RoboDoX
ENGINEERING A BRIGHTER FUTURE
Granada Hills Charter High School Robotics Team

Implementing PID for Velocity Control Using the WPI Library (C++)

1 Introduction

2 Velocity

2.1 Calculating RPM

2.2 Filtering RPM using an infinite set

2.3 Normalizing RPM for PID use

3 Input to PID

3.1 Creating a PID source

3.2 Sending the RPM to PID

4 PID output

4.1 Sending PID output to Speed Controller

1 Introduction

This paper regards the steps necessary to implement a PID loop for velocity control using the WPI Library in FIRST Robotics. The methods discussed are only compatible with the Iterative Robot class but can be easily modified for use with the Simple Robot class. To use PID for velocity control with the Simple Robot class, follow the red steps in addition to the normal steps of the guide. If using Iterative Robot, it is unnecessary to follow the red steps.

2 Velocity

In the 2012 version of WPILib used in the 2012 FRC season, PID control is only available for position and rate, but not yet for true velocity control. To implement PID for velocity control, it is first necessary to obtain velocity by calculating Revolutions Per Minute (RPM).

2.1 Calculating RPM

To calculate RPM, it is first necessary to obtain a few variables. You must create an Encoder object and a Timer object. The current Encoder value is necessary. It is also necessary to get the current time value.

Sample code will be shown, and then explained. **Note: the following sample was written using the IDE Notepad++ so it may not match your final code**

```
void CalculateRPM()
{
    static int rawValue          = 0;
    static int lastRawValue      = 0;
    static int deltaCounts       = 0;
    static double loopsPerSecond = 0;
    static double lastTime       = 0;
    static double currentTime    = 0;
    static double deltaTime      = 0;
    static double rpm = 0;
    if(ITERATIVE_ROBOT_PERIOD == 0)
    {
        loopsPerSecond = 50.0;
    }
    else
    {
        loopsPerSecond = 1/ITERATIVE_ROBOT_PERIOD;
    }

    rawValue = velocityEncoder.GetRaw();
    deltaCounts = rawValue - lastRawValue;

    currentTime = rpmTime.Get();
    deltaTime = currentTime - lastTime;

    rpm = deltaCounts * (60.0 / COUNTS_PER_REVOLUTION) * loopsPerSecond;
    rpm = deltaCounts * (60.0 / COUNTS_PER_REVOLUTION) / deltaTime;

    lastRawValue = rawValue;
    lastTime = currentTime;

    rpmSource.inputRPM(rpm / RPM_MAXIMUM)
}
```

Return to Table of Contents

For users of the Iterative Robot class who would like to not use time, ignore the code outlined with a red box. For users of the Simple Robot class or those wanting to use time to calculate RPM, follow the steps in the red box. All users must write the lines of code with an arrow pointing to them.

rawValue is the current raw encoder value used for calculation. lastRawValue is the previous encoder values, used for finding the difference in encoder values. deltaCounts is the change of counts between cycles of the loop and is used for calculation of the RPM. loopsPerSecond is only used by Iterative Robot to determine how many times the program will cycle in a second, this is used for time independent rpm. lastTime is the previous time used for time dependent calculation. currentTime is the current time used for time dependent calculation. deltaTime is the difference between currentTime and lastTime, used for calculation of RPM. Rpm is the rpm which we calculate.

The default Iterative Robot Period is defined as 0 but is actually interpreted as .02 seconds in the WPILib calculation. This gives us a frequency of 50 Hz or 50 loopsPerSecond. For non-default periods, the inverse is taken which will give us a frequency in Hz or loopsPerSecond.

The rawValue is obtained from the WPILib GetRaw() function in the Encoder class. We calculate deltaCounts before calculating rpm. The currentTime is obtained from the WPILib Get() function in the Timer class. We calculate deltaTime before calculating rpm.

rpm is calculated by multiplying the deltaCounts by the number of seconds in a minute (60) divided by the number of counts in a revolution(listed on the encoder wheel) which is multiplied by the number of loops in a second to get a final rpm. rpm is calculated by multiplying the deltaCounts by the number of seconds in a minute (60) divided by the number of counts in a revolution(listed on the encoder wheel) which is divided by the deltaTime for the loop to get a final rpm.

After calculating rpm, we set the rawValue and currentTime to lastRawValue and lastTime, respectively.

Finally, we output our rpm to the rpmSource which feeds the rpm to PID, this will be discussed further later on.

2.2 Filtering RPM using an Infinite Set

The following step is optional, only to be used if you get extremely noisy values from your rpm. To filter your values, it is necessary to add the following chunk of code to your existing calculateRPM() function.

Code will be shown, and then explained.

```
static double r      = .3;
static double rpmIIR = 0;

rpmIIR = r * rpmIIR + (1-r) * rpm;

rpmSource.inputRPM(rpmIIR / RPM_MAXIMUM);
```

r is the modifier for our rpm, the closer r is to 1, the less modification that will occur to average rpm. If r = 0, there will be no averaging. rpmIIR is our new averaged, filtered rpm. We calculate rpmIIR by multiplying r by the previous rpmIIr and adding that to rpm multiplied by (1-r). Then, we will take the new filtered rpm and input that into our rpmSource, replacing the old line of code.

Return to Table of Contents

2.3 Normalizing RPM for PID use

As it stands, the current rpm value is unusable in PID. This is because PID will only accept a value between -1 and 1 as input so that it can calculate a properly scaled output. To use this rpm in PID, we must normalize the value. To normalize it, we must divide our rpm by the physical maximum rpm, or the RPM_MAXIMUM. A slightly inaccurate value that can be used would be the free speed of the motor, which will be higher than what you will reach with load. The maximum rpm can be calculated by hand or can roughly be found by printing the rpm value when giving the motor a full command under the condition it will be used on your robot.

3 Input to PID

Because WPILib currently doesn't support velocity control for PID, we will need to create a custom input into their PID. This is done by creating a class which inherits from the PIDSource class in WPILib.

3.1 Creating a PID source

We will create a sample PID source class. We will create the .cpp and .h files for this class. We will call this class RPMSource. Sample code will be presented, and then explained. First, the header(RPMSource.h)

```
#ifndef RPMSOURCE_H
#define RPMSOURCE_H

class RPMSource : public PIDSource
{
public:
    RPMSource();
    virtual ~RPMSource();
    void inputRPM(double input);
    double PIDGet();
private:
    double rpm;
};

#endif
```

Firstly, we must define the header so that it can be accessed. Then we create a RPMSource class which inherits from PIDSource. We create an RPMSource constructor and destructor. The function input RPM is used to create the actual input which is returned by PIDGet(). The parameter, input, is the normalized rpm that we calculated earlier in the guide. rpm is a global variable so that it can be used in the inputRPM function and the PIDGet() function.

Next we will create our .cpp

Return to Table of Contents

```
#include "WPILib.h"
#include "RPMSource.h"

RPMSource::RPMSource()
{

}

RPMSource::~RPMSource()
{

}

void RPMSource::inputRPM(double input)
{
    rpm = input;
}

double RPMSource::PIDGet()
{
    return rpm;
}
```

Here we will include the WPILib header and our RPMSource header. We don't need anything in our constructor and destructor. Rpm is set by inputRPM, then it is output by PIDGet().

3.2 Sending the RPM to PID

In order to send the RPM to PID we must do two things. First we send the RPM to RPMSource, as we do at the end of our calculateRPM() function. Then we must be sure to create our PIDController object with our rpmSource as the source parameter.

Sample code, and then a further explanation.

```
PIDController(float p, float i, float d,
              PIDSource *source, PIDOutput *output,
              float period = 0.05);

PIDController(float p, float i, float d,
              rpmSource, PIDOutput *output,
              float period = 0.05);
```

The top code is an example of what the parameters would normally be for PIDController. On the bottom is the same example, with our new rpmSource object in place of PIDSource.

4 PID Output

This part of the guide will show how to get the PID output and send it to a speed controller. There is nothing different about using this PID output compared to others, so this simply serves to complete the guide.

4.1 Sending PID Output to Speed Controller

After the PID has calculated an output, you will need to set the output of PID to your motor. This should be done automatically for whichever speed controller is specified in the constructor, but you can do it manually. This example will show sample code to a victor.

Return to Table of Contents

```
victor1.Set(pidController.Get(), 0);
```
This sets the victor 1 to the PID output with a default syncGroup parameter of 0.

Return to Table of Contents