Gyro Programming for FRC Robots

By:
Matt Krass
Email: mattkrass@gmail.com
Last updated: May 20th,2006
Team 358
http://www.team358.org

One of the more common sensors in FIRST Robotics I've encountered is a Gyro. It is also known as a Gyroscope, Angular Rate Sensor, and Yaw Rate Sensor. The uses for them are plentiful, from detecting tipping, to making accurate turns, to driving in a straight line. They're also good as a simple detector of shocks in autonomous mode, and coupled with an accelerometer can be used as a simple detector of impact. They come in many forms and packages, with different ratings of resolution and output.

First things first, let's get it straight what a gyro does exactly. A lot of people are of the misconception the gyro returns an angle relative to where it powered up. It does not; it actually returns a value indicating how fast the gyro is rotating on a single axis. Typically the output is a variable voltage from 0V to 5V. It rests at 2.5V nominally, indicating no movement in either direction. As it rotates the voltage increases and decreases based on how fast and to which direction. Just that value alone is good because it can be a simple indicator of a drift in the drivetrain, or your robot getting knocked off course for some reason. The real magic comes from integrating (adding up) the output of the gyro over time to get a relative heading, with 0 degrees being the direction the robot was facing when the gyro was initialized.

There are several things to consider when selecting and using a gyro. One of the most overlooked would seem to be the maximum turn rate of the gyro. For example, a popular gyro from Analog devices comes in 150 deg/sec and 300 deg/sec maximum turn rate models. The gyro in the 2006 Kit was has an approximately 80 deg/sec maximum. This is important

because if you exceed the turn rate, there's no real indicator, and your integration is no longer valid. You'll need a definite base point to recalibrate your gyro to maintain your current heading. Generally you shouldn't exceed 80% of the rated maximum, as it's a nominal rating and the closer you get the more of a risk of overloading the gyro. Another thing to look into is the gyro's signal bandwidth. This is essentially how often the gyro updates. This can usually be found in the device's datasheet. This is important in choosing the sample rate *(the wording of the rest of this sentence makes your statement ambiguous. The reader doesn't know if this is a good thing or a bad thing. Might also want to mention that you'll cover sampling later on.)* because if you sample faster than the gyro updates, you're oversampling it. This does not have to be handled specifically, but it will greatly increase accuracy and stability of the gyro.

Gyros are usually integrated by means of an accumulator. This is a function of the program that runs at a constant frequency, sampling the gyro and adjusting the heading. It essentially works like a high school math problem:

The formula for Heading is:
Heading = Rate of Change * Time

For example, having an accumulator that runs at 100Hz, every 10 milliseconds, could sample the gyro rate of turn, compute how far it could have gone in 10 milliseconds, and add or subtract a global variable indicating heading. For example:

Gyro Output: +200 deg/sec
Time period: 10 milliseconds (0.01 seconds)
200 deg/sec * 0.01sec = 2deg
Heading + 2deg = New Heading

By having a function like that run constantly in the background, on a timer interrupt, you can monitor the gyro and keep a live reading of your heading. Here is an example of a very simple accumulator:

```
// The gyro is connected to analog input 1
#define GYRO_INPUT rc_ana_in01

// "heading" is constantly updated to current heading, it's a
// long to prevent problems from overflows occurring. It's a
// global so both the user and the accumulator can access it.
```

```
signed long heading = 0;

// This runs every 10 milliseconds, 100 times a second.
void BasicAccumulatorSimple()
{
      // Sample the gyro's output
      unsigned int gyrovalue = GetAnalogValue(GYRO_INPUT);

      // Now we want to convert the gyro output to signed.
      // Typically a value of 512 is "at rest" and below that is
      // movement in one direction, above in the other, so we
      // subtract 512 to make the base point ("at rest") 0.
      // In doing so we can just add it straight away to the
      // heading and be done with it. It is also typecasted to
      // the same variable type as heading, as a precaution.
      heading += (signed long)(gyrovalue – 512);

      // Ok, we're done here.
}
```

This is a very, very simple accumulator. It lacks any error checking, or filters, or serious precautions against a computer error. It does illustrate the point though. You might be wondering, where did I use the equation I just explained? The short answer is, I didn't. The long answer is that I did, in another form.

That accumulator runs at a constant frequency, as such, every sample is the same period of time. This makes the divisor of the equation a constant, and factored in to every number involved, which effectively cancels it out. Now all you need to worry about is accumulating the changes. This will give you your heading in an arbitrary raw unit. This might not seem like much use, but it will be consistent. Turning the gyro X degrees will always yield the same number, no matter what it is. This is of course within reason and real world tolerances. With this you can benchmark certain common angles, like 30, or 45 or even 90 degrees. Using simple `#define`s you can make DEG30, DEG45, and DEG90 to those values and use them later. The readings will also be consistent across multiplication. So you can use DEG90 * 2 to get the reading for 180 degrees.

One of the biggest flaws in that example is the assumption of 512 as the "at rest" point. This was a guesstimate I used based on the resolution of the Analog-to-Digital converter on the Robot Controller, which is 10 bits. This means 2^10, or 1024, possible values. 1024 / 2 = 512, which is the

midpoint. Most sensors tend to oscillate around that point, and a startup calibration period while not moving can help maintain the gyro's accuracy.

The best method I've found to do a quick calibration is to average a few seconds of gyro sampling and declare that the base point. This means you have to remain still during the calibration period. Here's an example of a modified accumulator that calibrates the gyro the first time it runs:

```
#define GYRO_INPUT rc_ana_in01 // Gyro on input 1
#define CAL_SAMPLES 300 // Take 300 samples (3 seconds) for the
                        // calibration

signed long heading; // Global variable to hold heading

void BasicAccumulatorCalibrate()
{
      unsigned int gyrovalue = 0;
      unsigned static char calibrated;
      unsigned static char samplecount;
      unsigned char gyrobase;

      // Sample the gyro's output
      gyrovalue = GetAnalogValue(GYRO_INPUT);


      // Has the gyro gone through it's power up calibration yet?
      if(!calibrated)
      { // It hasn't been calibrated
            if(samplecount < CAL_SAMPLES)
            { // Have we finished sampling?
                  gyrobase += gyrovalue; // Accumulate samples
            }
            else
            { // Let's average it now
                  // Divide samples to get average reading.
                  gyrobase /= CAL_SAMPLES;
                  // Tell the gyro it's calibrated
                  calibrated = 1;
            }
      }
      else // It has been calibrated.
      { // Accumulate it

            // Now we want to convert the gyro output to signed.
            // We do this by subtracting gyrobase, making the
            // reading center around 0.
            // Positive is one direction, negative another, then
            // we just integrate it
```

```
            heading += (signed long)(gyrovalue – gyrobase);
      }
}
```

This is a bit more complicated. It uses some more variables and a few conditionals. Essentially each run through it checks to see if it has been calibrated, if not it performs the calibration procedure repeatedly until a suitable value is achieved. The length of the calibration period is defined by CAL_SAMPLES, which is how many samples are averaged to get the base point. Once sufficient samples have been made, the average is calculated and the `calibrated` flag is set. Next run through the gyro will be sampled for actual heading. That part hasn't changed, except for using the freshly calculated base point instead of a constant. The more samples you take, the more accurate your base point is, and the more noise and 'glitches' it filters out. However every 100 samples is another second you have to wait, with my choice of 300 samples you have to spend the first 3 seconds not moving. Typically I use a much smaller number, such as 50, for about half a second of calibration. Note that the robot does not have to be enabled for this to work, so 3 seconds is OK since it takes you at least that long between powering up and the matches starting. This is an improvement, but it still does not accommodate noise, or sudden errors in the signal which can lower resolution. Also, the heading variable we've been modifying is fairly difficult to access, since it would cause data corruption if you happened to be accessing it when the 10 millisecond sampling timer goes off.

Let's tackle these problems one at a time, and build up our gyro software so it's robust and functional. The first problem is filtering the gyro output. There are all kinds of complicated software filters available, but you can get away pretty simply with a simple averaging one that dampens unexplainable spikes at the cost of a minor lag in the reaction. This may sound bad, but it hasn't had much impact in my experience, nothing I could measure on our 2006 robot to any extent. By simply storing the previous sample value, and averaging it with the new sample value before the accumulator every run through, you absorb spikes fairly well and the results are more accurate. Here's the function with the required modifications:

```
void BasicAccumulatorFinal()
{
      unsigned int gyrovalue = 0;
      unsigned static int prevgyro;
      unsigned static char calibrated;
      unsigned static char samplecount;
```

```
      unsigned char gyrobase;

      // Sample the gyro’s output
      gyrovalue = GetAnalogValue(GYRO_INPUT);


      // Has the gyro gone through it's power up calibration yet?
      if(!calibrated)
      { // It hasn't been calibrated
            if(samplecount < CAL_SAMPLES)
            { // Have we finished sampling?
                  gyrobase += gyrovalue; // Accumulate samples
            }
            else
            { // Let's average it now
                  // Divide samples to get average reading.
                  gyrobase /= CAL_SAMPLES;
                  // Tell the gyro it’s calibrated
                  calibrated = 1;
            }
      }
      else // It has been calibrated.
      { // Accumulate it

            // Now we want to convert the gyro output to signed.
            // We do this by subtracting gyrobase, making the
            // reading center around 0.
            // Positive is one direction, negative another, then
            // we just integrate it
            // Now we're also averaging the current sample with
            // the previous before normalizing it.

            // The following line is split over two for
            // readability
            heading += (signed long)(((gyrovalue + prevgyro) / 2)
                                                      – gyrobase);
      }
      // Last thing we do is store the gyrovalue for next time
      prevgyro = gyrovalue;
}
```

Now we’re getting somewhere, in a few short revisions we’ve gone from a simple accumulator to one that can calibrate the gyro, and filter it moderately well. It’s not quite done though. Now we need to talk about a few other problems.

(transition problem here. Above you talk about a “few” problems while the sentence below sounds like you only have one more. Might change “another” to “next” in some way.)

We still have another problem to resolve. Namely that unlucky timing on your use of the `heading` variable can cause a glitch. Depending on the situation, accessing the variable in and out of the interrupt can lead to some weird oddities. Namely that large, multibyte variables may change between the instructions that access them. Your value can sporadically become corrupted and unusable. It's recommended you disable the timer interrupt, copy the variable to a temporary working variable, and re-enable the timer. To simplify the process you can create a function that does all that and returns the variable for your use. Here's an example of such a function:

```
signed long GetHeading()
{
      signed long tempheading = 0; // Someplace to store the copy
      INTCONbits.GIEL = 0; // Disable low priority interrupts
      tempheading = heading; // Copy to a working variable
      INTCONbits.GIEL = 1; // Enable low priority interrupts

      return tempheading; // Return the copy of heading
}
```

This is used like this:

```
signed long robotheading = GetHeading();
```

Then you can use `robotheading` in all your calculations, and call `GetHeading()` only when you need an update. This relieves the problem of accessing an interrupt-owned variable.

Another problem that can't be solved very well is sensor drift. It's a fact of life, that depending on the hardware involved and the sampling rates, there will be a drift, it could be small, or it could be large, or miniscule. It's an unavoidable situation and the best you can do is tune your controls and sampling rates to minimize it over the 2 minute and change period you need it to work.

A lot of people have asked me why I work in "raw" units. I admit at first even I was skeptical of a unit that has no measurable quantity. But it does have its advantages. Namely, converting data is a guaranteed loss of accuracy, no matter how good the math is, errors are amplified by conversions. Since I had nothing to really gain other than readability for a conversion to "human-readable" units I chose to stay with raw. If you feel like you can accept the loss of resolution for the sake of readability you can

do the conversations fairly simply. It's a ratio between "gyro" units and something like degrees. This can be easily down by benchmarking a fixed point, such as 180 degrees, and then dividing the raw value by 180 to find out the raw unit for 1 degree. Then divide your units by that number to convert to degrees. This is where rounding errors and truncation errors become a problem, They can amplify a simple error multiple times over, especially if you keep converting the number. Floating point math is such a strain it's not worth it, and fixed point is a fairly advanced solution to a problem that doesn't really need to be solved in my opinion.

We're almost out of problems to solve here! Alright next up, remember when I told you about gyro's maximum turn rate? Well we need to deal with that too. If your gyro exceeds its maximum turn rate it simply stops counting, and you could be turning even faster than you know, which could easily get you lost. The obvious solution is "Don't turn fast!" but in reality (Darn that pesky place) we can't always control how the robot moves. Driver error, robot collisions, random black hole on the field, any number of things can disrupt the gyro and make it max out. In this condition the gyro is still good for relative measurements, but any heading you had stored is very unlikely to be valid still. The simplest way to detect max out is a simple if after sampling the gyro. If the value received is outside of a "safe" window of values, you've maxed out. There's not much you can do at this point really, without some other sensor to tell you where you're facing, you can't recover the heading. In the case of a gyro designed to measure tilt instead of heading, you can actually couple it with an accelerometer to re-calibrate while at rest, but that is beyond the scope of this paper. The gyro can be used for relative turns from that point, but if you don't know where you're starting, how can you know how far to turn?

Since I consider max out a sensor failure, and I consider failure detection to be an advanced topic, you can stop reading at this point if you just want a basic gyro driver. However if you're interested in failure detection and oversampling, keep reading and I'll explain those as well.

You're still here? Man you really want your gyro to work well, alright let's get started. First up, max out, before I said I used an `if` statement to detect it. This is pretty much all there is to *detecting* it. What you do with it is up to you. I prefer to set a flag and let the rest of the program handle it. After all, at the time it happens, it might not matter, so why risk screwing up something that wouldn't have otherwise been affected?

Here’s an example:

```
void AdvancedAccumulatorMaxOut()
{
      unsigned int gyrovalue = 0;
      unsigned static int prevgyro;
      unsigned static char calibrated;
      unsigned static char samplecount;
      unsigned char gyrobase;

      // Sample the gyro’s output
      gyrovalue = GetAnalogValue(GYRO_INPUT);
      // Is it too high?
      if(gyrovalue > GYRO_MAX_HIGH || gyrovalue < GYRO_MAX_LOW)
      {
            // Set global Gyro overloaded flag so you know
            gyro_overloaded = 1;
            // I don't recommend handling anything else here, let
            // something else see the flag and deal with it.
      }


      // Has the gyro gone through it's power up calibration yet?
      if(!calibrated)
      { // It hasn't been calibrated
            if(samplecount < CAL_SAMPLES)
            { // Have we finished sampling?
                  gyrobase += gyrovalue; // Accumulate samples
            }
            else
            { // Let's average it now
                  // Divide samples to get average reading.
                  gyrobase /= CAL_SAMPLES;
                  // Tell the gyro it’s calibrated
                  calibrated = 1;
            }
      }
      else // It has been calibrated.
      { // Accumulate it

            // Now we want to convert the gyro output to signed.
            // We do this by subtracting gyrobase, making the
            // reading center around 0.
            // Positive is one direction, negative another, then
            // we just integrate it
            // Now we're also averaging the current sample with
            // the previous before normalizing it.

            // The following line is split over two for
            // readability
            heading += (signed long)(((gyrovalue + prevgyro) / 2)
```

```
                                                        - gyrobase);
      }
      // Last thing we do is store the gyrovalue for next time
      prevgyro = gyrovalue;
}
```

Alright, now your code can detect max out and let the rest of the program know. Ok, so that wasn't incredibly advanced, the next part is pretty cool.

Oversampling is the technique of sampling a sensor faster than it updates, and using the "noise" generated to increase resolution. In the case of the gyro, it's a matter of sampling it faster than it reacts to changes in angular rate. The rate of change for the gyro is known as it's bandwidth, and is generally a measurement of Hz, or how many times a second the gyro's reading "updates". For example, the ADXRS300EB gyro board from Analog Devices has a bandwidth of 0.04KHz, or 40 Hz. This means the gyro's output changes 40 times a second, so if you're sampling at 200 Hz you're getting the same sample five times in a row, but you'll get slightly different values because of the "noise". Averaging these samples together increases your resolution. That's oversampling in a nutshell. *(Oversampling has one more important aspect: to avoid aliasing the result. If you sample at exactly the update frequency, for instance, you will actually lose information. For a general description see http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem in particular see http://en.wikipedia.org/wiki/Nyquist_rate. At a minimum we should always sample at twice the bandwidth just to get valid data.)*

Since oversampling is simply a ratio of bandwidth to sample rate, that's the easiest way to handle the math involved. After you sample the gyro each loop around, and subtract the neutral value, you can then multiply it by the ratio. That's all there is to it.

Here's an example, of the final version of the gyro driver:

```
...
#define SAMPLE_RATE   40/200
...

void AdvancedAccumulatorFinal()
{
      unsigned int gyrovalue = 0;
      unsigned static int prevgyro;
      unsigned static char calibrated;
```

```
	unsigned static char samplecount;
	unsigned char gyrobase;

	// Sample the gyro’s output
	gyrovalue = GetAnalogValue(GYRO_INPUT);

	// Handle the oversampling here
	gyrovalue = gyrovalue * SAMPLE_RATE; // Be explicit!
	// Is it too high?
	if(gyrovalue > GYRO_MAX_HIGH || gyrovalue < GYRO_MAX_LOW)
	{
		// Set global Gyro overloaded flag so you know
		gyro_overloaded = 1;
		// I don't recommend handling anything else here, let
		// something else see the flag and deal with it.
	}


	// Has the gyro gone through it's power up calibration yet?
	if(!calibrated)
	{ // It hasn't been calibrated
		if(samplecount < CAL_SAMPLES)
		{ // Have we finished sampling?
			gyrobase += gyrovalue; // Accumulate samples
		}
		else
		{ // Let's average it now
			// Divide samples to get average reading.
			gyrobase /= CAL_SAMPLES;
			// Tell the gyro it’s calibrated
			calibrated = 1;
		}
	}
	else // It has been calibrated.
	{ // Accumulate it

		// Now we want to convert the gyro output to signed.
		// We do this by subtracting gyrobase, making the
		// reading center around 0.
		// Positive is one direction, negative another, then
		// we just integrate it
		// Now we're also averaging the current sample with
		// the previous before normalizing it.

		// The following line is split over two for
		// readability
		heading += (signed long)(((gyrovalue + prevgyro) / 2)
                                                        – gyrobase);
	}
	// Last thing we do is store the gyrovalue for next time
	prevgyro = gyrovalue;
}
```

Ok, let's run through it, it's basically the last example with two additions. A new `#define`, which sets up the oversampling ratio, the formula for that is easy:

Gyro Update Rate (Hz) / Sampling Rate (Hz)

Then, each sample, after being centered around 0, is multiplied by the ratio. This is it, that's your oversampling code, seems too easy doesn't it? Keep in mind, you have to be explicit, since the fraction needs to be evaluated in the proper order of operations. If you shorten that statement, compiler optimizations will eat it, and default that to 0, making all your samples equal to 0, effectively disabling your gyro.

That about wraps up the paper, 358s 2006 robot uses a driver based on this original program, with numerous robot specific improvements. For more information on interrupts and timers and using them I recommend reading the IFI whitepaper on the subject, available on their website.

As always you can send me questions and comments to the above listed contact info, or post about this whitepaper on the forums. I hope this was helpful to teams and any way I can improve it is a welcome suggestion. Please see my PID Control Theory whitepaper, also available on ChiefDelphi.com for more information on how to control your robot using the gyro feedback.