Summary of Week Four

We did not get quite as far as we would like for week four, but we are still doing fairly well.

I had built RobotMain.java as my primary class, which I placed in package org.phs.code.robot. We initialize our state values and then we enter into a while loop that only terminates when our vehicle’s x and y coordinates make it to within 15 centimeters of the target. This is an arbitrary value, we chose it in the session. We also discussed how to compute it. We are going to subtract the x ordinate of the vehicle from the x ordinate of the target and the y ordinate of the vehicle from the y ordinate of the target, and then use the Pythagorean theorem to compute the length of the hypotenuse. That becomes the radius of the circle containing the target and the center of our vehicle.

I had you set up StateAssessment.java as a second class. It had two methods, one that set up the initial conditions and one that checked to see if the robot was at the target. You initialized this class early in your main module by calling its constructor:  
  
StateAssessment state = new StateAssessment();

With that set up you were able to call the init() method as state.init();.

With these set up, we were able to create a while loop to contain and control our code.

while (!state.atTarget()) { 🡨 this method checks to see if we should stop

// We are going to do some primitive testing here

X += 1;

Y += 1;

State.setPose(x,y,0); 🡨 I use x, y, and, z, but z is always 0 and we aren’t turning

}

This week we built on this by adding a new class, which we called Listener.class. What I prepared as I was getting ready for tonight’s session I will put a copy on the Google Drive, along with the logging program I use. The Logging.java file is just a class file that was created by the PowerHawks.

I was able to set up two listeners in the one class. This could be done in one class by passing the pin in the constructor. Here I am treating the two encoders as one device.

-------------------------------------------------------------------------------------------------------------------

**public** **class** SparkFunListeners {

**private** **long** countA;

**private** **long** countB;

**public** SparkFunListeners() **throws** InterruptedException {

**final** GpioController gpio = GpioFactory.*getInstance*();

Pin pinOutA = RaspiPin.***GPIO\_07***;

Pin pinOutB = RaspiPin.***GPIO\_08***;

PinPullResistance pullOutA = PinPullResistance.***PULL\_UP***;

PinPullResistance pullOutB = PinPullResistance.***PULL\_UP***;

countA = 0;

countB = 0;

**final** GpioPinDigitalInput outA = gpio.provisionDigitalInputPin(pinOutA, pullOutA);

**final** GpioPinDigitalInput outB = gpio.provisionDigitalInputPin(pinOutB, pullOutB);

outA.setShutdownOptions(**true**);

outB.setShutdownOptions(**true**);

outA.addListener(**new** GpioPinListenerDigital() {

@Override

**public** **void** handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent event) {

countA += 1;

}

});

outB.addListener(**new** GpioPinListenerDigital() {

@Override

**public** **void** handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent event) {

countB += 1;

}

});

}

Notice that the function GpioPinListenerDigital() { } is actually included right in the code that defines the listeners for the two Gpio Pin definitions. They could be set up as separate functions, but they are short and sweet. In modifying the sample code, I removed the Main() and put a constructor in its place. You can review constructors in Sololearn, but they are basically a method with the same name as the class that are called when the class in first instantiated.

If you stick with automation coding, those exception classes are going to become very important. Your code can’t quite or your robot is in danger. You have to figure out what can cause your program to fail and come up with means to recover from the failure. Right now, it is just going to fail, because we haven’t really handled the exception – that will be another summer or winter seminar all by itself.

What that does do is allow me to declare my class variable countA and countB, which are incremented whenever the pin goes to a high state.

I also created get methods and placed them after the constructor. Note that getCounts() returns an array of two values. We will use this more and more as we get deeper into the lesson. In my code, I would get the data from the encoders by executing

Int[] counts = listener.getCounts();

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**public** **long**[] getCounts() {

**long**[] counts = {0,0};

counts[0] = countA;

counts[1] = countB;

**return** counts;

}

**public** **long** getCountA() {

**return** countA;

}

**public** **void** clearCountA() {

countA = 0;

}

**public** **long** getCountB() {

**return** countB;

}

**public** **void** clearCountB() {

countB = 0;

}

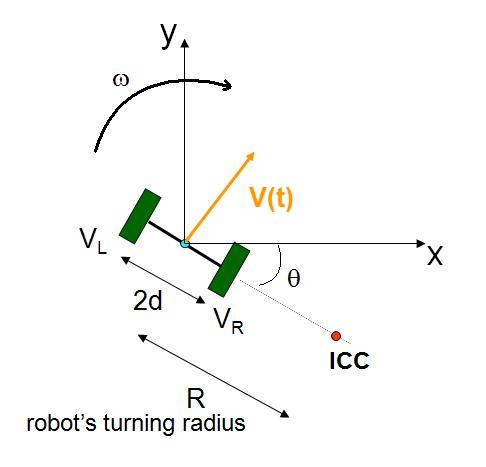
So, once we get this class executing, we will have our counts. How do we use them? Our problem is figuring out when our vehicle has made it a distance of 424 centimeters, plus whatever it takes to get around the central object, sure, but what if we got that wrong? All we would be doing is driving a certain distance and stopping. With the technology that we have placed on these vehicles, as cheap as it is, we can actually drive around an unknown field and still make our goal, just as long as we know where we are.  
  
To make this work, we are going to have to even out our loops and have things calculated across the same interval. This will allow us to pretend that the short segments we measure are able to be integrated into a set of equations that approximates our position on the field accurately. Rather than keeping track of the distance travelled, we can keep track of the distance left to go.  
  
If we have done this right, when we read the output from our encoders, we will find out that one rotation of the motor produces exactly 4 highs on the pins we are monitoring. If, by chance, we started out with our vehicle facing the target head on and drove until both wheels were lined up on either side of the target, we would have covered 424 centimeters with a couple of millimeters left over. Our wheels have a circumference of 20.42 centimeters, so they would have gone around 20.77 times. With them going around 20.77 times, the disks would have gone around 48 times that, or 997 times. Our sensors, with their four magnets, two poles each, would have triggered our input pin 3,989 times.

In this scenario, to get our vehicle to stop at the target, we could simply put the vehicle down facing the target, set both motor speeds the same, count to 3.989 ticks of the encoder, and tell the motor to stop. Since we have to steer the vehicle, we are actually going to use those ticks and a little geometry to figure out where on the field we are.

We are going to borrow another bit of math here. Of necessity, our two wheels share something called the Instantaneous Center of Curvature (ICC). This is a point going through the center of the wheels, the distance of which is determined by the relative speeds of the two motors and the direction of rotation.  
  
If the left wheel is not moving and the right wheel is moving in a forward direction, the ICC is the exact center of the left wheel. The right wheel will execute a complete circle around the left wheel that is equal to twice the length of the base of the wheels times pi(). The same relationship holds if the right wheel is stopped.  
  
If the left wheel is moving forward and the right wheel is moving backwards, or the other way around, the vehicle will pivot on its center. This time the distance travelled will be equal to the length of the wheel base time pi().  
  
In the case where both wheels are moving, but one is moving more slowly than the other, we can pretend that they are driving the circumference of a much larger circle and the current wheel velocities will hold true around the whole circumference of the circle. All we have to know is the distances that the two wheels have covered at any given instance – hence the meaning of the Instantaneous Center of Curvature. At the time we make our measurements, we have a count for a small fragment of time where we know the relationship between these two counters.

The concept is described here, and also at many other places on the web.  
  
<http://cse17-iiith.virtual-labs.ac.in/forwardkinematics/index.php?section=Theory>

In coding, we have a choice to make. Do we want to extract the formulas for the individual elements? Or do we want to code this as Matrix math?



Omega is the rate of rotation around the ICC. Theta is the angle from the x axis. R is the radius of the turning circle. In this case the circle itself is to the center of the axle. X and Y are the left to right axis and the front to back axis. V(t) is the velocity at the current time. The Velocity of the Left Wheel and that of the Right wheel are determined by the value of Omega and are equal to Omega (in Radians per second) times (R + d), for the Left Wheel, and Omega times (R – d), for the Right Wheel.

**long** dT = currentTime - prevTime;

**double** delta\_left = (leftCount - prevLeftCount) / ticks\_per\_centimeter;

**double** delta\_right = (rightCount - prevRightCount) / ticks\_per\_centimeter;

omega = (delta\_right/dT - delta\_left/dT) / l;

R = 0.5 \* ( delta\_right/dT + delta\_left/dT)/(delta\_right/dT - delta\_left/dT);

At this point we can use geometry to figure out our motion on the field and the vehicle orientation. As long as we keep that dT fairly constant, we can make use of the rate of change that we expect out of the system to keep our program in line.