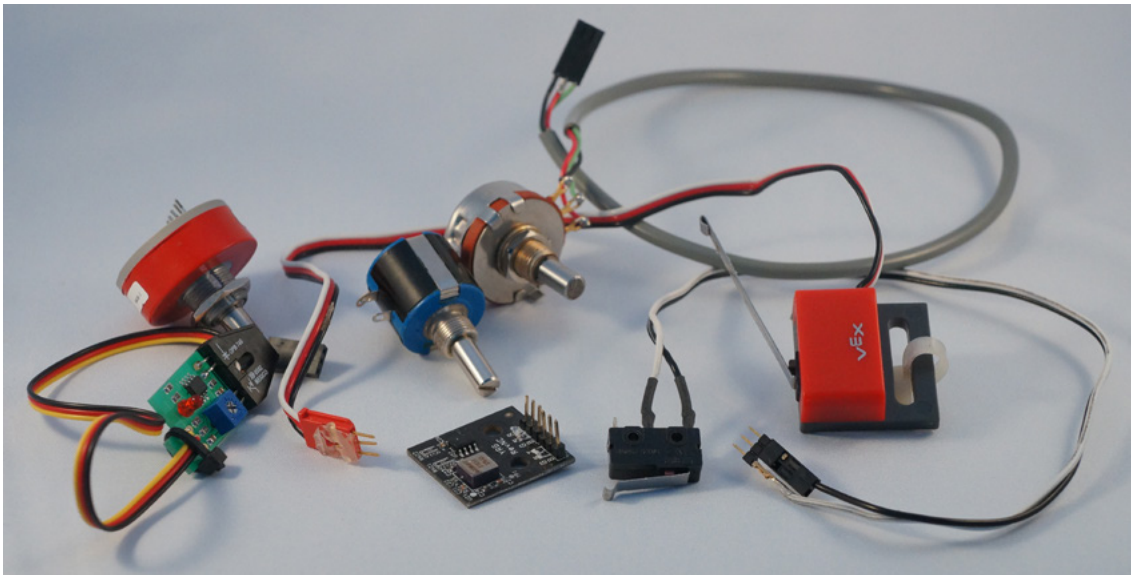
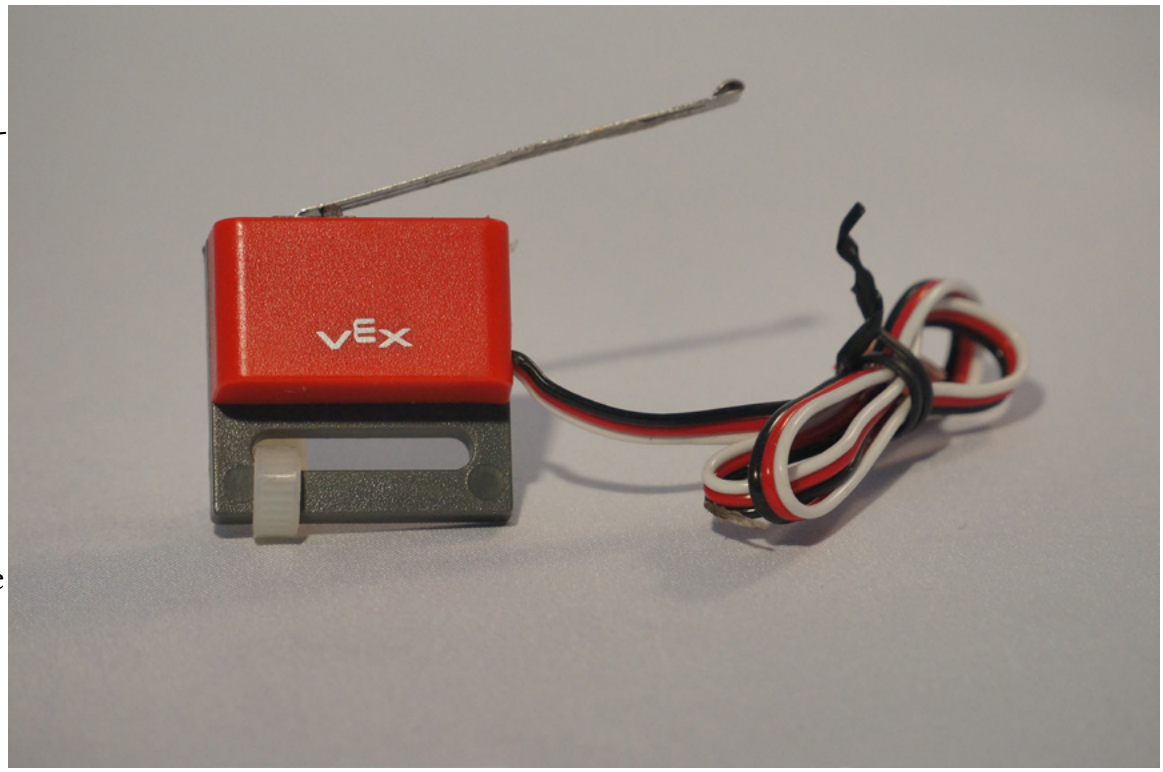
Chapter 6: Sensors and Control

One of the integral parts of a robot that transforms it from a set of motors to a machine that can react to its surroundings are sensors. Sensors are the link in between the physical robot and the code, which allows the robot to make intelligent decisions based on its surroundings. In this chapter, you are going to learn about gryos, encoders, potentiometers, switches, and two types of rangefinders, ultrasonic and infrared. In doing so, we'll learn the basics of how they work and how to use them with the Arduino. Additionally, you'll learn about a couple of ways to control your robot based on input, including proportional control.



6.1: Switches

A switch is, software-wise, one of the simplest of sensors available. A switch can certainly be a complex mechanism, but in the end it is a simple digital input. Take the Vex bump sensors: they are simple switches that return `HIGH` when not being pressed, and `LOW` when they are being pressed. The Vex bump sensor is configured for Normally Open behavior, often abbreviated as either n.o. or no. This means that in the relaxed state, where there is no external mechanical pressure on the switch, the internal contact is open. When the switch is pressed, contact is made, and the voltage is pulled `LOW`.

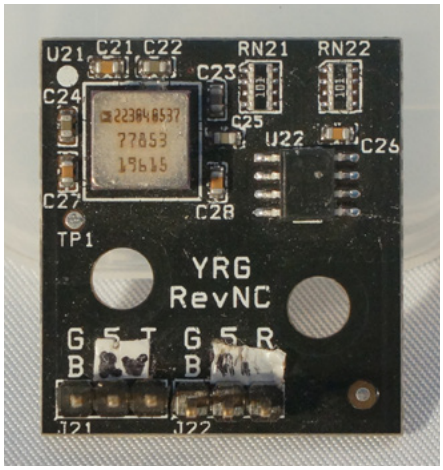


This switch is an example of a sensor that needs to have its port configured to be `INPUT_PULLUP` mode. It's connected between the signal and ground pins on the Arduino. This way, the Arduino input port will be pulled `HIGH` as long as the bump switch is not pressed. When the switch is pressed, the connection is made to ground, and thus the signal is pulled `LOW`. Without the pullup resistor the input is said to be "floating" and will be read as a random value.

6.2: Gyros: Angular Positioning

A gyroscope (abbreviated gyro) is a sensor that measures the rate of rotation, or angular velocity, of an object. This allows you to find the current angle, measured as distance from the starting angle, by integrating this velocity over time. Additionally, gyros have a maximum rate of rotation.

If you rotate faster than this, then the gyro won't be able to keep up, and the heading value computed by integrating will be incorrect. For a detailed explanation of how gyros work, take a look at this page: http://sensorwiki.org/doku.php/sensors/gyroscope.



6.2.1: Using a Gyro with the Arduino

It is not as simple to use a gyro with the Arduino as it is with a servo, as there are no libraries bundled with the Arduino designed to work with a gyro. However, the Arduino Playground, the official Arduino wiki, has an excellent article about using a gryo with the Arduino here: http://playground.arduino.cc/Main/Gyro. Please read, as the rest of the explanations will assume that you have read it.

As you can see from the example, you must start by converting from the integer values given by `analogRead()` to voltage. This is for a few reasons. The first is that the gyro specifications are given with volts as a unit, such as sensitivity, which is mV/deg/sec, or the zero voltage, given in V. Additionally, you want as much precision as possible when calculating position, so converting to floats avoids round-off error.

Next determine the center voltage of the gyro. This is the voltage the gyro will return is there is no movement. By subtracting the center voltage from the gyro output you'll have negative voltage for counter-clockwise rotation and a positive voltage for clockwise rotation.

Next, divide by the sensitivity. This allows you to convert the voltage into a change in position. The value is converted from volts to degrees per second with this division.

After conversion, you need to check to see if there is enough rotation speed to be a valid reading. In the example, the gyro has a lower threshold of 1 degree/second. This means that any values less than 1 degree/second can and should be considered minor fluctuations in sensor, and are ignored. Of course, this means that if you are actually turning at some speed greater than 0, but less than 1 degree/second it will be ignored by this algorithm. This introduces error, an important concept in sensor feedback.

If you have determined that the value is in fact valid, you start by accounting for the fact that you are running this code in a 10 ms loop. At the end of the example, you will see `delay(10)`. This means that after calculating the change, you wait 10 milliseconds, then do it again. The gryo, however, has its rotation values in degrees/second. Therefore, to convert from 10 milliseconds divide the rotation value by 100. Then, add the new value to the accumulator variable, integrating the value con-

verting from rate to angle.

The last step directly affecting the value is checking to see if the accumulator is still within legal bounds for a degree value. For example, 321 degrees is a legal value, but -19 or 378 are not. The sensor has to stay within 0 to 359 degrees, or the sensor has completed a full circle and the value should be modified to be within 0 - 359 degrees. In the example code, this is accomplished by either adding or subtracting 360, but this can also be done by using the modulo operator. This is the modulo operator in C: %. It returns the remainder of dividing the second number by the first. So for example, 2 % 3 = 2, and 3 % 2 = 1. Taking a look at the example values, -19 % 360 = 341, 378 % 360 = 18, and 321 % 360 = 321. Therefore, you can eliminate this block from the example code:

```
//Keep our angle between 0-359 degrees
if (currentAngle < 0)
  currentAngle += 360;
else if (currentAngle > 359)
  currentAngle -= 360;
```

And replace it with this one line:

```
//Keep our angle between 0-359 degrees
currentAngle %= 360;
```

For a more in depth explanation of the modulo operator, take a look here: https://en.wikipedia.org/wiki/Modulo_operation.

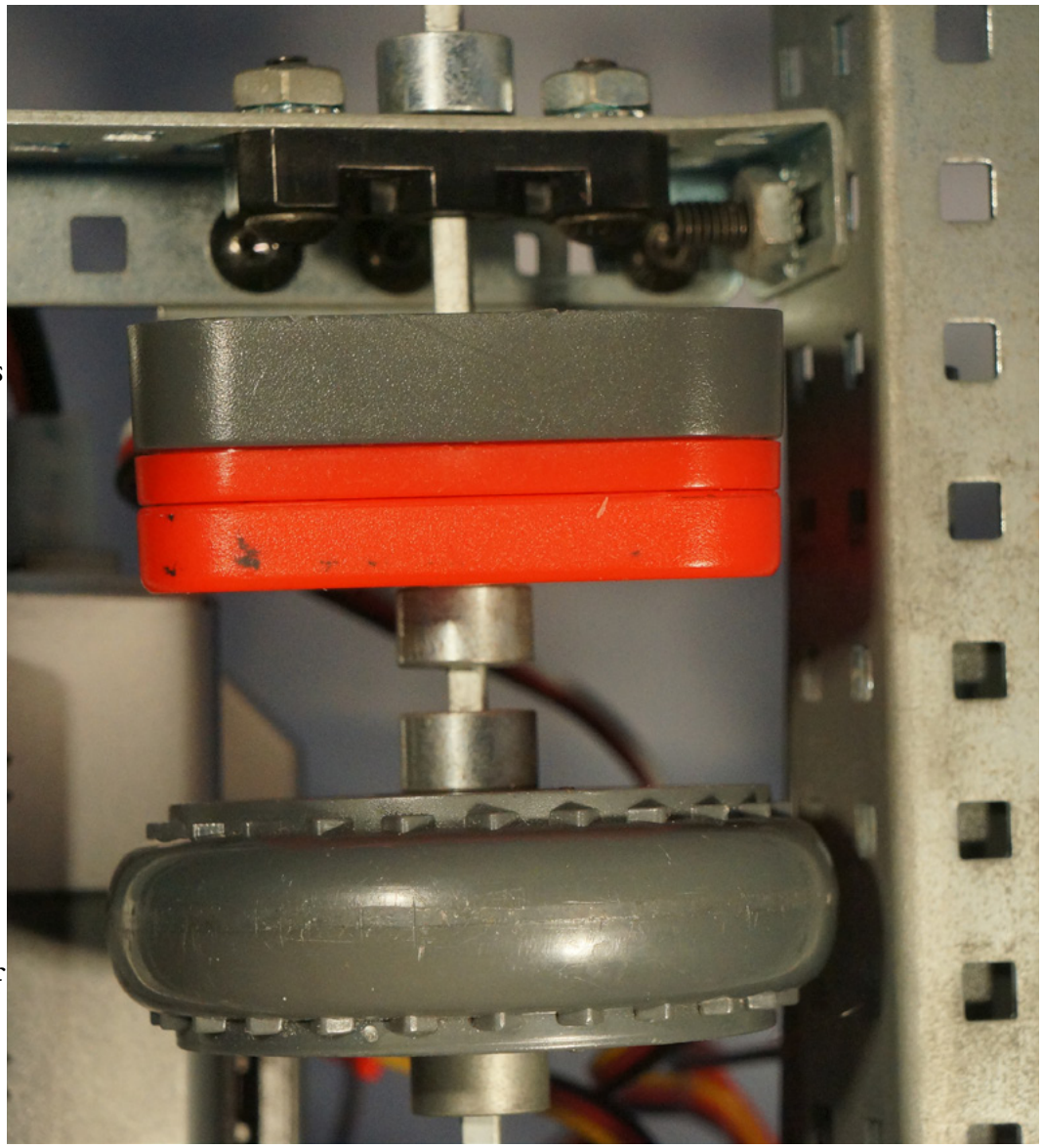
The last things that the example does is to print out the current angle, for debugging purposes, and to delay for 10 ms. This is to ensure that the program maintains the 10 ms loop that was specified earlier. All of the computations up to this point have taken a very small amount of time, small enough to be negligible. Therefore, it is fine just having the Arduino wait for the full 10 ms.

6.2.2: Sensor Error

Now is a good time to bring up one of the most important topics in the use of sensors: sensor error. No sensor is 100% accurate all of the time, and this generally introduces some amount of error into the value that you are reading. With a gyro, for example, if you are rotating slower than the sensitivity, then your value is no longer accurate. Some sensors can be far more accurate than others. You need to be sure that you think about sensor error when making a design, and choose sensors that will give you acceptable levels of error for a task.

6.3: Encoders

Encoders are another very common sensor used in robotics. An encoder measures motion. One of the most common uses of an encoder is to measure the distance a wheel has turned. This allows you to control the robot based on how far it has gone. You can also use this data to determine the speed of what you are measuring. Many car cruise-control systems use encoders to determine current speed. First, you'll learn about how a basic encoder work, and then about some of the different types of encoders.

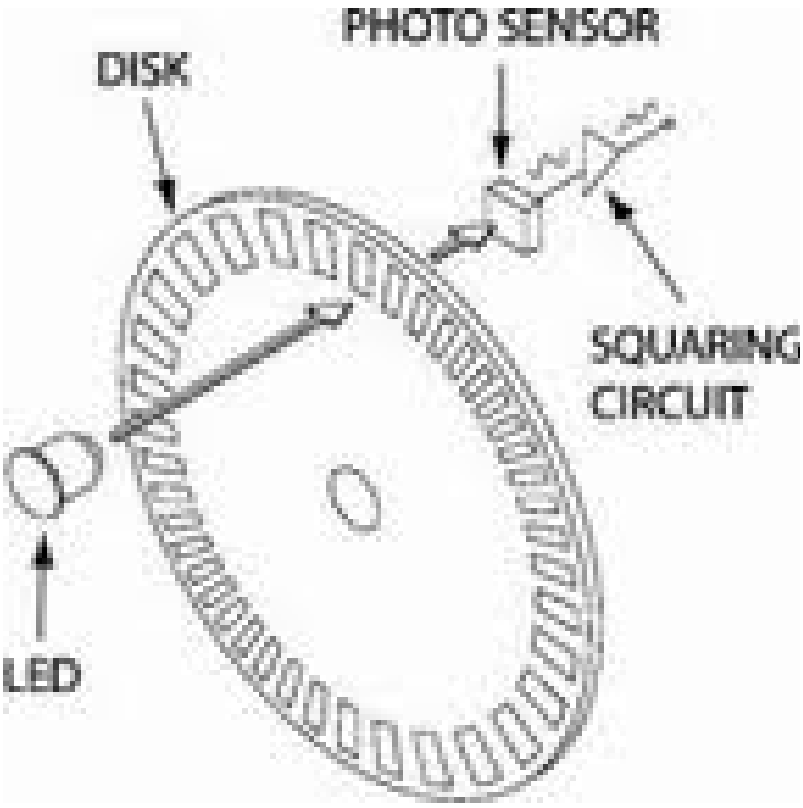


6.3.1: Basic Encoder Operation

All encoders are basically counters. They send a pulse every time a counting event happens. Optical encoders, for example, count pulses of light, sending an electrical pulse back to the controller when a counting event happens.

Here is a picture of an optical encoder. The ring is mounted to the rotating object you are trying to keep track of, quite often an axle. Every time that a hole passes between the LED and the photo sensor, the photo sensor triggers, and an electrical pulse is sent out. By counting the number of pulses, you can determine the absolute distance from start that you have moved. There are several other technologies that can be used to make an encoder as well. A common type are magnetic encoders. You

can even make encoders out physical switches, incrementing the count every time a switch is hit.

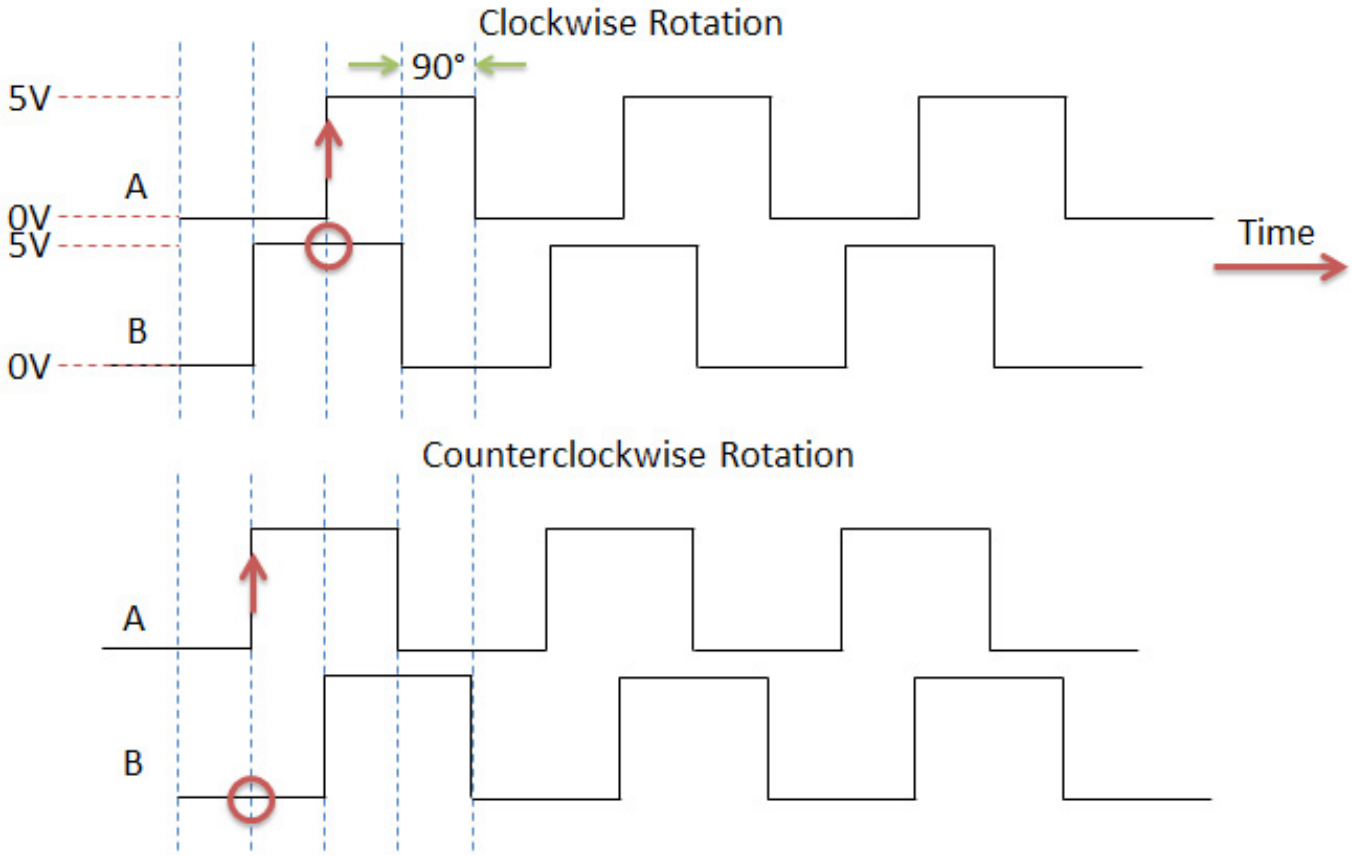


There are two main categories of encoders: rotary and linear. A rotary encoder measures angular movement, such as the rotation of a wheel. All encoders are basically counters. They send a pulse every time a counting event occurs. Optical encoders, for example, count pulses of light, sending an electrical pulse back to the controller each time a slot passes between the LED and the photo sensor. Linear encoders measure linear movement, such as the extension of a construction crane's arm. In addition to classifying encoders by the types of movement measured, encoders are also classified by the type of data they return. Encoders like the one in the picture are called single channel encoders: they have one output and simply pulse when a count is triggered. This allows for absolute displacement from the starting point. This is a very important point: single channel encoders cannot tell you direction of motion. All they can determine is that a count event occurred. If you need to determine direction, there are other types of encoders, including quadrature encoders, which will be covered in the next section. A final category of encoders have a slightly different concept. These encoders are called absolute encoders, and they tell you the absolute angular/linear position of an object, and reset after completing a full range of motion. These encoders can be useful when you have something that can rotate continuously in any direction, but you need to keep track of current angle, not current position. This could be an individual wheel on a swerve drive system, or something more complex. Absolute encoders will also be covered in the upcoming sections.

6.3.1: Quadrature Encoders

As mentioned previously, quadrature encoders allow you to have direction information, not just distance. This gives several abilities, such as allowing you to calculate speed on a drivetrain that moves forwards and backwards, and allows you to do control based on distance travelled. These encoders work by having two outputs, referred to as channel A and B. These channels are offset by 90 degrees from each other, so that if we

are rotating clockwise A will go `HIGH`, and then B will. Take a look at this picture of the output of a quadrature encoder:



As we can see, if A goes `HIGH` while B is `LOW`, the encoder rotation is counter-clockwise, and if A goes `HIGH` while B is also `HIGH`, the encoder rotation is clockwise.

Take a look at this code example for a quadrature rotary encoder on the Arduino: http://playground.arduino.cc/Main/RotaryEncoders. All that you should read is the introduction, example 1, and the further description section. The rest covers using interrupts and classes, which are not being covered here.

As you can see, this is a very simple method of counting the ticks. This example only checks one one of the transitions, A going from `LOW` to `HIGH`. If B is `HIGH` at this point, we know we are going one way. If it is `LOW`, then we are going the other direction.

An important term you may have noticed in the examples is resolution. This describes the number of ticks per revolution of the encoder. If an encoder has a resolution of 30, then that means there are thirty counts every time the encoder completes one revolution. Most encoders have much, much greater resolutions than this, usually between 100 and 6000 ticks per revolution. At 100 ticks/revolution, each tick has a precision of 3.6 degrees. For a 2 inch diameter wheel, with a circumference of approximately 6.28 inches, this means each tick represents .0628 inches of robot travel. This is very precise, usually within most tolerances. How-

ever, consider large applications, with wheel diameters of 2 feet. This is now .0628 feet/tick, or about ¾ of an inch. This may be outside the required tolerance, so more precision is needed. Of course, very small applications need encoders with high resolutions as well, such as in robotic surgery. In that situation, millimeters can mean the difference between a successful operation and rupturing a major artery.
Take a look at this white paper on encoders: http://www.usdigital.com/assets/general/whitepaper_encoders.pdf. USDigital is a large manufacturer of encoders, and this white paper is a very good introduction to the finer details of how they work. Additionally, it will serve as the introduction to absolute encoders, which we will talk about in the next section.

6.3.2: Absolute Encoders

The white paper provided an excellent introduction to absolute encoders. It mentioned that absolute encoders use an optical pattern for determining current position. Two common patterns are called binary and gray code, each of which has a unique code for each possible position. Like quadrature encoders, absolute encoders have multiple different tracks to represent each signal; however, absolute encoders have well more than just two tracks. Here's look at a three-track binary encoder. This means that the program receives three signals from the encoder, which are represented here as three binary numbers, such as 001, or 000, or 110. A three track encoder can sense 23 number of distinct positions, or 8 positions. Since there are 360 degrees of rotation, and 8 positions, the accuracy of this encoder is 45 degrees/position. If the number of tracks is increased to 9, then there are 29 positions, or 512 positions. This is pretty precise, as there is better than 1 degree/position location accuracy. From this example, you should be able to see a few of the advantages and disadvantages of coded absolute encoders. A major advantage over incremental encoders is that from the moment absolute encoders are powered on, they know exactly where they are, since every code is unique and corresponds to only one location. However, it will take a lot of digital inputs to get this information. To get accuracy to within one degree, 9 tracks are needed, which is nine inputs. To solve this problem, many absolute encoders move the calculation of current position onto the encoder itself, and instead relay position information to the controller using a more advanced communications protocol than a simple digital pulse, such USB or SPI. This means that you, as the programmer, have to work slightly harder to interface with the absolute encoder, but the advantages of not having to index on startup and not have to take up a large number of inputs could be worth it.

6.3.2.1: Binary Code

Recall that there are two methods of coding on a coded absolute encoder. The first is called binary encoding. As a note, despite the fact this system is called binary and the other method is not, they both are binary in the sense that each code is made up of only 0's and 1's. However, binary encoding follows the binary number system, whereas Gray encoding does not.

Binary encoding, as it sounds, goes from position to position by counting in the binary number system, adding 1 for each step. Here's a three track binary code:

000	100
001	101
010	110
011	111

To generate each new code, simply add 1 to the previous code. This is nice and simple, but using this code can produce a few problems. If you look at the code, there are three stages where the number changes more than one bit at the same time: 001 -> 010, 011 -> 100, and 101 -> 110. While it's convenient to think of the digital pulses as being perfect square waves and instantly switching on and off, this is not the case. If bit 2 takes a little longer to switch than bit 1, and we check the encoder (called sampling) at the wrong time, then this transition: 001-> 010 could look like this: 001 -> 000. Since it is possible to have the program sample only when a change occurs, it will check when bit 1 changes, even though bit 2 is not done changing yet. To correct this problem, most absolute encoders use a different code, called Gray code.

3.3.2.2: Gray Code

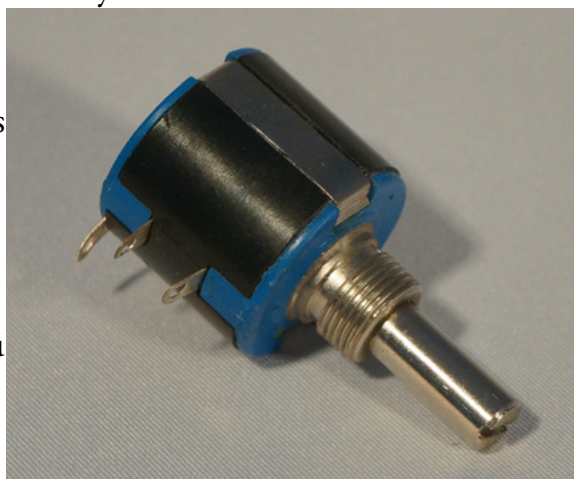
Gray code is a slightly different coding system that helps solve this problem by having every transition change only one bit at a time. Here's the Gray code for a three track absolute encoder:

000	110
001	111
011	101
010	100

As you can see, it never changes more than one bit when moving from state to state. This means that if the program only samples when it detects a change, there's no issue, as you know that this is the final position and not some transitionary state. Because of this, Gray code is used far more often than binary code.

6.4: Potentiometers

Potentiometers have already been briefly mentioned, but it's time to go into more depth on what they are, how the work, and what you would use them for. This is a picture of a potentiometer:



As mentioned earlier, a potentiometer is a variable voltage divider. Internally, the potentiometer consists of a wiper, controlled by the knob you can see on the top. By rotating the knob, you adjust the voltage divider. You then run a voltage through the outer two pins, and read the voltage off the middle pin. This voltage changes as the knob moves, allowing you to know the position of whatever is attached to the potentiometer.

Potentiometers can be among the simplest of sensors, but they are also very powerful. They are very useful in situations where you have a moving joint that has a constrained amount of movement. Take the arm on your base-bot, for example. It has around 180 degrees of motion. This makes it perfect for a potentiometer. It can be mounted directly to the rotation shaft, and you can then read the angle of the arm at any time.

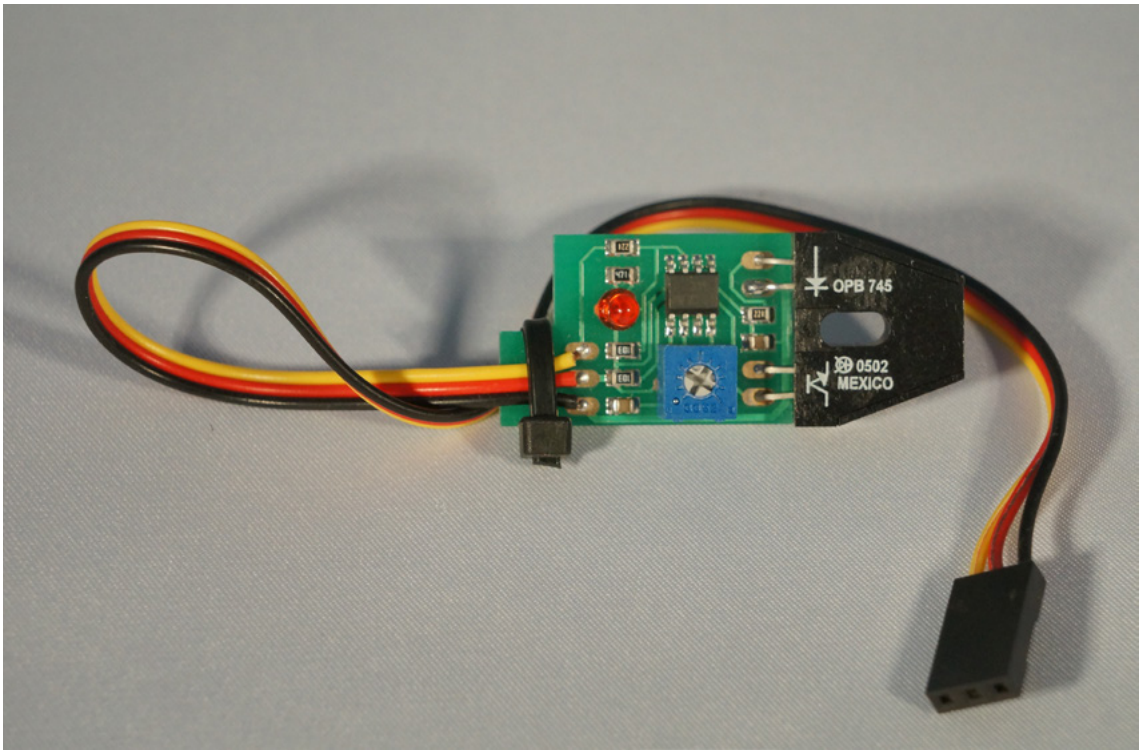
There are several different types of potentiometers. The most common ones used in robotics are rotation and linear potentiometers. Just like encoders, rotation potentiometers measure angular position, and linear potentiometers measure linear position. There are other other types of potentiometers; Wikipedia has an excellent article that you can find here: https://en.wikipedia.org/wiki/Potentiometer.

Rotation potentiometers are differentiated by the number of turns they have. Some, such as the one you used on your base-bot's arm, are a one-turn potentiometer. This means that they have 360 degrees (usually a little less) that they can rotate. If you rotate a potentiometer past the end, you'll end up breaking it. Another common turn-amount is 10 turns. Make sure that you use a potentiometer with enough turns, or you'll end up breaking it. It's usually a good idea to leave some space at both ends, just in case the joint you are trying to measure goes over your expected stopping point.

Potentiometers have several advantages. They can be quite small, and very efficient. Potentiometers also have several advantages over incremental encoders, they know their position upon startup. And unlike coded absolute encoders, they just return a simple voltage, so you can just use an analog port. Because of these advantages, it is usually a good

idea to use a potentiometer in place of an absolute encoder if you don't need the ability to spin freely.
Just as absolute encoders can spin freely, there are continuous rotation potentiometers. Like an absolute encoder, these potentiometers can spin freely, and reset upon completing a full rotation. These potentiometers are more expensive, though. They have another disadvantage: There is a small area between the maximum voltage and 0 volts where there is a short between source and ground, and you read an open circuit. If you use a continuous rotation potentiometer, make sure that you take is gap into account.

6.5: Rangefinders



Knowing how far your robot has travelled is very useful, but now it's time to talk about sensing how far your robot is from other objects. These types of sensors are often called rangefinders, as they help us find our distance, or range, from other things. Two main types of rangefinders: ultrasonic and infrared. However, they both rely on the same basic principles.
Ultrasonic sensors send out a pulse of high-frequency sound and measure the time it takes for the pulse to come back, which works on exactly the same principle as a bat's echolocation. Infrared rangefinders sends pulses of infrared light, and it used a sensor that can tell where the light is returned. Based on the location on the return sensor, it is possible to construct a triangle to tell where the object is.

6.5.1: Ultrasonic Rangefinders

Ultrasonic sensors consist of a speaker and a microphone. It starts by sending out an ultrasonic sound pulse. It then times how long it takes for the echo to return, and that time tells the code how far away the closest object is. This works well in both air and water, but since the speed of sound in water is different than it is on the air, you need to change the math depending on which situation you are working in.

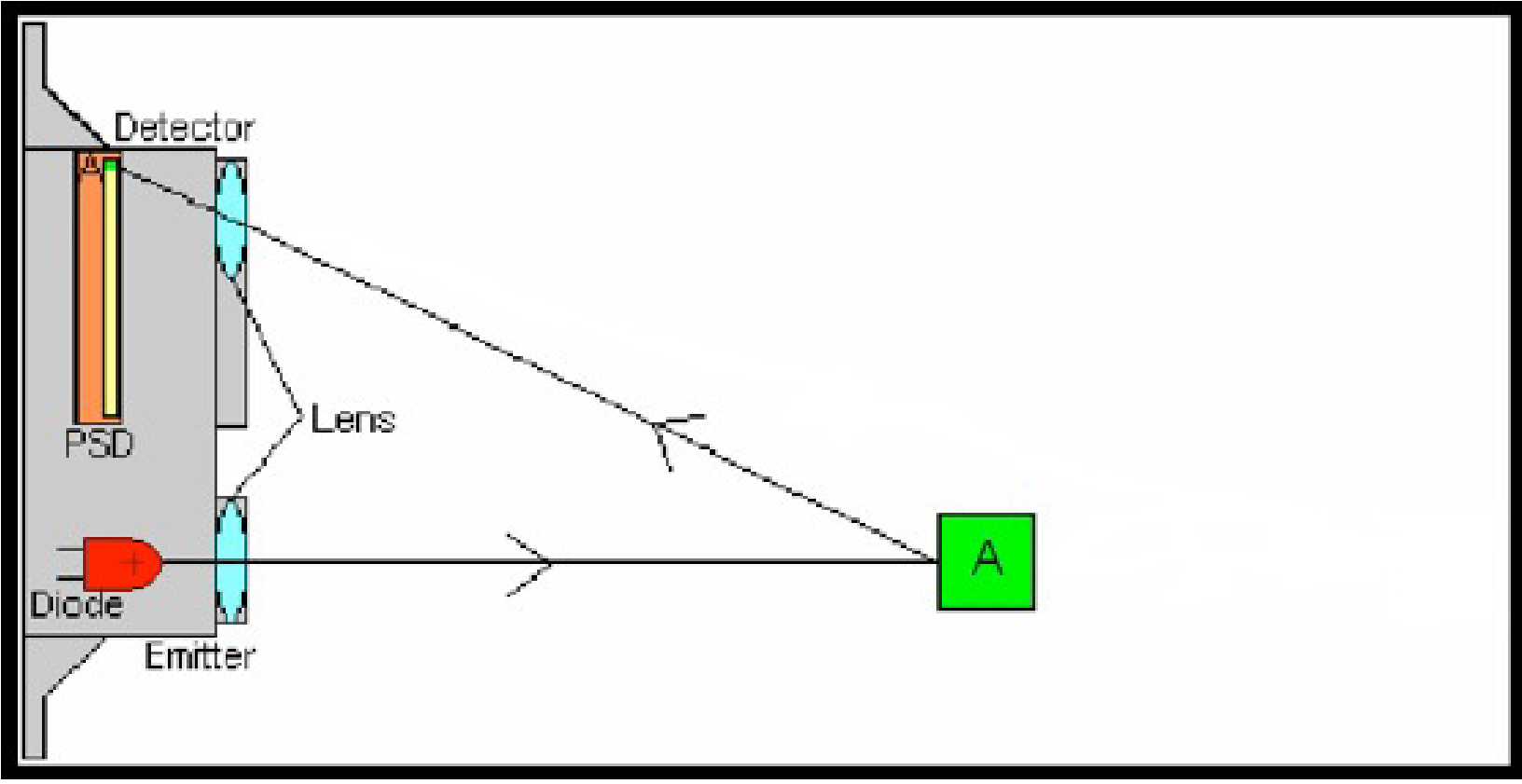
In the air, the speed of sound is approximately 343 m/s, or 1125 ft/s. To calculate distance away, start by measuring the time it take the sound to hit the object and come back. Since this is the round trip time, you only want half of the time, so divide the time by 2. Next, multiply the speed of sound by the time, and this result is the distance. As you may be able to guess, this requires a very accurate timing mechanism, as the time sound takes to travel 1 foot is 1/1125 of a second, or approximately .99 milliseconds. The Vex ultrasonic rangefinders are capable of measuring distances as small as 1.5 inches. The time it takes for an ultrasonic pulse to travel this far is about .09 milliseconds. That's about 1/8th of the time that a baseball remains in contact with a bat as it is hit.

One of the ultrasonic sensors available in the lab is a Maxbotix ultrasonic rangefinder. There are multiple ways of communicating with this sensor. It has serial communications support, and it will communicate the distance in ASCII code. It also has the ability to return a raw voltage, which can be converted to inches with a factor of ~9.8 mV/inch, when running at 5V. The Vex ultrasonic sensors are more complex to program, and we don't use them with the Arduino.

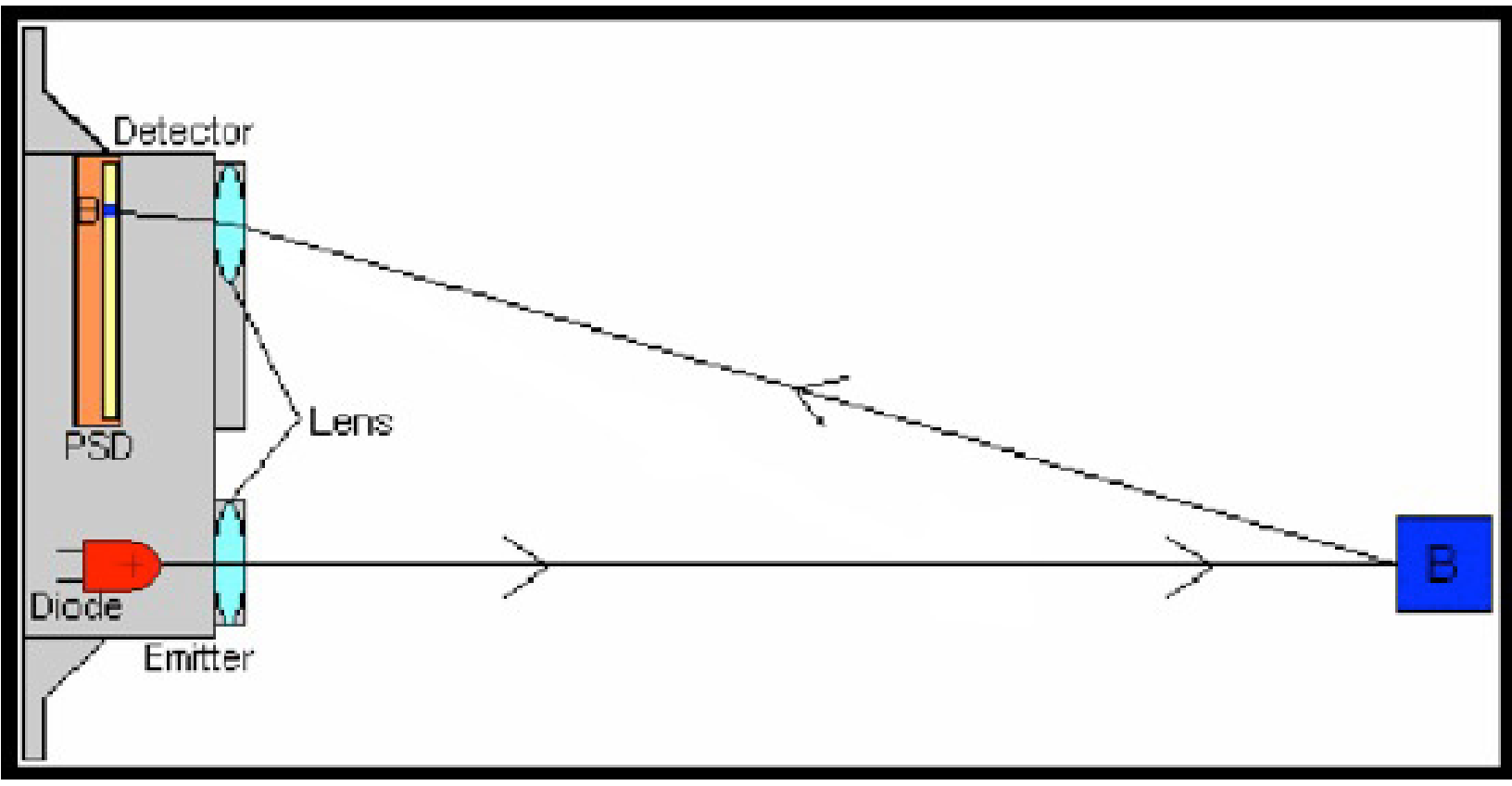
6.5.2: Infrared Rangefinders

Infrared rangefinders work on a slightly different principle than ultrasonic rangefinders. They don't measure the time it takes for a light pulse to bounce and come back, as it is way too fast. Take something that is .5 feet away. An ultrasonic sensor can accurately measure this, as it takes almost a millisecond to travel this distance and come back. By comparison, light takes about 1/1000th of this time, at ~1 nanosecond. Increase the total distance traveled to 3 meters, and the round-trip time is only up to ~3.3 nanoseconds. This time difference is incredibly small, so there needs to be a different solution to measure distance with light.

To solve this issue, infrared rangefinders instead measure the angle of the returning light. Infrared sensors have an two ports: an emitter and a detector. The emitter sends a tight beam of infrared light out, and the detector measures the angle at which it returns. With that angle, you can construct a 90 degree triangle, and then use trigonometry to figure out the distance from the sensor.



Here, the light leaves the emitter and hits A. It then returns, hitting the PSD (which is a light sensor) at a specific point. This will return a certain voltage, which can then be converted to a distance.
Here's another image, with a different target farther away:



As you can see, it reflects on a different point on the sensor, which returns a different voltage.
Here's an excellent explanation of one of the most popular infrared rangefinders, which is available in the lab for use on your robots: http://www.societyofrobots.com/sensors_sharpirrange.shtml.
As you just read, infrared rangefinders have a couple of important issues. The first is a minimum range distance. If you are within this distance, then the values that you will get are ambiguous, and there is no way to resolve to one of two distances just looking at the value. A couple of solutions to overcome this exist. You can have multiple sensors that cover each other's blind spots, or you can position the sensor so that nothing can get into the bad value zone.
The second issue is that the output of the sensor is non-linear. This can be very difficult to deal with. The article also mentioned two ways to deal with this problem: you can create a lookup table with commonly used values. This works fine if you just want to look for a couple of values, and take action when you see those values, but such a lookup table can get very, very big very quickly. The second solution to this issue is to create a function that models the values of the sensor. This can be done using programs such as Mathematica or Maple, but we won't be going into how to do this in this book.

6.6: Control

Now that you know about all of these sensors and how to get information from them, it's time to talk about how to use them in your programs to control a robot. Switches are among the simplest sensors, and using them to provide control is very easy: just wait for the switch to be pressed (or released, depending on your situation) and then take an action. For example, you might have a limit switch on an arm. When the arm gets to its maximum reach, it hits the limit switch, and your program stops moving the arm farther. You might use code like this:

```
// Create the arm motor variable
Servo armMotor;

void setup() {
     // Set up the servo and the port for the limit switch
     // The limit switch is a pullup,
     // so it is normally HIGH, and switches to low
     // when pressed
     armMotor.attach(6);
     pinMode(1, INPUT_PULLUP);
}

void loop() {
     // The variable joystick is assumed to be input from a
     // joystick/other input device.
     // We are not actually getting the input to simplify the code,
     // but imagine that we are
     // Take the opposite of the input on pin 1.
     // If it is HIGH, then move to the else
     // if it is LOW, then the limit switch is pressed,
     // so control based on whether
     // the arm is moving the right way.
     if(!digitalRead(1)) {
          if(joystick <= 90) {
               // If we are moving the right direction (90 or less)
               // drive the motor based on the input
               armMotor.write(joystick);
          } else {
               // The program is trying to go the wrong way, so set it
               // to stop
               armMotor.write(90);
          }
     }
// The arm is not at the limit, so we don't care
// about checking for going over
else {
          armMotor.write(joystick);
}
```

This is a simple example of using a limit switch. Limit switches are very easy to use, as state is digital and all that is needed is a simple read. However, it is possible to have much more complex control based on the input of other sensors.

6.6.1: Proportional Control

Proportional control is a very simple concept, but very powerful when used correctly. Suppose you are driving a car approaching a stop sign. As you get closer to the sign as read by the sensors (your eyes) the car speed decreases. The difference between the cars current position and the desired position (the stop sign) is called the error. How fast should the car drive as it approaches the stop sign? Ideally the speed is proportional to the error, i.e. the closer the car is, the slower it goes until it gets to the sign. At that point the error is zero, and the cars speed goes to zero.
For example, suppose there is a robot with left and right motors and a gyro. This robot starts out at an angle of 0 degrees, and should turn to an angle of 90 degrees. The turn towards the target angle (90 degrees) should be at a rate proportional to the difference between 90 degrees and the current heading. When the robot starts off at 0 degrees it turns fast, and as it approaches 90 degrees the turn rate decreases. To accomplish this, you might use code like this: (This example assume an function called `readAngle()` that returns the current gyro heading as an int and the motors are set up).

```
void loop() {
     int targetAngle = 90;
     int error = targetAngle - readAngle();
     error += 90;
     leftMotor.write(error);
     rightMotor.write(error);
}
```

In this program the error is computed by subtracting the target angle from the current angle read from the gyro. This is a value that is zero if the robot's heading is 90 degrees, and positive or negative if it's off course right or left. The magnitude of the error is how far from the desired heading the robot is facing.
Remember that motors speeds are 90 for stopped and greater or less than 90 for clockwise or counterclockwise. Since the program should stop the robot when the error reaches 0, it needs to fit this equation: 0 (error) + x = 90 (motor stop). It therefore adds 90 to the error to satisfy this equation.
There could be a slight problem with this code, however. As the robot approaches 90 degrees, it might no longer have the ability to actually turn the motor. This is because values close to 90 don't necessarily have the ability to actually turn the motor. To compensate for this, there is the proportional constant, commonly abbreviated as the `Kp` The error is multiplied by `Kp`, and then centered on 90. Therefore, as you get smaller and smaller values, the p value boosts them up just enough for the motors to get to the final destination. Tuning p values is often best

done manually through trial and error. There are formulae to to figure out what the best values are, but these will be covered in more advanced classes such as controls.

The new code with a `Kp` value of 2 would look like this:

```
void loop() {
     int targetAngle = 90;
     int Kp = 2;
     // read the current angle
     int curAngle = readAngle();
     // compute the error as described in the text
     int error = targetAngle - curAngle;
     // Multiply the error by the p constant
     error *= Kp;
     error += 90
     // Since we are multiplying the value by 2,
     // it could be too large, so we want to constrain
     // it to be no greater than 180 and no less than 0
     // We don't use modulo here because you could have
     // a final value of -1, which is
     // full speed turn to the right.
     // However, -1&180=179, which is full speed
     // turn to the left
     if (error < 0) {
          error = 0;
     }
     if (error > 180) {
          error = 180;
     }
     leftMotor.write(error);
     rightMotor.write(error);
}
```

As you can see here, this code will approach 90 more slowly, giving more power up until the very end.

Proportional control works for any of the sensors covered in this chapter with the exception of the limit switches. For all of these sensors, the error is target_value - current_value. In all cases, you will have to convert the range given to the range that will drive your motors, which in the case of Arduino servo's is 0 to 180.

6.7: Conclusion

This ends the chapter on sensors and control. To recap, this chapter looked at the basics of how various sensors including limit switches, gyros, encoders, potentiometers, and ultrasonic/infrared rangefinders work, and how to use them with the Arduino. You then learned how to use limit switches for control, and a more complex method of control called proportional control.