Steering with the Modern Robotics gyro – an introduction

v4 dated 1/22/16, clarified heading directions
v3 dated 1/20/16, added new signed heading
v2 dated 12/11/15

This article is a low-tech introduction to steering a robot autonomously using the new Modern Robotics Integrating Gyro sensor. The focus here is learning, rather than simply providing copy-and-paste code. Examples are given mostly in **pseudocode** which can be adapted for languages such as Java, RobotC, C++, etc.

The MR gyro sensor gives rate of rotation about X, Y and Z axes. It also performs an on-board calculation (integration) to give the current heading about the vertical Z axis. This article discusses only the heading value, for steering. It's an integer value in degrees, initialized to zero at calibration, where zero is straight ahead (I2C cable at the back).

For simplicity, this tutorial has heading values increase with clockwise (CW) sensor rotation, looking down from above. This is true only when using **basic headings** obtained with the default/generic gyro command. See the end of Section 11 for actual directions.

The sensor should be placed horizontal (parallel to the ground), close to the robot's center of rotation, and as low as possible. Convenient access is not required, although it helps to see its glowing LED to indicate the sensor is connected.

1. Heading Values

The sensor can give two types of heading value. The **basic heading** type ranges only from 0 to 359 degrees, never a negative number. So, continuous CW rotation from zero gives positive values that increase to 359 degrees, then jump back to zero. Counterclockwise (CCW) rotation from zero immediately jumps to 359 degrees, then 358, and so on.

The other type is a **signed heading** that can be positive or negative, and extends without limit. So, counterclockwise (CCW) rotation from zero gives negative values such as -1 degree, then -2 degrees, and so on. And there's no rollover at 360 degrees, so two full CW rotations would result in a heading of +720 degrees.

For both types of headings, turning actions accumulate, unless the sensor is purposely reset to zero. If the sensor rotates CW 30 degrees from zero, then rotates CCW 10 degrees, the heading value will now read +20 degrees. But when the sensor rotates through zero, you must be aware of the adding and subtracting process. If the sensor rotates CW 10 degrees, then rotates CCW 30 degrees, the two heading types will give different results. The basic heading will be 340 degrees, while the signed heading will be -20 degrees.

Just choose the type of heading that you prefer, become familiar with its math, and stick with it. They have different characteristics for autonomous driving, as you will see below. In general the basic heading needs more managing, covered by Sections 3 and 5.

2. Target Headings

Autonomous steering is typically done by comparing the current heading to a target heading. You calculate the difference, or error, and command the motors accordingly.

When choosing a target heading, you must consider the type of heading value (basic vs. signed). Suppose you are working with signed values, and you want the robot to turn 90 degrees to the left. This steering destination could be represented by either a target of 270 degrees or a target of -90 degrees. Which one to use? If the current heading near zero is compared to +270, the error is very large, and standard steering math would sharply spin the robot around the wrong way (clockwise). You'll see that math later in Section 6. So, here you would use -90 as the target.

If you are working with basic heading values, the same caution applies plus the math is complicated by the "jump" that happens around zero. This is the topic of Section 3, which can be skipped if you are not using basic values.

A lesson here is that the programmer must know the intended driving plan. Do not rely blindly on one method or "the math" to cover all situations. Manually step through sample scenarios, including the anticipated extremes and values near zero. This valuable exercise is described later in Section 12.

3. Basic Heading Conversion

If you prefer using **basic heading** values, they may need to be converted before use. If you are using **signed heading** values, skip this section.

As noted above, CCW rotation from zero immediately jumps to 359 degrees, then 358, and so on. This complicates the math used to calculate deviation from a target heading. A common solution is to convert this high angle to a small negative angle, by subtracting 360 degrees from large sensor headings. Thus 350 degrees, for example, becomes -10 degrees. This -10 degrees can flow seamlessly to, say, +10 degrees as the sensor rotates CW, with no "jump" at zero. This works because the 360 is subtracted only from high values, not low values.

When choosing a target heading, the conversion method must be considered. Look at the example from Section 2. With the sensor pointing roughly forward, the initial heading value could be a high positive number, converted to a low negative number. If compared to +270, the error value is very large; standard steering math would sharply spin the robot around the wrong way (clockwise).

So, when you use negative values for CCW targets, here is adjustment pseudocode for rotating CW or CCW from zero (or any low value):

```
currentHeading = sensorGyro.getHeading();          // get and store the heading
if (currentHeading > 180) {                        // if high value, say 181 to 359,
  currentHeading = currentHeading - 360;           // make it negative
}
```

But suppose for some reason you need or want the target heading to be designated as +270 instead of -90. With the sensor pointing roughly forward, the initial heading could be a low positive number, not converted by the above pseudocode, giving a very large error value. Standard steering math would again sharply spin the robot around the wrong way (clockwise). In this case it's better to convert a low positive heading into a value **greater than** 360 degrees. For example 5 degrees would become 365 degrees, and the steering math would bring the robot around counterclockwise as desired.

So, when you use high positive values for CCW targets, here is adjustment pseudocode for rotating CCW from zero (or any low value):

```
currentHeading = sensorGyro.getHeading();        // get and store the heading
if (currentHeading <  180) {                     // if low value, say 0 to 179,
  currentHeading = currentHeading + 360;         // make it higher than 360
}
```

Obviously both of these methods must be adjusted if the steering might pass near or through an absolute heading of 180 degrees. In these examples there's nothing special about "180", it's just an intermediate number to safely distinguish low positive headings (above zero) from high positive headings (below 360), for the purpose of conversion.

Again, the programmer must know the intended driving plan. Do not rely blindly on one method or "the math" to cover all situations. Manually check the key values.

4. Looping

Steering actions are typically placed inside a **while loop**, constantly retrieving the sensor's heading values. When the loop's conditional expression is no longer true, the loop ends. For turning, the loop's condition might be "sensor's actual heading hasn't yet reached target heading".

How should you define "reached"? Use **greater-than** or **less-than**, as applicable. Look for a threshold, not an exact match. Beginning programmers often loop "while currentValue is not equal to (!=) targetValue". Robot sensors are famously jumpy and cycle times are unpredictable, so the gyro sensor could easily skip over the exact target value… the loop continues, yielding a funny spinning robot for YouTube.

Here is simple pseudocode for a steering loop.

```
targetHeading = 90;                              // want to turn CW
currentHeading = get signed heading;             // get heading value from gyro sensor

while ( currentHeading <  targetHeading ) {      // loop while condition is true

  [ steering/driving code ]
  currentHeading = get signed heading;           // update the sensor value
}                                                // end while loop

[ stop motors ]
```

You might think the condition should always read “while current heading less than target heading”. This works for CW rotation from zero towards a positive target. But what if the target is -90 degrees? A current heading near zero is not less than -90, the conditional is false, and the while loop will not run at all. A skipped loop can be hard to notice and frustrating to debug.

So, pay attention to the choice of greater-than or less-than. It depends on your target heading (e.g. -90 or +270) and your estimated current heading. Again, plan each intended driving action, have some idea of its physical scenario, and test key values. Don’t blindly copy-and-paste your steering code throughout the program.

You might have the bright idea of using the absolute-value function, to help ‘standardize’ your steering code. Certainly you can find examples of this online. But this is less intuitive, complicates the debugging, and anyway will not be a true universal solution. Stick with the actual numbers, and have a simple plan for each action.

5. Looping with Basic Headings

If you prefer using **basic heading** values, they should be converted before looping. If you are using **signed heading** values, skip this section.

With looping, basic headings pose a special risk. If the sensor value hasn't yet been converted, the conditional might already be false, and the while loop won't run at all. For example when turning CW, the condition might be "heading less than 90". An near-zero initial heading of 358 would be **false**, and the loop is **skipped**. Likewise for turning CCW, the condition might be "heading greater than 270"; an initial heading of 2 would be false.

A very crude solution might be to first steer in the intended direction for a **time duration** estimated to force the heading above 359 or below 0 as needed. A better solution is to perform a one-time conversion **before the loop**, and have the loop test the converted value.

```
targetHeading = 90;                                 // want to turn CW

currentHeading = sensorGyro.getHeading();           // read sensor value
if (currentHeading > 180) {                         // if high value, say 181 to 359,
   currentHeading = currentHeading - 360;           // make it negative
 }

while ( currentHeading < targetHeading ) {          // loop using converted heading

  [ steering/driving code ]

  currentHeading = sensorGyro.getHeading();         // update sensor reading
  if (currentHeading > 180) {                       // same conversion in loop
     currentHeading = currentHeading - 360;
   }
}                                                   // end while loop
[ stop motors ]
```

A shorter solution uses a **do...while loop**, which always executes at least once. Thus the conversion code doesn't need to appear both outside and inside the loop.

```
targetHeading = 90;                                  // want to turn CW
do {                                                 // use a do...while loop
    currentHeading = sensorGyro.getHeading();        // read sensor value
    if (currentHeading > 180) {                      // convert inside the loop
        currentHeading = currentHeading - 360;
    }
   [ steering/driving code ]
} while ( currentHeading < targetHeading );          // evaluate after each cycle
[ stop motors ]
```

6. Proportional Steering

As for the driving code itself, a simple method is proportional steering. Proportional means that the amount of steering varies (linearly) **in proportion to** the amount of error:

1. calculate error (deviation from target)
2. use error and gain to adjust a preselected 'middle' power level
3. run the motors, each adjusted, to steer

Here is sample pseudocode for steering while driving forward. If you once learned about **line following** in *FIRST* LEGO League, this might look familiar!

```
headingError = targetHeading - currentHeading;    // a decreasing positive error,
                                                  // for positive target
driveSteering = headingError x driveGain;         // gain is predetermined by testing

leftPower = midPower + driveSteering;             // positive adjustment
rightPower = midPower - driveSteering;            // negative adjustment

RunLeftMotor (leftPower);
RunRightMotor (rightPower);
```

If midPower is zero, the robot can spin in place (if it's capable) to the desired heading.

To get a sense of how this works, manually step through the code with sample values. The beauty of proportional code is that it automatically steers left or right as needed. If the error becomes negative (overshoot the target), the "positive" adjustment to leftPower becomes negative, *vice versa* for rightPower, and the robot steers the other way. Magic!

Through testing, gain should be selected to use the full desired range of motor power, and for steering response. A good choice of gain allows a driving robot to settle quickly into the desired heading and maintain near-zero error. When you manually step through the math for a few scenarios including the start, pick a rough initial value for gain, and fine-tune it from there.

Gain is sometimes called a proportional constant or **scaling factor**, since it also serves to "fit" the converted values to the available range of motor power. Some motor controllers look for values between 0 and 1, other motor controllers want values between 0 and 100.

7. Power Limits

The above calculations of motor power could give values outside the allowed range, possibly resulting in run-time errors or random motor commands. In general, a program needs to do something about calculated values that may give unpredictable behavior. Here's a sample correction:

```
leftPower = midPower + driveSteering;
if (leftPower > 100%)  leftPower = 100%;        // don't exceed maximum
if (leftPower < 0)  leftPower = 0;              // don't allow driving backwards
```

To allow spin moves (if the robot is capable), the low limit could be set to -100% (and midPower is zero). To reduce the top speed, adjust the high limit (and possibly gain).

8. Motor Directions

The above pseudocode samples ignore the fact that most robots have **opposite-facing drive motors**. Fortunately most programming libraries offer a single-line function to always reverse one of the motors. This leaves the programmer free to concentrate on code logic, without worrying about one motor always running 'backwards'.

Which one to reverse? Old habits may dictate reversing, say, the right-side motor. But the better approach is to first determine which motor naturally runs forward. Namely, giving that motor a **positive** power value makes the robot drive **forward**. If it's the left motor, then the old habit works fine. But many motor axles run clockwise with positive voltage, which (on the left side) often runs the drive train backwards. Your code would need to use all negative values for driving forward.

So, identify which motor runs forward naturally. Use the auto-reverse function on the other motor. You will save much stress and confusion by making the natural choice, rather than debugging 'double negatives' under pressure. Positive is always forward. Make it so, Earthling.

Also -- first make sure the declared LeftMotor is truly running the left-side motor. Make a simple test program to verify this. Seems obvious, but you'd be surprised!

9. Safety Timeouts

Action loops need safety timeouts. Repeat that to yourself five times. Your program must deal with an autonomous robot getting stuck in a loop, for whatever reason. Start a system timer when entering the loop, then monitor it and take action when needed.

One method is simply to add 'time' to the loop conditional. If the time limit is reached, the loop exits normally and the program continues. Here is an outline in pseudocode:

```
targetHeading = 90;                          // want to turn CW
currentHeading = get signed heading;         // get heading value from gyro sensor

ResetTimer;                                  // start loop safety timer, in seconds

while ( currentHeading <  targetHeading &&  Timer <  10 ) {     // loop while both
                                                                 // conditions are true
  [ steering/driving code ]

  currentHeading = get signed heading;       // update the sensor value

}                                            // end while loop

[ stop motors ]
```

Another method is to check the timer inside the loop, and take actions such as setting a timeout flag (boolean variable) and/or immediately exit the loop (*break* command in Java). The flag can be used after the loop to take further action if desired.

More purposeful methods are possible too, such as: take a corrective action (still within the loop), reset the timer, increment a counter, then continue the loop.

10. Reaching the Target

For spin moves and some other maneuvers, proportional steering can sometimes fail to achieve the full desired target heading. Why? As the error gets smaller and smaller, so do the power levels. At some point the robot may be unable to move further; the code is "stuck" in the loop.

This can be addressed by adjusting the minimum power levels, although this may risk overshooting the target. Or, if testing shows the robot gets stuck consistently short of the target, you could use a purposely higher target. When the loop "times out", the robot heading is known.

Low closing power is a characteristic of "P-only" control. Various improvements are offered by proportional-integral (PI) and proportional-integral-derivative (PID) controls, but they are outside the scope of this introduction.

11. Sample Outline

Putting all this together, here is a sample outline for "steer right to 90 degrees" using **basic headings**; sensor values are converted before use.

```
DeclareLeftMotor (portL);                   // connect to devices/controllers/ports
DeclareRightMotor (portR);
DeclareSensor (portG);
sensorGyro.calibrate();                     // calibrate and set gyro heading to zero
                                            // may need some time!
AutoReverse (LeftMotor);                    // right-side motor runs forward
midPower = 40%;                             // average driving speed is 40%
driveGain = 0.7;                            // depends on task and robot
targetHeading = 90;                         // steer CW to 90 degrees

ResetTimer;                                 // start loop safety timer, in seconds

do {                                        // begin do...while loop
  currentHeading = sensorGyro.getHeading();
  if (currentHeading >  180)  {             // if high value, say 181 to 359,
     currentHeading = currentHeading - 360; // make it a small negative
   }

  headingError = targetHeading - currentHeading;     // a decreasing positive error,
                                                     // for positive target
  driveSteering = headingError x driveGain;  // gain is predetermined by testing

  leftPower = midPower + driveSteering;      // positive adjustment
  if (leftPower > 100%)  leftPower = 100%;   // fix out-of-range calculated values
  if (leftPower < 0%)   leftPower = 0%;      // don't allow driving backwards

  rightPower = midPower - driveSteering;     // negative adjustment
  if (rightPower > 100%)   rightPower = 100%;
  if (rightPower < 0%)   rightPower = 0%;

  RunLeftMotor (leftPower);                  // run the drive motors
  RunRightMotor (rightPower);

} while ( currentHeading <  targetHeading    // end loop,
          &&  Timer <  10 );                 // if either condition is false

RunLeftMotor (0%);                           // stop motors
RunRightMotor (0%);
```

This is **pseudocode** only; you have the fun of writing the real code in your own programming language. Modern Robotics provides gyro set-up commands in Java: http://www.modernroboticsinc.com/integrating-3-axis-gyro

For simplicity, this tutorial has headings increase with CW rotation. This will be true only for **basic headings** obtained with the method getHeading from the default/generic class GyroSensor. The MR web page shows how to use their class ModernRoboticsI2cGyro, where method getHeading retrieves **basic headings** (0 - 359), and getIntegratedZValue retrieves **signed headings** (- & +); both increasing with CCW rotation.

12. Verifying Solutions

For each planned driving action, always step through the code manually. Yes, this means with paper and pencil; ask an older person for those.

Make a chart. The first column is **gyro heading**, the others are variables in order: currentHeading (after adjustment, if any), headingError, driveSteering, leftPower and rightPower (after high/low clipping, if any). In the final column called **Action**, draw a little arrow indicating the approximate robot driving direction produced by leftPower and rightPower.

Now plug in gyro sensor headings one at a time, starting with possible initial values. For starting near zero, for example, try +5 and +355. Remember that basic heading values range only from 0 to 360, never negatives. Now try values that change, say, every 10 degrees, in the intended steering direction: CW higher, CCW lower. Finally try a number just before your target, and a number just after your target (in case of overshoot). You can even test for an imagined obstacle or collision!

For each sensor input value, draw the arrow for output action. **If every tested scenario does what you want, your entire loop should work fine.**

This exercise, more than anything, will give a true understanding of proportional steering with the Modern Robotics gyro. Best of luck!

13. Background

This guide was written for young teams after trying to help a rookie FTC team at their first tournament in December 2015. The rookie programmers struggled to debug a simple autonomous program to drive straight and make a couple of turns. The code structure was good, but the programmers lost track of the pluses and minuses, forwards and backs, lefts and rights, and especially the logic of gyro conversions near zero degrees.

Tournament pressure added to the confusion. At one point they were dealing with **triple negatives**: positive power drives backwards, rightMotor is on the left side, but automatically reversed. Also they forgot whether the gyro headings increase CW or CCW, and forgot that their treads could not drive backwards, separately or together. An interesting day!

Searching online for clear guidance was not fruitful. Here are two examples:

> **"This is an example LinearOpMode that shows how to use the Modern Robotics Gyro."**
> http://stackoverflow.com/questions/33723341/moving-methods-into-a-different-class-without-passing-variables-java
>
> **"Gyros to control robot driving direction"**
> http://wpilib.screenstepslive.com/s/3120/m/7912/l/85772-gyros-to-control-robot-driving-direction

So, this basic guide was created. It all seems so simple now! But really it wasn't, and hopefully other new teams can now use this valuable sensor. Comments and corrections are welcome, please email WestsideRobotics@verizon.net