This lesson gives you a chance to practice coding, while introducing a new concept called "modulo", which takes a bit of thinking to understand in a mathematical concept, but as you will see is easy to use in terms of writing code. You learned about the gyroscope in an earlier lesson when you used it to turn lights on and off based on the rotation of the robot. In this project, we're going to use it to ensure that the robot drives in a straight line and can maintain a given orientation.

Note: This lesson is a bit heavy on math. The concepts aren't super advanced, but there is a fair amount of arithmetic that goes into it. If it starts getting confusing reading a lot of numbers, take your time, slow down, read it again slowly, and then proceed.

The gist of this project is that we will keep track of a target heading, and then constantly measure the current heading relative to the target heading and "add turning" relative to the difference between the target and the actual. If the difference is large, we'll add a lot of turning. If the difference is small, we'll add a small amount of turning.

Heading

Let's talk about what the word "heading" means in context. We will declare a variable representing the target heading of the Romi in the Drivetrain class. Here's an example. You can go ahead and create a new RomiReference project called DriveStraight and copy this into your code in Drivetrain.java:

```
14  public class Drivetrain extends SubsystemBase {
15    private static final double kCountsPerRevolution = 1440.0;
16    private static final double kWheelDiameterInch = 2.75591; // 70 mm
17
18    private double targetHeading;
```

This heading will be relative to whatever the heading of the robot was the last time the gyroscope was reset. You can reset the gyro anytime you want in code, but if you don't do that, by default the last time it is reset would be when the gyro first initializes when you turn the robot on. Assuming you're not rotating or moving the robot during its boot sequence, you don't need to worry about this too much – you can assume that the "default" orientation is whatever direction the robot is facing when it turns on, and therefore that orientation would be a heading of zero. As you saw last time we used the gyro, you can use the getAngleZ() method to return the current heading in degrees. Before we continue, let's make sure to initialize our targetHeading so that it does, in fact, use the heading of the robot when it boots. You can add this line of code to line 45 of Drivetrain.java. When the Drivetrain object is initialized, the current heading will be stored as the target heading.

```
39  /** Creates a new Drivetrain. */
40  public Drivetrain() {
41    // Use inches as unit for encoder distances
42    m_leftEncoder.setDistancePerPulse((Math.PI * kWheelDiameterInch) / kCountsPerRevolution);
43    m_rightEncoder.setDistancePerPulse((Math.PI * kWheelDiameterInch) / kCountsPerRevolution);
44    resetEncoders();
45    targetHeading = m_gyro.getAngleZ();
46  }
```

The getAngleZ method returns a double, and even though headings are relative to a 360 degree circle, getAngleZ() does NOT necessarily return a value between 0 and 360. If you turn the robot left (counterclockwise) 90 degrees, you'd get a value of -90. If you spin the robot right (clockwise) two full

revolutions from zero, you'd get a value of 720. If you spin the robot one and three-quarters revolutions left from zero, you'd get a heading of -630. This is important because if you're starting at zero, you might assume a negative number means you rotated the robot left, but -630 is the same orientation as positive 90. Rotating the robot left one and three-quarters revolutions puts it at the same orientation as rotating it one quarter revolution right.

Modulo

When we made the lights turn on and off, we didn't worry about this. You could load your program and spin the robot a full revolution and check this for yourself – even if it's at the proper orientation, the red light will stay on because we did not account for this. For that project, that was fine, but if you want to actually use the gyro to do useful things with the robot this is a problem. If you're actually driving the robot around you don't want to worry about how many times you've spun in a circle in order for your code to work. So we need a way to account for this, and luckily for us there's an easy solution using what's called "modulo".

You may have never heard the term modulo before but you're already familiar with it if you've ever used a clock or a calendar. What time is it right now? Chances are, it's some hour between 0 and 12 or 0 and 24, depending on what kind of clock you like to use. And what day of the month is it? Chances are it's something between 1 and 31. But obviously you've lived a lot more than 24 hours and a lot more than 31 days. But after each cycle, these numbers reset and start counting again. This is because we're interested in how much time has passed since the start of the most recent cycle. It's the same thing with the gyro – we're interested in how much change there has been since the last time the heading passed the origin point, regardless of whether it read 0, 360, -720, or whatever else when it passed the origin.

How do you *do* modulo math? Well, you've probably done this before as well, if you've ever done division and calculated the remainder. Let's go back to our calendar example. If you asked someone what hour it is when it was, say, 11 AM on January 1st 2021, they'd probably say it's 11. They *could* say it's 17,703,971, which is approximately the number of hours that have passed since the beginning of the day on January 1st at the start of the AD calendar, but they wouldn't do that. What they *would* do, without realizing it, is take 17,703,871, divide it by 24, and tell you the remainder of that operation, which happens to be 11. This is called *modulo division* – instead of giving you answer to 17703871/24, which is 737665.45833, they'd round that to an integer and give the remainder.

You're familiar with the basic mathematical operators: +, -, /, and *. But there's an additional operator you might not be familiar with: the modulo operator, "%". In Java, the percent sign will do remainder division, or modulo, on the number on its left. So for example:

```
System.out.println(10 % 3);
```

Outputs "1", because if you were do integer division of 10 by 3, you would get 3 with a remainder of 1. By the way, "10 % 3" would be read as "ten modulo three", or "ten mod three" for short. Modulo works for negative numbers as well; just like 10 mod 3 is 1, -10 mod 3 is negative -1 because 3 goes into 10 negative three times, with a remainder of -1. Modulo division has a lot of everyday uses. Obviously time and date stuff is a big example, but it also becomes relevant for things like splitting up food and figuring out how much will be left over. Try to think of a couple examples in real life. As a quick project, go back to your lesson four project – the methods deep dive project where you created several calculator-type methods like add, subtract, etc., and create a modulo method that takes two integers as parameters,

mods one by the other, and returns the result to your main method where you output it. So your main method would include a line like this:

```
int modulo = modulo(firstNumber, secondNumber);
System.out.println(modulo);
```

Focusing back on the robot, we're working with a degree system with 360 logical degrees, but a range which we'd like to keep in between -180 and 180. We can use modulo to take any heading, and make it between 0 and 360, by modding by 360. When modding by 360, any value between -360 and 360 will not change. (This range is NOT inclusive: -360 and 360 would each return 0, but 359.999 would be unchanged.) Any value with *absolute value* (which you learned about in a prior lesson) of 360 or higher will be brought down to be within the range of -360 to 360, without changing its logical meaning. 450 will become 90, 560 will become 200, etc. Here's an example of a line of code that does this, assuming the heading is stored in a double variable called "heading". You don't need to write this anywhere just yet, this is simply an illustration of the concept.

```
  // Mod the heading by 360 to bring it within the range of -360 to 360
  heading %= 360;
```

This takes us halfway there, but we want to ensure the final output is between -180 and 180, instead of between -360 and 360. If we were to use a range of between -360 and 360, it would actually still work orient the robot properly, it just might turn the long way around – e.g. instead of turning ten degrees left, it might turn 350 degrees right. Considering you would only need to veer one degree off course to then trigger a 359 degree correction, this would be functionally undriveable even though it's "mathematically correct". So we definitely need to reduce our range further.

Logically we can do this range reduction using a couple if statements. We know that any value between -360 and -180 corresponds to a value between 0 and 180. For example, -350 corresponds to 10, -190 corresponds to 170, etc. All we have to do is add 360. The same goes for the range of values between 180 and 360: they correspond to values from -180 to 0. For these values we need to *subtract* 360 instead of adding it. We can use simple if statements to accomplish this – if the value is less than -180, add 360 to it to make it between 0 and 180. If it's greater than 180, subtract 360 to make it between -180 and 0. Here's an example of that, and again this is not something you need to write somewhere yet, but an illustration of the concept:

```
// Bring the heading to between -180 and 180 by adding or subtracting
// one revolution's worth of degrees if it is outside of that range
if (heading < -180) {
    heading += 360;
}

if (heading > 180) {
    heading -= 360;
}
```

These chunks of code demonstrate how to get a heading value to within a desired range using modulo. But how do we use any of this in our robot code? The trick is going to be using our targetHeading variable declared earlier and *comparing it to the current heading*. The difference between

two headings is also a heading. For example, if you have a target heading of 60 degrees and a current heading of 30 degrees, then the difference between those headings is 30 degrees, which is itself a heading. We will add turning from how much output we send to the wheels based on the magnitude of this heading. If the difference between the headings is small, say, just one degree, we'll only add a very small amount of turning power. If it's large, such as 30 degrees, we'll add a large amount of turning.

Modifying Turning Power

You've modified the turning power of the robot before, way back in lesson 2 when you opened the Drivetrain.arcadeDrive method and modified the zaxisRotate variable, and then again in lesson 5 when you added cut power mode. We're going to do the same thing here but instead of scaling it by a static amount, we'll scale it by a dynamic amount based on the difference between the target and current headings as described above. To do this you'll make a method that handles all the logic described in the prior section, but first, here's an example of how you can set up your arcadeDrive method to apply the result. Note how you call a method which will have a dynamic result, and then simply add the value to zaxisRotate. In this case you actually subtract it because of how the math works out, but of course subtraction is simply "addition of a negative number", so that's why you can think of this as "adding turning". You can go ahead and write this code, and then we'll create the getGyroAdjustment method.

```
48    public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
49      double gyroAdjust = getGyroAdjustment();
50      zaxisRotate -= gyroAdjust;
51
52      m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate);
53    }
```

Getting the Gyro Adjustment

With arcadeDrive setup, it's time to apply everything you learned about headings and modulo to calculate the adjustment. It was a lot of reading, but as you saw, a fairly short amount of code, so this method isn't too hard. Follow these steps to complete it. An example is below, but see how much you can complete yourself before looking at the example.

1. Declare the "getGyroAdjustment" method, which should be private and return a double.
2. Declare a double variable "headingDifference" and set it equal the gyro's current z angle minus the target heading.
3. Mod the heading difference by 360.
4. Apply the two if statements shown above to make sure the range of the heading difference is between -180 and 180.

At this point, the headingDifference variable is ready to go. However, if you think about what you added to arcadeDrive, you're subtracting gyroAdjust from zaxisRotate. The final value of zaxisRotate needs to be between -1 and 1, and if it's outside of that range it will simply be truncated to that range. E.g. if you provide a value of 15, that will get truncated down to 1, and a value of -200 will get truncated to -1. Either of these would result in sending 100% turning power to zaxisRotate, which is usually not what you want. If the headingDifference is 1, that means the robot is off from its

target by one degree – you definitely do not want to compensate for a one degree difference by turning at full throttle. So you'll need to apply a scaling factor to the headingDifference in order to get a reasonable final value for your gyro adjustment. Declare a constant in the constants class for this (use the lightbulb to import Constants in Drivetrain.) You can play around with what value you want, but for starters .01 works alright. If you set this value too low, then your code will not do much, because you've scaled the effect down to almost 0. If you set it too high, your robot will start spinning back and forth uncontrollably because it will turn with too much power, overshoot the target, and then turn back the other with full power and overshoot it again, etc. So it's generally wise to start with a low number and bump it up slowly if you need more power, than to start with a high number. Here's an example of the declaration in Constants.java and the line in Drivetrain.java applying the scale. Note how a new variable "gyroAdjust" is declared; this isn't strictly necessary although after you apply the scale, the variable is longer logically a "headingDifference" so declaring a new variable to represent what the value logically is can be a good idea.

```
15    public final class Constants {
16        public static final double GYRO_ADJUST_SCALE_COEFFICIENT = .01;
17    }
```

```
double gyroAdjust = headingDifference * Constants.GYRO_ADJUST_SCALE_COEFFICIENT;
```

After you've done this, the last thing you need to do is use the return keyword to return your newly created gyroAdjust variable. Here's an example of the completed method:

```
private double getGyroAdjustment() {
  double headingDifference = m_gyro.getAngleZ() - targetHeading;
  headingDifference %= 360;

  if (headingDifference < -180) {
    headingDifference += 360;
  }

  if (headingDifference > 180) {
    headingDifference -= 360;
  }

  double gyroAdjust = headingDifference * Constants.GYRO_ADJUST_SCALE_COEFFICIENT;

  return gyroAdjust;
}
```

Testing & Improving

At this point, you can deploy your code, and if you test it, you will notice that your robot will do its best to return to its starting orientation. There are two good ways to test this. The first is to put the Romi on some small object that you can lift up and rotate around, for example a tray or a large plate. If you spin the tray around, you'll see the Romi start spinning the opposite direction, trying to find its original orientation again. It will get pretty close. Our code isn't perfect – for example, it might get close,

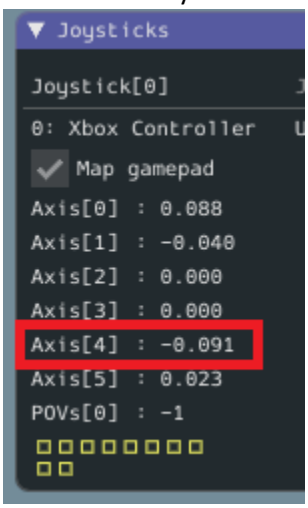
but then the power it sends to try and turn is not enough to break surface friction of the wheels on the tray. There are more advanced things you can do to account for this problem but for today's lesson, this is good enough. Another way you can test is by turning the robot using the joystick. If you do this you'll again see it try to snap back to the original heading. Since the way we're re-orienting the robot is by modifying the zaxisRotate value, you might even notice that you can stall the robot in a different orientation if you apply partial power to the rotation joystick, because you'll be applying a turning power that cancels out the turning correction.

Having your robot continuously self-adjust back to the original orientation is pretty cool! But, if you did the test where you turned the robot using the joystick and it snapped back, you might have already noticed a problem with our code. Sure, you can drive straight, but what if you want to turn? It may be possible using the stalling trick described in the last paragraph, but realistically, you can't turn your robot with this code. The problem is that the robot's target heading is never updated when we want to give it a new target heading. Let's finish out this lesson by making a small improvement to the code to update the target heading whenever we turn. Then the robot will have a new target heading instead of trying to correct back to the old one.

Detecting Turning & Deadband

A first pass at detecting whether or not the driver is turning the robot is pretty simple. In the arcadeDrive method, you have the variable zaxisRotate that you're modifying to change the turning value, but its original value comes from the user's control over the rotation joystick. In theory, if this value is nonzero, the driver is trying to turn. In practice, since it's a physical joystick, at rest it may not measure zero – it will probably be close to zero, but if the driver releases it, it won't return *exactly* to zero. There is a concept in robot control called "deadbanding" which basically means creating a *deadband*, or a zone where inputs get reduced to zero. For example, you could decide that any value between -.1 to .1 should be counted as zero. We'll talk about this more in future lessons when we discuss other ways to improve user input. For now just keep in mind that you'll need to check if the magnitude of zaxisRotate is bigger than some value, and if it is, you can assume that the user is trying to turn. When you're checking the magnitude of a value, that's usually a good use case for using absolute value, which you learned about in a prior lesson. Take your best shot at writing an if statement that checks whether or not the user is trying to turn. As for a value to compare the zaxisRotate to, it will depend on your physical controller. With a newer, stiffer controller that returns to center well, .1 might be a good number. If you have an older controller and the joystick has more play, you might need a bigger number, even as large as .25 or .3. You can use the Joysticks window of in the Robot Simulation interface to check this; find the joystick you're using for control, flick your joystick a few times, and see what number it comes to rest on. Pick a number slightly higher than the largest value you see after a few flicks. With an Xbox controller this is axis 4. Once you have a value, declare it in your Constants file and

use it to finish your if statement.



Once you've created your if statement, the last thing you need to do is adjust the target heading if the statement returns true. This is simple: just set the target heading to the *current* heading. You don't know what the target heading will be when the robot *finishes* turning, but since your if statement will run constantly while the robot is turning, it will keep updating the target on the fly every time. Here's a sample completed if statement:

```
public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
  if (Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE) {
    targetHeading = m_gyro.getAngleZ();
  }

  double gyroAdjust = getGyroAdjustment();

  zaxisRotate -= gyroAdjust;

  m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate);
}
```

Congratulations, you're now done! Test your code. You'll see that after turning, the robot does not snap back to its original orientation, but if you do the same test where you set it on a tray and rotate the tray, it will still rotate. You might notice one issue: after turning, the robot will snap back a little bit in the opposite direction, and then snap back a second time in the same direction you were originally turning. This happens because the target heading is being set *while the joystick is being held*, but it takes a small amount of time to stop turning after the joystick is released due to the robot's current momentum. However, the target heading will be locked in as soon as the joystick is released, meaning that the robot will overshoot its target, and then start correcting for it. There are ways to deal with this problem and we'll look at one of them in lesson 11. For now, you've done enough to complete this lesson.