B355 Manipulation Report

Robert Kellems

Introduction

For this project, we were tasked with building and writing code for a robotic limb consisting of two joints that could pick up three soda cans in a row in a completely autonomous manner. The range in which these cans could be placed had to be long enough to include at least four cans at one time. In addition, we needed to include safety measures in the mechanical and/or software design that would prevent the claw on the end of the arm from hitting the table. We built a manipulator which consists primarily of an upper arm made up of two rails and a forearm made of one rail, each powered by a gear system designed to provide sufficient torque. Our code makes use of both inverse kinematics equations and our own observations to take the horizontal distance at which the can is located and calculate the point in space at which the claw needs to be in order to grab the can. Upon grabbing the can, our robot drops the can off to its behind and continues to pick up cans until there are no more left in its range. We both use limit switches and rely on our predictable range of potential values to prevent the claw from hitting the table.

Mechanical Design

We did not have to make many major revisions as we were building our manipulator, although there were some adjustments made along the way to help combat issues that we encountered. Perhaps the main change that was made was shortening the upper arm in order to make the length more comparable to that of the forearm, thus giving the robot better range; this was done by simply sliding the metal rails extending out from the metal rails connected to the body of the robot inward and reattaching them there. The pair of metal pieces which bind the two sides of the upper arm were also not originally a part of our design, but were added later once we

realized that the two rails had the tendency to move outwards and thus needed something added for stability. Other than these few issues, the designing and building of the manipulator went smoothly and required very little backtracking.

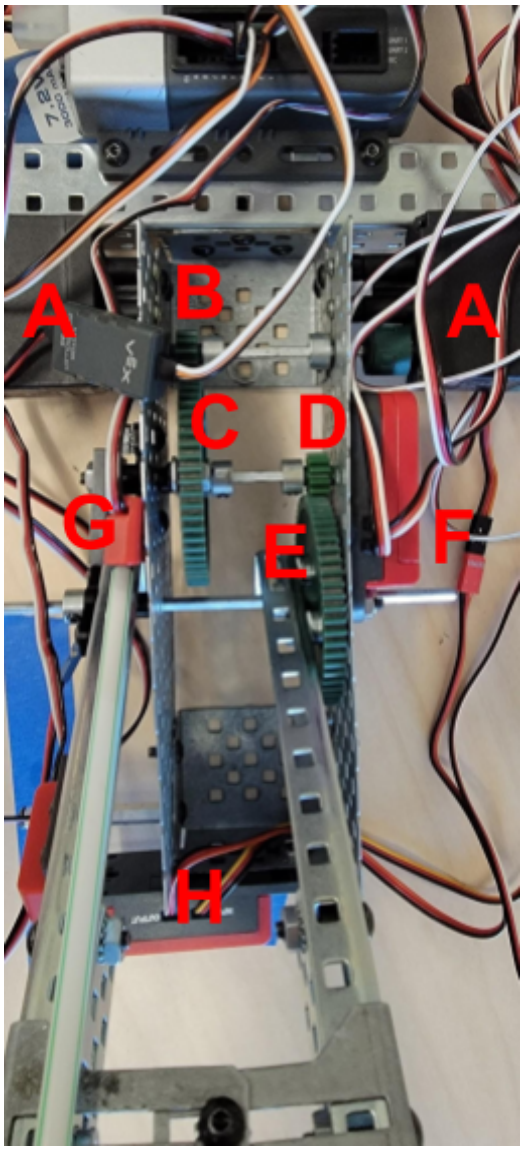

Fig. 1. The enclosure containing the gears for the shoulder joint, viewed from above.

The shoulder joint of our robot is powered by a system containing two motors ("A" on Fig. 1) and a group of gears connected to a rectangular structure made of various metal pieces. The two motors are connected to each end of one beam near the back of the base. On this beam

there is a 12-tooth gear (“B” on Fig. 1) which is connected to a 60-tooth gear (“C” on Fig. 1) on another beam; this next beam also has a 12-tooth gear (“D” on Fig. 1) on its opposite end which connects to a 60-tooth gear (“E” on Fig. 1) on yet another beam, this beam being the one that is connected to the two rails that make up the upper arm. Each pair of 12-tooth and 60-tooth gears would provide 5 times the amount of torque provided by the motor itself on their own (since the gear ratio is 60:12 = 5:1); with both of them being connected, our gear system provides 5 * 5 = 25 times the original amount of torque. The shaft encoder (“F” on Fig. 1) which we use to track the movement of the shoulder joint is attached to the middle beam in our gear system on the left side of the structure enclosing the gears. On the right side of the enclosure is a limit switch (“G” on Fig. 1) with a straw that points upward; the purpose of this switch is to kill the two motors if the upper arm begins to move too far back. The sonar sensor (“H” on Fig. 1) which we use to find the distance of a can from the manipulator is attached to the front face of the enclosure. It is also worth noting that our microcontroller is attached to a base made of metal rails which is attached to the back of the gear enclosure (Fig. 2); this could provide some extra stability for our robot, although it’s unclear to what extent it helps.

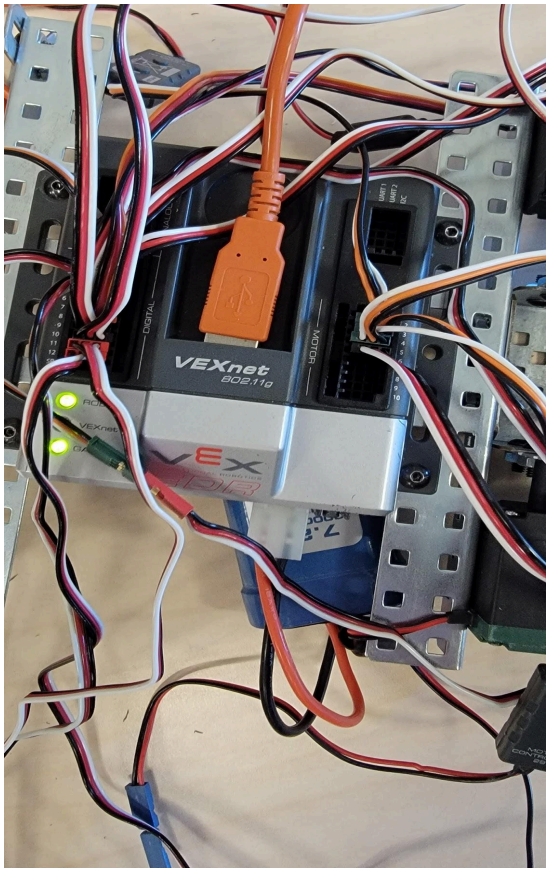

Fig. 2. The base holding the microcontroller, viewed from above.

As mentioned previously, the upper arm of our manipulator primarily consists of two metal rails (“A” on Fig. 3), each attached to one of two sides on the final beam in our gear system. Two small T-shaped pieces (“B” on Fig. 3) are attached to each rail and are screwed together in the gap between the two in order to provide stability. Connected to the right side of the right rail is a limit switch pointing downwards (“C” on Fig. 3); if the shoulder moves too far down, then this limit switch is triggered by a beam sticking out of the right side of the enclosure. An additional metal rail (“D” on Fig. 3) is tucked into each of the metal rails attached to the gears to extend the length of the upper arm. These additional rails hold the gear system which powers the elbow joint of our manipulator; the left rail hosts a shaft encoder (“E” on Fig. 3) and the right rail hosts a motor (“F” on Fig. 3), both of which are connected to a beam which has two

12-tooth gears (“G” on Fig. 3) near the center. Each of these 12-tooth gears are attached to a 36-tooth gear (“H” on Fig. 3) located on a beam higher up the arm which also hosts a pair of 12-tooth gears (“I” on Fig. 3). These 12-tooth gears are then connected to 60-tooth gears (“J” on Fig. 3) located on the beam which moves the forearm. With gear ratios of 36:12 = 3:1 and 60:12 = 5:1, the torque at the elbow is 3 * 5 = 15 times that of the motor alone. The fact that there are pairs of gears rather than just one does not affect the torque; however, the added gears do provide useful support for the two rails making up the upper arm.

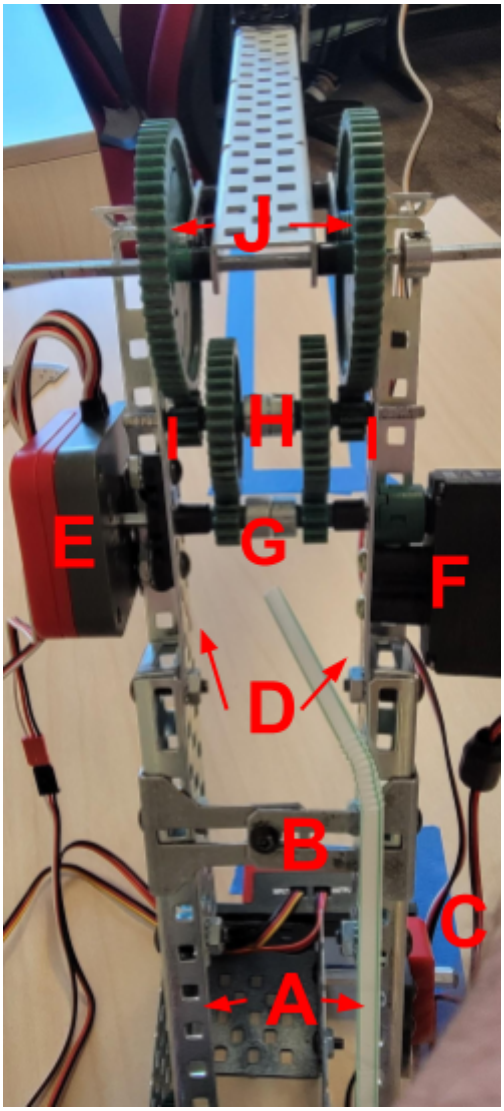

Fig. 3. A view of the upper arm and gear system powering the elbow joint.

The forearm consists of just one metal rail extending from the elbow joint, with a claw ("A" on Fig. 4) attached near the end such that the opening of the claw begins where the rail ends. A servo ("B" on Fig. 4) with a small beam is attached to the right side of the claw; there is a 12-tooth gear ("C" on Fig. 4) on the beam which then connects to the claw's gears so that the claw's grip can tighten/loosen. Unlike the upper arm, the forearm does not have any buttons/switches in place to stop its motor when it is moving too close to the table. As explained later, the code prevents the forearm from going too low (and thus putting the claw at risk). The remaining limit switches and buttons were used for a manual movement mode that we used for testing and calibration, with different switches/buttons being mapped to the up/down movement of upper arm and forearm.

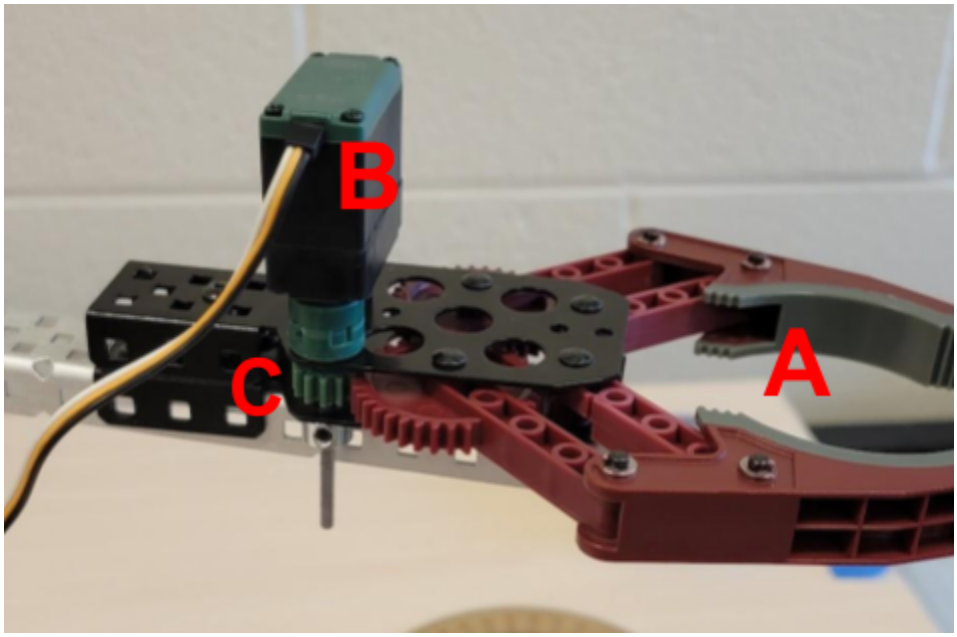

Fig. 4. The manipulator's claw and associated parts, viewed from the side.

Software Design

Like the mechanical design of the robot, the overall structure of the code remained mostly the same throughout our work, with changes occurring in the smaller details. Initially we

wanted to create a more general movement function than the one in the final code, with both moving toward the can and moving back into the dumping position being handled by this function with inverse kinematics. However, we quickly realized that dealing with both forward movement and backward movement cases with inverse kinematics was unnecessary since the dumping position would remain constant, so we instead used constant shaft encoder values for backward movement and calculated these values for forward movement only. We struggled with consistently getting the arm in the correct position throughout the robot's range (e.g. the correct endpoint would be reached when the can was close to the robot but the calculated endpoint when the can was far away would be completely wrong), which is what caused us to add/subtract certain constants from the desired shaft encoder value based on which section of the range a can was detected in.

At the beginning of our code (see Fig. 5 for a high-level outline), we define a few constants which are used later, those being "y" (the height at which we want the claw to grab the can), "l1" (the length of the upper arm), and "l2" (the length of the forearm, claw included); all of the measurements are in inches. We also set the values of both of the shaft encoders to 0, which is done for calibration purposes. For the best results we would use the aforementioned manual movement mode and a protractor to reset the joints to the angles at which we had originally set the shaft encoder values to 0 while writing the code (55° for the shoulder and 130° for the elbow). At this point the arm moves into its dumping position to prepare to grab a can (this being necessary because the movement function can only handle forward movement as discussed earlier); this is done with a while loop that moves the two upper arm motors backward until the upper arm has reached the proper position (determined by the shaft encoder value) and a subsequent while loop which does the same for the forearm. Once the robot is in dumping

position, the claw is opened and a while loop which breaks when the sonar sensor value is either above 22 inches or below 10 inches is entered. This while loop ensures that the robot will only attempt to grab a can if it is located within this specific range. Since we know how the arm will behave within this specific range, we can be certain (barring errors beyond our control) that the claw will not be at risk of hitting the table and breaking.

Once we're in this while loop, we first calculate "x", or the distance from the shoulder joint to the can, which is found by taking the sum of the distance from the joint to the sonar sensor (4.3 inches) and the sonar sensor value. Using x, y, l1, and l2, we then calculate θ_1 and θ_2 (the desired angle in radians for the shoulder and the elbow respectively to reach point (x,y)) using these inverse kinematics formulas from the class slides:

$$\theta_2 = cos^{-1}\frac{x^2+y^2-(l_1)^2-(l_2)^2}{2l_1l_2}, \theta_1 = tan^{-1}\frac{y}{x} - sin^{-1}\frac{l_2 sin\theta_2}{\sqrt{x^2+y^2}}$$

We then add π to θ_2 and call the movement function using our θ values for its arguments. In both cases the θ values are converted to degrees, but for θ_2 we use the difference of θ_2 in degrees and 365 for the argument. Initially we tried to determine what needed to be done to the radians values to make them usable for our robot through just math, but we eventually found ourselves playing with the θ values until we found something that worked; adding π to θ_2 in radians and then subtracting 365 from θ_2 in degrees seemed to do the trick. There is likely a more elegant solution, but we stuck with the one which gave us success.

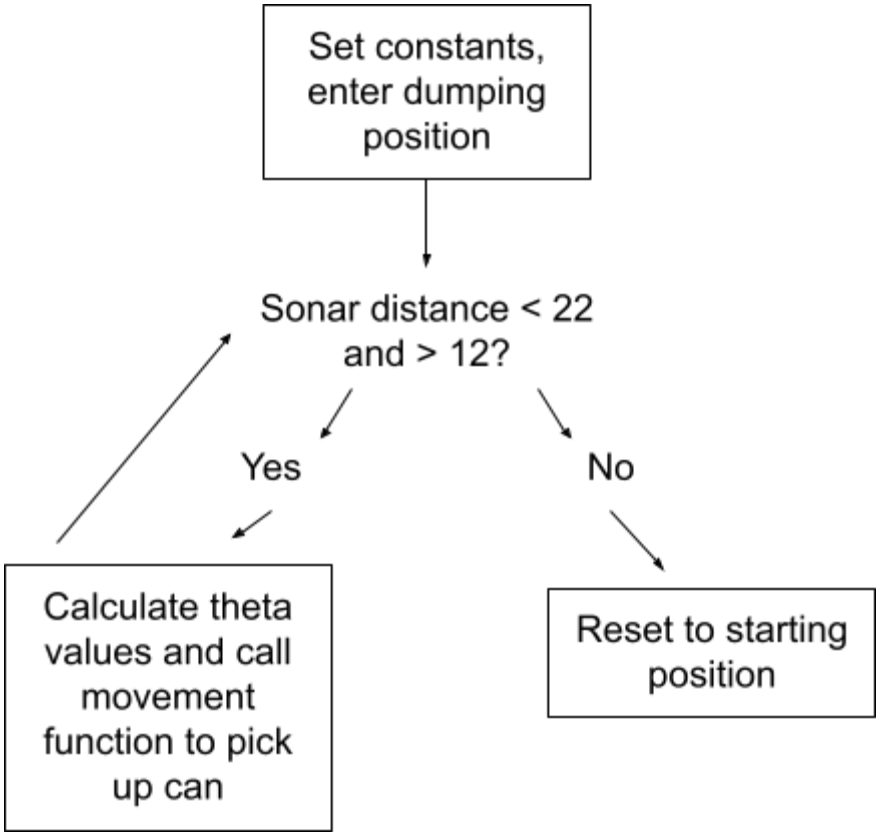

Fig. 5. A basic outline of the code for our manipulator.

The movement function (see Fig. 6 for pseudocode) begins by calculating the desired shaft encoder values ("se1" and "se2") to get the manipulator in the right position for the can from the angles passed in as arguments. These are calculated like so:

$SE_1 = (|\theta_1| - 55)\ *-\ 5, SE_2 = (\theta_2 - 130)\ *-\ 10.5$

The angles for the shoulder and the elbow are subtracted by 55 and 130 respectively since those are the angles at which those joints have a shaft encoder value of 0. The constants which they are multiplied by represent our rough estimates of the shaft encoder value:angle ratio based on observation. There is then a group of if/else statements which check if the sonar value is less than 15, greater than 15 but less than 18, or greater than 18; two values called "se1T" and "se2T" are set depending on which conditional is met. Once we enter the while loops which will move the

manipulator, se1 and se2 will be subtracted by se1T and se2T respectively. The purpose of this is to split the manipulator's range into three discrete sections and change the desired shaft encoder values based on which section the can is in; as mentioned earlier, certain parts of the range would cause issues while others wouldn't with certain settings, so we went with this approach to essentially correct for whatever inaccuracies may have been introduced in the conversion from angle to shaft encoder value. Like the constants used to calculate se1 and se2, the values for se1T and se2T are based on repeated observations of our manipulator with different parts of its range.

```
def movement(theta1, theta2):
    se1, se2 = values based on theta

    if sonar < 15:
        set se1T and se2T accordingly
    else if sonar < 18:
        set se1T and se2T accordingly
    else:
        set se1T and se2T accordingly

    while shoulder shaft encoder < se1 - se1T:
        if either limit switch is pressed:
            break
        continue moving upper arm
    stop upper arm

    while elbow shaft encoder < se2 - se2T:
        continue moving forearm
    stop forearm

    close claw
```

Fig. 6. Pseudocode implementation of the movement function from our code.

The while loops which move the manipulator to the can take the same basic form as the ones which move it into dumping position: keep the motor(s) moving until a certain shaft

encoder value is detected, and then stop. However, the while loop for the upper arm does have an additional if statement which causes the loop to break if either of its limit switches are pressed, preventing it from going too far back or too far down. Once the upper arm and then the forearm are in the desired position, the servo is set so that the claw closes around the can without crushing it, and the movement function ends.

After returning to the main function, the same code from the beginning that was used to move the manipulator into its dumping position is repeated. Once the arm is in the proper position, the claw is opened by the servo, after which the while loop restarts. When the while loop has been broken out of (presumably because there are no more cans within the robot's range), execution moves on to another pair of while loops meant for moving the upper arm and forearm, although these are meant to move the two into the position at which their shaft encoder values are equal to 0. It is worth noting that the values which these while loops are checking for are not 0; this is because we noticed that the arm would often overshoot the desired point, so we made the values lower than 0 to compensate. Once this movement is complete, autonomous movement has ended and the robot should be close to the correct position for it to continue working on at least a few subsequent runs until manual calibration is needed.

Performance Evaluation

Overall, our manipulator was able to accomplish the task of picking up three cans autonomously fairly consistently, especially if we had just manually calibrated it. Although there were often instances of the arm pushing the next can in line while picking up the front can (especially at closer distances), the push was typically very slight and would not prevent the manipulator from being able to pick up the can. The range at which our manipulator can pick up cans is from 10 to 22 inches away from the sonar sensor, which was just enough to meet the

requirement of having a range that could include four cans at one time. We did not test the manipulator using objects that were placed above or below the table's surface; it is possible that the robot would be able to pick up such objects if we were to change the constant "y" in the code, but it is probably more likely that other constants would need to be changed as well since they were used with the specific height associated with the cans on the table in mind. For a robot like this to truly be able to pick up objects at different heights autonomously, we would probably need to employ a servo in order to allow the sonar sensor to scan up and down to determine the height at which the object is located as well as the horizontal distance.

Although our usage of kinematics allowed the upper arm to typically get into a position close to one which was appropriate for picking up a can, we struggled quite a bit to get the forearm to get to the right position. Ultimately we opted to use more observational techniques for getting the correct positioning to assist our (likely shoddy) employment of inverse kinematics, as seen with se1T and se2T in the code. Knowing this, it is almost certain that if the lengths of our upper arm and forearm were different in such a way that the range for the manipulator was different, we would have to find new values using observation and change them in the code for the robot to work correctly. Ideally, one would have code that relies solely on the results of the inverse kinematics equations and could thus be generalized to different arm designs; if we had truly tried to understand how to properly utilize these equations and perhaps been more careful with our mapping of angles to shaft encoder values, it is possible that we could have achieved something similar to this.

Within its range, the manipulator is almost always successful in picking up a can. The two most common types of misses (once we had the manipulator successfully picking up cans consistently, that is) were the claw barely scraping the can with the tips of its pincers, thus

pushing the can forward, and the claw overshooting the can slightly such that the can was located directly in the center of the claw's grasp, where our designated grip strength was not strong enough to grip the can. We would encounter these issues when the robot had not been manually calibrated in a few runs; after calibration, cans placed at the same distance as before would almost always be picked up successfully. We did not experiment with objects other than cans, although I think that with some minor changes to the code (e.g. changing the "y" value and grip strength) it would be able to successfully pick up objects that are at least a little bit heavier than the average can, since the arm did not seem to struggle at all with a can's weight. With objects much heavier than a can, we would likely need to increase the number of gears powering the joints in order to provide the system with more torque, although I am unsure at what point this addition would become necessary.

Conclusion

Our manipulator was successful in autonomously picking up three cans within its given range with only minor issues relating to calibration. In addition, the safety measures which we had in place allowed us to avoid damaging the claw. However, the way in which we calculate the point at which the claw grabs could rely more heavily on inverse kinematics in order to make the code more generalizable for other manipulator designs. We could also allow for more options regarding where the claw could pick up objects, including objects at heights greater or less than that of the table and/or objects located to the left or right of the arm.

Appendix

Below is the code for autonomous movement:

```
#pragma config(Sensor, dgtl1,  shoulderSwitchF, sensorTouch)
#pragma config(Sensor, dgtl2,  shoulderSwitchB, sensorTouch)
#pragma config(Sensor, dgtl5,  quad,         sensorQuadEncoder)
#pragma config(Sensor, dgtl7,  sonar,        sensorSONAR_inch)
#pragma config(Sensor, dgtl9,  foreQuad,     sensorQuadEncoder)
#pragma config(Motor,  port2,           rightMotor,
tmotorVex393_MC29, openLoop)
#pragma config(Motor,  port3,           leftMotor,
tmotorVex393_MC29, openLoop)
#pragma config(Motor,  port4,           foreMotor,
tmotorVex393_MC29, openLoop)
#pragma config(Motor,  port5,           servo,
tmotorServoStandard, openLoop)
//*!!Code automatically generated by 'ROBOTC' configuration
wizard               !!*//

//function for autonomous movement
//takes theta1 and theta2 (in degrees) as input
void movement(float t1, float t2) {
    //desired shaft encoder values for...
    int se1 = (abs(t1)-55)*-5; //shoulder
    int se2 = (t2-130)*-10.5; //elbow
    int se1T;
    int se2T;

    //choosing constants to apply to se1 & se2 based on
different sections in our range
    if(SensorValue[sonar] < 15) {
      se1T = 80;
      se2T = -60;
    } else if(SensorValue[sonar] < 18)  {
      se1T = 40;
      se2T = 130;
    } else {
      se1T = -20;
      se2T = 520;
    }

    //printing desired shaft encoders for testing
    writeDebugStreamLine("SE1: %d", se1);
    writeDebugStreamLine("SE2: %d", se2);

    //shoulder movement
```

```
    while (SensorValue[quad] < se1 - se1T) {
      //checking if limit switches are pressed
      if (SensorValue[shoulderSwitchB] == 1 ||
SensorValue[shoulderSwitchF] == 1) {
           break;
      }
      motor[leftMotor] = 20;
      motor[rightMotor] = 20;
    }
    motor[leftMotor] = 0;
    motor[rightMotor] = 0;

    //elbow movement
    while (SensorValue[foreQuad] < se2 - se2T) {
           motor[foreMotor] = 40;
    }
    motor[foreMotor] = 0;

    wait1Msec(500);
    motor[servo] = 28; //closing claw to grab can
    wait1Msec(500);
}

task main()
{

    float y = 4.5; //height at which can is grabbed
    float l1 = 11.375; //arm length
    float l2 = 16.75; //forearm length w/ claw

    //resetting quad encoder values for calibration purposes,
    //only necessary occasionally
    SensorValue[quad] = 0;
      SensorValue[foreQuad] = 0;
    wait1Msec(500);

    //moving to dumping position so that the arm only moves down
when grabbing can
    while (SensorValue[quad] > -70){
      motor[rightMotor] = -30;
      motor[leftMotor] = -30;
      }
    motor[rightMotor] = 0;
    motor[leftMotor] = 0;

    while (SensorValue[foreQuad] > -2000) {
      motor[foreMotor] = -40;
```

```
      }
    motor[foreMotor] = 0;

    wait1Msec(500);

    motor[servo] = -127; //opening claw

    wait1Msec(1000);

    //while within the possible range for our arm
    while(SensorValue[sonar] < 22 && SensorValue[sonar] > 10){

      float x = 4.3 + SensorValue[sonar]; //distance from can,
4.3in is distance from shoulder joint to sonar sensor
      //inverse kinematics equations
      float theta2 = acos(((x*x) + (y*y) - (l1*l1) -
(l2*l2))/(2*l1*l2));
      float theta1 = atan(y/x) -
asin((l2*sin(theta2))/sqrt((x*x)+(y*y)));

      //shifting theta2 to the desired part of the unit circle
      theta2 = theta2 + PI;

      //printing theta values
      writeDebugStreamLine("Theta 2 Radians: %0.2f", theta2);
      writeDebugStreamLine("Theta 1 Radians: %0.2f", theta1);
      writeDebugStreamLine("Theta 2: %0.2f", abs(theta2 *
(180/PI)-365));
      writeDebugStreamLine("Theta 1: %0.2f", theta1 * (180/PI));

      //subtracting 365 degrees for moving to desired part of
unit circle
      movement(theta1 * (180/PI), abs(theta2 * (180/PI)-365));

      wait1Msec(500);

      //dumping the can
      while (SensorValue[quad] > -70){
           motor[rightMotor] = -30;
           motor[leftMotor] = -30;
      }

      motor[rightMotor] = 0;
      motor[leftMotor] = 0;
      while (SensorValue[foreQuad] > -1700) {
           motor[foreMotor] = -40;
      }
```

```
      motor[foreMotor] = 0;

      wait1Msec(500);
      motor[servo] = -127; //open claw to drop can
      wait1Msec(1000);
    }

    //reset to initial position once all cans have been picked
up
    //desired quad values are not 0 since the motors are usually
late to stop
    while (SensorValue[quad] != -10){
      motor[rightMotor] = 30;
      motor[leftMotor] = 30;
    }
    motor[rightMotor] = 0;
    motor[leftMotor] = 0;
    while (SensorValue[foreQuad] != -80) {
      motor[foreMotor] = 40;
    }
    motor[foreMotor] = 0;
}
```