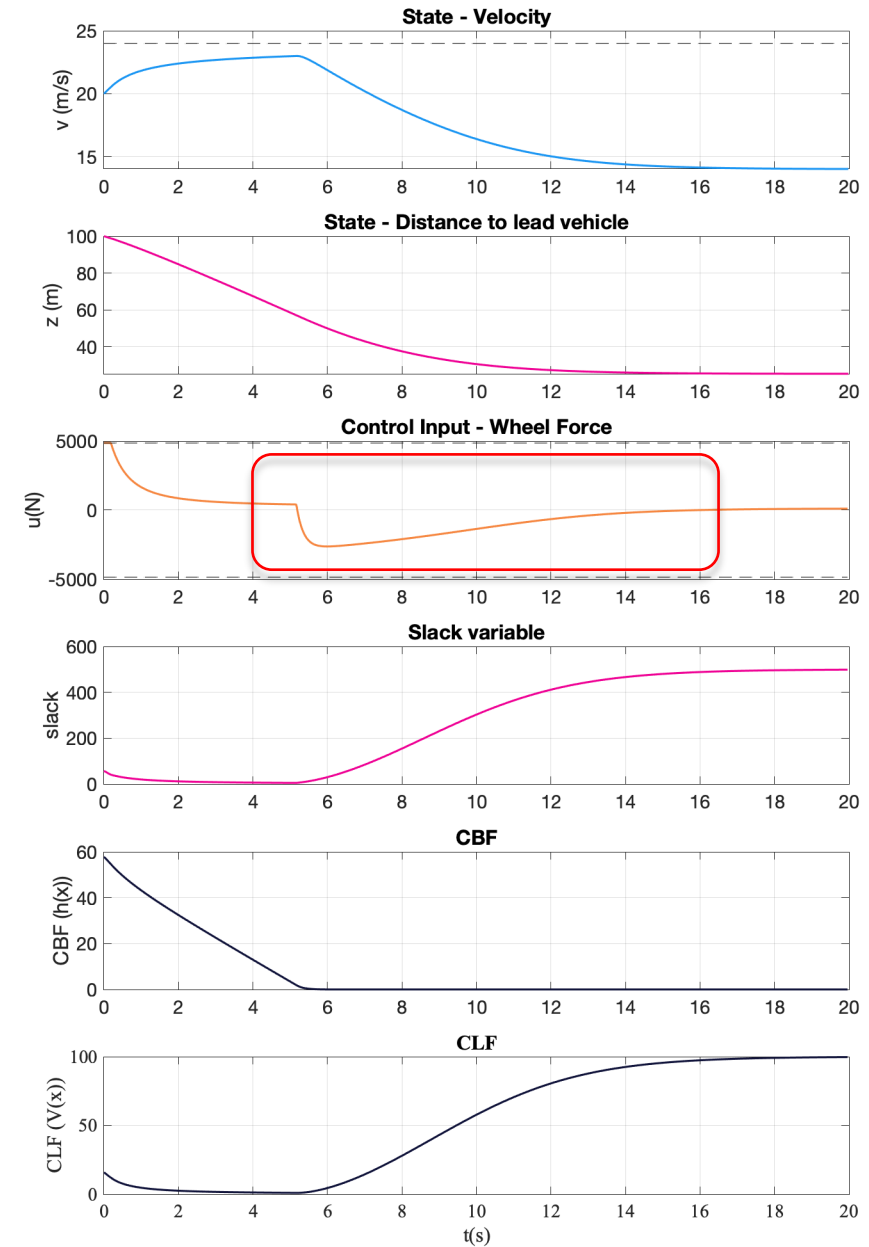
# Results

1. The vehicle tries to reach the target velocity at the beginning (~5.2 sec).
2. As vehicle gets closer to the lead vehicle, because of the CBF constraint, it starts to decelerate.
  - Note that the CLF constraint is actively relaxed to give priority to the CBF constraint.



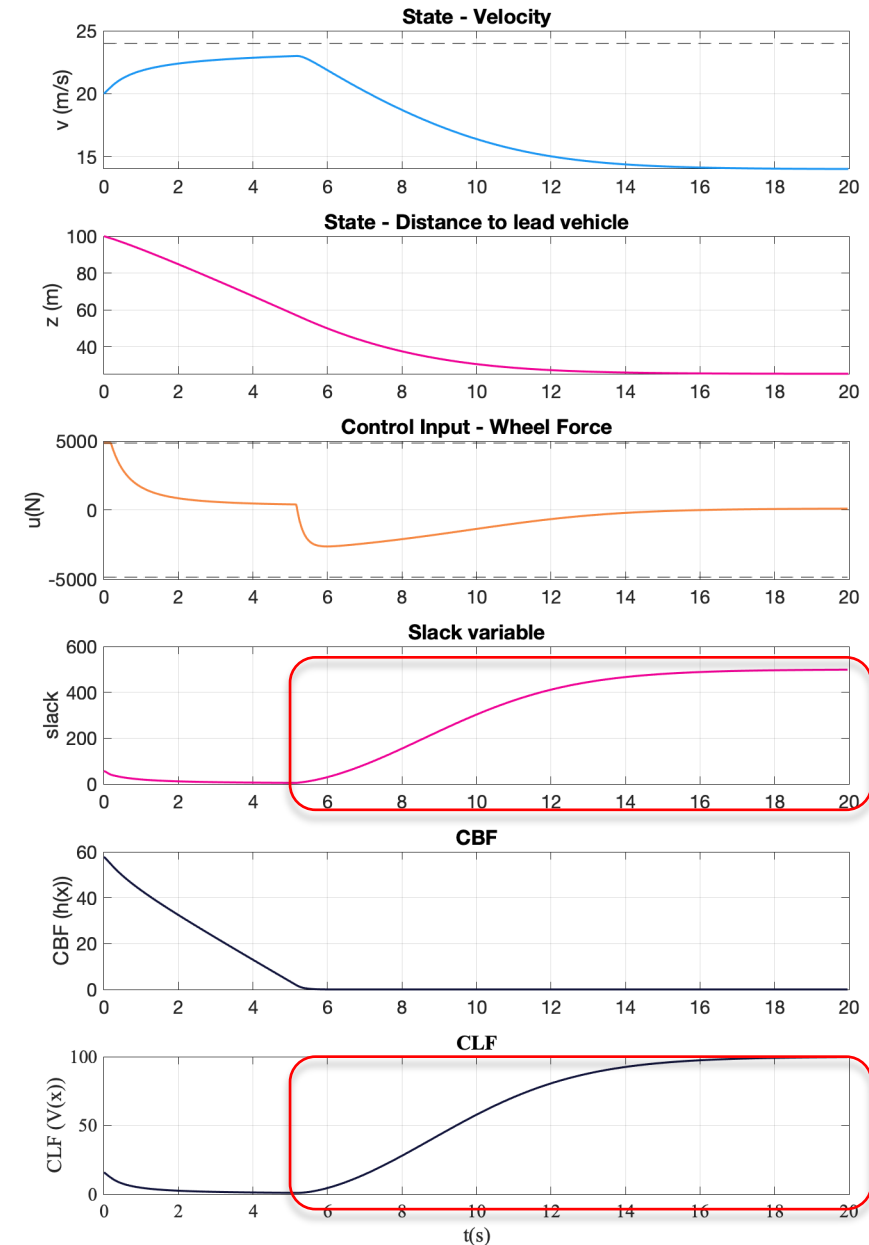
# Results

1. The vehicle tries to reach the target velocity at the beginning (~5.2 sec).
2. As vehicle gets closer to the lead vehicle, because of the CBF constraint, it starts to decelerate.
  - Note that the CLF constraint is actively relaxed to give priority to the CBF constraint.



# Results

1. The vehicle tries to reach the target velocity at the beginning (~5.2 sec).
2. As vehicle gets closer to the lead vehicle, because of the CBF constraint, it starts to decelerate.
  - Note that the CLF constraint is actively relaxed to give priority to the CBF constraint.



# Remarks

# Effects of hyperparameters

$$\underset{u}{\operatorname{argmin}} \quad (u - u_{ref})^T H(u - u_{ref}) + p\delta^2$$

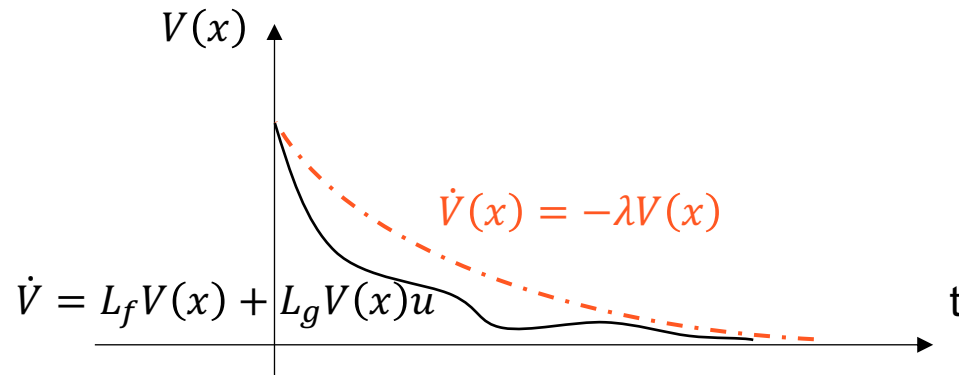
$u$ : control input

$\delta$ : slack variable

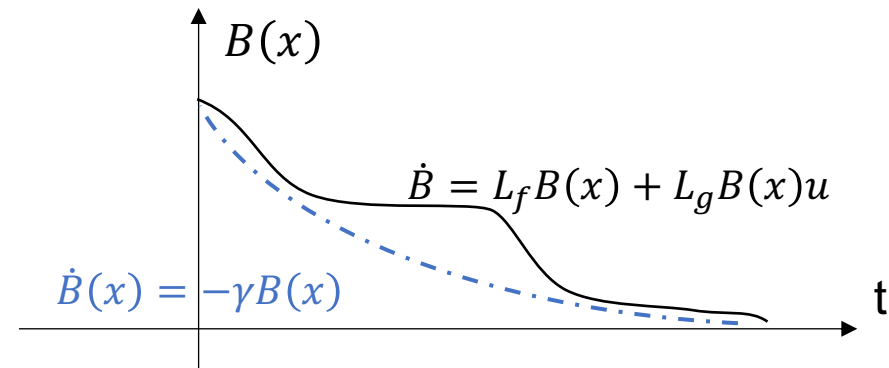
$$\begin{aligned} \text{subject to: } & L_f V(x) + L_g V(x)u + \lambda V(x) \leq \delta \\ & L_f B(x) + L_g B(x)u + \gamma B(x) \geq 0 \\ & u \in U \end{aligned}$$

- Selecting appropriate hyperparameters is very important. They directly affect the performance of the controller. More importantly, inappropriate value of  $\gamma$  can make your QP infeasible.
- Strictly speaking, the range of valid  $\lambda$  and  $\gamma$  should be considered in the design process of valid CLF and CBF. However, it is hard to be done in practice.
- In practice, people often design “simple” CLFs and CBFs and tune  $\lambda$  and  $\gamma$  based on the outcome.

# Effects of hyperparameters - $\lambda$ and $\gamma$



$$\dot{V}(x, u) + \lambda V(x) \leq 0, \text{ for } \exists u \in U, \lambda > 0$$



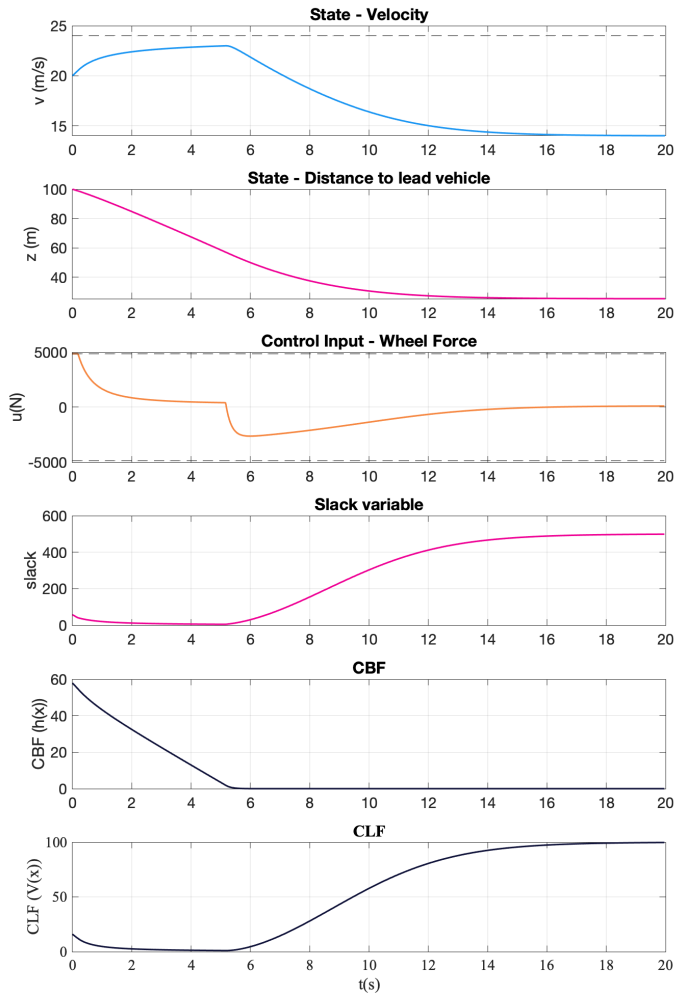
$$\dot{B}(x, u) + \gamma B(x) \geq 0, \text{ for } \exists u \in U, \gamma > 0$$

- As  $\lambda$  **gets bigger**, the CLF constraint imposes **stricter condition**. It requires  $V(x)$  to decay more quickly.
- On the other hand, for the CBF constraint, the **stricter condition** is imposed by  $\gamma$  **becoming smaller**. It requires  $B(x)$  to decay more slowly.

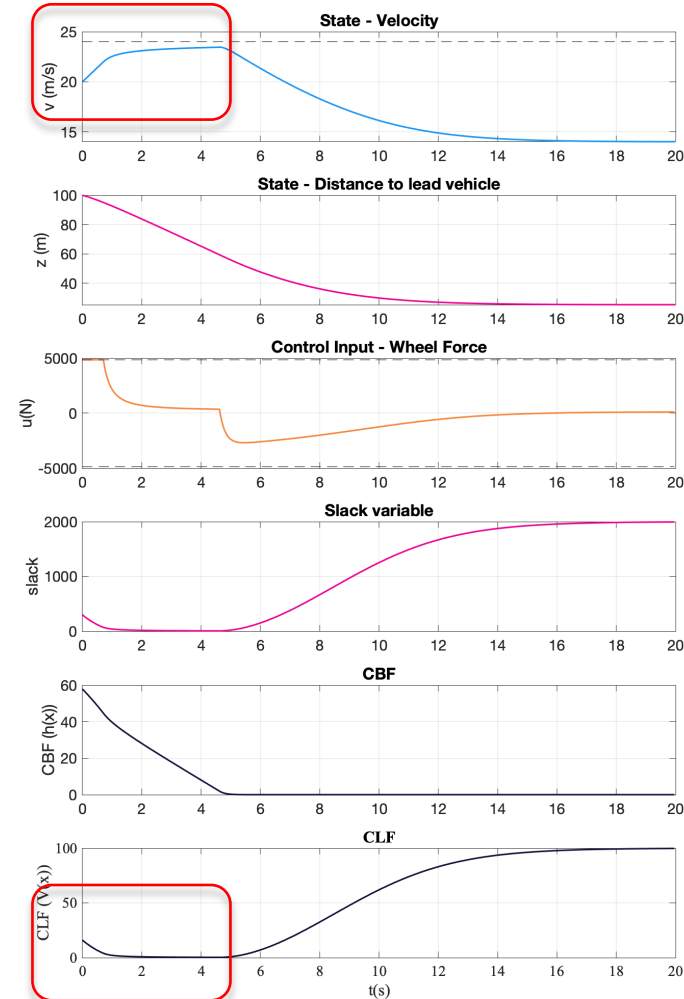
# Effects of hyperparameters - $\lambda$

ACC example:

$$\lambda = 5, \gamma = 5$$



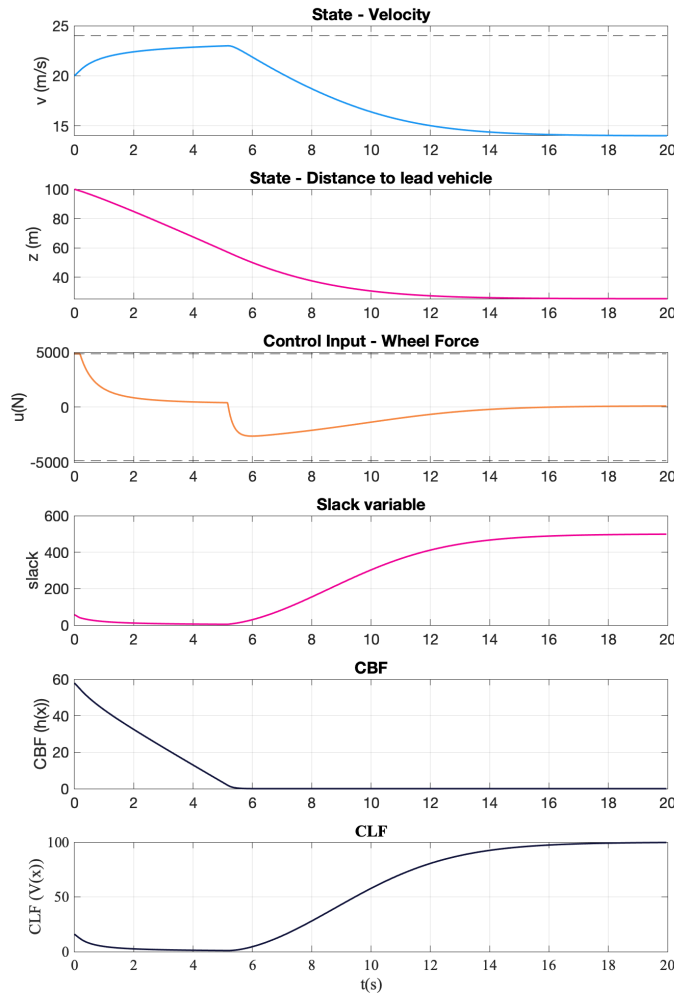
$$\lambda = 20, \gamma = 5$$



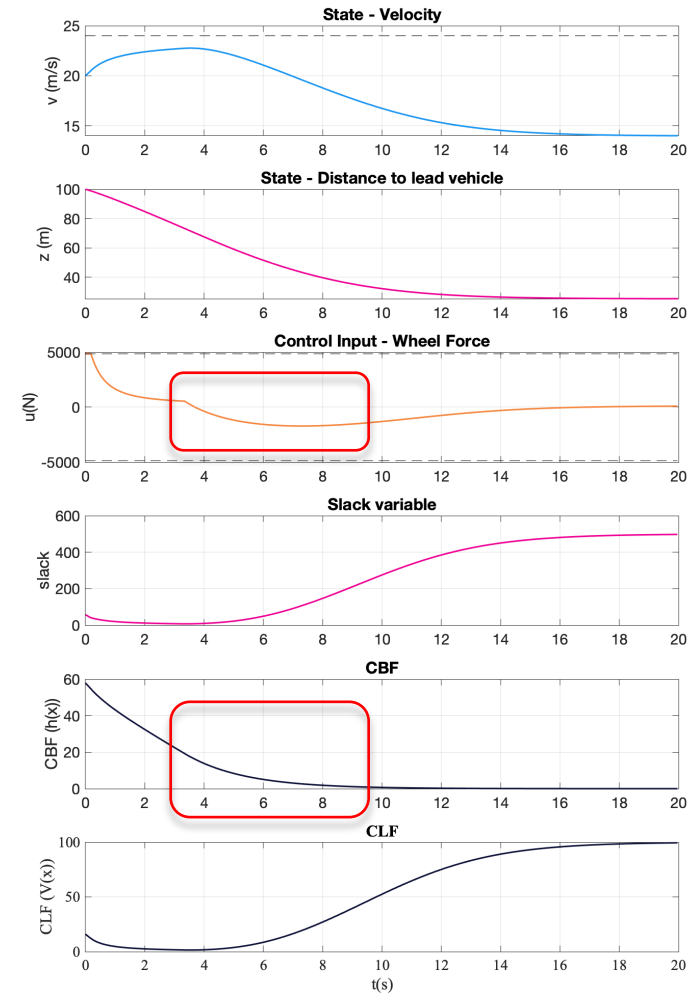
# Effects of hyperparameters - $\gamma$

ACC example:

$$\lambda = 5, \gamma = 5$$



$$\lambda = 5, \gamma = 0.5$$

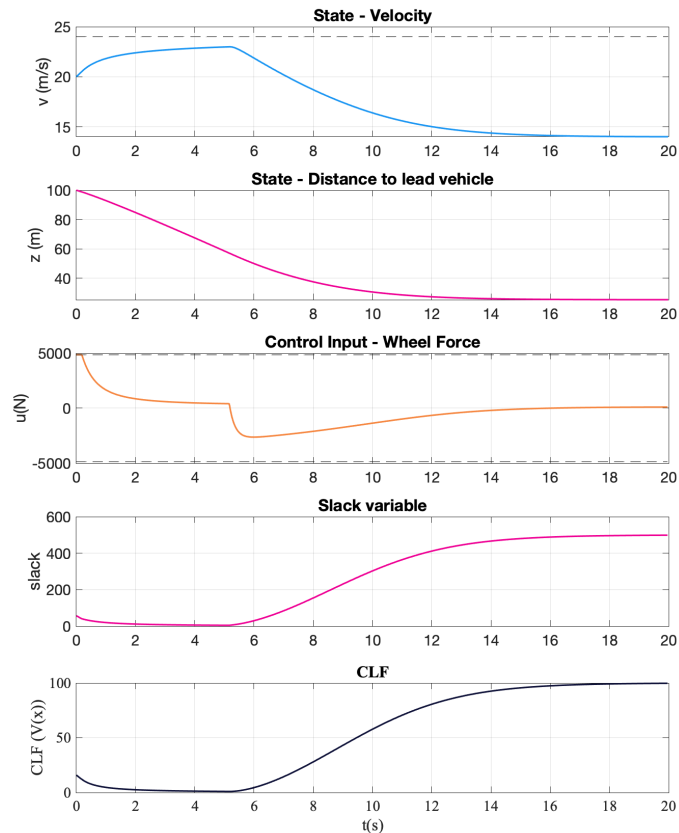




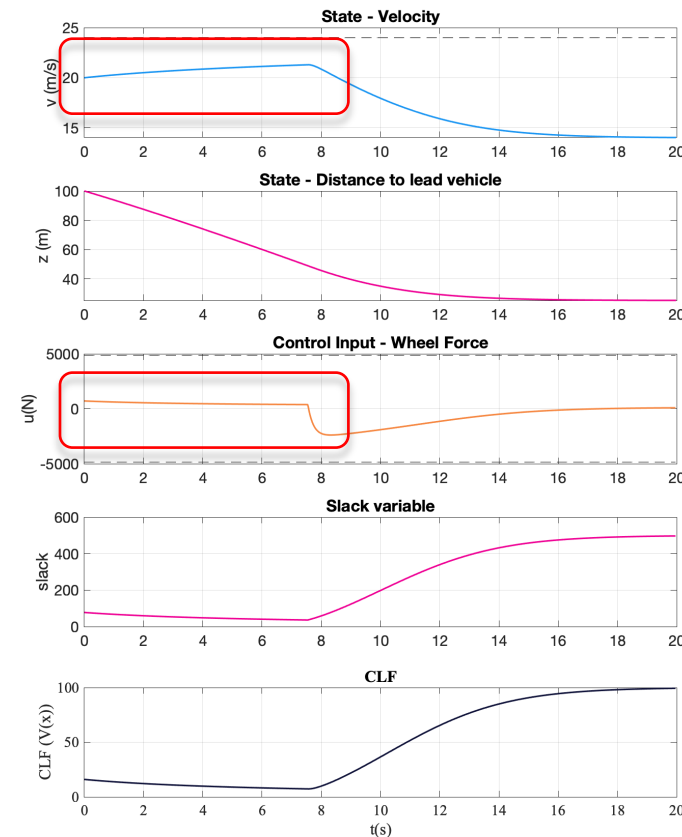
# Effects of hyperparameters — Slack cost weight ( $p$ )

Relaxation of the CLF constraint is regulated by the slack cost. Therefore, if the weight is too small, CLF constraint is “actively” relaxed to reduce the norm of the control, which is an undesired behavior.

ACC example:  $(\lambda = 5, \gamma = 5), p = 0.02$



$p = 0.001$



# Relative Degree

$$\sup_{u \in U} [L_f B(x) + L_g B(x)u] + \alpha(B(x)) \geq 0$$

- One implicit assumption applied when we defined the CBF before is that  $L_g B(x) \neq 0$ . We call this  $B(x)$  having **relative degree 1**.
- However, if  $L_g B(x)$  becomes 0,  $\dot{B}$  does not depend on the control  $u$ .
- For such cases, we differentiate  $B(x)$  with respect to time multiple times until finally the actuation term appears on the derivative. If this is possible, we can still regulate  $B(x)$  to stay positive, based on the concept called Exponential CBF\* or High Order CBF\*\*.

$$B^{(i)} = L_f^i B(x) \text{ for } i = 1, \dots, \gamma_b - 1, \quad B^{(\gamma_b)}(x, u) = L_f^{\gamma_b} B(x) + L_g L_f^{\gamma_b - 1} B(x)u$$

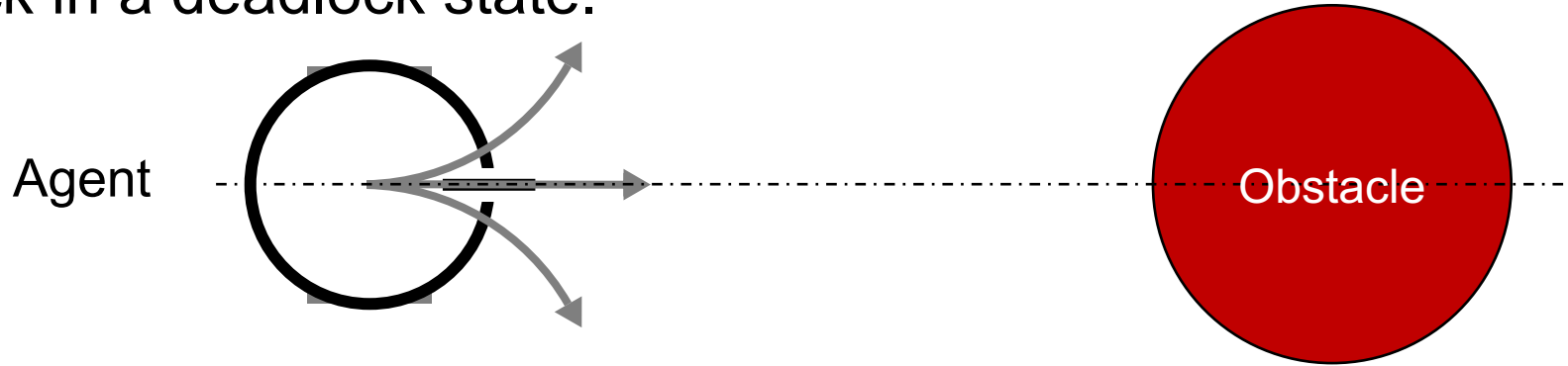
- $\gamma_b$  is called the **relative degree of  $B(x)$** .
- An example (2D double integrator) of relative degree 2 case can be found in the library.

\* Quan Nguyen and Koushil Sreenath. "Exponential control barrier functions for enforcing high relative-degree safety-critical constraints." ACC 2016

\*\* Wei Xiao and Calin Belta. "Control barrier functions for systems with high relative degree." CDC 2019

# Deadlock caused by symmetry

- For mobile robot applications, for specific conditions when the CBF constraint and the possible control actions are symmetric, the robot can get stuck in a deadlock state.



- For such cases, small random perturbations to the control input can help to avoid the deadlock.
- More details on [Grover et al. "Why Does Symmetry Cause Deadlocks?", IFAC, 2020](#)

# Other toy examples

- 2D Double Integrator reaching a target while avoiding an obstacle.
  - [run\\_cbf\\_clf\\_simulation\\_2DDI.m](#)
  - $u \in \mathbb{R}^2$
  - Relative degree 2.
- Dubins Car
  - [run\\_cbf\\_clf\\_simulation\\_dubins\\_car.m](#)
- Inverted pendulum
  - [run\\_clf\\_simulation\\_inverted\\_pendulum.m](#)
  - CLF-QP