# Control Barrier Functions - A Simple Case Study

Recently, I have started flying a quadcopter our lab's flight arena, and I realised it would be good to implement some safety filters before I start flying the quad around using any new algorithms. I'll use this as an opportunity to demonstrate how control barrier functions can be used.

## Set up

The drone is running a pixhawk flight controller, running px4, and using off-board control from a Raspberry Pi. The raspi is mounted on the drone and sends commands over a UART connection. Since we have access to a vicon motion capture system, I'm feeding the position data into pixhawk, through the raspi. On the raspi, I'm running ROS, and im monitoring everything by SSH-ing into the raspi. Maybe someday I'll write up details on how I set up everything, but thats not the focus of this article.

The drone can be flown in many modes, but Im operating 3 modes, of increasing autonomy:

- *stabilised mode*: the drone uses its IMU to estimate how fast its tipping over, and trys to control it such that it hovers - since it doesnt have any position feedback, it drifts (sometimes pretty fast) and so the radio is used to manually control it and keep it in the arena.
- *position mode*: the drone is receiving position feedback from the vicon, and so is much better at maintain a hover. I can use the radio to push the drone around in different directions, but once I let go, it just sits there.
- *offboard mode*: in this mode, ROS has full control, and I can send either:
  - position commands (set a target set point)

- velocity commandas (set a target speed to maintain)
- motor commands (set the rpm of the individual motors)

In position and velocity command mode, internal PIDs are running to get the drone to the target. In fact, in motor command mode there also are PIDs that get the motors to spin at a target RPM.

Today, I flew the drone in position mode, without a safety filter. I just do a simple check to make sure I didnt accidentally send a position command thats beyond the boundaries of the netted area (an if-else kind of check) but apart from that, I just send the commands straight to the pixhawk.

The objective now is to send velocity commands, but also make sure that the robot doesnt keep crashing into the net if I forget to slow it down earlier.

*So what does a safety filter mean?*

We want to create a node between a commanded velocity, say from my algorithm, and the commanded velocity that the pixhawk gets, such that the drone wont hit the boundary of the net.

It takes as arguments the current drone position (`x, y, z`), and the commanded velocity (`vx, vy, vz`) and returns a new, filtered, velocity (`vx_safe, vy_safe, vz_safe`).

I will assume the drone arena is a cubiod, with known x, y, z bounds, denoted as `xmin, xmax`, `ymin, ymax`, `zmin, zmax`.

## Approach 1: If-Else

The most straight-forward, and frankly easiest, method is to do an if-else based safety filter. One implementation could look like this:

```
# the if-else filter
def filter(x, y, z, vx, vy, vz):

  if x <= xmin || x >= xmax:
    vx = 0

  if y <= ymin || y >= ymax:
    vy = 0

  if z <= zmin || z >= zmax:
    vz = 0

  return (vx, vy, vz)
```

which simplify says, if the drone is outside the bounds, it must stop - so it works right?

Well not really - there are at least two issues:

- suppose the drone does go outside the bounds - it will stop moving in that axis, but it wont be able to get back to the inside of the netted area
- suppose the drone was commanded to go towards the net with a very high speed - by the time it crosses the bounds, it might be too late, and the drone might not be able to slow down fast enough - and it will crash into the net.

The first issue can be handled by modifying the algorithm a bit:

```
# the modified if-else filter
def filter(x, y, z, vx, vy, vz):
```

```
if x <= xmin:
  vx = max(vx, 0.0)

if x >= xmax:
  vx = min(vx, 0.0)

# same for y and z...

return (vx, vy, vz)
```

The second issue is still challenging. You could introduce a maximum speed, but that may be too constricting.

Furthermore, how do you handle noise? Both from the vicon set up, and the drone just vibrating due to its PID. You could create a buffer, i.e. define your flight region to be smaller than it actually is, but thats kinda sad.

We want something that makes the drone slow down as it approaches the boundary, but only if it wants to keep going into the net…

## Approach 2: (Simple) Control Barrier Functions

Lets think about just the x-axis, and for now, just that we want to keep `x < xmax`.

The Control Barrier Function (CBF) approach is a really simple - we start by stating “the safety condtion”:

```
If x <= xmax, the system is *safe*
```

this gets translated into a mathematical function:

$$h(x) = x_{max} - x$$

and now we have the safety condition in a standard form:

```
If h(x) >= 0, the system is *safe*
```

Simply put, the function $h$ is simply a function that takes the robot's state, and returns a scalar number. There are a few more technical requirements, like differentiability and the relative degree, but these only appear for more complicated systems, so I'm going to skip the detials here for the sake of clarity.

In any case, we have now stated that we want to keep $h$ positive. But how does our control input affect $h$? We can use chain rule:

$$\dot{h} = \frac{dh}{dt} = \frac{dh}{dx}\frac{dx}{dt} = -1 \cdot v_x$$

which means that if $v_x$ is large and positive $\dot{h}$ is large and negative - if we are speeding towards $x = x_{max}$, the safety function/score $h$ is decreasing quickly - the intuition checks out.

But notice something really interesting: if we want $h > 0$ for all time, we and $h > 0$ at the initial time, all we need to do is to **enforce**

$$h = 0 \implies \dot{h} \geq 0$$

which in this case looks like:

$$x = x_{max} \implies -v_x \geq 0$$

or

$$x = x_{max} \implies v_x \leq 0$$

which turns out to be the same condition as our modified if-else filter! (for the curious, this is called Nagumo's condition)

Unfortunately, this is not a smooth controller, so we modify the condition slightly: we will **enforce** the condition:

$$\dot{h} \geq -\alpha h$$

where $\alpha > 0$ is any positive number. Notice that when $h = 0$, we have $\dot{h} \geq 0$, and so it is **mathematically impossible** for $h < 0$, and therefore the system will remain safe!!

What does this look like for our system?

$$-v_x \geq -\alpha(x_{max} - x)$$

and therefore

$$v_x \leq \alpha(x_{max} - x)$$

So what is $\alpha$? Its essentially the inverse of a time-constant. If we integrate $\dot{h} = -\alpha h$, we get $h(t) = h_0 e^{-\alpha t}$, and therefore, if $\alpha$ is large, the time-constant is small, and so the system is allowed to reach the boundary of the safe set much more rapidly. If $\alpha$ is small, the time constant is large, and so the system can only approach the boundary slowly. Ultimately, it is a tuning parameter, upto the roboticist to decide. I often start with $\alpha = 1$ if I'm unsure.

If we repeat the same analysis for the safety condition

$$x > x_{min}$$

by defining a new $h$ function as

$$h(x) = x - x_{min}$$

we arrive at the safety constraint

$$v_x \geq -\alpha(x - x_{min})$$

and so overall we have that

$$-\alpha(x - x_{min}) \leq v_x \leq \alpha(x_{max} - x)$$

Note: in general, combining multiple safety conditions into a single constraint is not possible - in these specific cases, it is.

And so the code can be implemented as follows:

```python
# the basic CBF
def filter(x, y, z, vx, vy, vz, alpha=1.0):

  # filter for the max side
  vx = min(vx, alpha * ( xmax - x) )

  # filter for the min side
  vx = max(-alpha * (x-xmin), vx)

  # same for y and z...

  return (vx, vy, vz)
```

## Approach 3: (Real) Control Barrier Functions

For completeness, I will show you how this works in general, and why CBFs are so powerful. The previous section showed that its really quite intuitive, but here I will show you why we like it.

Consider a non-linear control affine dynamical system. Thats just fancy speak for consider a system that behaves like

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

where $x$ is the n-dimensional state vector, $u$ is the m-dimensional control vector, and $f, g$ are functions that only depend on the current state of the system. While it may seem like quite a restriction to assume the above form, many (many) systems do in fact follow this general form, perhaps with input bounds. For now, we ignore the input bounds.

The safety condition is still

$$h(x) \geq 0$$

and so we have to enforce

$$\dot{h}(x, u) \geq -\alpha h(x)$$

which can be written out as

$$\frac{dh}{dx}(f(x) + g(x)u) \geq -\alpha h(x)$$

or

$$\frac{dh}{dx}f(x) + \frac{dh}{dx}g(x)u \geq -\alpha h(x)$$

which is infact a linear constraint on $u$: it looks like

$$a^T u \geq b$$

where $a = \frac{dh}{dx} g(x)$ and $b = -\alpha h(x) - \frac{dh}{dx} f(x)$. $a and b$ only depend on $x$!

Since this is a linear constraint on $u$, we can solve for a good $u$ quite easily. We must solve the following optimization problem:

$$\begin{array}{ll} \underset{u}{\text{minimize}} & ||(u - u_{des})||^2 \\ \text{subject to} & \frac{dh}{dx} f(x) + \frac{dh}{dx} g(x) u \geq -\alpha h(x) \end{array}$$

which is simply a quadratic program - we know there is exactly one minimum point to this problem, and there exist many methods to find it - above we simply found the analytic solution, but in general I use tools like `convex.jl, cvxpy` or `osqp`. In `cvxpy` the filter can be implemented as

```
# the CVXPY method
import cvxpy as cp

# where m is the number of control inputs
# where h(x) defines the safety score
# where f(x) and g(x) define the dynamics
# where Lfh(x) = dhdx(x) * f(x)
# where Lgh(x) = dhdx(x) * g(x)

def filter(state, u_des, alpha=1.0):

  u = Variable(m)

  obj = cp.sum_squares(u - u_des)
```

```
    cons = [Lfh(x) + Lgh(x) @ u >= -alpha * h(x)]
    # the @ symbol is used to mark a linear-algebra type product. In this case a inner/dot product

    prob = cp.Problem(cp.Minimize(obj), cons)

    prob.solve()

    return u.value
```

The incredible power of this result is that we can find provably safe controllers for non-linear systems, and solve them easily! That was pretty difficult before CBFs came around.

## Approach 4: Roboust CBFs

Finally, we want to consider some robustness into the problem. There are a number of different ways to achieve this (see Input-to-State-Safe CBFS, Tunable ISSF CBFs, robust CBFs and adaptive CBFs), but the simplest is to figure out how much the quadrotor jumps around.

From few experiments, it seems that the position jumps around by upto approximately $\Delta = 5$ cm, so we will simply say that safety is defined by

```
If xmax-x >= \Delta, the system is *safe*
```

which is implemented as, ensure:

$$\dot{h} \geq -\alpha(h - \Delta)$$

or

```
# the basic CBF, with some robustness
def filter(x, y, z, vx, vy, vz, alpha=1.0, Delta=0.05):

  # filter for the max side
  vx = min(vx, alpha * ( xmax - x - Delta) )

  # filter for the min side
  vx = max(-alpha * (x-xmin - Delta), vx)

  # same for y and z...

  return (vx, vy, vz)
```

I hope this actually works…

By Devansh Agrawal.