Operating the robot with feedback from sensors (PID control)

Without feedback the robot is limited to using timing to determine if it's gone far enough, turned enough, or is going fast enough. And for mechanisms, without feedback it's almost impossible to get arms at the right angle, elevators at the right height, or shooters to the right speed. There are a number of ways of getting these mechanisms to operate in a predictable way. The most common is using PID (Proportional, Integral, and Differential) control. The basic idea is that you have a sensor like a potentiometer or encoder that can measure the variable you're trying to control with a motor. In the case of an arm you might want to control the angle - so you use a potentiometer to measure the angle. The potentiometer is an analog device, it returns a voltage that is proportional to the shaft angle of the arm.

To move the arm to a preset position, say for scoring, you predetermine what the potentiometer voltage should be at that preset point, then read the arms current angle (voltage). The different between the current value and the desired value represents how far the arm needs to move and is called the error. The idea is to run the motor in a direction that reduces the error, either clockwise or counterclockwise. And the amount of error (distance from your setpoint) determines how fast the arm should move. As it gets closer to the setpoint, it slows down and finally stops moving when the error is near zero.

The WPILib PIDController class is designed to accept the sensor values and output motor values. Then given a setpoint, it generates a motor speed that is appropriate for its calculated error value.

Creating a PIDController object

```
Joystick turretStick(1);
Jaguar turretMotor(1);
AnalogChannel turretPot(1);
PIDController turretControl(0.1, 0.001, 0.0, &turretPot, &turretMotor);

turretControl.Enable();  // start calculating PIDOutput values

while(IsOperator())
{
   turretControl.SetSetpoint((turretStick.GetX() + 1.0) * 2.5);  (1)
   Wait(.02);      // wait for new joystick values
}
```

The **PIDController** class allows for a PID control loop to be created easily, and runs the control loop in a separate thread at consistent intervals. The **PIDController** automatically checks a **PIDSource** for feedback and writes to a **PIDOutput** every loop. Sensors suitable for use with **PIDController** in WPILib are already subclasses of **PIDSource**. Additional sensors and custom feedback methods are supported through creating new subclasses of **PIDSource**. Jaguars and Victors are already configured as subclasses of **PIDOutput**, and custom outputs may also be created by sub-classing **PIDOutput**.

A potentiometer that turns with the turret will provide feedback of the turret angle. The potentiometer is connected to an analog input and will return values ranging from 0-5V from full clockwise to full counterclockwise motion of the turret. The joystick X-axis returns values from -1.0 to 1.0 for full left to full right. We need to scale the joystick values to match the 0-5V values from the potentiometer. This is done with the expression (1). The scaled value can then be used to change the setpoint of the control loop as the joystick is moved.

The 0.1, 0.001, and 0.0 values are the Proportional, Integral, and Differential coefficients respectively. The **AnalogChannel** object is already a subclass of **PIDSource** and returns the voltage as the control value and the Jaguar object is a subclass of **PIDOutput**.

The **PIDController** object will automatically (in the background):

- Read the **PIDSource** object (in this case the turretPot analog input)
- Compute the new result value
- Set the **PIDOutput** object (in this case the turretMotor)

This will be repeated periodically in the background by the **PIDController**. The default repeat rate is 50ms although this can be changed by adding a parameter with the time to the end of the **PIDController** argument list. See the reference document for details.

Setting the P, I, and D values

The output value is computed by adding the weighted values of the error (proportional term), the sum of the errors (integral term) and the rate of change of errors (differential term). Each of these is multiplied by a scaling constant, the P, I and D values before adding the terms together. The constants allow the PID controller to be tuned so that each term is contributing an appropriate value to the final output.

The P, I, and D values are set in the constructor for the PIDController object as parameters.

The SmartDashboard in Test mode has support for helping you tune PID controllers by displaying a form where you can enter new P, I, and D constants and test the mechanism.

Continuous sensors like continuous rotation potentiometers

The PIDController object can also handle continuous rotation potentiometers as input devices. When the pot turns through the end of the range the values go from 5V to 0V instantly. The PID controller method SetContinuous() will set the PID controller to a mode where it will computer the shortest distance to the desired value which might be through the 5V to 0V transition. This is very useful for drive trains that use have continuously rotating swerve wheels where moving from 359 degrees to 10 degrees should only be a 11 degree motion, not 349 degrees in the opposite direction.

Controlling the speed of a motor

Controlling motor speed is a a little different then position control. Remember, with position control you are setting the motor value to something related to the error. As the error goes to zero the motor stops running. If the sensor (an optical encoder for example) is measuring motor speed as the speed reaches the setpoint, the error goes to zero, and the motor slows down. This causes the motor to oscillate as it constantly turns on and off. What is needed is a base value of motor speed called the "Feed forward" term. This 4th value, F, is added in to the output motor voltage independently of the P, I, and D calculations and is a base speed the motor will run at. The P, I, and

D values adjust the feed forward term (base motor speed) rather than directly control it. The closer the feed forward term is, the smoother the motor will operate.

Note: The feedfoward term is multiplied by the setpoint for the PID controller so that it scales with the desired output speed.

Using PID controllers in command based robot programs

```
1  package org.usfirst.frc190.GearsBot.subsystems;
2  import edu.wpi.first.wpilibj.*;
3  import edu.wpi.first.wpilibj.command.PIDSubsystem;
4  import edu.wpi.first.wpilibj.livewindow.LiveWindow;
5  import org.usfirst.frc190.GearsBot.RobotMap;
6
7  public class Elevator extends PIDSubsystem {
8      SpeedController motor = RobotMap.elevatorMotor;
9      AnalogChannel pot = RobotMap.elevatorPot;
10
11     public Elevator() {
12         super("Elevator", 1.0, 0.0, 0.0);
13         setAbsoluteTolerance(0.2);
14         getPIDController().setContinuous(false);
15         LiveWindow.addActuator("Elevator", "PIDSubsystem Controller", getPIDController());
16     }
17
18     public void initDefaultCommand() {
19     }
20
21     protected double returnPIDInput() {
22         return pot.getAverageVoltage();
23     }
24
25     protected void usePIDOutput(double output) {
26         motor.pidWrite(output);
27     }
28 }
29
```

The easiest way to use PID controllers with command based robot programs is by implementing PIDSubsystems for all your robot mechanisms. This is simply a subsystem with a PIDController object built-in and provides a number of convenience methods to access the required PID parameters. In a command based program, typically commands would provide the setpoint for different operations, like moving an elevator to the low, medium or high position. In this case, the isFinished() method of the command would check to see if the embedded PIDController had reached the target. See the Command based programming section for more information and examples.