

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

Library: [github link](#).

Jason Choi

Hybrid Robotics Lab

Introduction

- Control Lyapunov Function and Control Barrier Function based methods are effective in many safety-critical control problems. Each deals with safety in terms of stability and set invariance, respectively.
- CBF-CLF-Helper library 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 for example systems.
- 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 library.
 - User manual for the CBF-CLF-Helper library.
 - Built-in demonstrations.
 - Practical tips and issues on designing the controller.

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)
- Guideline to CBF-CLF-Helper
- Other examples
 - Multiple control input & High relative degree - 2D Double Integrator
 - CLF-QP
 - Dubins Car
 - Wheeled Inverted pendulum
 - Anderw Taylor – L4DC2020
- Higher relative degree example.
- Issues
 - Effects of hyperparameters λ, γ
 - Slack Cost
 - Relative Degree
 - Deadlock under Symmetry

Backgrounds

Dynamics – Control Affine System

Expression for dynamics of a general nonlinear controlled system:

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

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

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

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

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

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

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

Control Lyapunov Function (CLF)

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

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

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

Then $V(s)$ is a local **Control Lyapunov Function** and Ω_c is a region of attraction (ROA), i.e. **every state in Ω_c is asymptotically stabilizable to s_e .**

Derivative of $V(s)$ along the control affine dynamics

$$\begin{aligned}\dot{V}(s, u) &= \nabla V(s) \cdot \dot{s} \\ &= \nabla V(s) \cdot f(s) + \nabla V(s) \cdot g(s)u \\ &= L_f V(s) + L_g V(s)u\end{aligned}$$

$(L_p q(s) := \nabla q(s) \cdot p(s)$ is a Lie derivative operator used to make formulas concise.)

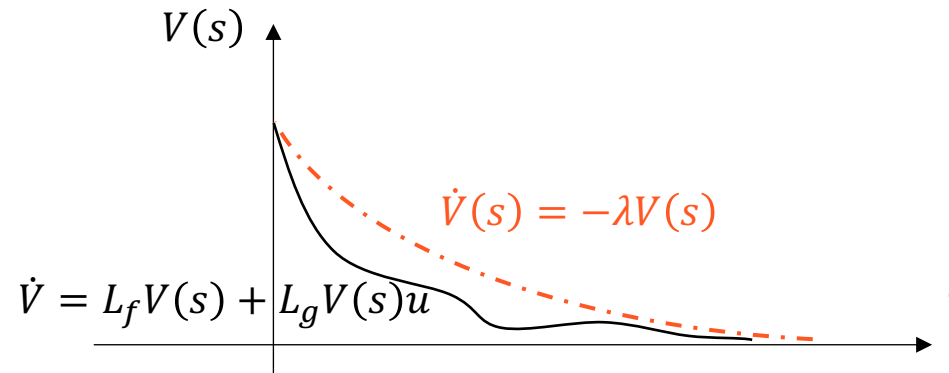
is affine in u .

Exponentially Stabilizing Control Lyapunov Function (CLF):

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

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

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



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

CLF-QP

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

- CLF constraint is relaxed with slack variable to guarantee feasibility of the problem.
- Input constraints should be linear.

Control Barrier Function (CBF)

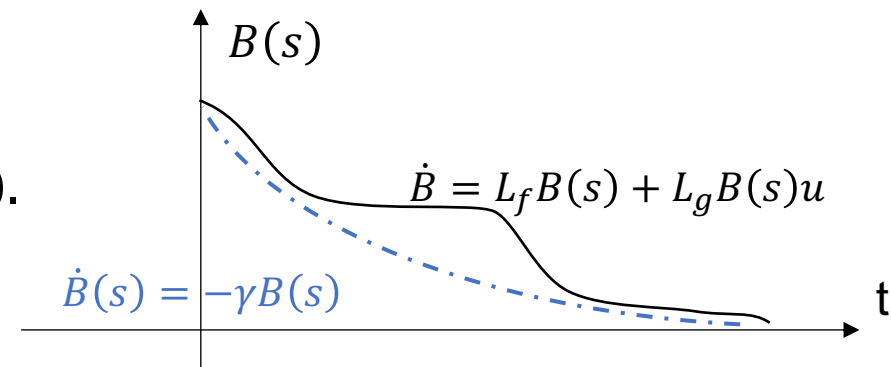
Let $B(s): \mathbb{R}^n \rightarrow \mathbb{R}$ a continuously differentiable function whose zero-superlevel set is \mathcal{C} and $\nabla B(s) \neq 0$ for all $s \in \partial\mathcal{C}$.

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

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

for all $s \in \mathcal{D}$, then $B(s)$ 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 γ is often used as $\alpha(\cdot)$; $\alpha(B(s)) = \gamma B(s)$. Then, γ serves as a decay rate of a lower bound of $B(s(t))$.



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

CBF-CLF-QP

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

- If $B(s)$ 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.

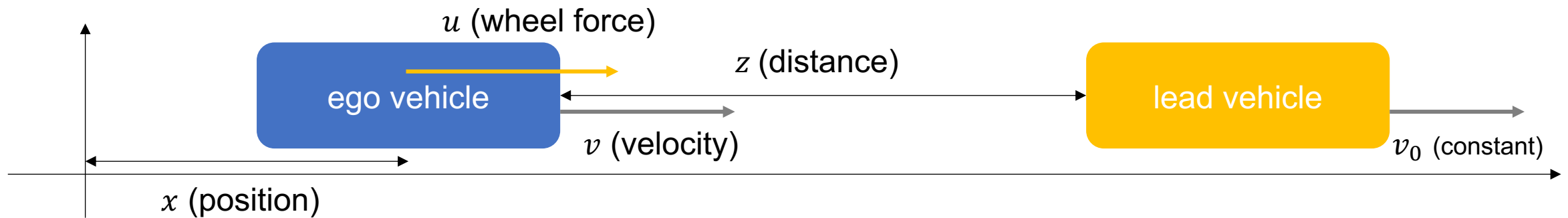
Design Steps

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,” in Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on. IEEE, 2014, pp. 6271–6278.

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



State

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

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 $s_e([\cdot \ v_d \ \cdot]^T)$ and positive everywhere else.

Let's try the most intuitive one: $V(s) = (v - v_d)^2$
 $\nabla V(s) = [0 \ 2(v - v_d) \ 0]$

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

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

The value of rolling resistance F_r is minute compared to u here.

If $v < v_d$, if we accelerate sufficiently large, the second term of RHS will be positive and we can make the constraint satisfied. The opposite case ($v > v_d$) holds same.

Therefore, we conclude that this is a valid CLF.

Remark: If the value of λ becomes too large, depending on the input constraints, the CLF might 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— $B(s) = z - T_h v$.

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

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

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

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

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

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 :

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

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

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

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

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 λ and γ , the feasible space of the CLF constraint and CBF constraint might vary. (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.

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, s)
            v = s(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 [s, f, g] = defineSystem(~, params)
```

```
    syms x v z % states
```

```
    s = [x; v; z]; ← state s
```

```
    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]; ←  $f(s)$ 
```


```
    g = [0; 1/m; 0]; ←  $g(s)$ 
```

```
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 $f(s)$ and $g(s)$ separately.
- This function will allow your class instance to hold function_handle `obj.f(s)` and `obj.g(s)` 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;   $V(s)$   
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_handle `obj.clf(s)`, `obj.lf_clf(s)`, and `obj.lg_clf(s)` 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);  $\leftarrow B(s)$   
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(s)`, `obj.lf_cbf(s)`, and `obj.lg_cbf(s)` 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.  
s0 = [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 = 10; ←  $\lambda$ 
```

```
params.cbf.rate = 1; ←  $\gamma$ 
```

```
% 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** (λ in CLF constraint)
 - **params.cbf.rate** (γ 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);
s = s0; t = 0;
% initialize traces.
ss = 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);
ss(1, :) = s0'; ts(1) = t;
% Run simulation.
for k = 1:total_k-1
    % Determine control input.
    Fr = accSys.getFr(s);
    [u, slack, h, V] = controller(s, Fr); ←  $u_{ref} = F_r$ 
    us(k, :) = u'; hs(k) = h; Vs(k) = V;

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

    ss(k+1, :) = s';
    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{S} , 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.