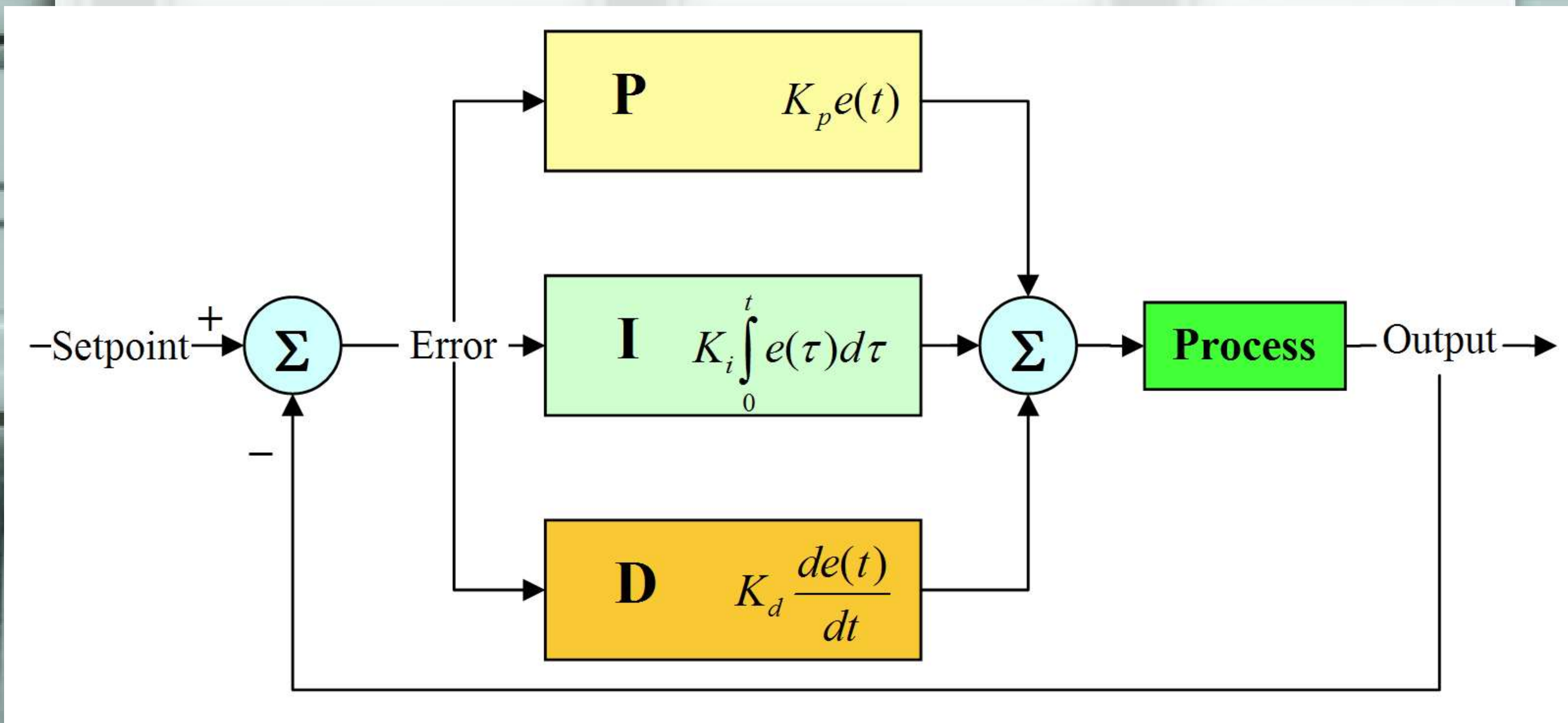
What is a PID controller

- A feedback loop control system
  - It looks at what happened in the past to help adjust its output in the future
- You give it a target, and the current position. It tries to get the error to zero between the target and your position
- Proportional – error * fixed amount (Kp)
- Integral – sum of error * Ki
- Derivative – Try to slowdown as the rate of change increases to prevent overshoot * Kd

What is a PID controller

- Target = set point
  - Where the controller is trying to get to
- Error = current reading – set point
  - The difference between the desired location and the current location
- Output = the amount we want to affect the control variable to try and reduce the error

What is a PID controller

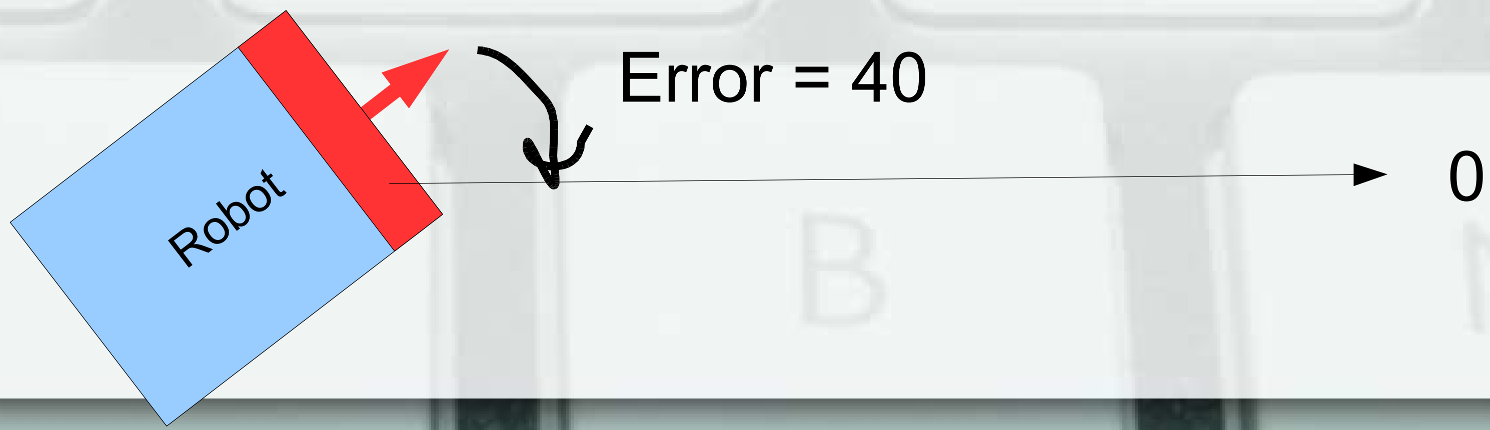


Our Problem

- We want the robot to drive forward in a certain direction
- We need to turn and then move

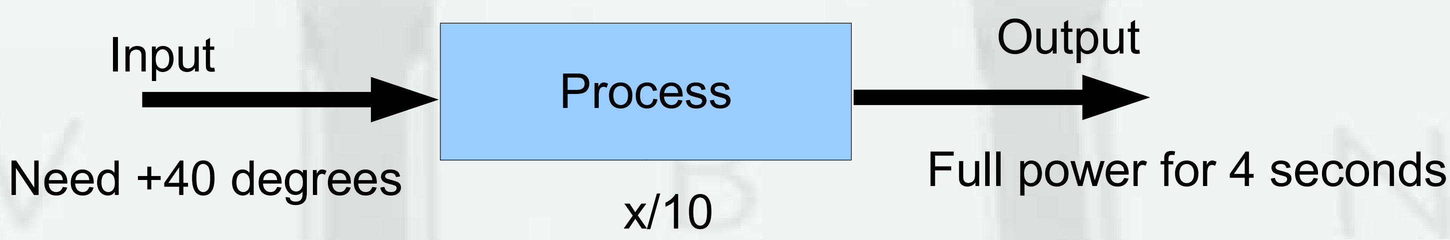
First we turn

- We have an error in our current angle
- “Forward” will always be relatively 0 degrees regardless of what direction that is as the direction we want to move is at 0 error
- If we make “forward” 90 degrees then we are trying to create an error of 90 in our system – which is weird
- i.e. the front of the robot is forward, not compass North



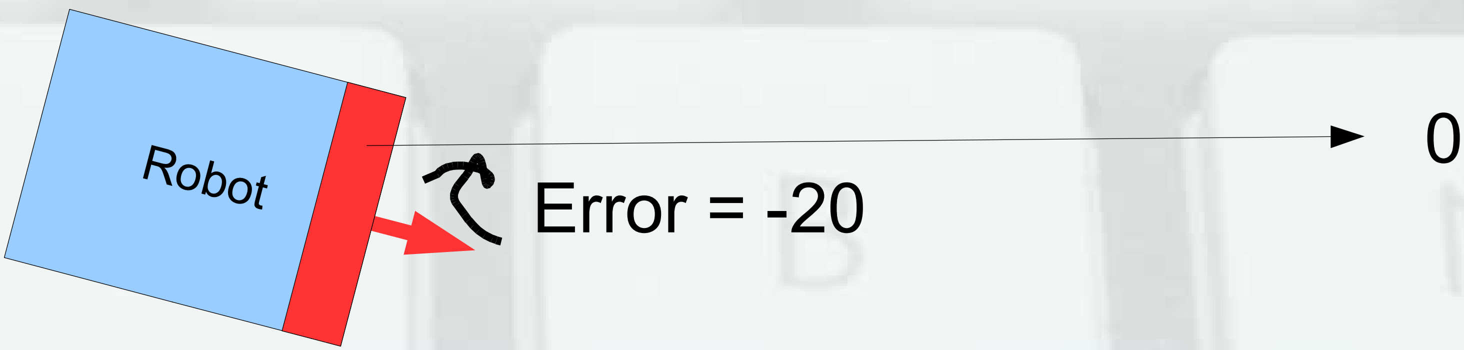
Open loop controller

- An open loop controller has no looping of output to input (open)
- If we move 10 degrees per second and need to move 40 degrees then we need full power for 4 seconds: 10 * 4 = 40



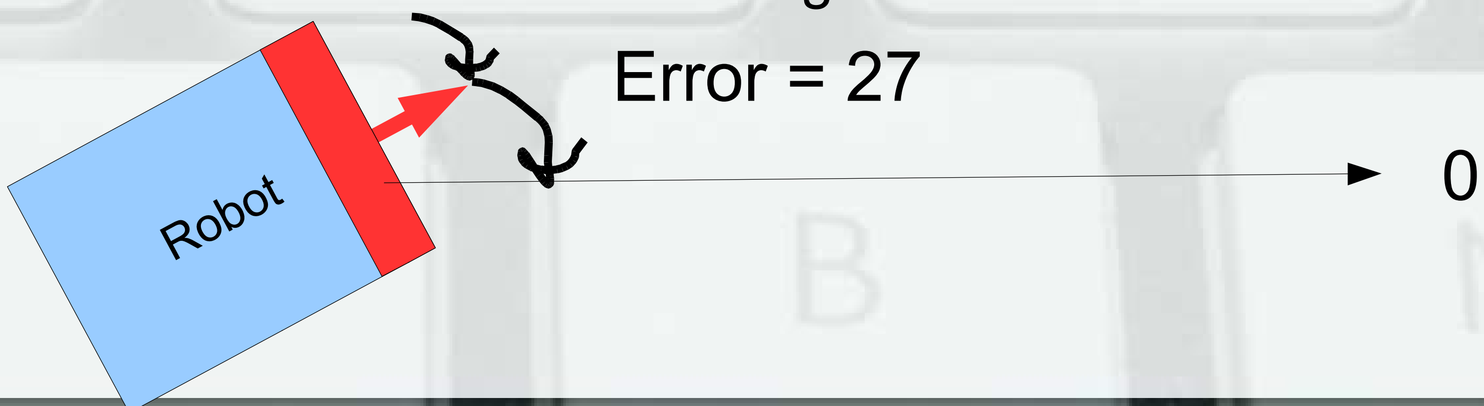
Open loop turn result

- What happened?
- I guess we moved 15 degrees per second, so we have an error of -20 (i.e. we need to turn -20 degrees)
- We could fix this by initially moving for 2.6 seconds from our original position: 2.6 * 15 = 40



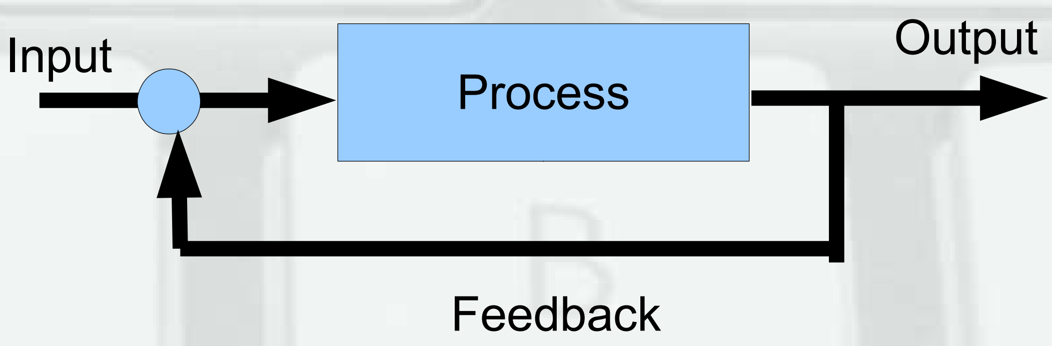
Open loop turn result 2

- We still have an error of 27. WTF?
- The output signal to movement mapping isn't 1:1
- The robot turns 5-15 degrees per second
    - exaggerated, but real systems are like this
- i.e. if we apply a power of 12V to the motor we move somewhere between 5-15 degrees



Closed loop controller

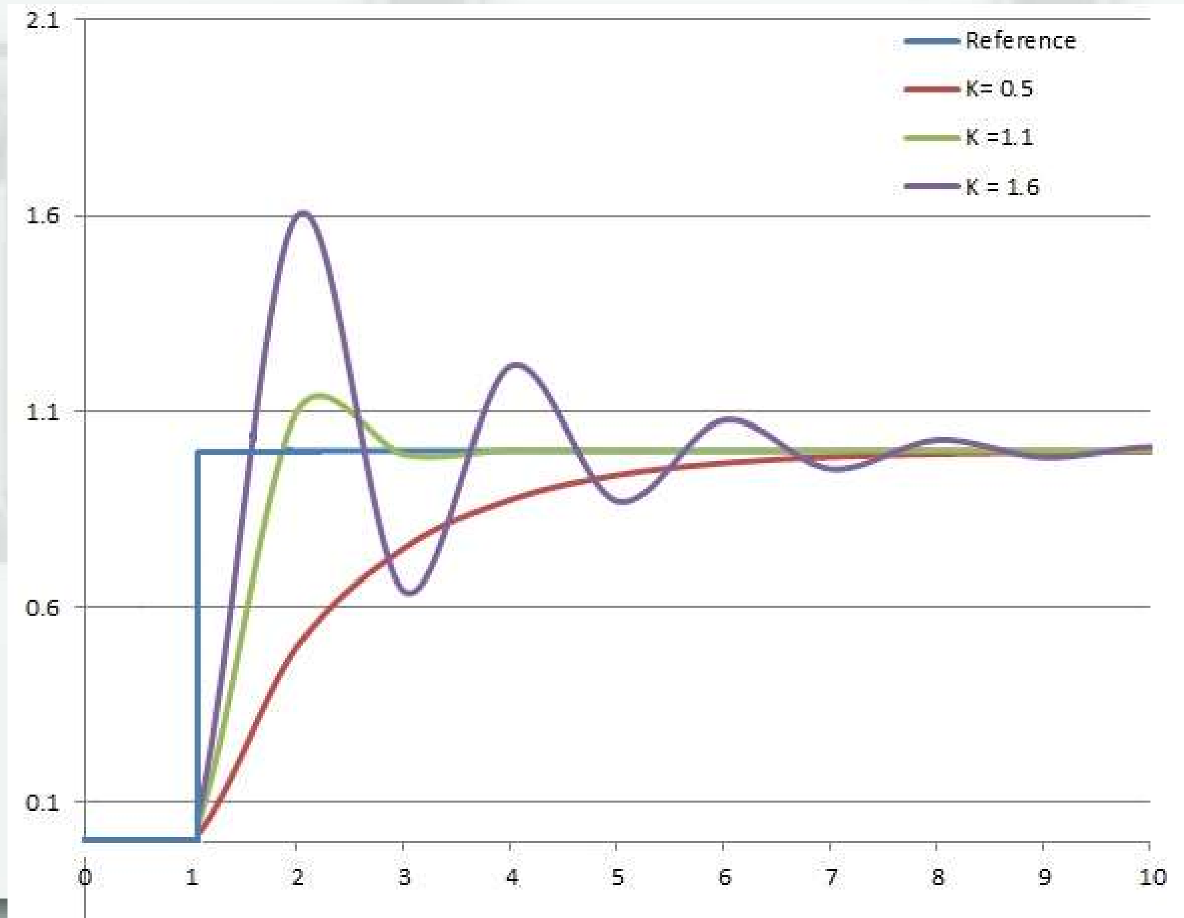
- Feedback the error we are seeing in the system and keep changing our output to match
- Rather than “move for 4 seconds” tell it to move for 0.1 seconds and then check again



Proportional Controller

- Error term * Kp
- i.e. angle error * Kp (0.5)
  - 40 * 0.5 = 20 power to motor
  - 32 * 0.5 = 16 power to motor
  - 28 * 0.5 = 14 power to motor
  - 24 * 0.5 = 12 power to motor
  - Etc... until we get to 0
- We're getting their... slowly...

Proportional Controller



Being close enough

- 0.005 degrees * 0.5 = 0.0025 power
- 0.004 degrees * 0.5 = 0.002 power
- 0.004 degrees * 0.5 = 0.002 power
- 0.004 degrees * 0.5 = 0.002 power
- … 0.002 power isn't enough to move robot ...
- We instead say “if you're within +/- 0.1 degrees then exit controller” (return 0 power)

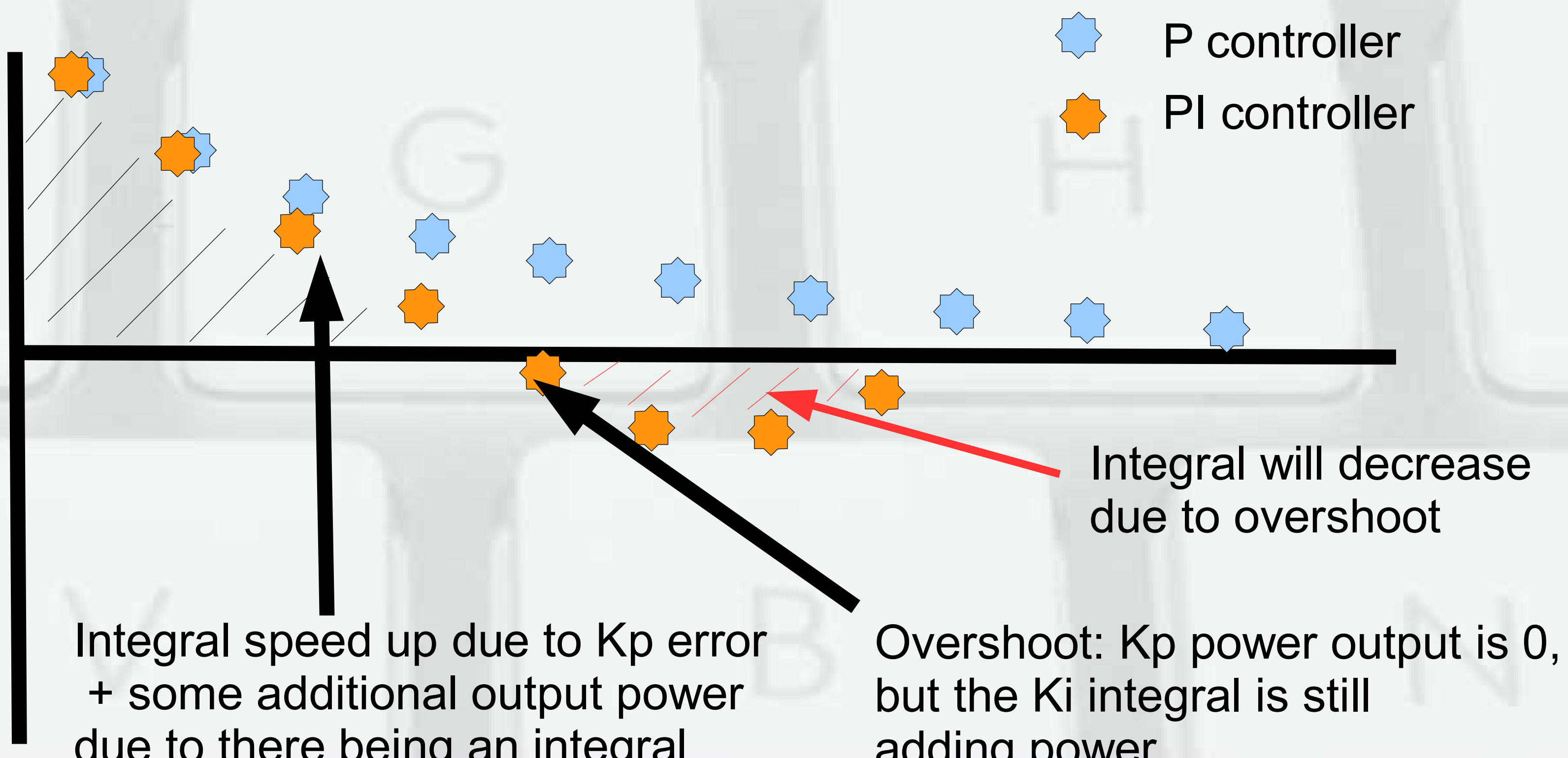
Proportional Controller

- Error term * Kp
- i.e. angle error * Kp (0.5)
  - 40 * 0.5 = 20 power to motor
  - 32 * 0.5 = 16 power to motor
  - 28 * 0.5 = 14 power to motor
  - 24 * 0.5 = 12 power to motor
  - Etc... until we get to 0
- We're getting their... slowly...

We're far away, but slowing down

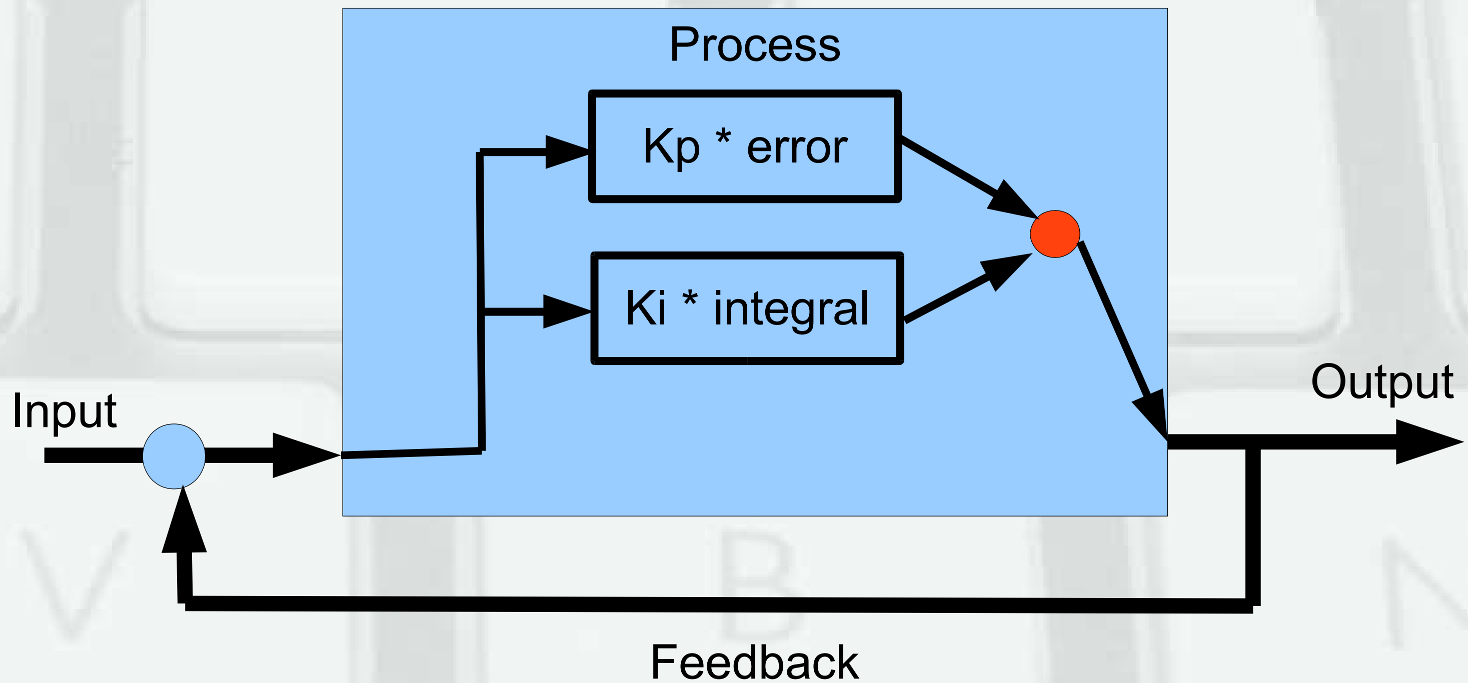
Integral Control

- Integrate the error and add more output

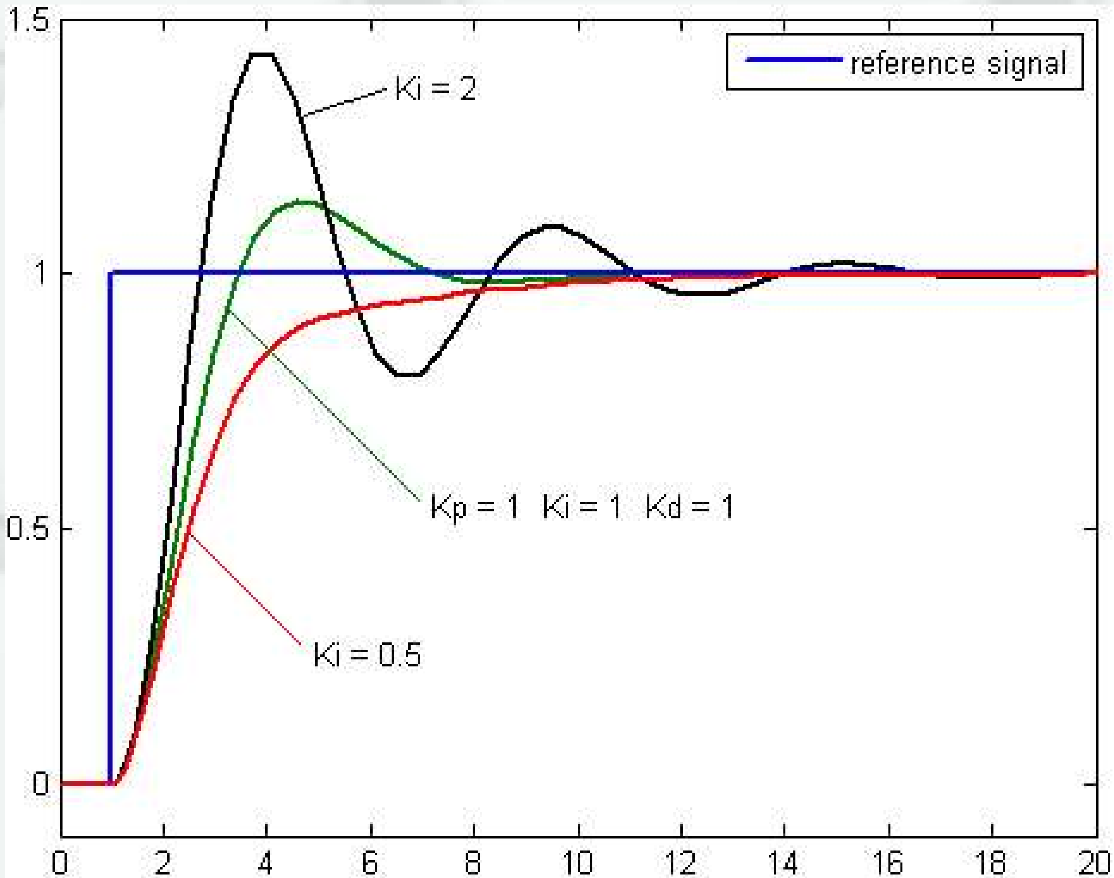


Integral Control

- Add the output from the Kp and the Ki terms together to get the total



Integral Control



Offset error – P only

- Some of Rob's donut get stuck in the left wheel so the robot keeps moving 0.5 degrees to the left
- P controller: we are currently 0.5 degrees off course
  - Error 0.5 * Kp (2) = 1 power, which moves 0.5 degrees right
  - 0.5 degrees right (from P controller) + -0.5 to the left due to donut = 0 degrees movement
  - So we are still pointing 0.5 degrees off course

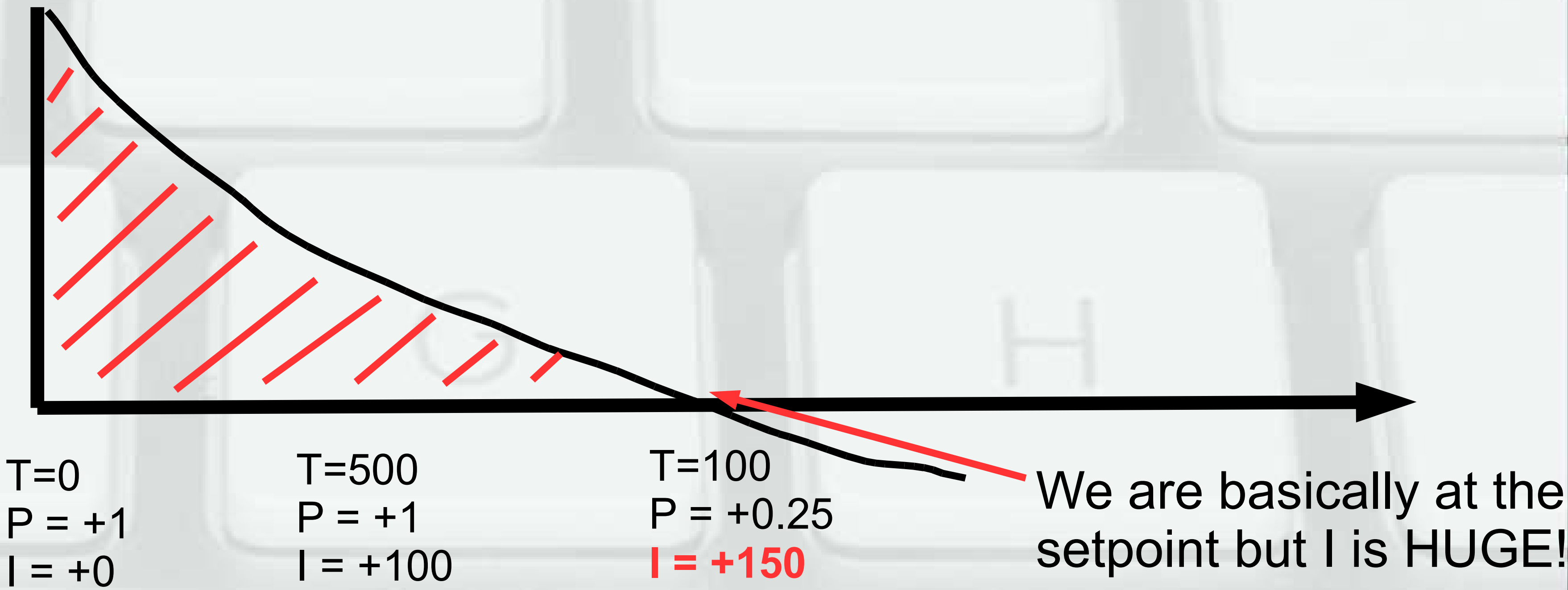
Offset error – P and I (PI controller)

- PI controller: we are currently 0.5 degrees off course
  - Error 0.5 * Kp (2) + intg(0) * Ki (1) = 1 power, which moves 0.5 degrees right
  - 0.5 right + -0.5 left = 0 degrees movement
  - Error 0.5 * Kp (2) + intg(0.5) * Ki (1) = 1.5 power, which moves 0.75 degrees right
  - 0.75 right + -0.5 left = 0.25 degrees right (now @0.25)
  - Error 0.25 * Kp (2) + intg(0.75) * Ki (1) = 1.25 power, which moves 0.625 degrees right
  - 0.625 right + -0.5 left = 0.125 (now @0.125)
- The integral build up is forcing us to turn right despite the constant -0.5 left turn imbalance

So PI is always better than P?

- No, not always
- It **can** increase over shoot
- It **can** increase set time (time to get to 0)
- It **can** reduce stability (keeps messing with the output)
- Windup can be an issue

I windup



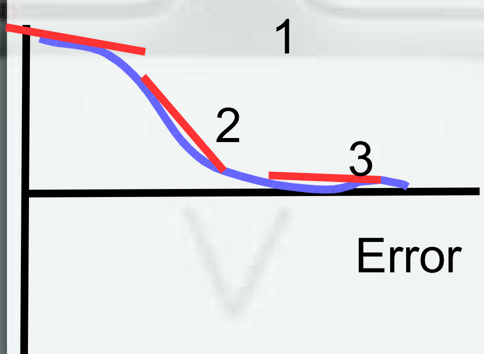
- **I** keeps getting bigger if we start far from the setpoint
- We we reach the setpoint **I** will push us past it for a long time

Prevent I windup

- There is lots of in-depth research into PID controllers
- We can just keep it simple
- Limit the size of I to prevent too much build up
    - If (I > 50) then I = 50
- Set I back to zero the first time we cross the Y axis
    - If (cross Y axis && first_cross) I=0
    - This will allow us to accelerate towards setpoint if we are far away, but stop once we are relatively close

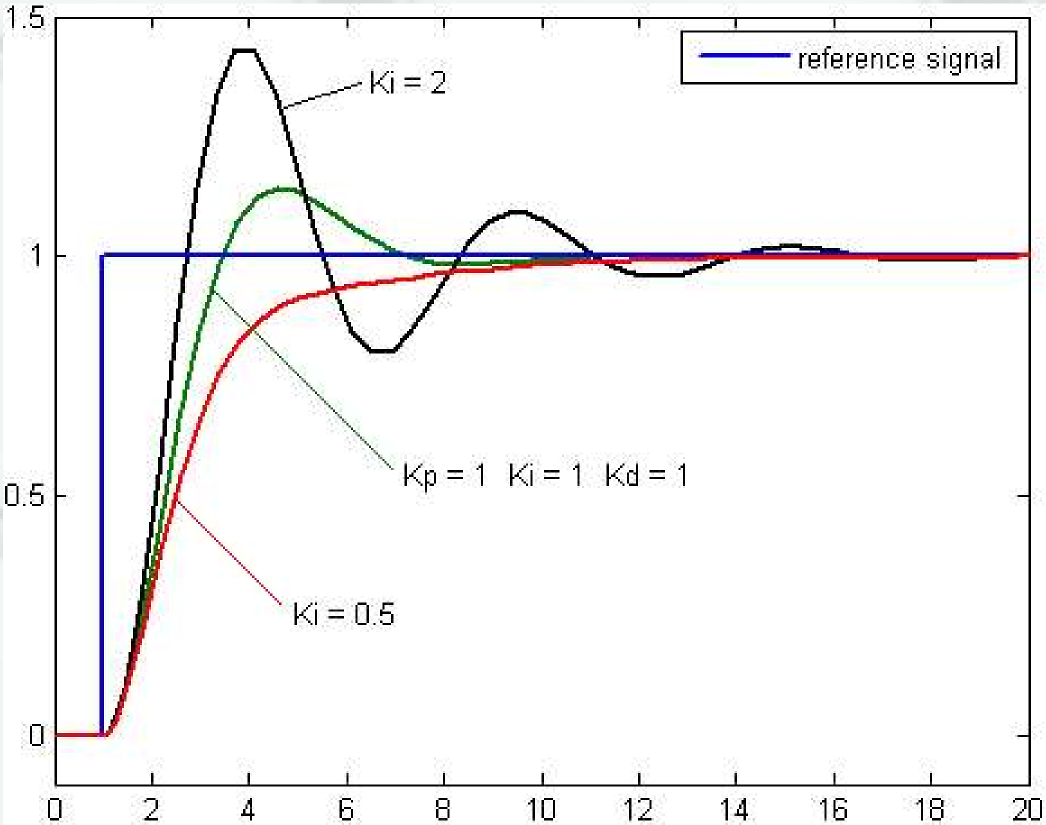
Derivative Control

- If in doubt, just use PI and skip the D
- Predict the future and adjust the output according
- The faster the **rate of change** of the error the more D will scale back the output



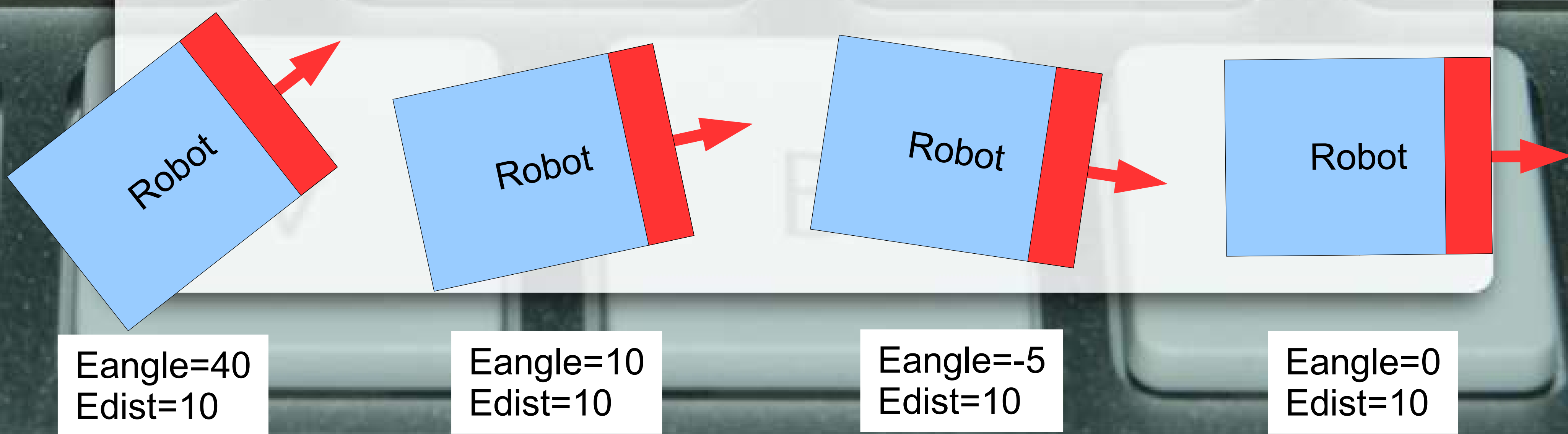
What D is saying:
1. Error's not changing much, I won't do much
2. We're changing a lot, we should slow down
3. We're not changing much again, I won't do anything

Derivative Control

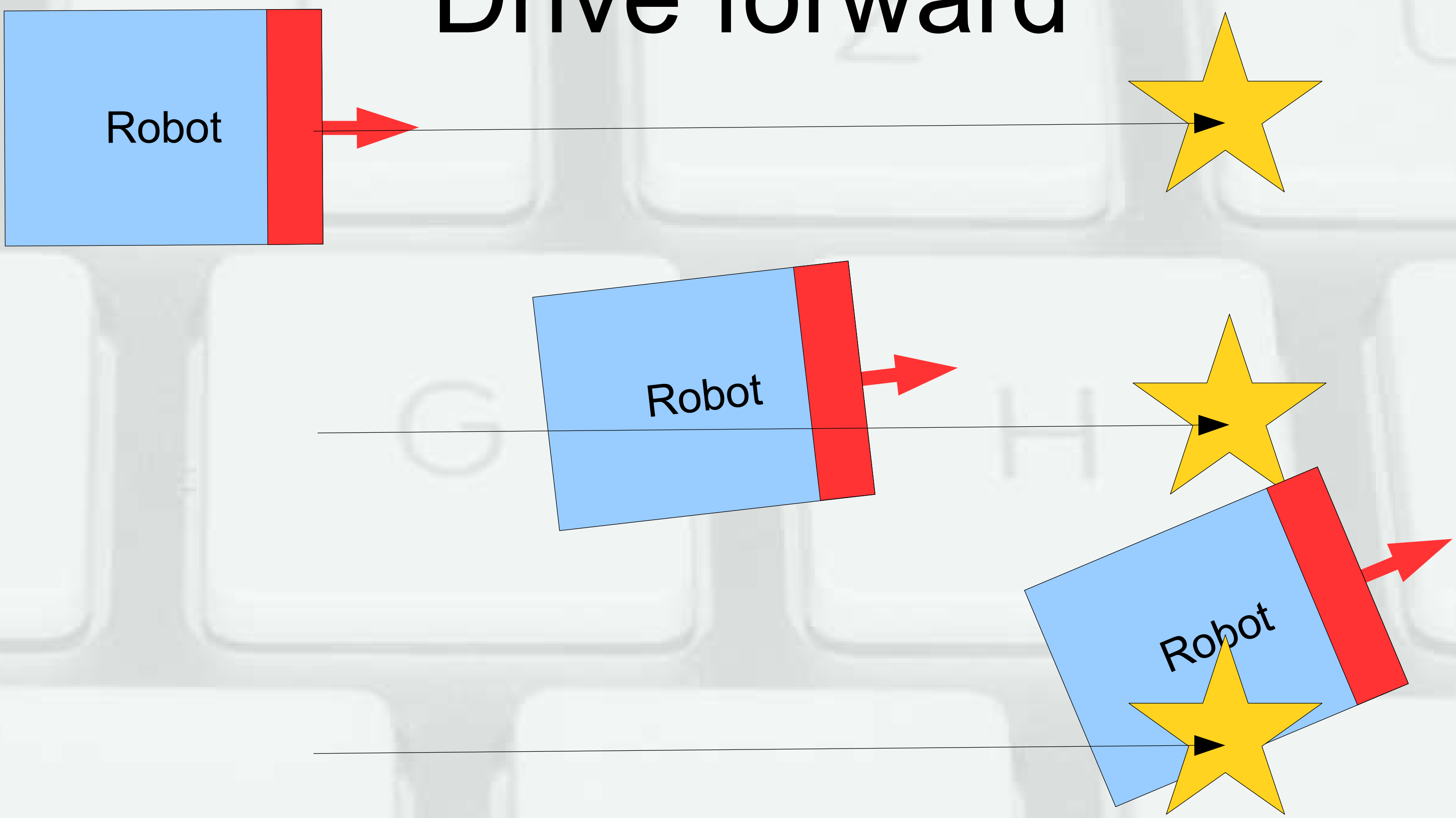


Weren't we going to drive forward?

- Use the turn PID controller



Drive forward



- We forgot about Rob's donut in the wheel!

What to do?

- We run our turn controller in 2 modes:
  - Turn then exit
  - Continuous course
- We first use the controller to turn until we are pointing in the correct direction and exit so that we can start the drive forward command
- As part of driving forward we start the turn PID controller again at 0 degrees with 0 error to stay on our current heading
- We will stop the drive and turn controllers once we have traveled the correct distance

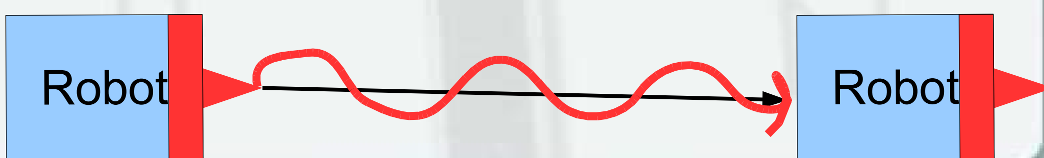
A PID controller is like a tire

- You make a tire. It goes on a car and allows it to drive. So everything is good?
  - What if it's a race car?
  - What if it's an off road car?
  - What if it's an 18 wheeler truck?
  - What if it's a moon buggy?
- One tire doesn't fit every car, one PID controller doesn't fit every problem
- You need to customize, tweak, and tune

Potential Issues

- Minimum movement unit
- Momentum
  - Turning
  - Going forwards/backwards
- Gravity
- Long windup time
- Offset errors
- Steady state error

No controller present, that's just how mechanical objects work



Potential Solutions

- Exit once output is below a threshold
- Pause the controller for a time to allow momentum to take its course
- Consider known effects/offsets
  - Gravity will never make us fall up
    - It's easier for the motor to "push" downwards
- Have a high gain and low gain controllers for large or small distances – big move tuned and little tweak tuned
- Measure the variability/errors under best-case conditions so that you're not trying to achieve something impossible
  - Can we move from 15.001 degrees to 15.002?

We did a gain slowdown

```java
/**
 * If the current error is within one of the thresholds then slow down
 * @param cur_input
 */
private void doSlowdown(double cur_input) {
    double cur_error = (setpoint - cur_input);

    // Do the outer slowdown
    if (!startedOuterSlowdown && Math.abs(cur_error) < outerThreshold) {
        Kp = this.origKp * outerSlowdown;
        Ki = this.origKi * outerSlowdown;
        Kd = this.origKd * outerSlowdown;

        maxoutput_low = this.origMaxoutput_low * outerSlowdown;
        maxoutput_high = this.origMaxoutput_high * outerSlowdown;

        this.startedOuterSlowdown = true;
    }

    // Do the inner slowdown
    if (!startedInnerSlowdown && Math.abs(cur_error) < innerThreshold) {
        Kp = this.origKp * innerSlowdown;
        Ki = this.origKi * innerSlowdown;
        Kd = this.origKd * innerSlowdown;

        maxoutput_low = this.origMaxoutput_low * innerSlowdown;
        maxoutput_high = this.origMaxoutput_high * innerSlowdown;

        this.startedInnerSlowdown = true;
    }
}
```

We did a momentum wait

```
public boolean isDone() {
    boolean superDone = super.isDone();

    if (superDone)
        isNormallyDone = true;

    return isNormallyDone && (currentRetryTimeMs >= maxRetryTimeMs);
}
```

We tried to stop windup

```
public void calculate(double cur_input, boolean clamp) {
    long currentLastCalledTime = lastCalledTime;

    doSlowdown(cur_input);

    // Now do the calculation as normal (maybe with slowed down values)
    super.calculate(cur_input, clamp);

    // If the controller is normally done we might want to wait for the
    // robot to settle. We will set the output to 0.0 to settle
    if (this.isNormallyDone && currentSettleTimeMs < settleTimeMs) {
        output_value = 0.0;

        // reverse any integral windup for this period
        if (!preventWindUp) {
            double cur_error = (setpoint - cur_input);
            integral -= cur_error * dt;
        }

        // Add some time to the settle time (at least 1ms)
        currentSettleTimeMs += Math.max(1l, System.currentTimeMillis() - currentLastCalledTime);
    }
    // Otherwise, if we would normally be finished by we are doing the after
    // settle control increment that time
    else if (this.isNormallyDone) {
        currentRetryTimeMs += Math.max(1l, System.currentTimeMillis() - currentLastCalledTime);;
    }
}
```

What We Should Have Done - hindsight is a wonderful thing