Improving Joystick Deadbands for More Precise Control

By: Collin Ostrowski

I have seen many different threads about deadbands and joysticks but many just use a simple IF statement for their deadband which is not the best way to go.

TLDR: Basic deadband code causes the Value of the joystick to jump greatly once the joystick has passed it's deadband range. The best idea is to scale the joysticks output to allow all values to be hit to improve drivability. Just use the second larger section of code to fix this.

For those of you who don't know a deadband is a small area that is applied to a joysticks range and when it is in that area its value is set to 0. The reason for this is because joysticks are not perfect they will not always return to exactly 0. So without a deadband when you take your hand or finger off a joystick your robot might continue to slowly roll forward or an arm might move without you wanting it to. This is unacceptable for safety concerns as well as driveability.

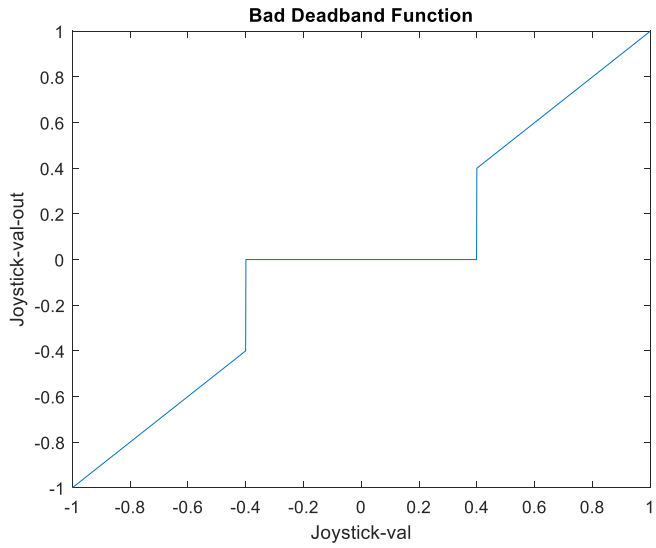
For Example, the usual code used for a deadband is:

```
//Joystick_val is the value the joystick is outputing
//Joystick_calc is the value after the code was applied
//Deadband is a double that is the positive and negative range of the
deadband

if( abs(Joystick_val) < Deadband ){
      Joystick_calc = 0;
}
else {
      Joystick_calc = joystick_val;
}
```

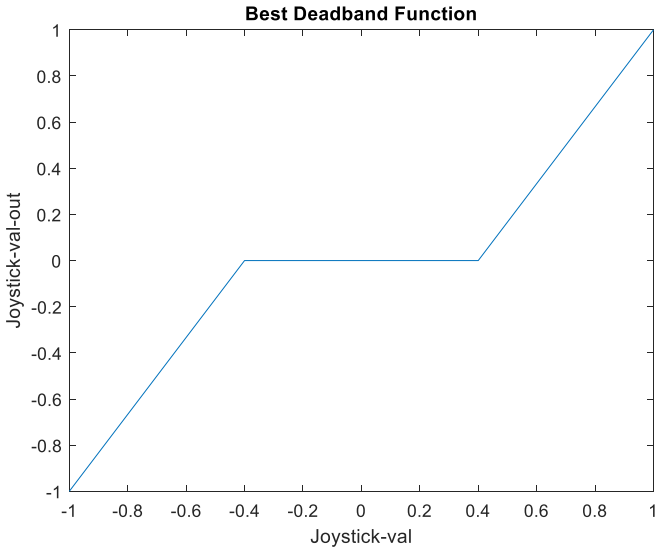
That usual IF statement work fine for deadbands as long as the deadband is very small, however, depending on how poorly manufactured or damaged the joystick is your team may choose for a larger deadband. Which causes the code to make the jump higher and higher.

This graph shows the code above plotted where the X axis is the value from the joystick and the Y axis is the output of the code. The deadband value is very extreme at 0.4 meaning the joystick has to be 40% percent before it will output a value that is not 0.



As you can see the from the graph once the value passes the deadband it jumps from 0 to the value of the deadband. Which can be extremely detrimental for controlling many different mechanisms. For example, let's say your cars throttle can only be at 0 or a range from 40% to 100%. It would be extremely difficult to drive the car slowly with any measure of control or consistency. If you attempted to inch your car forward which is very common for lining up your robot in FRC it would be almost impossible without having to fight against your own control.

Ideally the function would allow any output percentage while having a deadband of any size. With the same deadband of 40% the ideal functions graph would look like:



In this case when the joystick passes the deadband it won't immediately jump to 40% even though there is a deadband of 40%. This is great for when you need to inch the robot forward or for any mechanism that needs only a tiny bit of power to move just a tiny bit.

For the more advanced teams feeding this into a control loop would also yield more precise movements as well.

The way this works is by adjusting the slope to make it so that 100% of the joystick is 100% and at the deadband value is 0%. At very large deadbands like above you can see that the slope is much greater than before. This will cause the joystick to be a bit more sensitive than before, but it allows small adjustments of the robot.

The best code a mentor and I developed is:

```
//Joystick_val is the value the joystick is outputting
//Joystick_calc is the value after the code was applied
//Deadband is a double that is the positive and negative range of the deadband

if( abs(JoystickValue) < Deadband ){
	Joystick_calc = 0;
else{
	Joystick_calc = (1 / (1 - Deadband)) * (Joystick_val + (-Sign(Joystick_val) * Deadband));
}
```

Essentially it is based off of the Point slope form $Y = M(X-X_0)+Y_0$. However, Y_0 is not used because we want the value to start at 0. X_0 is the deadband adjusted to be positive or negative depending on whether the joystick is in the positive or negative direction. The slope was based off of the slope formula where slope = $(Y_2 - Y_1) / (X_2 – X_1)$. Y_2 is equal to 1 for achieving 100% on the output of the function and X_2 is 1 for an input of 100%. Y_1 is 0 because X_1 is equal to the deadband, because at the deadband we want the function output to be 0% and at 100% we want the function output to be 100%.

The only problem with this code is that in a few programming languages the Sign() function is not naturally available. The function will output a -1 if the input value is less than 0. It will output 1 if the input value is greater than 0 and 0 if the input value is 0. The quick fix for this is to either make the function or by replacing the sign function with:

```
(input_val > 0) - (input_val < 0)
```

It uses the fact that TRUE is equal to 1 and FALSE is equal to 0.

Appendix

The code can be easily modified to add extra abilities. If your team likes to have a creep mode you can replace the 1 with a MaxSpeed variable to set the max percentage speed allowed in that mode for example it would look like:

```
Joystick_calc = (MaxSpeed / (1 - Deadband)) * (Joystick_val + (-Sign(Joystick_val) * Deadband));
```

So say in your creep mode you want 50% to be the max speed allowed. What you can do in this case is have MaxSpeed equal to 0.5 so when the joystick is at 100% the robots set speed will only be at 50%.

If you also want your robot to reach its max speed before the joystick reaches 100% then you can change the lower 1 to the desired percentage. Maybe because your joystick is broken at 100 or cannot ever reach a value of 1. For example lets say you want the joysticks 90% to be the robots 100% then you can do:

```
Joystick_calc = (1 / (0.9 - Deadband)) * (Joystick_val + (-Sign(Joystick_val) * Deadband));
```