Lab 1: Problem Solving with Software

Objectives

In this lab, you will

i. Learn how to implement basic estimation logic in Robot C.
ii. Learn how to solve basic control problems in Robot C.
iii. Create software that has acceptable qualities of readability, maintainability and easily upgradability.

Overview

Robotics systems must use sensors and actuators to convert between real world physical parameters such as position, temperature, and speed with values in the controlling software. A sensor is a device used to make measurements of real-world parameters and supplies values to the software. For example, you a sensor could convert the physical distance between two objects to a numerical value in the software. The converse device to a sensor is an actuator which is a device which converts a control signal to some physical effect in the real world. An electrical motor connected to a microcontroller such as your VEX controller is an actuator which converts a number within your software to an electrical voltage which the motor, in turn, then converts to the physical rotation of a part on your robot.

A problem encountered in robotic systems is imprecision in either the senor measurements and/or actuator actions. That is, sensors do not give complete information about the parameter of interest or the relationship from actuator control input to output has some random effects. While these problems can be somewhat reduced by using more expensive components, in real world systems, these devices are built under economic limitations. Robot system designers must often use software to compensate for the limitations of the components used in the robots. In this lab, you will be working with limitations of sensors and motors to try and obtain higher precision from the VEX sensors and motors.

Handling Imprecise Sensors

A key problem in robotics is how to obtain highly precise measurements from low cost sensors. Here we will derive a system for obtaining precise measurements of an environmental value using a cheap sensor by taking multiple measurements and combining the results together using software.

An example of a cheap sensor is a coarsely quantized sensor where the measurement only indicates if a parameter of interest is above or below to a given value. For example, old baking ovens would use a sensor like that shown in Figure 1. The sensor consists of bimetallic strip next to a conductive strip with an adjustable distance between the two strips. The different expansion coefficients of the two metals in the bimetallic strip causes it to bend differently at different temperatures. When the bimetallic strip bends enough, electrical contact is made between the bimetallic strip and the plain conductive strip. When an electric current can flow from the bimetallic strip to conductive strip then we know the temperature is above a certain point since the bending of the strip increases with temperature. The exact temperature when the strips are just touching can be fixed by adjusting the distance between the bimetallic and plain conductive strips at one end. For example, you can adjust the sensor so that there is conduction between the two strips when the temperature is higher than 95° C. At this setting, if we have conduction then we know the temperature is above 95° C, and if there is no conduction then the temperature is below 95° C. While the measurements that can be made by this device are crude, the sensor is also cheap, robust, and durable. This combination of attributes made this device the mainstay temperature sensor for consumer-grade ovens for many decades.

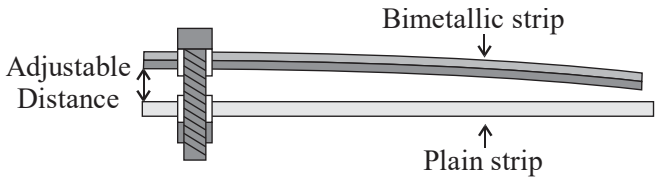

Figure 1: Cheap Temperature Sensor

With the advent of cheap computer controllers, a crude sensor like this can be used to provide precise measurements with the use of signal processing. In these systems, the distance between the two strips is adjusted with an actuator such as a stepper motor[1] so the boundary temperature where electrical contact is made can be changed. Assume that a system is developed so that if the temperature, T, is above some set value, a, in degrees Celsius then electrical contact is made where the set value, a, can be changed by the control system interactively. Let us also say we have the two measurements given below and assume that the temperature remains constant between the two measurements:

1. electrical contact when $a = 150°$ C, and
2. no electrical contact made when $a = 200°$ C.

From these two measurements, we know that the temperature T is less than 200° C but greater than 150° C. To get a more precise measurement, we can adjust a to 175° C to get a measurement at this point. If electrical contact is made for $a = 175°$ C then we know that $175° \text{ C} < T < 200° \text{ C}$. Where if no electrical contact is made for $a = 175°$ C then $150° \text{ C} < T < 175 °$ C. This can be summarized with the ranges graphed in Figure 2 for the two measurements 1 and 2 listed above combined with the measurement of no contact when $T = 175°$ C where the thicker lines indicate regions that can contain the temperature value.

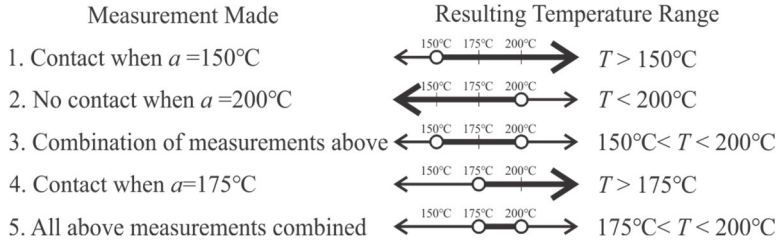

Figure 2: Measurement Combination

As you can see from Figure 2, adding additional measurements reduces the size of the range containing the value of the temperature.

We would like to create some software which can make crude temperature measurements like that described above but will be able to determine the temperature precisely by making multiple measurements at different set points for the sensor. Let's assume, to simplify our derivations, that the temperature can only take discrete values in the range from MIN to MAX degrees; that is, the temperature T is in the set $T \in \{MIN, MIN + 1, \cdots, MAX\}$. With no other information available, we have to assume that each value is equally likely. We define the function $P[A]$ as the probability function so that $P[A]$ is equal to the probability of the event A and is a value between 0, indicating event A will never occur, and 1, indicating that the event A will occur with 100% certainty. Using our assumptions above:

$$P[T = a] = \begin{cases} \frac{1}{MAX - MIN + 1} & MIN \le a \le MAX \\ 0 & \text{otherwise} \end{cases}.$$

To make a measurement, we set the measuring point of the sensor as a. To make our problem slightly simpler, we will assume our sensor is a little more sophisticated than the bimetallic strip sensor above in that it can detect equality in addition to if our test value if less than or greater than the set point. That is, our system will return a result indicating that $T > a$, $T = a$, or $T < a$. The goal of our software is to select a measurement set point so the resulting range of the possible temperature value is minimized. The probabilities of the three different results given a set point a are:

$$P[T > a] = \frac{MAX - a}{MAX - MIN + 1}$$
$$P[T = a] = \frac{1}{MAX - MIN + 1} \text{ where } MIN \le a \le MAX.$$
$$P[T < a] = \frac{a - MIN}{MAX - MIN + 1}$$

[1] A stepper motor, https://en.wikipedia.org/wiki/Stepper_motor, is a specialized motor which moves a predetermined distance in response to a short electrical pulse. As well as the application described here, they are widely used in printers (both imaging and 3D).

If $T = a$, we are finished with the measurement process since we have determined the temperature exactly. If we get one of the other results, more measurements will have to be taken to reduce the number of possible values for T further. For the result that $T < a$, we obtain that the temperature is in the range of $MIN \leq T < a$ which is of length $a - MIN$. For the result that $T > a$, we obtain the range that $a < T \leq MAX$ which is of length $MAX - a$. If we obtain the result that $T = a$ then the output range is of length 1. If we average over all possible temperatures T where $MIN \leq T \leq MAX$ with the measurement set point of a, we can calculate the average length of the output range after the measurement. This calculation is done by considering the true value of T to equal each sample value of t in the range of $MIN \leq t \leq MAX$ and computing the output range for possible values when the measurement is set to point a, and then computing the average length of the output range. If you perform this computation, you get the average length of the output range:

$$\begin{aligned} AVERAGE\ LENGTH &= (MAX - a)P(T > a) + (a - MIN)P(T < a) + 1 \times P(T = a) \\ &= (MAX - a)\frac{MAX - a}{MAX - MIN + 1} + (a - MIN)\frac{a - MIN}{MAX - MIN + 1} + \frac{1}{MAX - MIN + 1} \\ &= \frac{(MAX - a)^2 + (a - MIN)^2 + 1}{MAX - MIN + 1} \end{aligned}$$

To find the value of a which minimizes $AVERAGE\ LENGTH$, we take the derivative of the above equation and see what value of a makes this derivative equal to zero. This gives us the following:

$$\frac{d}{da} AVERAGE\ LENGTH = \frac{2(MAX-a)(-1)+2(a-MIN)}{MAX-MIN+1} = \frac{4a-2(MAX+MIN)}{MAX-MIN+1} = 0.$$

Solving this equation, gives us the solution that $a = \frac{(MAX+MIN)}{2}$ will minimize the $AVEARGE\ LENGTH$ since $\frac{d^2}{da^2} AVERAGE\ LENGTH > 0$ for all values of a. When a is not an integer value, we must use one of the nearest integer values to the result above to minimize the value of AVERAGE LENGTH.

With this selection of measurement point, after the measurement the length of the range of possible temperature values will shrink by a factor of about 2 if the measurement point does not match the true value. You can then refine the MIN and MAX values to match your new knowledge of the range of possible values and repeat the process. An algorithm to find the temperature is shown in Figure 3 as a flow chart.

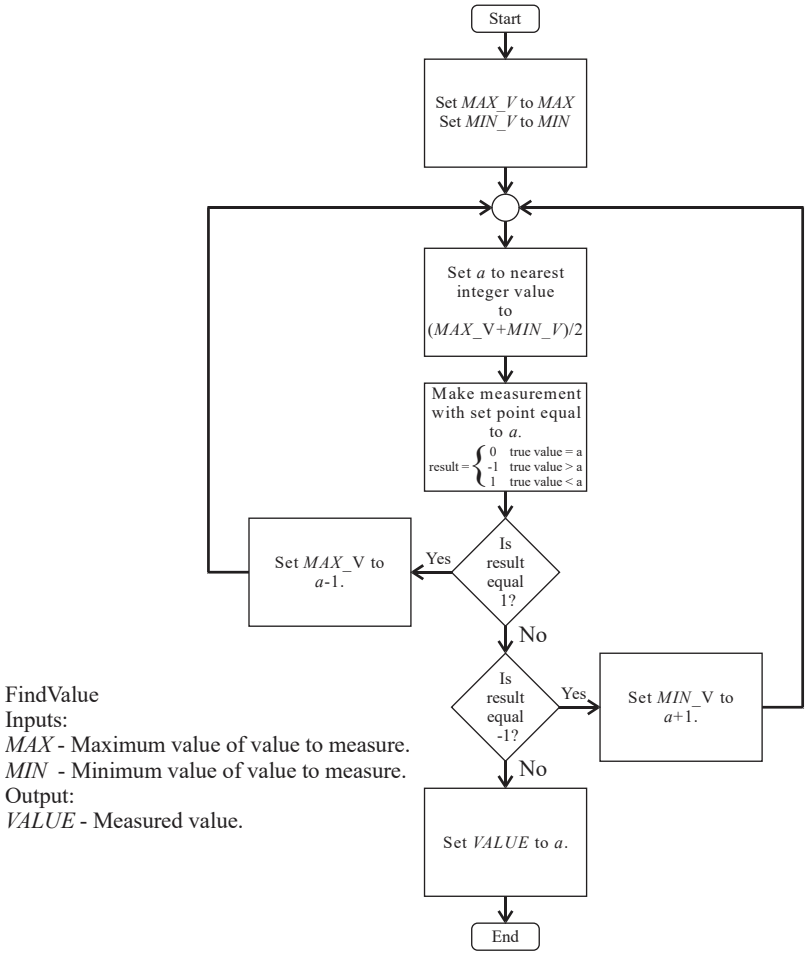

Figure 3: Flow Chart of Algorithm to Find Value from Crude Measurements

Handling Imprecise Actuators

Modern robotics systems have digital actuators which convert values from the control software to a physical parameter such as an angle, position, or speed of a robot's component. A problem is that the conversion from software values to physical parameter values is rarely exact since there are random real-world effects that actuators must deal with such as friction, physical loads on the robot components, or changing environment conditions. Therefore, the output of a physical actuator to a control signal can give slightly different results when applied at different times or under different circumstances. This is the problem of imprecise actuation. To handle this case, more sophisticated software control must be used to compensate for these random effects so that precise control can be achieved from imprecise actuators. The process of handling these imprecise actuators is the converse to imprecise sensors problem where a fixed real-world parameter results in an imprecise value in the software.

There are several types of imprecision and some engineers spend their entire careers implementing mitigation strategies for different types of actuator imprecision. Here we will examine the problem of how to handle inertia in motors. The force on the rotor in a DC motor is proportional to the voltage applied to the control terminals of the motor. The typical time and speed relationship of a motor is shown in Figure 4. When a control signal is applied to a motor, it takes the motor some time to accelerate to the set speed corresponding to the control signal. In addition, when the control signal is removed from the motor, the motor takes some time to deaccelerate to a stop.

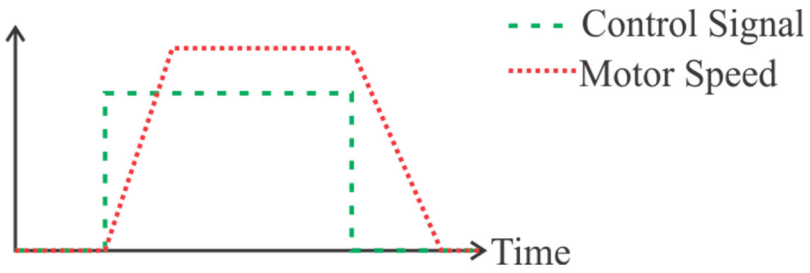

Figure 4: Motor Control/Speed Time Relationship

The acceleration and deacceleration profile of motors with respect to the control signal must be considered when you wish to move the motor to a given position. If you turn off the control signal when the motor hits the target position, the motor will slide past the position as it deaccelerates to a stop. To get an accurate position control, you must turn off or reduce the motor control signal before you reach the target position.

Control Motor
Inputs:
MAX - Maximum amplitude of motor control signal.
TARGET - Target position of motor.
ACC -Desired accuracy of position.

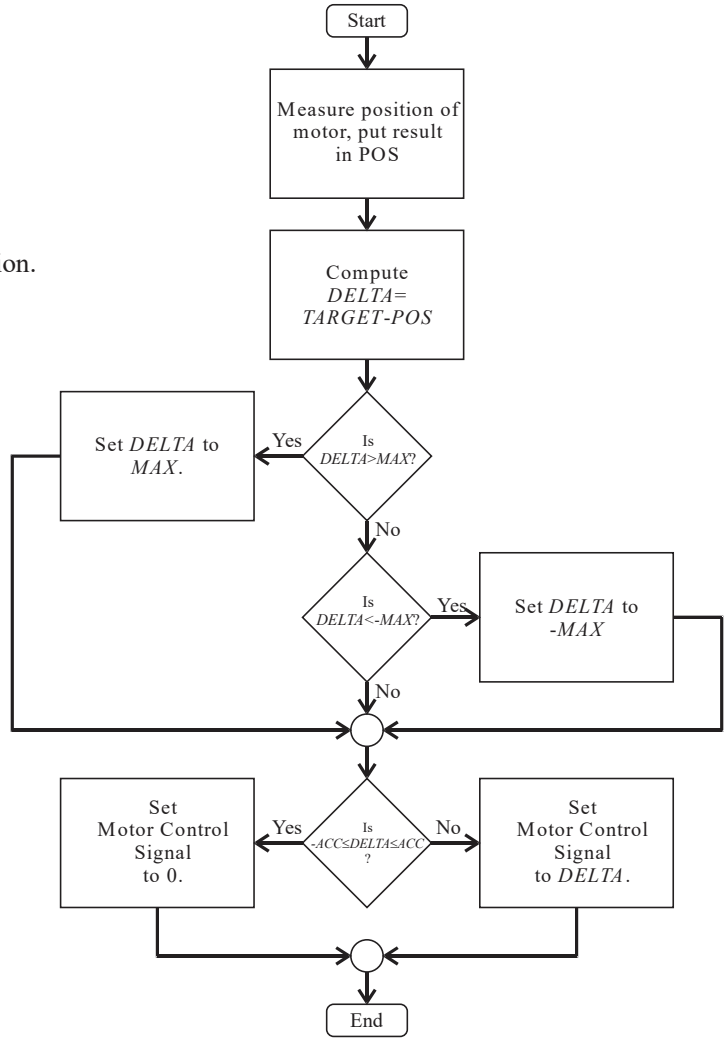

Figure 5: Motor Control Algorithm

A simple solution to this problem is to use a positional feedback control algorithm as shown in Figure 5. This algorithm takes in three input parameters: *POS* which specifies the target position of the motor, *MAX* which specifies the maximum amplitude of the control signal, and *ACC* which specifies the intended accuracy of the control. The control algorithm uses these input values as well as the current measured position to compute a control value and then sends this a value to the motor. For each iteration of a control loop, this algorithm measures the position of the motor, computes the difference between the current position and target position, and then sends a control signal to the motor based on this difference. If the difference is within the desired accuracy setting of the algorithm then the motor control signal is set to zero. If the magnitude of the difference between the desired position and current position is greater than the desired maximum amplitude of the control signal, the motor is sent either *-MAX* or *+MAX* depending on whether the current position is greater or less than the target position, respectively. If the magnitude of the difference is less than the desired maximum magnitude then the motor is sent the difference as the control signal. The net result is that the motor is given a large control signal if the motor position is far from the target position and this control signal's magnitude is reduced as the motor moves closer to the target position. As a result, the motor slows down as it approaches the target allowing for more precise control.

In the lab, you will implement this algorithm and see what values of MAX and ACC allow for accurate positioning as well as rapid convergence to the desired target values.

Deliverables:

By the end of this lab, you will need to:

i. Write a RobotC program which will implement the Imprecise Sensor Algorithm for an simulated sensor.
ii. Build a simple VEX test rig for testing a motor controller.
iii. Write a RobotC program which will implement the Imprecise Actuator Algorithm for a simple motor controller with a potentiometer position sensor.

Lab Preparation (2 marks)

Before starting, you will need to download firmware onto your controller.

1. Read this laboratory description. Be prepared to answer questions about the material above.
2. Build the VEX motor control rig described below for Exercise 2 shown in Figures 6 and 7. The description of the mechanism is in the Exercise 2 section below.
3. If you are unfamiliar with C, please review the RobotC tutorials on the CourseSpaces website.

Exercise 1 – Imprecise Sensors (4 marks)

In this exercise, you will implement the algorithm described above for a simulated sensor. You will start with an implementation of a simple sensor algorithm which sweeps over the range of available values until a matching value is found. For this exercise, your code will be executed on the VEX controller with a simulated sensor.

Instructions:

1. Connect your VEX controller to the computer.
2. Open the RobotC program on your computer and open the "Exercise1.c" code.
    a. This code simulates a sensor on the VEX controller. A new random value is generated when the "`generate_new_value()`" function is called. This function generates a random sensor value, X, in the range *min_value≤X≤max_value*. You obtain measurements by calling the function "`resolve_measurement( int meas_point )`" which returns "1" if `meas_point` is greater than the random sensor value, "0" if the `meas_point` is equal to the random sensor value, and "-1" if `meas_point` is less than the random sensor value. This function records the number of measurements made so that the efficiency of the imprecise sensor algorithm can be made.
    b. The code runs 1000 trials of generating random sensor values and estimating the value. The total number of measurements is then reported.
    c. The basic code uses a very inefficient algorithm as compared to the algorithm described above. This algorithm sweeps the measurement set point value from `min_value` to `max_value`, stopping when equality is reported by the `resolve_measurement` function.

3. Run the code and see how many measurements are needed on average to estimate the sensor value with the original inefficient algorithm.
    a. Compile the code and download it onto the VEX controller. Select the menu option "**Robot→Compile and Download Program**".
    b. Open the debugger window so you can see the text output of your program. Do this by selecting the menu option "**Robot→Debugger Windows→Debug Stream"**. The "Debug Stream" will only show up if the Menu Level is set to "Expert" or "Super User".
    c. The code prints to the "Debug Stream" with the "`writeDebugStream("%d trials and %d guesses with %d correct\n",current_trial,total_meas,correct_guess)`" function call. To see how to use the function please see http://robotc.net/w/index.php/Debug_Stream. The string formatting is based on the standard C "`printf`" command. Please see https://en.wikipedia.org/wiki/Printf_format_string for more information.
    d. Feel free to add more debug prints to your code if it can help you debug.
    e. Run the code and see how many trials this code takes. You should see that 1000 trials takes about 50000 guesses to find the sensor measurements to find all of the sensor values.
4. Modify the code to implement the algorithm described in Figure 3.
    a. Adjust the code in the function.
    b. If you write the code correctly it should take about 5800 guesses to find sensor values for about 1000 trials.
    c. For tips on debugging and stepping through your code please see the CourseSpaces webpage.
5. Save your code for submission at the end of the lab.

STOP! ***Call a TA over and be prepared to demonstrate your code***

Exercise 2 – Motor Position Controller (4 marks)

In this exercise, you will be building a motor controller for the mechanism shown in Figure 6. You may need to use a longer shaft to connect to the motor than is shown in the photo below.

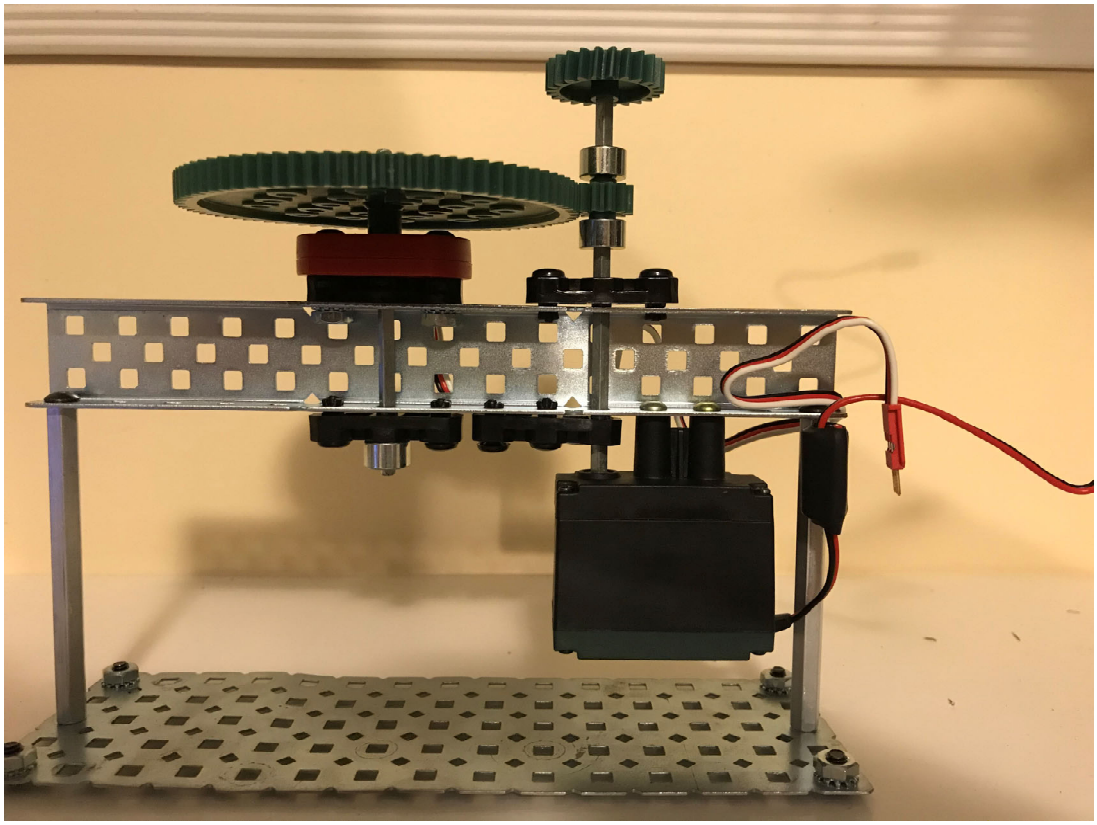

Figure 6: Motor Controller Mechanism

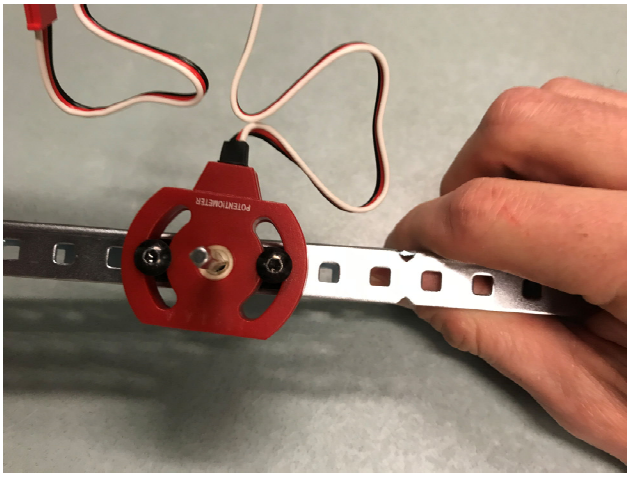

Figure 7: View for Top of Left Shaft with Top Gear Removed.

This mechanism connects the motor to a small gear which then interlocks into a large gear connected to a potentiometer. Please see the construction instructions for the Tumblerbot and Protobot on the ENGR120/121 CourseSpaces page for instruction on how to connect the different parts from your VEX kit together.

The connections to the VEX controller are as follows.

1. The motor should be a 2-wire motor connected to motor port 1 of the VEX controller. This motor should be configured as a VEX 393 Motor with the "**Robot→Motors and Sensor Setup"** menu option under the "Motors" tab
2. The potentiometer should be connected to Analog port 1 of the VEX controller. The potentiometer is a device which returns a value from 0 to 4095 which is proportional to the rotational angle of the shaft it is connected to. This is configured as a Potentiometer sensor under "**Robot→Motors and Sensor Setup"** under the "VEX 2.0 Analog Sensors 1-8" tab.

Instructions:

1. Open the RobotC program on your computer and open the "Exercise2.c" code.
    a. This code is a very basic controller which runs the motor at a constant magnitude with a sign determined by the difference from the target and measured position. This code will work but will constantly switch the motor direction when the position is near the target value and will have a large error value. You will improve the performance greatly by implementing the algorithm in Figure 5 in step 4 below.
2. Ensure that the motor and potentiometer are configured correctly. The motor and potentiometer and motor need to be configured so when the motor is activated with a positive control signal the potentiometer measurement should increase.
    a. Download the "Exercise2.c" code on the VEX controller.
    b. This code runs the motor controller for 5 seconds. If the motor and potentiometer are properly configured then the potentiometer sensor value should approach the value stored in the `target_value` constant. To see the potentiometer sensor reading, you open the sensor window with the menu selection "**Robot->Debugger Windows->Sensors**" and then select the "Sensors" tab in the debugger windows on the bottom of the RobotC window.
    c. Execute the code. View the potentiometer value under the `motor_angle` sensor value under the Sensors Debugger window. If the `motor_angle` does not approach the `target_value` when the code is run then the motor is not properly configured. This can be corrected by going under "**Robot→Motors and Sensor Setup**" under the "Motors" tab and changing the value in the "Reversed" checkbox for port1, the motor named "`motor1`". If the `motor_angle` sensor is already close to the `target_value` when the code starts executing so you see no change, you can either rotate the motor shaft or change the target value in the software and run the code again.

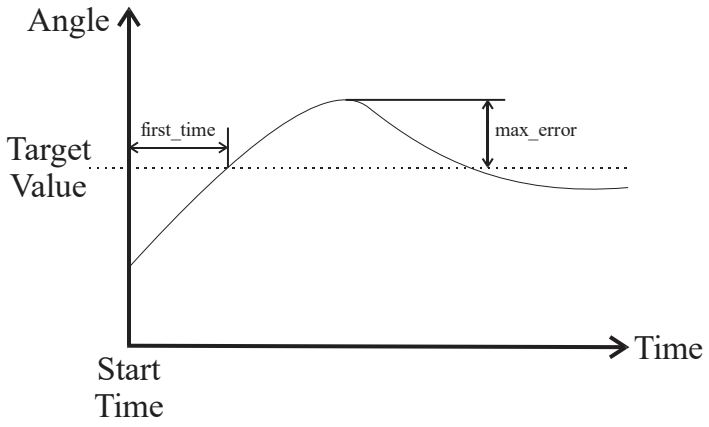

Figure 8: Accuracy Measures

3. The code executes the algorithm and records the time it takes to first hit the target value in milliseconds in `first_time` and the maximum error from the target value after the first time it is encountered in the variable in the variable `max_error`. See Figure 8 for a graphical description of these values. These two values are both printed to the Debug Stream.
    a. Set the `target_value` to 1500 and manually rotate the motor shaft so the potentiometer sensor is within 25 points of 3000. You will find adding a gear to the top of the motor shaft will make it easier to turn manually.
    b. Run the code and note the time taken to first encounter the target value and the maximum error.
4. Implement the algorithm shown in Figure 5 in the "`set_motor()`" function. The "`set_motor`" function should compare the `target` input argument and compares it with the currently measured motor shaft angle and then sends the appropriate control value to the motor for a single iteration of the algorithm shown in Figure 5. Set the intial values of the parameters to be MAX=50 and ACC=10.
5. Adjust the *MAX* and *ACC* values so that you can obtain better `first_time` and `max_error` values than the simple control algorithm supplied to you.
    a. For each trial manually adjust the potentiometer to start near the 3000 value point.
    b. Find the *MAX* and *ACC* values which give the best `first_time` value.
    c. Find the *MAX* and *ACC* values which give the best `max_error` value.
6. If you have time, see if you can find an algorithm which will allow you to simultaneously minimize both `first_time` and `max_error`.
    a. Only do this, if you have more than half an hour left in your lab period.
    b. This portion is only for bonus marks.
7. Submit code on CourseSpaces website.
    a. Create C files which contains the new code for each of your exercises.

- The filenames of your code files should be "BXX_GNNN_Lab1_Ex1.c" for your exercise 1 code and "BXX_GNNN_Lab1_Ex2.c" for your exercise 2 code with the "BXX" being replaced with your lab section and "GNNN" being replaced with your group number preceded by a 'G'. Use of other filenames will prevent your code from being marked.
- It would be wise to send a copy of your code via email to yourself for safety in case of a CourseSpaces error.

Your code will be marked for readability and functionality. (4 marks)

END OF LAB