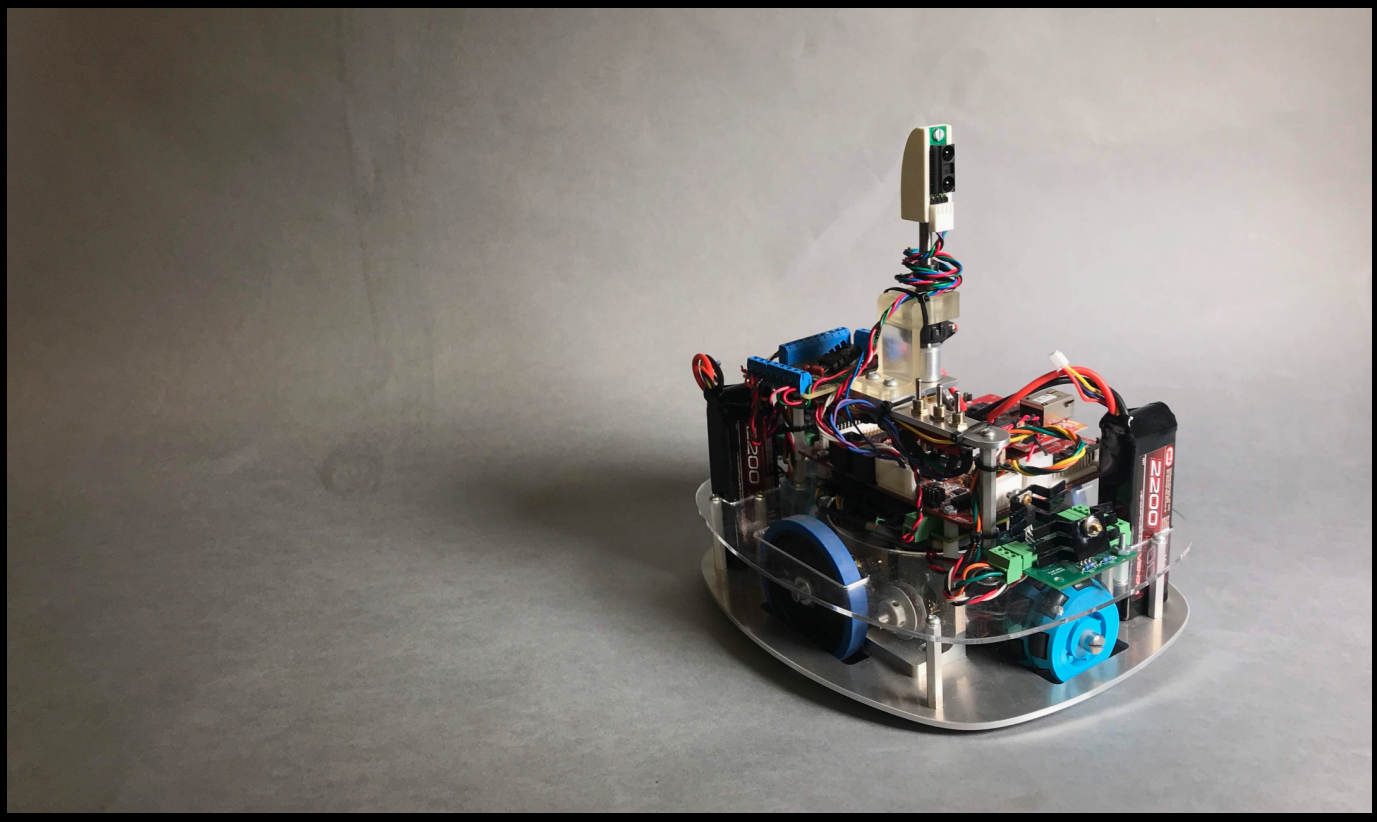


ENGS 147 Final Project Report

Jennifer Jain, Dan Magoon, Jonah Sternthal, Joe Leonor

Table of Contents

Robot Specifications

We have the following specifications for the robot ranked in order of importance:

1. The robot should complete a simple course without hitting any barriers consistently.
2. The robot should not leave the boundaries of the course
3. The robot should be able to recognize and escape concavities.
4. The robot should complete a simple course relatively quickly (less than a minute).
5. The robot should complete a moderately difficult course without hitting any barriers consistently.
6. The robot should complete a moderately difficult course relatively quickly (less than 3 minutes).
7. The robot should complete a challenging course without hitting any barriers consistently.
8. The robot should complete the challenging course relatively quickly (less than 4 minutes).

Mechanical System Design

CAD Model and Part Drawings

Mechanically, we intended the robot to be easy to assemble and disassemble, easily manufacturable, and sturdy. The sturdier the robot was, the better the measurements we would aquire through sensors such as the encoders and IR sensor. With this in mind we opted to keep machined parts to a minium, but not shy away from them when it came to structural stability.

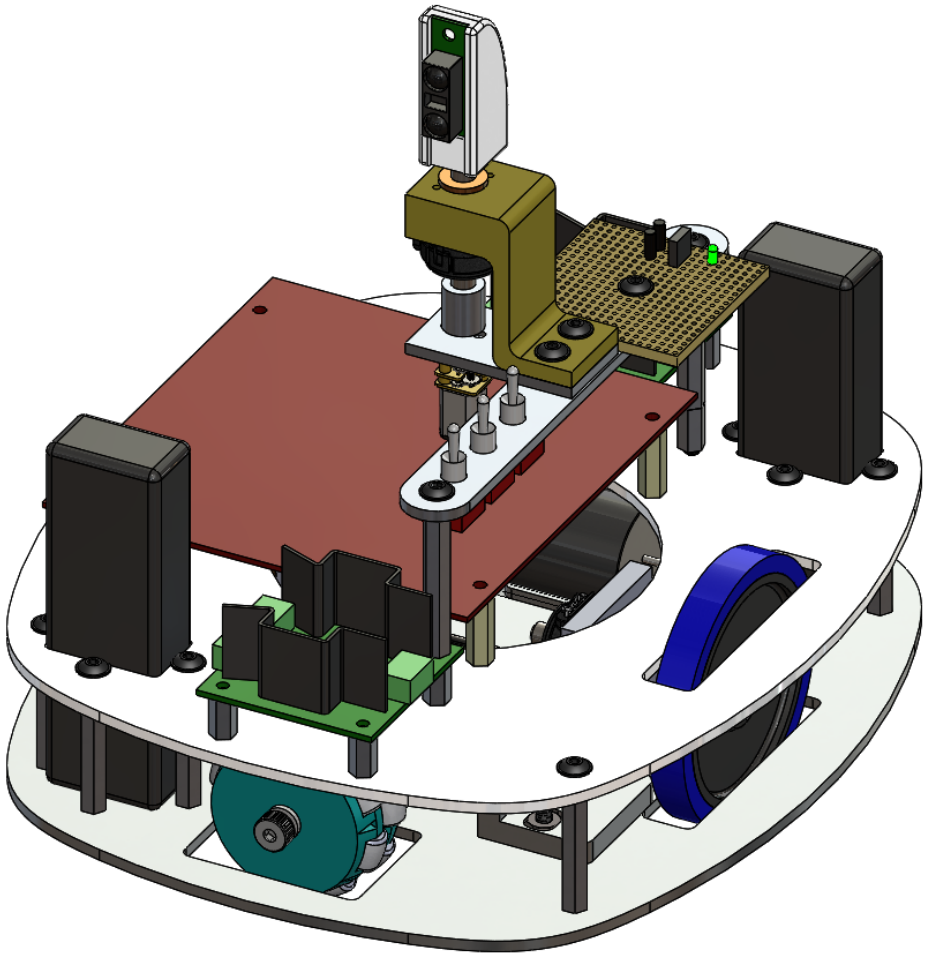


Figure 1: Trimetric View

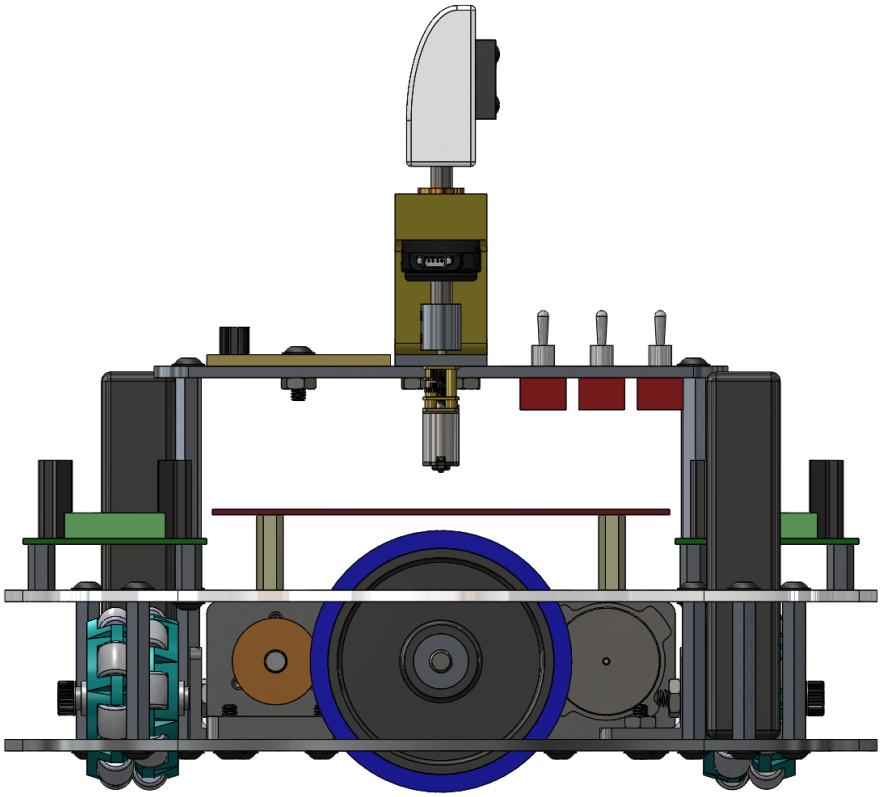


Figure 2: Side View

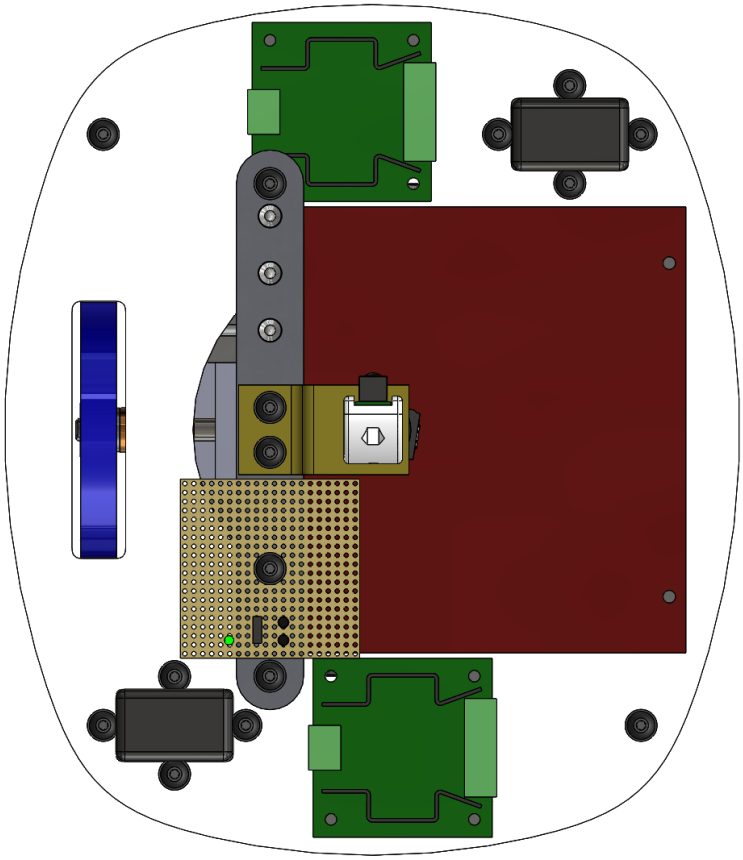


Figure 3: Top View

The base plate was machined from a 12” x 12” plate of 1/8” aluminum and had a 2D profile of approximately 9” x 9.” The extra material was used to create the IR sensor mount and cross beam. The drive train brackets machined from 1.5” x 1.5” blocks of aluminum and ultimately had walls ¼” thick. This was more than enough to handle the loads the robot would experience. We chose the larges wheel size possible as this would result in higher encoder resolution – our drive train would certainly be able to handle the additional torque a large wheel create. The encoders were mounted directly on the wheel axles – this way any slop in the drive train would not affect our position estimates. Additionally, placing encoders on axles spinning at a faster rate than the wheel would decrease our encoder resolution.

All other components and electronics were placed on an acrylic shell that rested upon the sturdy motor brackets and a set of standoffs that doubled as a battery cage. The shell was easily removed with the removal of a few bolts to allow for easy motor access.

The IR sensor was moutned on the 1/8” cross-beam along with the perf-board, additional circuitry, and three switches, the function of which are discussed below. The IR mount consisted of two parts, a base and tower. The division of the mount into two parts allowed for easy installation of the IR sensor encoder. The base provided a surface for the Polulu Motor and gear box to mount to. A D-Shaft coupler was manufactured to pair the small Polulu motor shaft with the ¼” IR sensor d-shaft. The D-shaft holding the IR mount was simply supported within the tower with two brass bushings.

Part drawings can be found in Appendix A.

Drivetrain Design Analysis

Using the final weight of the robot (2248.7 g), we performed analysis in Matlab to determine the requirements for our motor (see Appendix B for code and output). We calculated rolling resistance and wheel angular velocity, and converted this to a power requirement assuming the drivetrain efficiency of 1 gearbox and 1 belt drive. We then interpolated this requirement onto the specifications of the venerable Faulhaber Motor to determine an operating point. See table below summarizing our inputs and findings:

Table 1: Motor Analysis Inputs and Outputs

Variable:	**Value:**	**Units:**
Robot Mass	2248.7	g
Number of wheels & motors:	2	
Desired Max speed:	0.4	m/s
Desired Acceleration time:	0.5	s
Wheel radius (the largest wheels):	0.0365	m
Wheel angular velocity:	105	rpm
Rolling resistance torque:	0.0141	N*m
Acceleration torque:	0.0328	N*m
Factor of Safety:	1.5	
Gearbox efficiency:	0.8	
Belt drive efficiency:	0.85	
Motor Power Requirement:	0.3406	Watts

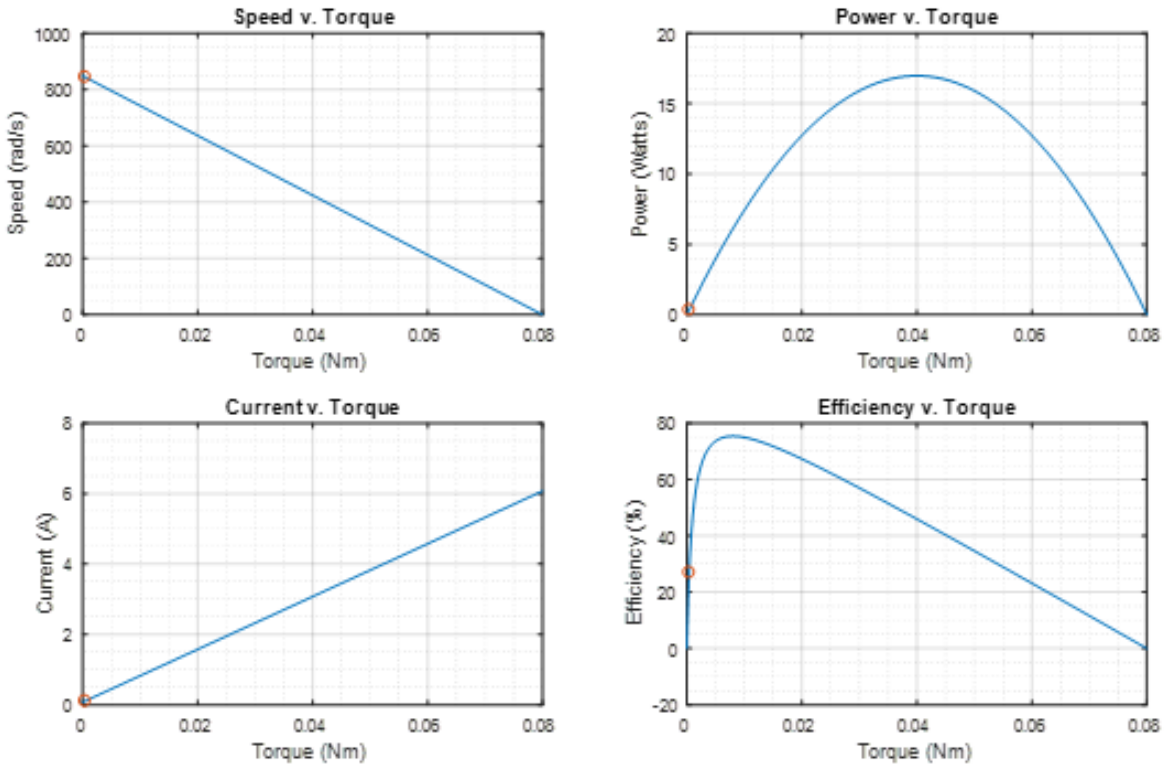


Figure 4: Motor Analysis Specifications

Based on our requirements, the Faulhaber motor would operate at 8060 RPM and with an ideal gear ratio of 77:1. Since the existing gearbox that attaches to the motor is 66:1, a 1:1 timing belt drive would be close to an ideal final drive.

However, using this gear ratio, our wheels would be able to produce 3.6 Nm of torque at max acceleration, which corresponds to 98 N of force. With a coefficient of friction of 0.8 for rubber to concrete, this force would require 25 kg of total robot mass in order to hold traction. First, this analysis indicates that weight should be as low and centrally located as possible. Second, we clearly will need to control the motor speed to avoid loss of traction and ensure proper encoder function.

Despite the fact that numerous other, smaller motors would work just as well for our needs, we are initially selecting the Faulhaber motor since it is available and easily exceeds our design requirements. We intend to use the 66:1 planetary gear reduction and a 1:1 timing belt as the final drive, and roll on two 7.3 cm blue rubber wheels.

To drive the sensor shaft, we are going to use a 250:1 Micro Metal Gearmotor HPCB. This motor serves the purpose of rotating the shaft with the necessary torque while minimizing the additional mass from an added motor.

Bill of Materials

Note 1: Stock is grouped by letter. Stock to purchase is listed at the bottom of the table.
Note 2: Items without a price we have claimed or can find in Couch Labs, Machine Shop, or are provided by the instructor.

Table 2: Bill of Materials

ITEM NO.	Description	Part No.	Vendor	Unit Price	QTY.	TOTAL PRICE	Stock
1	Axle Bracket		Machined		2	$ -	A

2	1/4" D-Profile	8632T139	McMaster Carr		3	$ -	C
3	Wheel Hub		Thayer		2	$ -	
4	Large Wheel		Thayer		2	$ -	
5	20 Teeth MXL Series Timing Belt Pulley	1375K21	Thayer		2	$ -	
6	8-32 Hex Nut	90480A009	Thayer		15	$ -	
7	8-32 Flanged Button-Head Socket Cap Screw	91355A075	McMaster Carr	$ 7.13 per 25	12	$ 7.13	
8	6-32 Phillips Head 3/4"		Thayer		24	$ -	
9	Bushing (unflanged)		Thayer		2	$ -	
10	1/4" Oil-Embedded Flanged Sleeve Bearing	6338K411	McMaster Carr	$ 0.67	2	$ 0.67	
11	1/4" Flanged Ball Bearing		Thayer		1	$ -	
12	1/4" Unflanged Ball Bearing		Thayer		1	$ -	
13	Oil-Embedded Thrust Bearing	5906K562	McMaster Carr	$ 0.48	2	$ 0.96	
14	Encoder	E4T-360-250-S-H-D-B	US Digital		3	$ -	
15	Motor Bracket		Machined		2	$ -	B
16	Faulhaber Motor and GearBox		Thayer		2	$ -	
17	30 Tooth MXL Series Timing Belt Pulley	1375K28	McMaster Carr	$ 12.31	2	$ 24.62	
18	Drive Belt	7887K15	McMaster Carr	$ 2.67	2	$ 5.34	
19	Chassis Base	89015K18	McMaster Carr		1	$ -	D
20	Omni Bracket		McMaster Carr		2	$ -	E
21	10-24 x Shoulder Bolt	91259A544	Thayer		2	$ -	
22	10-24 Hex Nut	90480A011	Thayer		2	$ -	
23	Kornylak Omniwheel	FXA317 (2052BX CAT-TRAK)	Kornylak	$ 8.80	2	$ 17.60	
24	Upper Plate	8560K354	McMaster Carr		1	$ -	A

25	Rabbit MicroController		Thayer		1	$ -	
26	Cross Bar	8975K578	McMaster Carr		1	$ -	D
27	Power Amplifier		Thayer		2	$ -	
28	0.5" Standoff		Thayer		5	$ -	
29	1.5" Standoff		Thayer		10	$ -	
30	2" Standoff		Thayer		2	$ -	
31	Tower Bushing Mount		McMaster Carr		1	$ -	A
32	Tower Motor Mount		McMaster Carr		1	$ -	A
33	250:1 Micro Metal Gearmotor HPCB 12V	3044 (Pololu)	Thayer		1	$ -	
34	D-Shaft Coupler		Machined		1	$ -	A
35	6-32x3/16 Set Screw	94105A143	Thayer		2	$ -	
36	Lithium Ion Battery	VNR1577	Horizon Hobby	$ 22.99	2	$ 45.98	
37	3Pos Toggle Switch		Thayer		3	$ -	
38	IR Sensor Bracket		Thayer		1	$ -	
39	Sharp IR Sensor - Short Range	GP2Y0A60SZLF	Pololu	$ 11.95	1	$ 11.95	
40	Perforated Electrical Board		Thayer		1	$ -	
41	L7805 5V Regulator	LM7805CT/NOPB	Thayer		1	$ -	
42	X Term Block 5.08MM HORZ 2POS PCB	ED2609-ND	Thayer		1	$ -	
43	X Term Block 5.08MM HORZ 3POS PCB	ED2610-ND	Thayer		7	$ -	
44	0.33 microF, 25 V electrolytic capacitor		Thayer		1	$ -	
45	0.1 microF, 25 V electrolytic capacitor		Thayer		1	$ -	
46	**Stock A**	In Possession	Thayer		1	$ -	
47	**Stock B** - 12" x 2" x 2"Aluminum L-Bracket (1/4" Wall Thickness)	8982K36	McMaster Carr	$ 12.05	1	$ 12.05	

48	**Stock C** - 1/4" D-Shaft Profile	8632T139	McMaster Carr	$ 9.10	1	$ 9.10	
49	**Stock D** - 1/8" x 12" x 12" Aluminum Stock	89015K18	McMaster Carr	$ 26.14	1	$ 26.14	
50	**Stock E** - 12" x 1" x 1" Aluminum L Bracket (1/8" Wall Thickness)	8982K4	McMaster Carr	$ 2.59	1	$ 2.59	
		TOTAL				$ 164.13	

Analysis of Sensors and Other Components Selected

Sensor Selection and Specifications

IR Sensor

We incorporated the SHARP IR sensor model GP2YOA60SZLF, which has a minimum range of 10 cm and a maximum range of 150 cm. We chose to use an infrared sensor due to its high resolution, low cost, and faster response times. Additional factors include its acceptable sensitivity when the beam is not directly perpendicular to object and its high accuracy when sensing white objects. Although the IR sensor has non-linear characteristics when outputting voltages, these are well known and can easily be modeled/linearized.

Encoders

We incorporated 3 US Digital encoders (model E4T-360-250-S-H-D-B), each of which resolves position of the shaft at 1440 counts per revolution. We chose to use these encoders because they were readily available and easy to implement. From the encoder readings, we can calculate orientation, position on the map, total distance traveled, and velocity.

Battery Selection and Specifications

Listed below is a summary of the major electronic components required for the robot, as well as each component's operating voltage and max current draw.

Table 3: Electrical Max Current Draw

Part	Quantity	Operating Voltage (V)	Max Current Drawn (mA)	Total Current Drawn (mA)
Microcontroller (BL2600)	1	11.1	1081	1081
Faulhaber Moten DC Motor 2342L012CR	2	12	1400	2800
Sensor Motor	1	12	800	800
Sharp IR Sensor (10-150cm)	1	2.7-5.5	33	33
			Total Max Current Drawn	4714

We can safely assume that our electronic components will not be operating at max current. Using capacity = (max current drawn) * (hours) with the specification of minimum battery life lasting at least ~20 minutes, the minimum capacity required for the battery can be calculated as follows. 1.65Ah = 4.714A * 0.35 hours. Increasing the capacity beyond 1.65Ah (1650mAh) can increase the battery life. Thus, we decided on using two 2200mAh LiPo batteries to enable both forward and backward movement, reduce the overall load on the robot, and account for the current drawn by the electronic components.

Table 4: Battery Specifications

Battery Selected	Voltage (V)	Capacity (mAh)	Discharge Rating
VENOM LiPo Battery (VNR1577)	11.1	2200	20C

Since the VENOM battery has a discharge rating of 20C, the maximum safe continuous current draw can be computed by 20 * 2.2A = 44A. Thus, 44A is the maximum sustained load that can be safely put on the battery, which is much higher than our intended load.

Electrical System Design

Power Distribution System

For the 5V regulator, we used the Texas Instruments model LM7805CT/NOPB voltage regulator. This model outputs 5V (with a 2% margin) and has a maximum 1.5A output. A 0.33**µ**F electrolytic capacitor from input to the 5V regulator to ground and a 0.1**µ**F electrolytic capacitor from output of the 5V regulator to ground were added. The input capacitor is added to filter the supplying voltage while the output capacitor is added to improve transient response and stability. A heat sink was added to the 5V regulator to dissipate heat and raise the potential maximum current output. A total of 8 terminal blocks were used to facilitate power bussing. A perfboard with solder pads was used to hold the power distribution components in place. A standoff was added to separate the cladded perfboard from the surface of the robot. Three 3-position DPDT switches were incorporated into the robot. A 15k**Ω** resistor connected to ground was connected in parallel to each 5V input from the switch farthest from the sensor. Two power amplifiers were implemented, and both have unity gain. One power amplifier receives two inputs, each controlling a wheel, from the BL2600 and outputs each specified voltage to its respective motor. The other power amplifier receives one input from the BL2600 controlling sensor position and outputs the specified voltage to the sensor motor.

Switches

The first switch (closest to the sensor) turns the BL2600 board and the power distribution system on (flip switch towards BL2600) and off. The second switch (middle) turns the power amps on (flip switch away from BL2600) and off. The third switch (farthest from sensor) controls the state of the program. To start the program, the third switch is flipped away from the BL2600 board. To pause the program, the third switch is in the neutral position. To resume the current program, the switch is flipped from neutral to start. To print a map, the switch is flipped towards the BL2600 board. To restart the program, the switch must go from positions print to start.

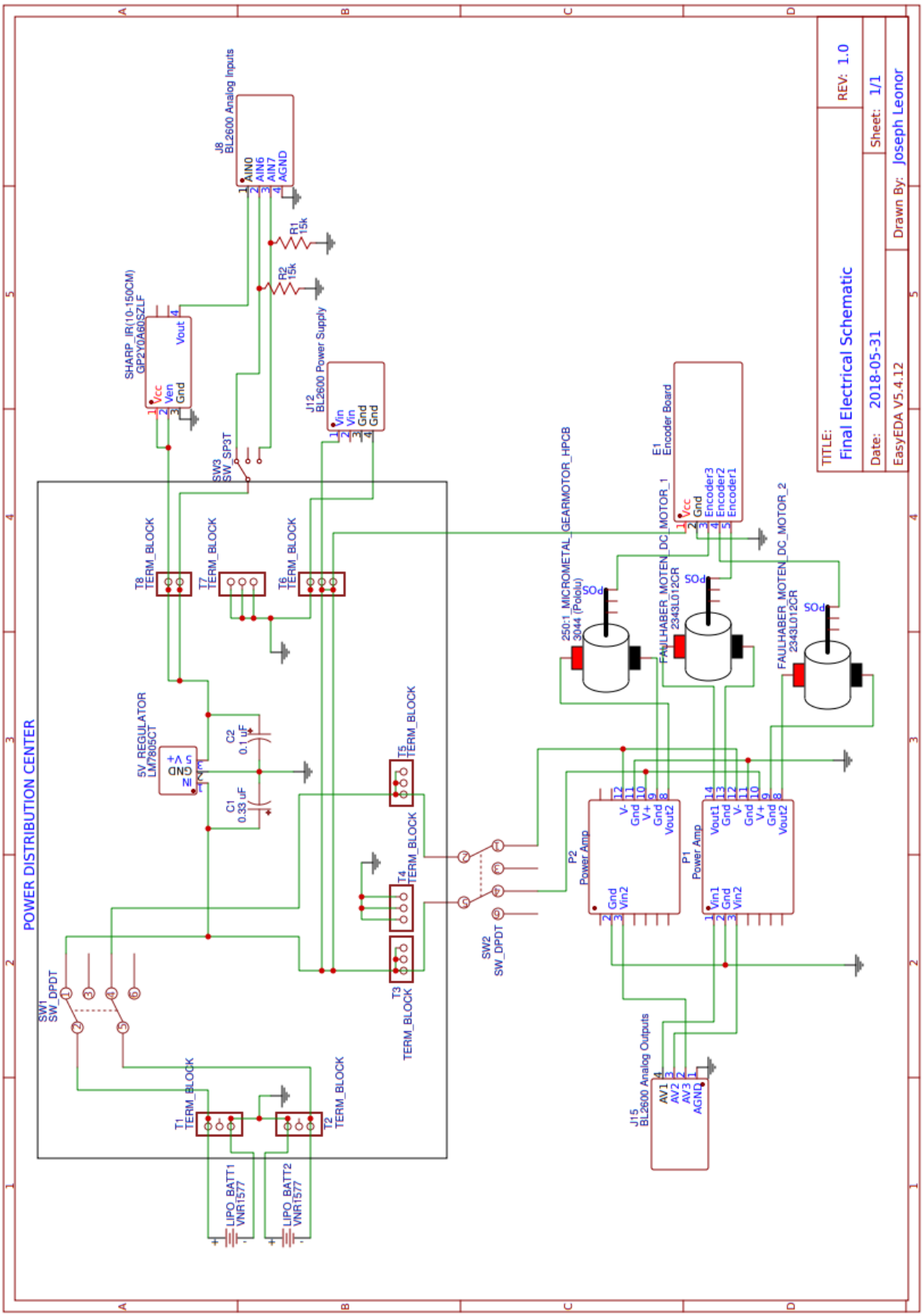


Figure 5: Schematic of Power Distribution System

Modelling

Sensor Modelling

In order to characterize the SHARP IR sensor model GP2YOA60SZLF, the sensor voltage was recorded at various distances away from an obstacle.

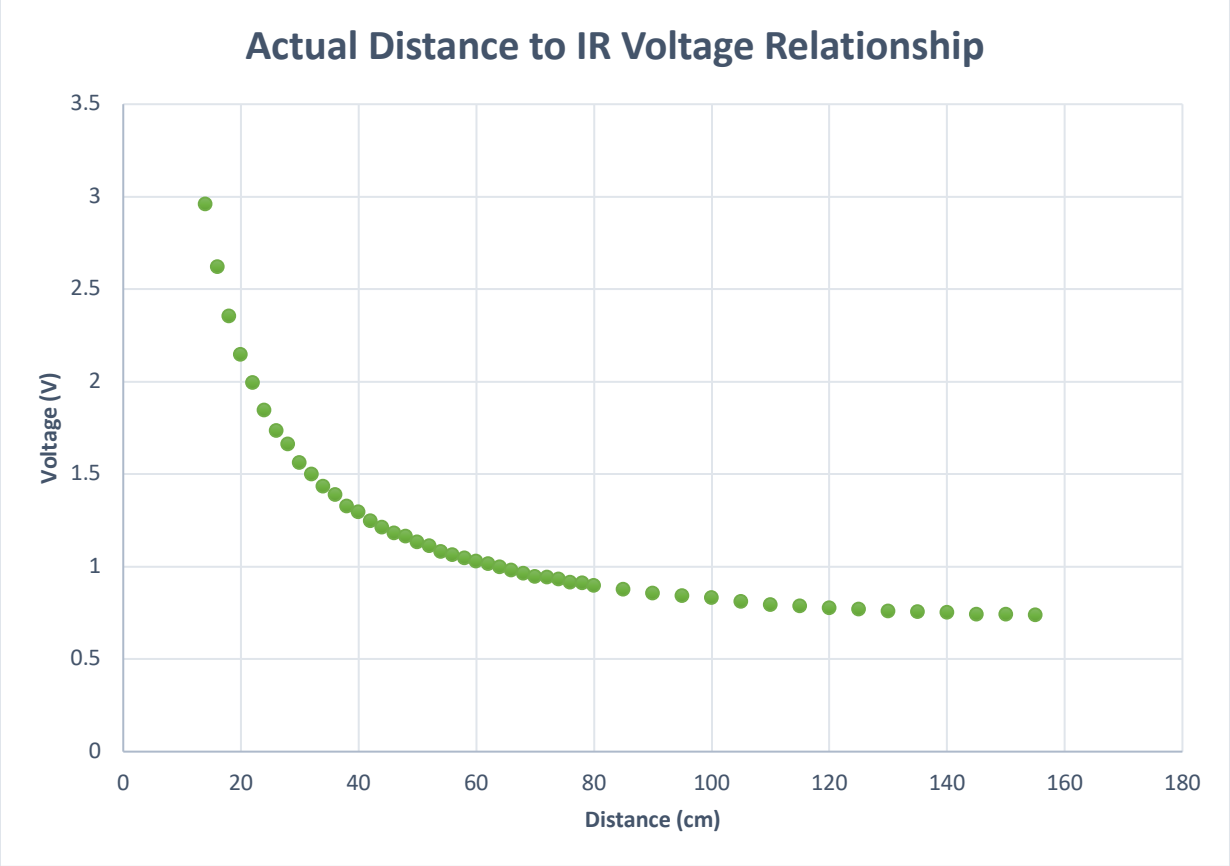


Figure 6: Actual distance to obstacle in cm vs IR voltage reading relationship

This relationship between distance to obstacle in centimeters and IR voltage readings were characterized using seven different models on Matlab's cftool including but not limited to rational fits, inverse voltage polynomial fit, and inverse distance exponential fit. The R-squared values for all the models tested were greater than 0.99, therefore the models closely correspond to the actual data.

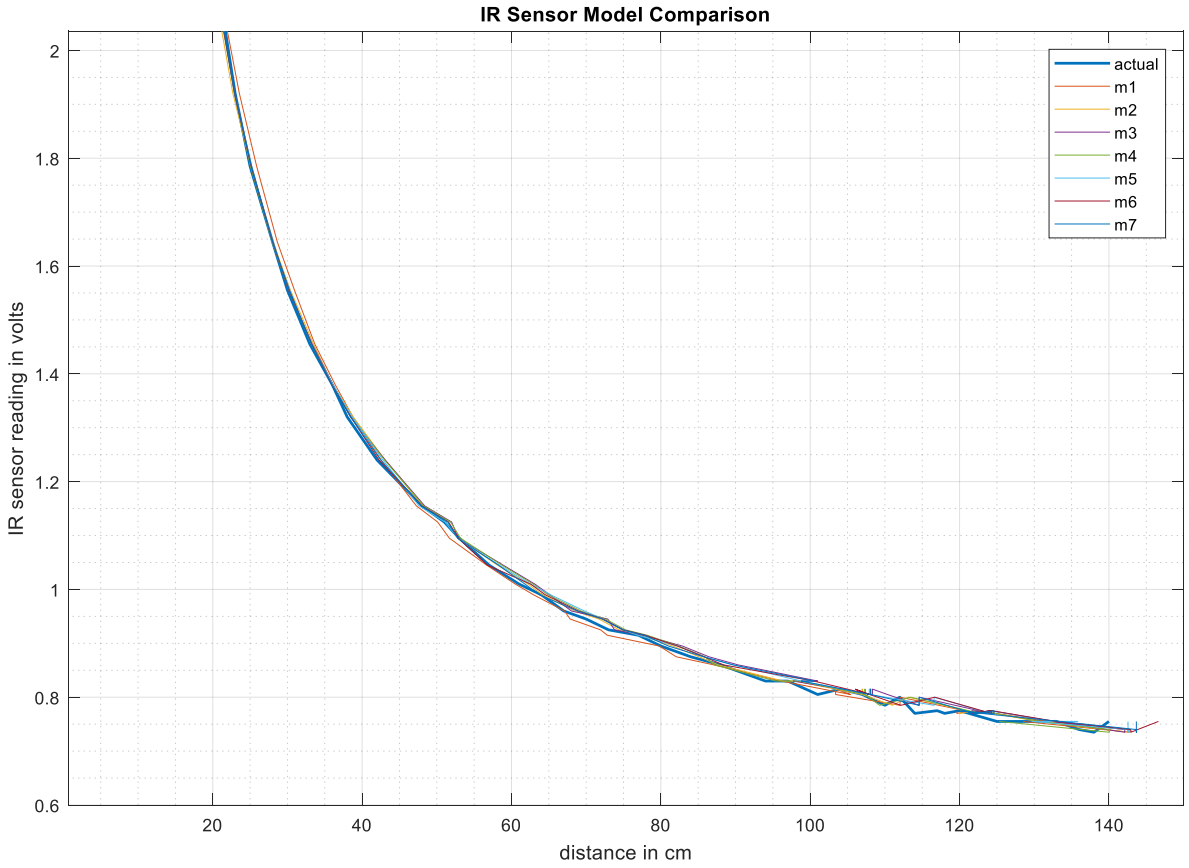


Figure 7: Zoomed in view of 7 different models experimental accuracy.

The model that worked the best experimentally and minimize the error (SSE) turned out to be model 2 which is a 4 degree polynomial fit bisquare model.

4 degree linear polynomial bisquare model:
f(x) = p1*x^4 + p2*x^3 + p3*x^2 + p4*x + p5
Coefficients (with 95% confidence bounds):
p1 = 152.4 (94.02, 210.7)
p2 = -381.3 (-582.5, -180.1)
p3 = 399.8 (151.7, 648)
p4 = -134.4 (-263.2, -5.651)
p5 = 26.98 (3.495, 50.47)

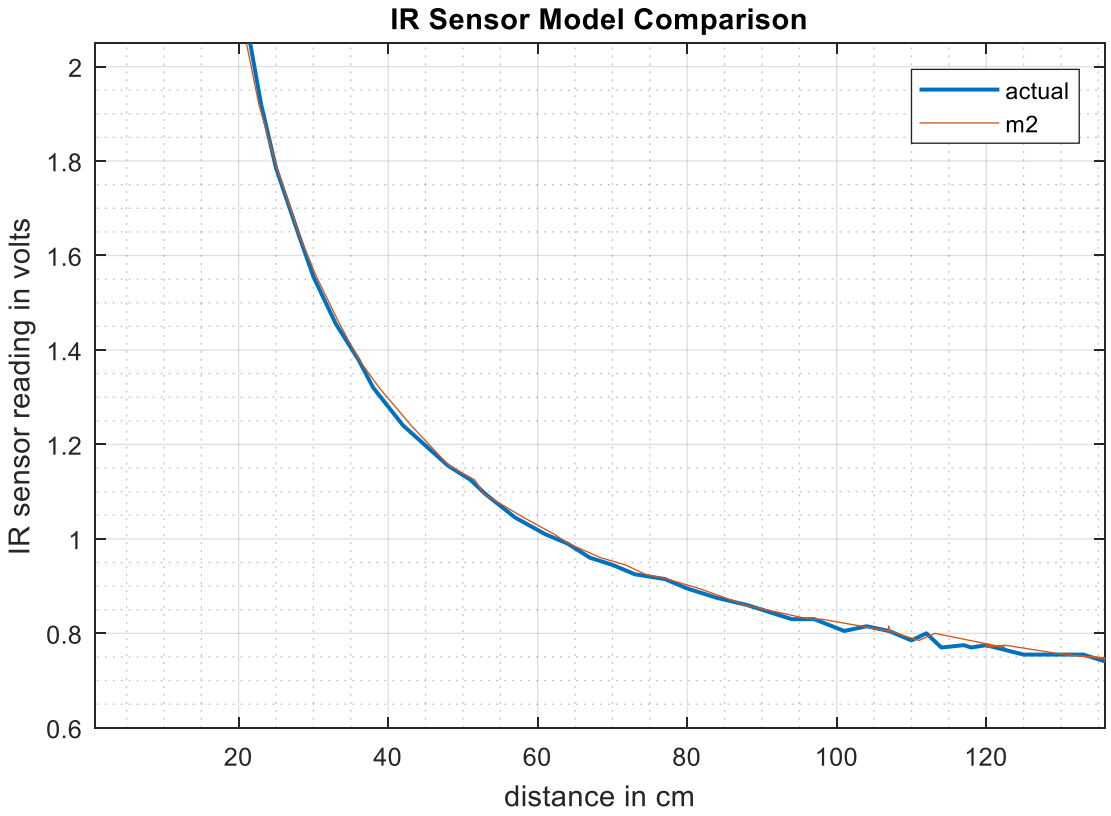


Figure 8: Model 2 (selected 4 degree polynomial model) prediction plotted against actual distance

See Appendix C for the Matlab code that compares several different model predictions with experimental data as well as Appendix D with the C-code that contains the code that was used to collect experimental vs actual distance data with every model.

Sensor Motor Modelling

In order to accurately model the motor controlling sensor position, 2 volts were inputted into the motor while the encoder recorded position.

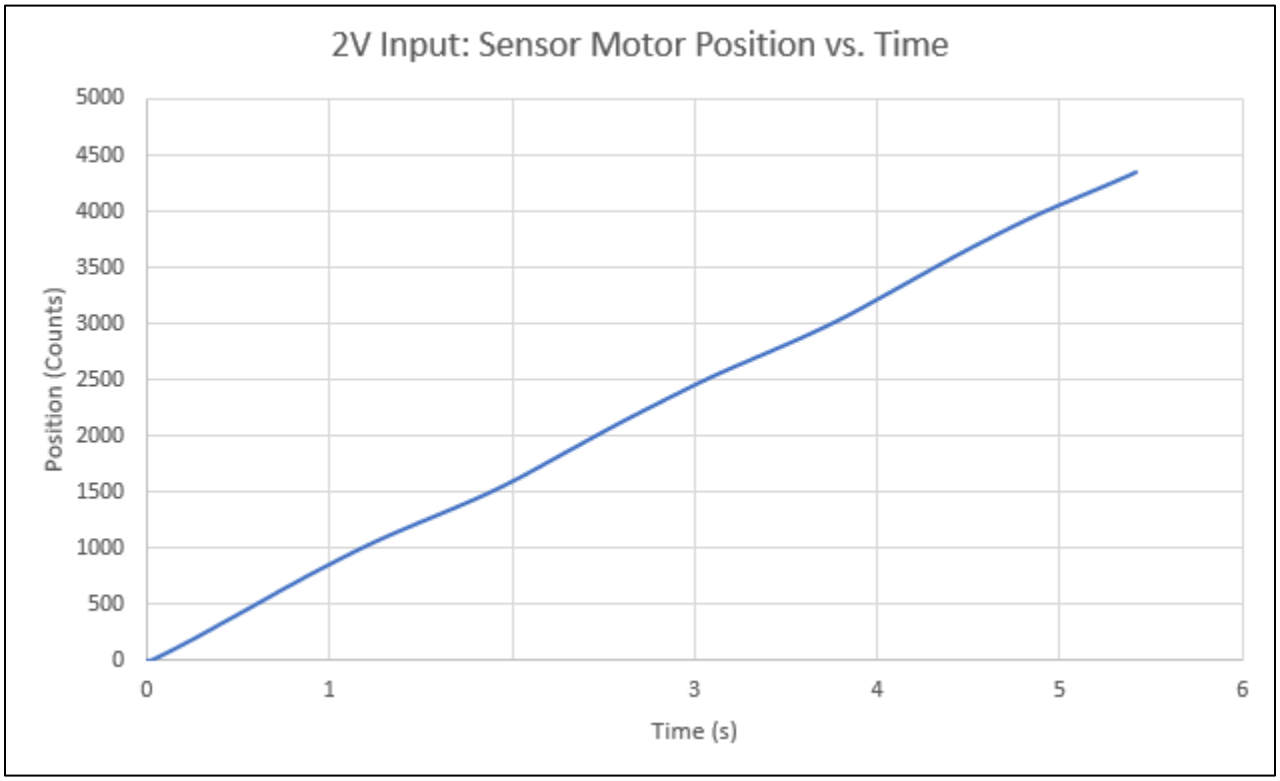


Figure 9: Sensor Motor Position vs. Time

From the position values, we are able to estimate instantaneous velocity at any given time.

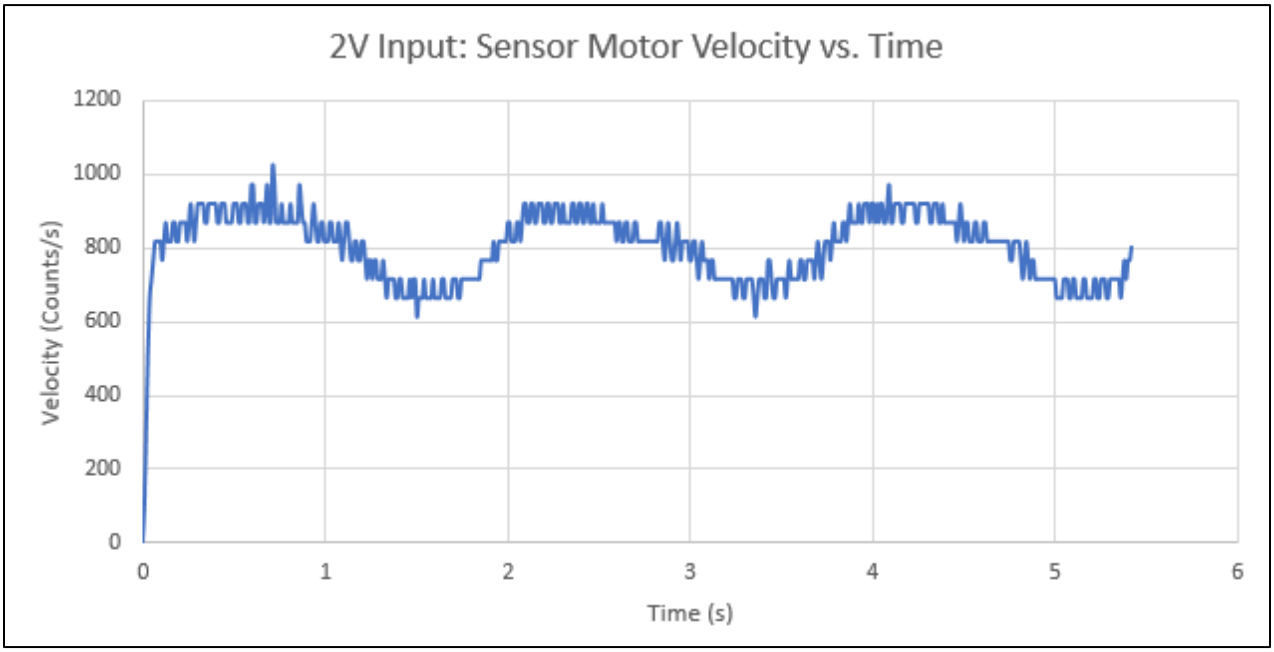


Figure 10: Sensor Motor Velocity vs. Time

As we can observe, the sensor's instantaneous velocity is sinusoidal (i.e. it has a period). This is due to disturbances introduced when rotating the sensor 360°, most likely caused by small asymmetries in the rotating parts. However, these fluctuations in instantaneous velocity are negligible since the sensor's position over time is mostly linear.

Drive Motor Modelling and Controller Design

We modelled the entire driveline as an ideal motor with no resistance, friction, or inductance. Our model goes from the input signal to the power amp, through the motor and belt drive to the output shaft where the encoder is mounted. The driveline thus follows the ideal motor equation:

$$V = k*\omega$$

V = voltage applied to the motor
k = constant relating motor speed to voltage
ω = the rotational speed of the motor

To characterize the model we delivered voltage to the power amp and compared these to the output speed of the motor, in encoder ticks per time tick. This provided us with a k value of exactly 3 volts per encoder tick per time tick.

Using this extremely simple motor model we designed a proportional integral motor controller in the form:

$$V = kP * \omega_{command} + kI * \omega_{error\ total}$$

V = voltage applied to the motor
kP = proportional gain (3 volts per encoder tick per time tick)
$\omega_{command}$ = the desired rotational speed of the motor
kI = integral gain (1 volt per encoder tick per time tick * # of time intervals)

We arrived at the value for the "integral" gain simply by increasing the value until the motors strongly resisted us trying to slow them down by hand while running the motor controller code. When we implemented the controller on both left and right wheels at once, the robot drove in an extremely straight line. However, even if the controller had been prone to drifting left or right, the navigation controller would ultimately account for the the error and allow the bot to drive without issue.

Heading Controller Design

Our heading controller makes the drive motor controllers look complicated! It outputs a commanded difference between right and left wheel velocities proportional to the difference between commanded and current heading. A fairly complete design is included in the Matlab modelling of navigation control section, but we will replicate the equations here for clarity:

$$dv = k\theta * (\theta_{command} - \theta_{bot})$$
$$vR = v_{avg} + dv$$
$$vL = v_{avg} - dv$$

dv = the desired difference between the left and right wheel velocity
$k\theta$ = the heading control gain
vR = the velocity command to the right wheel
vL = the velocity command to the left wheel
v_{avg} = the current velocity of the bot

We did not ever test the steady state error of the heading controller, but simply tuned the proportional gain to make it snappy, with the understanding that any steady state error would be accounted for in the navigation controller.

Dynamics and Kinematics Modelling

Dead Reckoning

The most important task in modelling was to keep track of the cartesian position of the robot at any time using only a series of encoder outputs. After surprisingly fruitless research, we opted to derive the kinematic position equations ourselves in a more intuitive way. We are working entirely using the change in encoder ticks between measurements at a specific time interval. Below we construct a crude approximation of our position that will converge to an exact calculation as the time

step goes to zero. We then introduce an exact equation, but will explain why this equations is undesirable. This sequence of equations walks through the process of dead reckoning in a way that is very easy to code, and could be shared with future mechatronics students who struggled as we did.

Change in heading:

$$\Delta\theta = (nl - nr) * step / wheelbase$$

$\Delta\theta$ = change in heading between last two measurements (radians)
nr = number of right encoder ticks between last two measurements
nl = number of left encoder ticks between last two measurements
step = linear distance that the wheel rolls per encoder tick
wheelbase = distance between left and right wheel

Current heading:

$$\theta_{current} = \theta_{previous} + \Delta\theta$$

$\theta_{current}$ = current heading (radians)
$\theta_{previous}$ = previous heading (radians)
$\Delta\theta$ = change in heading between last two measurements (radians)

Average heading and path distance travelled:

$$\theta_{avg} = (\theta_{current} + \theta_{previous}) / 2$$
$$\Delta d = [(nl + nr) / 2] * step$$

θ_{avg} = average heading between readings (radians)
Δd = path distance travelled by by center of bot between measurements (this works for any shape path; if the path is an arc Δd is the exact arc distance)

Approximate change in X and Y location:

$$\Delta X = \Delta d * cos(\theta_{avg})$$
$$\Delta Y = \Delta d * sin(\theta_{avg})$$

ΔX = change in X position between last two measurements
ΔY = change in Y position between last two measurements

Current X, Y location:

$$X_{current} = X_{previous} + \Delta X$$
$$Y_{current} = Y_{previous} + \Delta Y$$

$X_{current}$ = current X position
$Y_{current}$ = current Y position
$X_{previous}$ = previous X position
$Y_{previous}$ = previous Y position

Exact estimated change in X and Y location derived from SAS law of cosines:
These equations can only be used when $\Delta\theta$ is greater than zero. While geometrically more exact, they will give results that vary with speed. This is because with some constant level of random variation in encoder values between left and right (mostly due to driveline lash between the encoder and wheels), shorter distance steps between measurements will tend to decrease the value of *R*. Consequently, the path will be calculated as a sum of small radius arcs, and the calculations will

underestimate the total distance travelled. With longer distances between measurements the calculations will become more accurate, but will fail to account for variable radius turns. We found the approximate equations to be much more accurate overall in testing.

$$R = \Delta d / \text{abs}(\Delta\theta)$$
$$d_{linear} = R * \text{sqrt}(2 * [1-\cos(\Delta\theta)])$$
$$\Delta X = d_{linear} * cos(\theta_{avg})$$
$$\Delta Y = d_{linear} * sin(\theta_{avg})$$

R = radius of turn being made between two measurements
d_{linear} = linear distance between the endpoints of the path arc
ΔX = change in X position between last two measurements
ΔY = change in Y position between last two measurements

To finish up our kinematic model of the robot, all we simply needed to empirically acquire the step value (linear distance the wheel rolls for one encoder tick) and the wheelbase (distance between the neutral axis of the left and right wheel). We estimated step as:

$$step = \pi * (\text{wheel diameter}) / (\text{encoder ticks per revolution})$$

We then performed linear distance trials using our path following controller and adjusted the value of step until the robot could move an exact commanded distance. Our analysis estimated a value of 0.158 [cm/tick] which we corrected to 0.0160 [cm/tick] through experimentation. For wheelbase we performed a similar process. We first measured wheelbase to be 15.30cm. We then programmed a large square path using our path following controller. If the bot returned to the endpoint too far left, this meant that effective wheelbase was actually smaller than programmed wheelbase, and too far right meant a larger effective wheelbase than programmed. A programmed wheelbase of 15.40cm was required for navigational accuracy. These two constants and the above equations form a complete kinematic model of our differential drive robot.

Ignoring Dynamic Effects

We opted to forgo dynamic modelling for two reasons: First, the bot functions at fairly low speeds where dynamic effects would be inconsequentially different from kinematic expectations. Moreover, torque of the motors is extremely large relative to the forces required to accelerate and decelerate the bot, so the bot is unlikely to overshoot or undershoot speed commands by any significant margin or for any significant amount of time. Second, we programmed a ramping function that limited all speed requests to the motor to a maximum acceleration to ensure that the wheels would not slip.

Modelling a Vector Based Potential Navigation Controller

As a contingency to A* path planning, we modelled potential based pathfinding to gain intuition about its possible implementation. We produced a simple, animated map view of the robot that updates based on kinematics equations almost identical to those developed above, and uses the same parameters (see Matlab code in Appendix E). The robot model drives on a grid map, which is stored as a two dimensional matrix of ones (obstacles) and zeros (free space). At every timestep it performs vector summation of a unitized attractive vector to the endpoint and repulsive vectors from each obstacle that are proportional to the inverse square of their distance according to the following equations:

$$d_{obstacle} = \text{sqrt}[\ (X_{bot} - X_{obstacle})^2 + (Y_{bot} - Y_{obstacle})^2\ ]$$
$$X_{repulsive} = \sum[(1-d_{obstacle})*(1-d_{zero})]*[(X_{bot} - X_{obstacle})/(d_{obstacle})^2\ ]$$
$$Y_{repulsive} = \sum[(1-d_{obstacle})*(1-d_{zero})]*[(Y_{bot} - Y_{obstacle})/(d_{obstacle})^2\ ]$$
$$\theta_{end} = \text{atan2}[(Y_{end} - Y_{bot})\ ,\ (X_{end} - X_{bot})]$$
$$X_{attractive} = \cos(\theta_{end})$$
$$Y_{attractive} = \sin(\theta_{end})$$

$d_{obstacle}$ = distance to an obstacle point
X_{bot} = X position of bot (from dead reckoning)
Y_{bot} = Y position of bot

$X_{obstacle}$ = X position of a single obstacle (from mapping of IR sensor outputs)
$Y_{obstacle}$ = Y position of a single obstacle
$X_{repulsive}$ = Sum of repulsive vector X components for all obstacles closer than d_{zero}
$Y_{repulsive}$ = Sum of repulsive vector Y components for all obstacles closer than d_{zero}
d_{zero} = A chosen distance at which a repulsive vector is no longer generated
θ_{end}= angle from current position to end point
X_{end} = X position of end point
Y_{end} = Y position of end point
$X_{attractive}$ = X component of unitized endpoint attractive vector
$Y_{attractive}$ = Y component of unitized endpoint attractive vector

A commanded heading can then be calculated by taking the arctangent of the vector addition as follows:

$$\theta_{command} = \text{atan2}[\ (ka*Y_{attractive} + kr*Y_{repulsive}\ ),\ (ka*X_{attractive} + kr*X_{repulsive}\ )\ ]$$

$\theta_{command}$ = the desired heading of the bot

Robot heading control is then performed using a simple proportional controller to the error between the heading of the bot and the command heading:

$$dv = k\theta * (\theta_{command} - \theta_{bot})$$
$$vR = v_{avg} + dv$$
$$vL = v_{avg} - dv$$

dv = the desired difference between the left and right wheel velocity
$k\theta$ = the heading control gain
vR = the velocity command to the right wheel
vL = the velocity command to the left wheel
v_{avg} = the current velocity of the bot

The simulation allowed us to immediately diagnose 3 problems with this style of control and solve them very simply. First, the simulated robot became unstable when travelling parallel to walls. To solve this, we calculate the repulsive vector based on a lead point rather than the bot's center point. This is analogous to adding a derivative term, but it is computed geometrically. The equations for the X and Y position of the robot are now:

$$X_{bot} = lookahead * \cos(\theta_{bot})$$
$$Y_{bot} = lookahead * \sin(\theta_{bot})$$

lookahead = lead point distance to use in obstacle vector computation (we used 10cm for our simulation and on the actual robot)

Second, the simulated robot would get stuck in even the shallowest of local minima. To escape all local minima with edge angles less than the radial lines from the endpoint, we implemented a unitized momentum vector to be summed with the repulsive and attractive vectors. The momentum vector brings the robot's behavior closer to that of a conservative potential field, which would have no local minima. The momentum vector gain must be larger than the attractive vector gain for this method to be effective.

$$X_{momentum} = \cos(\theta_{bot})$$
$$Y_{momentum} = \sin(\theta_{bot})$$

$X_{momentum}$ = X component of momentum vector
$Y_{momentum}$ = Y component of momentum vector

Lastly, the simulated robot could still not escape an extreme concavity with wall angles steeper than radial lines from the endpoint (a square well for example). To solve this, we implemented a windup torque, which attempted to aim the robot back to the endpoint by rotating it back in the direction it came from.

$$\tau_{windup} = (\theta_{end} - \theta_{bot})$$

τ_{windup} = windup torque, limited to between $\pm 3\pi/2$ with logic

Combining the three corrections and tuning the gains in the simulator allowed us to escape any reasonable local minima and subsequently succeeded when implemented on the actual bot. Our navigation equations can be best described as vector-based potential navigation with soft, geometric wall following to escape local minima. Adding together all corrections gives us a new expression for the commanded heading and the heading controller itself:

$$X_{command} = ka*X_{attractive} + kr*X_{repulsive} + km*X_{momentum}$$
$$Y_{command} = ka*Y_{attractive} + kr*Y_{repulsive} + km*Y_{momentum}$$
$$\theta_{command} = \text{atan2}(Y_{command}, X_{command})$$
$$dv = k\theta * (\theta_{command} - \theta_{bot}) + k\tau * (\tau_{windup})$$

$X_{command}$ = X component of command vector
$Y_{command}$ = Y component of command vector
ka = attractive gain
kr = repulsive gain
km = momentum gain
$k\theta$ = heading control gain
$k\tau$ = windup torque gain

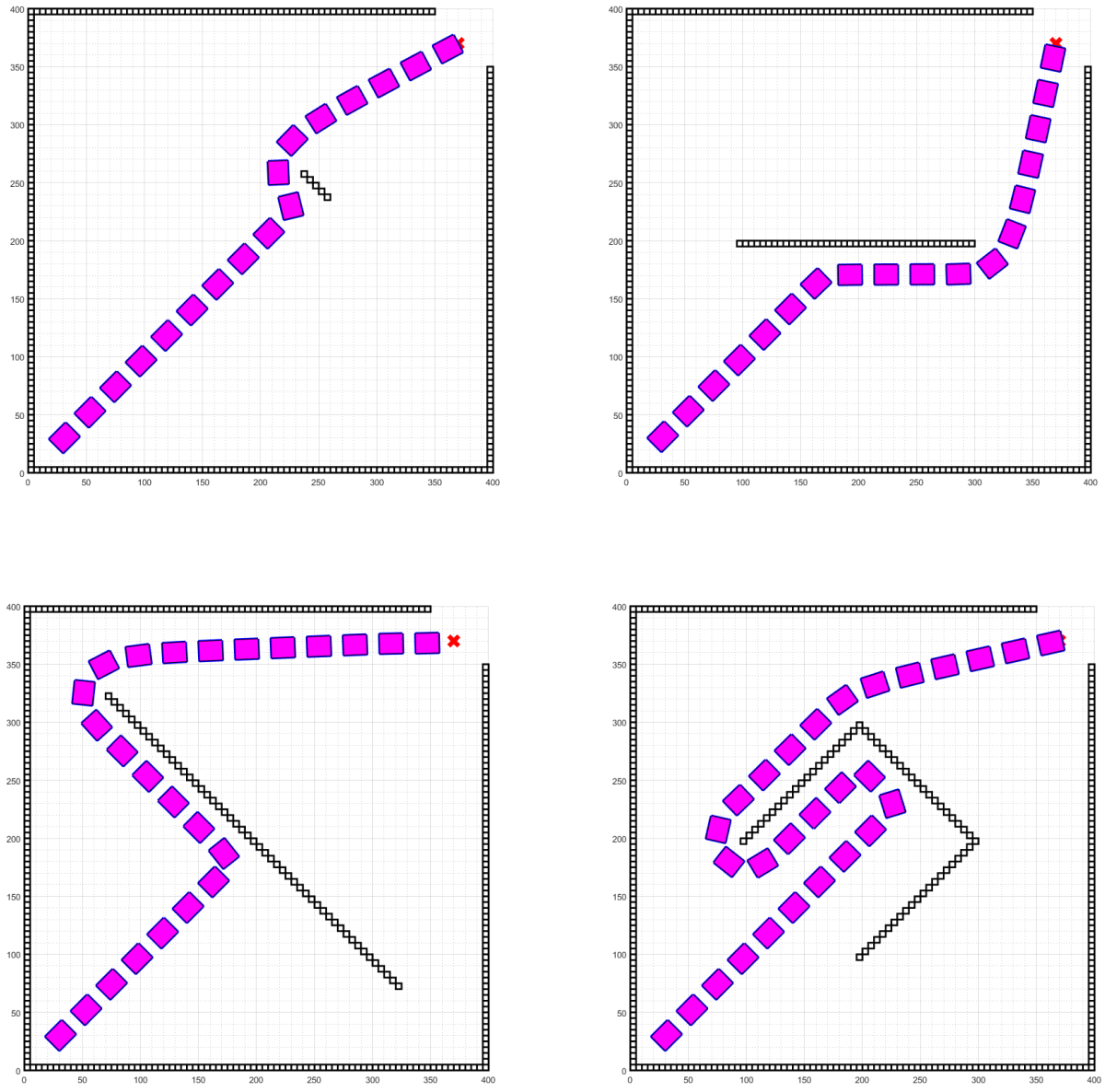


Figure 11: Freeze frame depictions of testing our navigation algorithms in our own animated Matlab simulation. Start location is in the bottom left, end location is in the top right. The magenta rectangle depicts the position of the bot over time. All of the navigation gains found by experimentation in the simulation worked properly when implemented on the actual robot microprocessor.

Software Design

See Appendix F for the C-code that runs the robot.

Vector Based Potential Navigation Algorithm

The potential navigation algorithm we implemented in our C-code on the BL2600 faithfully replicates the Matlab model of this algorithm described above and operates using all of the same mathematical expressions. In our C-code we implement potential navigation as a series of functions inside the motor controller costate that is timed to run every 100 time ticks.

Costate 2 below summarizes the implementation of this navigation function. See the Modelling section for detailed information on how the vector based potential navigation algorithm works.

Mapping Algorithm

The map of the 12ft x 12ft grid is represented by an 77 x 77 2D matrix represented using an array of 5929 nodes. Each node represents a 5cm x 5cm box in which either an obstacle or the robot could lie. The map is primarily used to store obstacles that the IR sensor picks up while sweeping.

Every voltage reading that the IR sensor picks up every 15 ticks is translated into a predicted distance using the `getDistance()` function at which an obstacle lies. This distance is then represented within the map by using the `IRMap()` function to increment the value at the corresponding node location (derived from the distance value, sensor angle, current bot position and heading) by 1.

The map is frequently parsed using the `parseMap()` function to filter out noisy IR data that lies close to the robot and increase the accuracy and reliability of our mapping algorithm. This is done by finding the location of the bot and doing an outward search of the number of grids defined in the local variable `searchradius` which is currently set to 10 grid boxes in each direction. This algorithm searches through the square radius and checks whether the element being searched has a number of IR points on the map greater than or equal to `REQWEIGHT` of 4 but less than the `MAPFILTER` value of 100. If this is true, all the neighbors of this node are checked and if the value of points placed on the neighbor's node ID is greater than or equal to the `REQWEIGHT` of 4, the number of valid neighbor points is incremented and the neighbor is marked as live by setting the value in the array equal to `MAPFILTER`. Once we reach 2 neighbors (as defined by `NEIGHBORS`), the value of the current node is also set to `MAPFILTER` thereby being marked at live and we have officially blocked these points on the map.

If the count of live neighbors set by `NEIGHBORS` has not been met, we decrement the neighbor `DEWEIGHT` (currently set at 0) below the `REQWEIGHT` marking the point as possibly a true block but not guaranteed to be an obstacle and give it an opportunity to pick up a few more points so it can meet the `REQWEIGHT` specification.

The `parseMap()` function is designed to prevent resetting the repulsive value of walls that were originally initialized to be repulsive while generating the map array.

Software Architecture and Overview of Major Functions

We designed our robot around switch interrupts and costates in order to do any preprocessing before the program runs and to gather data and control our robot using different timing specifications in order to generate an accurate map and control our bot around obstacles towards the goal.

Running the Program

After compilation and flashing the program to the BL2600, we set up a map array with a repulsive border and make the end box corner non repulsive so our bot does not steer clear of the end goal.

Wait Switch Interrupt

After turning on the motor, the wait switch starts up the program. Once this switch is triggered, we carry out the following initialization functions:

`stateInit()` – Initializes the values in the botState struct.

`sensorInit()` – Orients sensor upon program start and stores the start position of sensor into botState struct.

`checkInit()` – Initialize array of points to reference when checking the neighbors of a node.

`stateUpdate()` – This function takes in the current state of the bot in the form of a botState struct and updates all the parameters, checks for wheel position change rollovers, calculates change in x and y positions of the bot and gets a reading from the sensor. All this information is stored for future use in the struct.

The Costates

When the wait switch is triggered, a series of costates make up the state machine that controls the bot. There are 4 different costates as shown below.

Costate 1: Runs costate for robot control and obstacle avoidance, this costate runs once the program is started and waits for 100 ticks after finishing the functions.

`stateUpdate()` – Simply collects data from every input and performs the kinematics calculations to determine the (x, y) position and heading of the bot.

`obsVect()` – This function takes in the state of the bot, x, y coordinates for an obstacle and a map array to conduct an outward search from a robot to check whether there are obstacles around it. If there are obstacles that surround the robot, a repulsive x and y vector are calculated and stored.

`avoidObs()` – Sums the obstacle repulsive vector, the endpoint attractive vector and the momentum vector weighted with unique, tunable gains in order to compute a commanded heading. It also implements the windup torque to cause the bot to wall follow out of any local minima in which it becomes trapped. In the C-code, if the bot wall follows to the edge of the map the torque vector will reset and it will turn and wall follow back in the correct direction. `avoidObs()` also outputs the left and right wheel velocity requests based on the heading controller design outlined above. It even speeds up or slows the bot depending on the degree of peril it is facing from the surrounding obstacles.

`reqVel()` – The current bot state's requested left and right velocities are commanded to the left and right motors respectively. The change is done gradually by using a defined RAMP value to prevent the wheels from slipping by limiting the acceleration of the motors.

`controlLoop()` – Takes the velocity commands and faithfully forces the motors to spin at the commanded speeds using the drive motor controller explained above.

The entire sequence of functions runs 10 times a second to compute the best correction to make next. While it does not compute the shortest path, if the map is correct the bot will always reach the endpoint.

Costate 2: Runs costate for picking up frequent IR sensor readings and updating the map based on the mapping algorithm, this costate runs after costate 1 passes and waits for 15 ticks.

`stateUpdate()` – Updates the bot state as described above.

`IRxy()` – Turns an IR sensor reading distance into an x, y coordinate on the grid in centimeters, useful for figuring out where to map the sensor readings using the mapping algorithm.

`IRMap()` – maps the IR sensor x, y point in centimeter to a square in the map. The point at that square is incremented in the map array.

`sensorReverse()` – Changes the sign of the voltage sent to the IR sensor motor based on the desired `SENSORSPIN` set in the definitions at the beginning of the program. This function ensures that the sensor rotates back and forth covering a certain spin distance.

Costate 3: Runs costate for parsing and filtering the map, this costate runs after costate 1 passes and waits for 100 ticks.

`parseMap()` – This function takes in the state of the bot, the map and two arrays of neighboring grid coordinates. See the mapping algorithm section for more detail about this function. Ultimately, the function filters out noisy IR data and doesn't mark noisy points as obstacles that the robot must avoid.

Costate 4: Costate that senses any other switch interrupts enabling the bot to pause, break and restart the program, this costate runs after going into costate 1 waitfor of 100 ticks.

If the switch that started the program has been triggered to the middle setting, we pause the program, if flipped to the left we break the program and allow a short time for the debugging functions to complete. We can restart the program by flipping this switch back to the right.

See the section on switches for more detail about how the switches work.

Other Implementation Considerations

Dynamic Mapping

The mapping algorithm was developed further to become dynamic and therefore make points which were live at one point insignificant after a certain amount of time elapsed. While this feature is not present in our current version of the code, we had worked on implementing a mechanism that would take all the points above `MAPFILTER` and decrement them 1 below the `REQWEIGHT` after 2 minutes elapsed and the bot had not reached the goal.

This mechanism would make the bot more versatile and able to overcome situations where the bot traps itself into a corner by marking noisy points as live. Something that is likely to happen if the bot is faced with a narrow opening. Due to time constraints, we were unable to include this feature in the version of the code presented within this report.

A* Path Finding and Potential Function Hybrid

An alternate pathfinding algorithm we implemented was an A* optimal pathfinding algorithm combined with obstacle avoidance using a potential function. Since the BL2600 does not support multithreading running A* directly on the bot would require the bot controls coming to a halt while A* recomputes each time the bot senses a new obstacle.

We had optimized the speed of A* by running it on a low-resolution 12 x 12 grid and precomputing the heuristic using the Manhattan distance. Running A* on a low resolution grid outputs a list of points around 30 cm away from one point to the next. Each point was fed into the potential algorithm where the potential would take over as the robot moved from the previous point to the next point and thus avoid obstacles. We had started implementing the interaction between the high resolution grid and the A* low resolution grid and when the next point is sent to the potential function.

Although our A* worked precisely, there were some issues that aroused close to the deadline while merging the A* algorithm with the potential function and working out the logical interaction between the two functions and grids. Thus, we did not present this solution in detail within this report, but given more time, the hybrid function would have resulted in similar robot behavior as we currently have along with prioritizing the most optimal path to the end goal.

Performance and Evaluation

Motor Performance and Robot Speed

Our design target for the robot's maximum velocity was greater than 0.4 m/s. In order to evaluate this metric, we measured the amount of time required for the robot to travel 2 meters while at a steady-state velocity. Though the robot has potential to run much faster, we artificially limited the maximum speed in the program. The amount of time required to travel 2 meters was 4.21 seconds. The max velocity of the robot was calculated to be around 0.4751 m/s, which exceeded our design target. Our robot has the potential to complete the course at relatively high speeds; however, it is restricted to lower speeds due to limitations introduced by the sensor's ability to update and provide accurate distance values in a timely manner. If the robot traveled at higher speeds, it would not have enough time to react to information gathered from its environment to avoid obstacles. Thus, we program the robot to move at speeds around 0.15 m/s in order to account for these limitations and optimize course completion performance.

Mapping Performance

The mapping algorithm significantly improved the mapping accuracy of the bot. Without the algorithm, the bot would detect spurious points and avoid non-obstacles haphazardly and often lead itself into situations where the robot would be unable to move.

Below is a map of a two-walled course the robot generated before the mapping algorithm was implemented. The robot placed many spurious points from noisy IR readings which led to several locations without any obstacles that the robot would avoid.

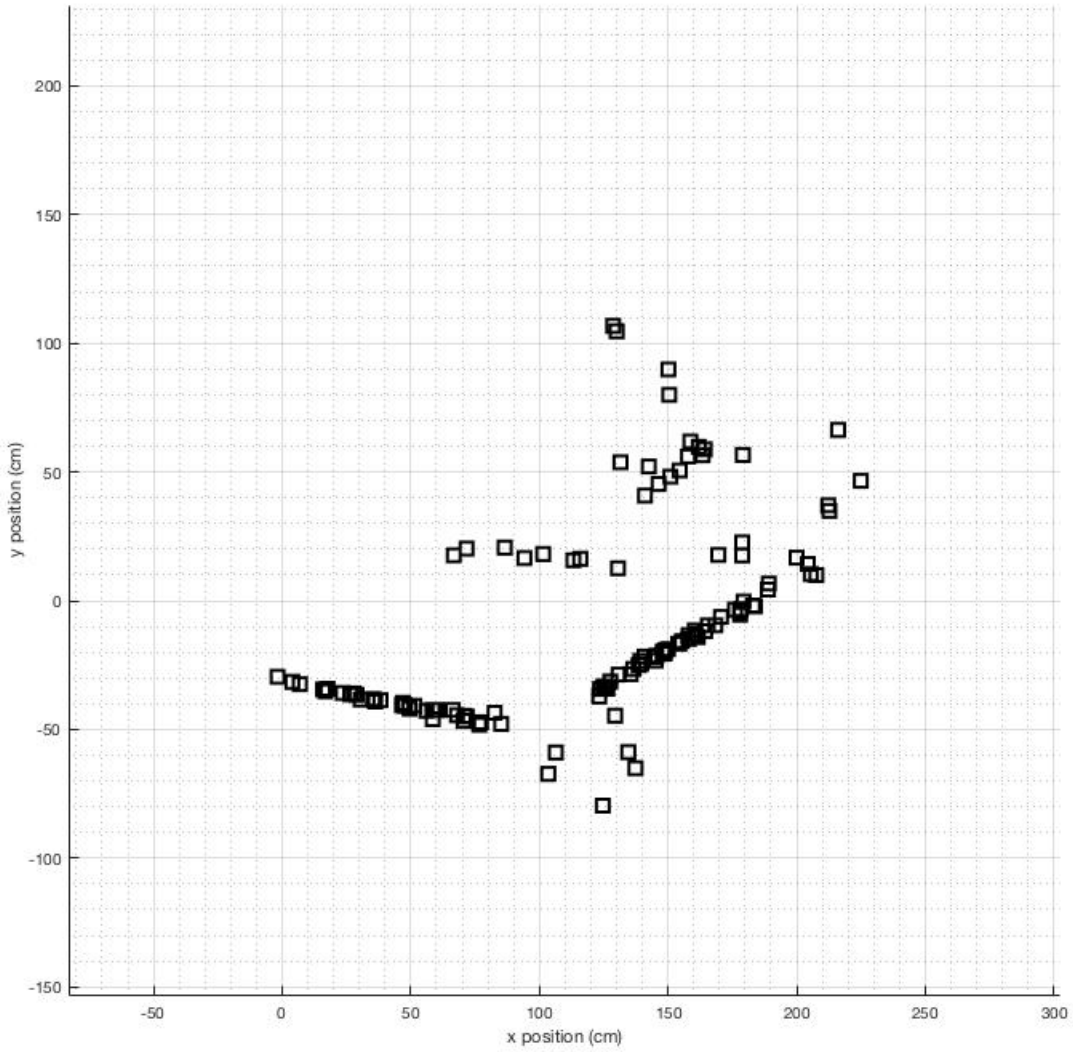


Figure 12: Generated map before implementation of mapping algorithm

After implementing the mapping algorithm, our mapped values become more accurate and defined, the robot filters the obstacles around it which means that the robot will never see single points on the map as long as the points are parsed. Note that if the points are not in the parse radius around the robot, the single points will stay present. Below are side-by-side images of two mapping tasks. The map quality generated increases significantly with the mapping algorithm.

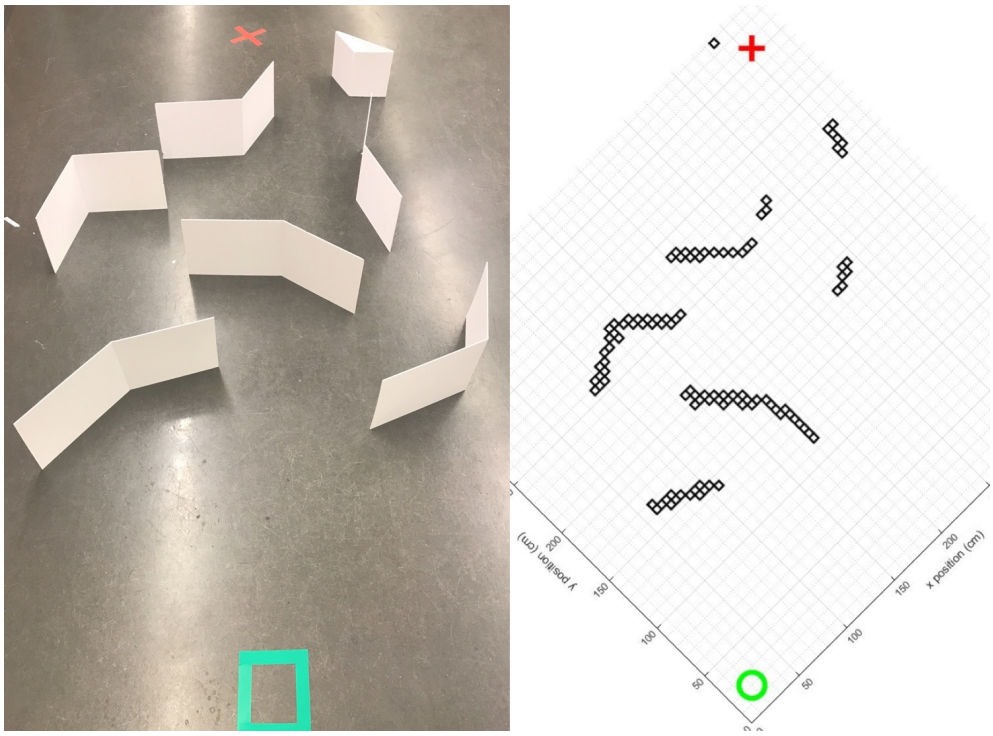


Figure 13: Mapping task 1, actual map (left) vs generated map (right)

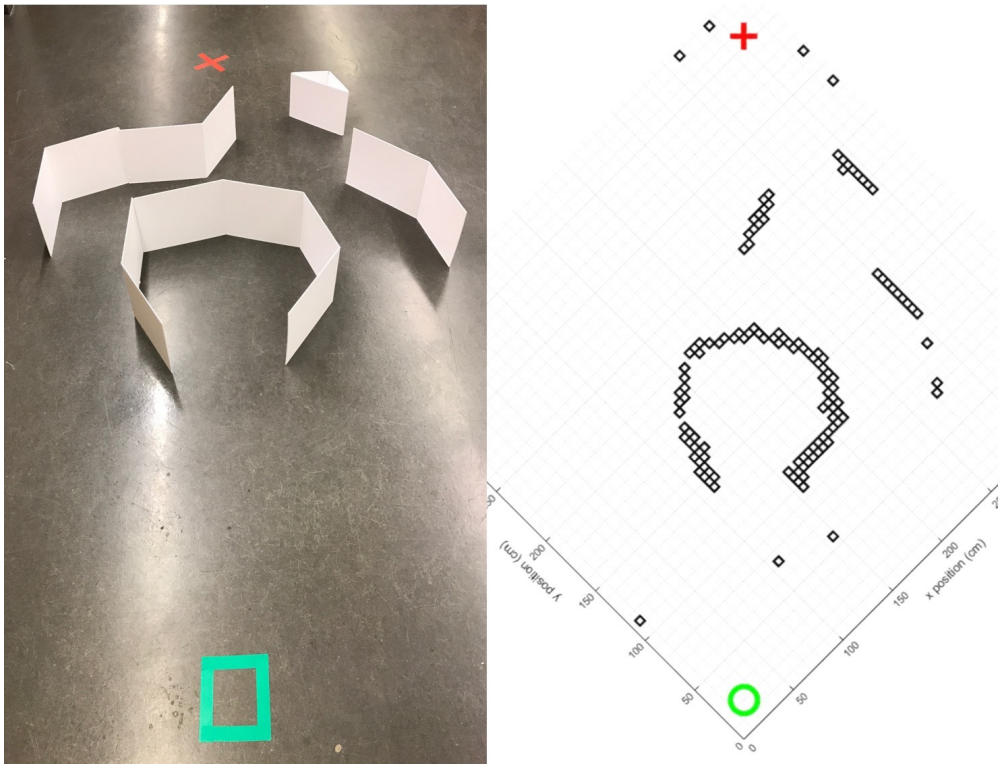
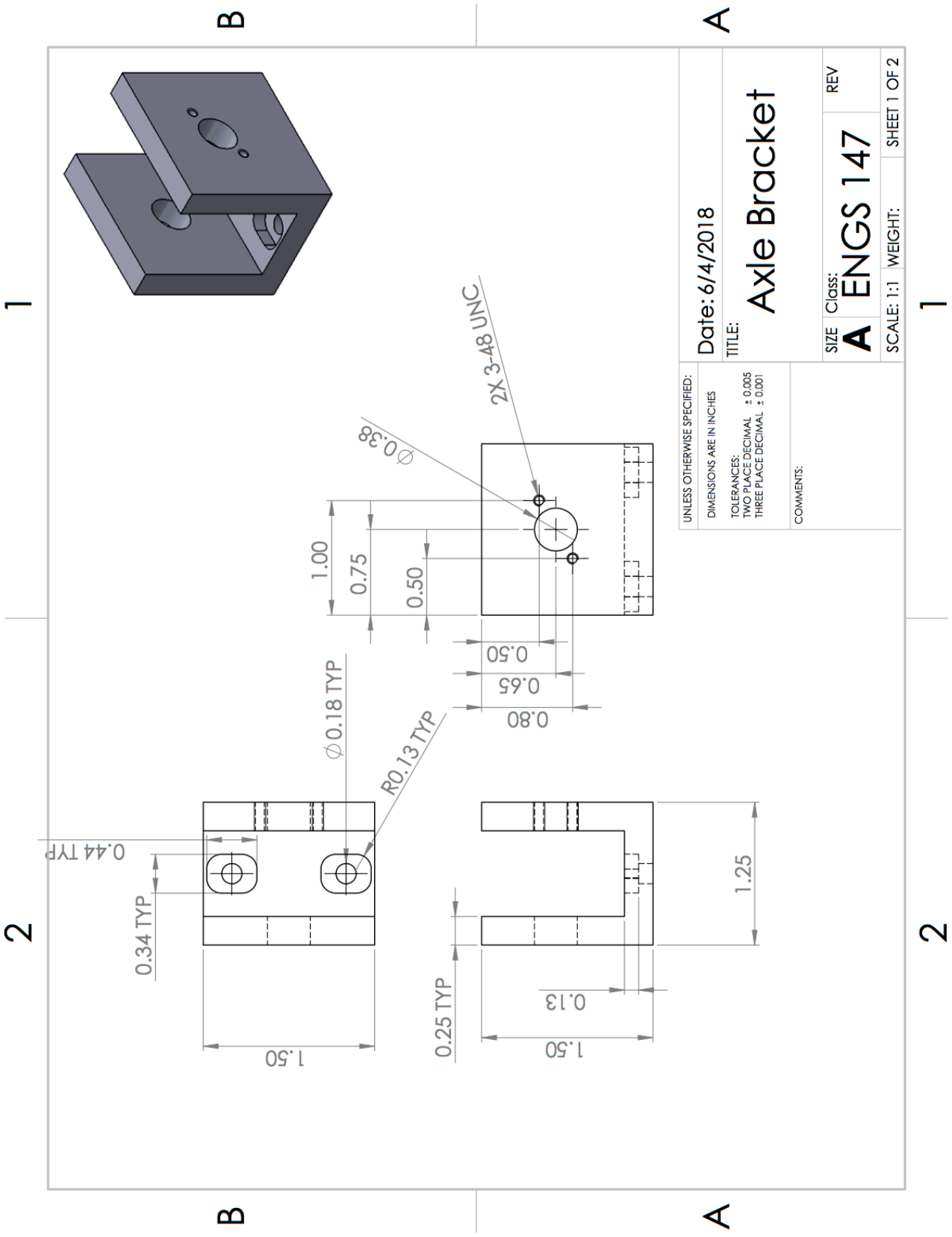


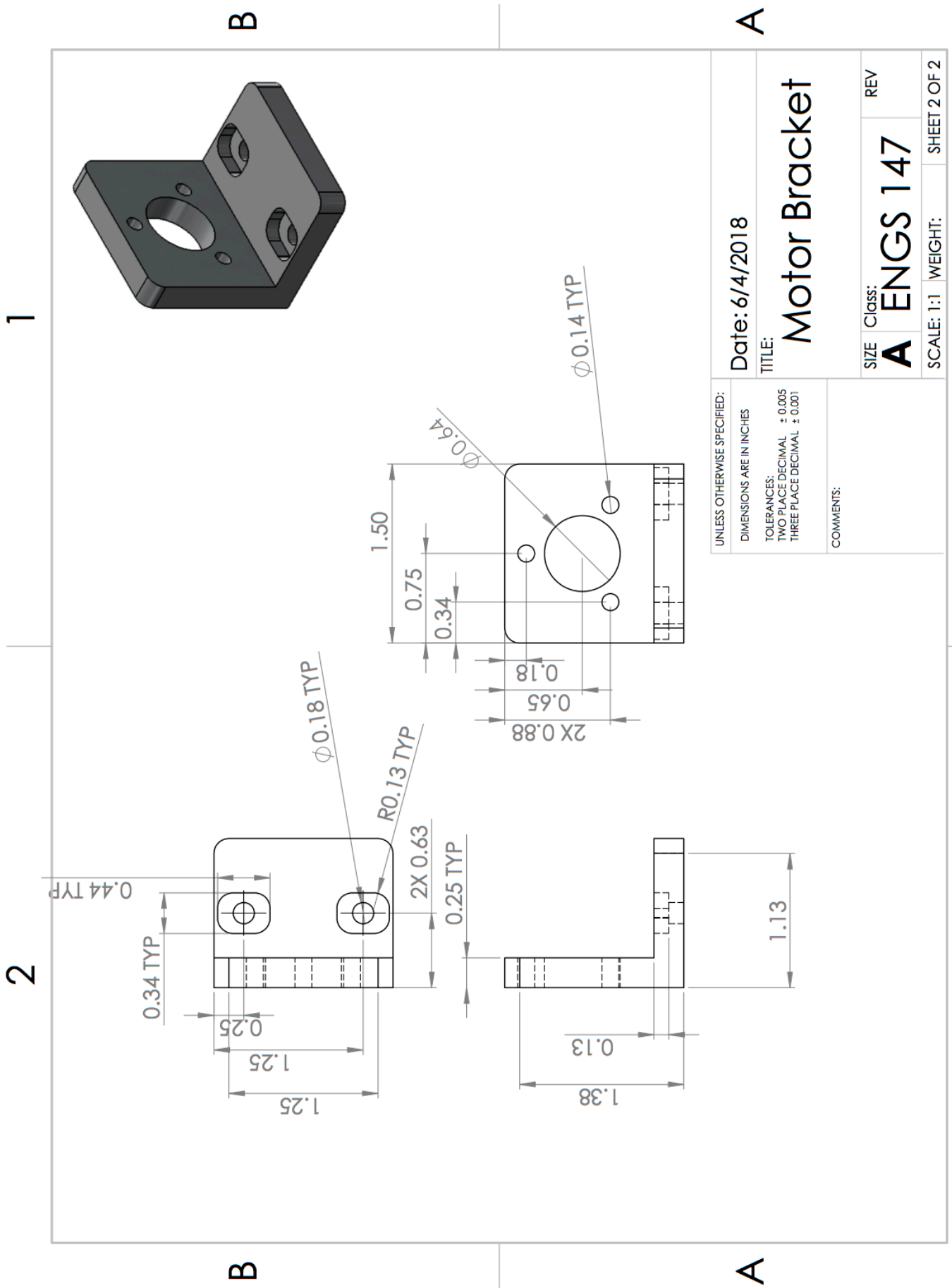
Figure 14: Mapping task 2, actual map (left) vs generated map (right)

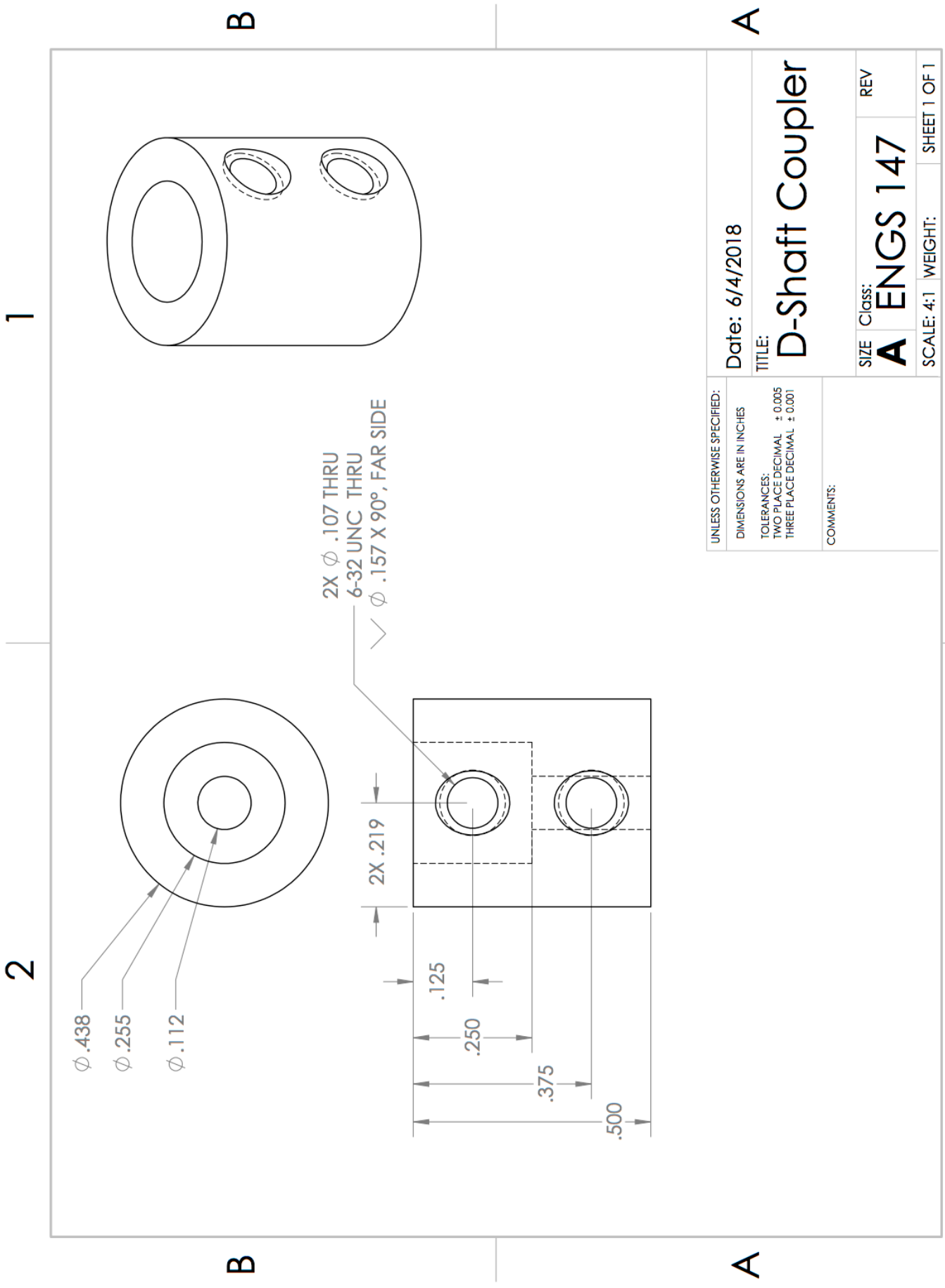
Performance Videos

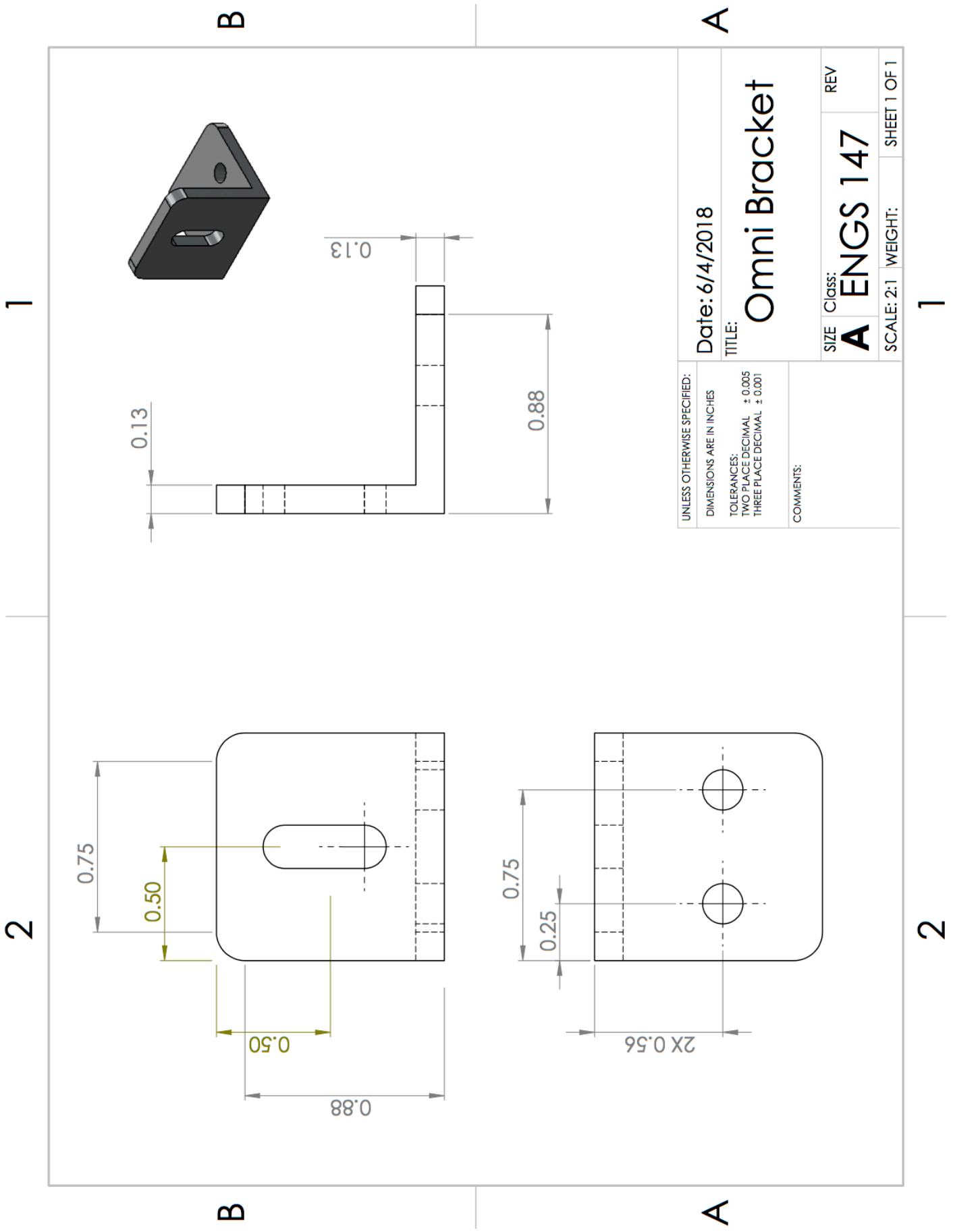
This YouTube video shows two different mapping tasks that our robot handled as well as the mapping data it generated for the course.
https://youtu.be/XPGX2dMUnhE

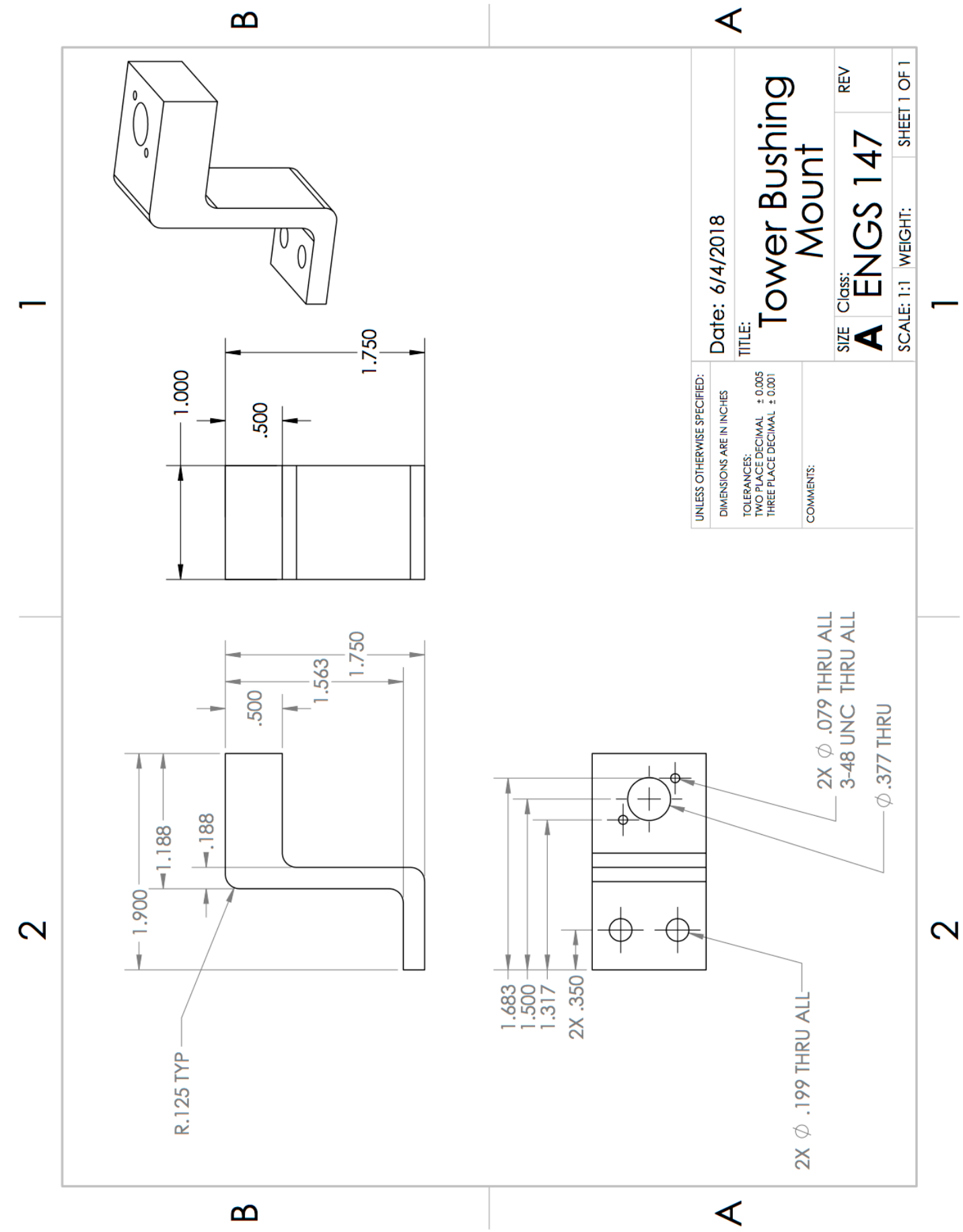
Appendix A – Part Drawings

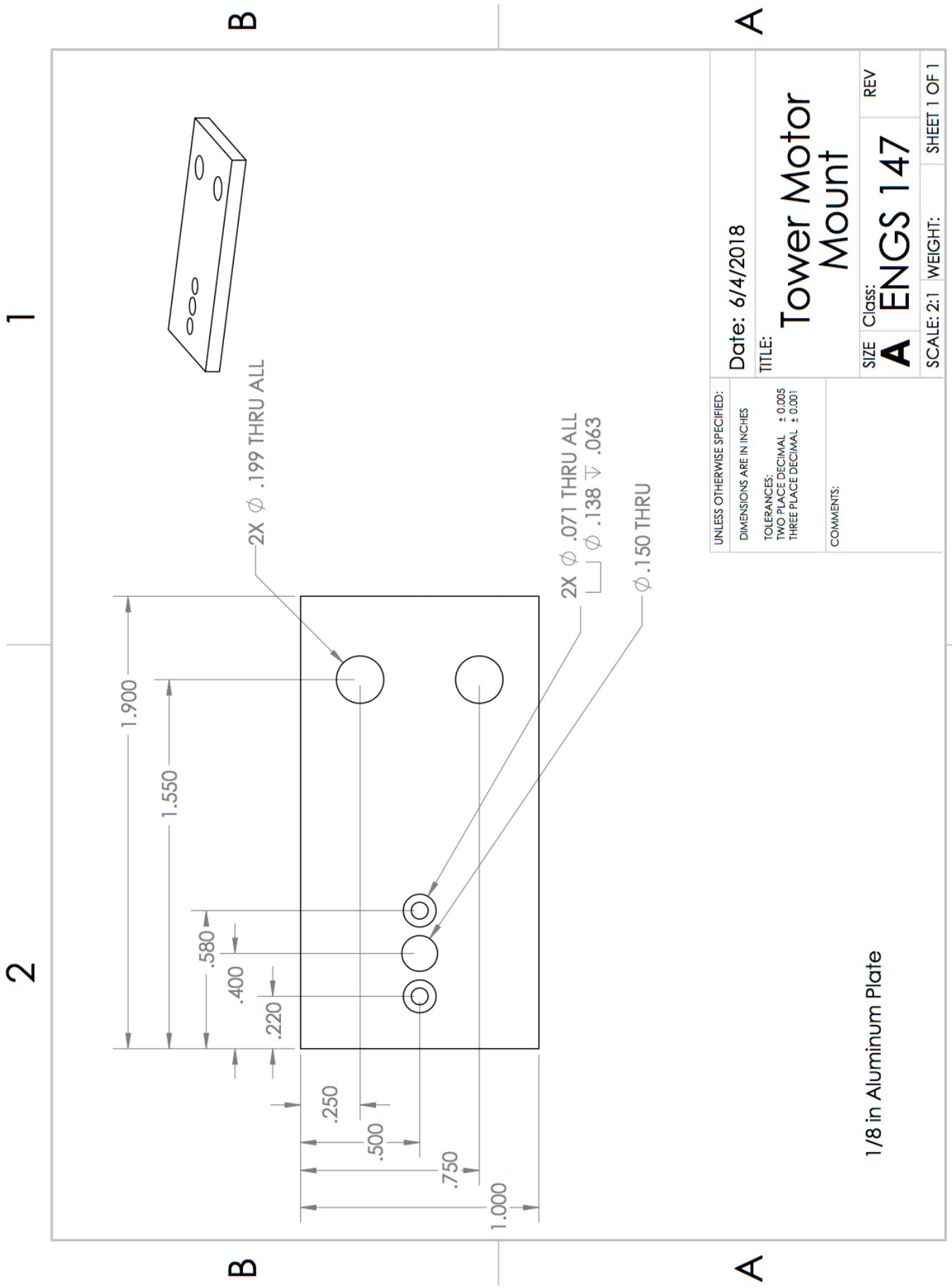

Axle Bracket
Date: 6/4/2018
Class: ENGS 147
SIZE A
REV
SCALE: 1:1
WEIGHT:
SHEET 1 OF 2
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
2X 3-48 UNC
⌀0.38
1.00
0.75
0.50
0.50
0.65
0.80
⌀0.18 TYP
R0.13 TYP
0.44 TYP
0.34 TYP
1.50
0.25 TYP
0.13
1.50
1.25

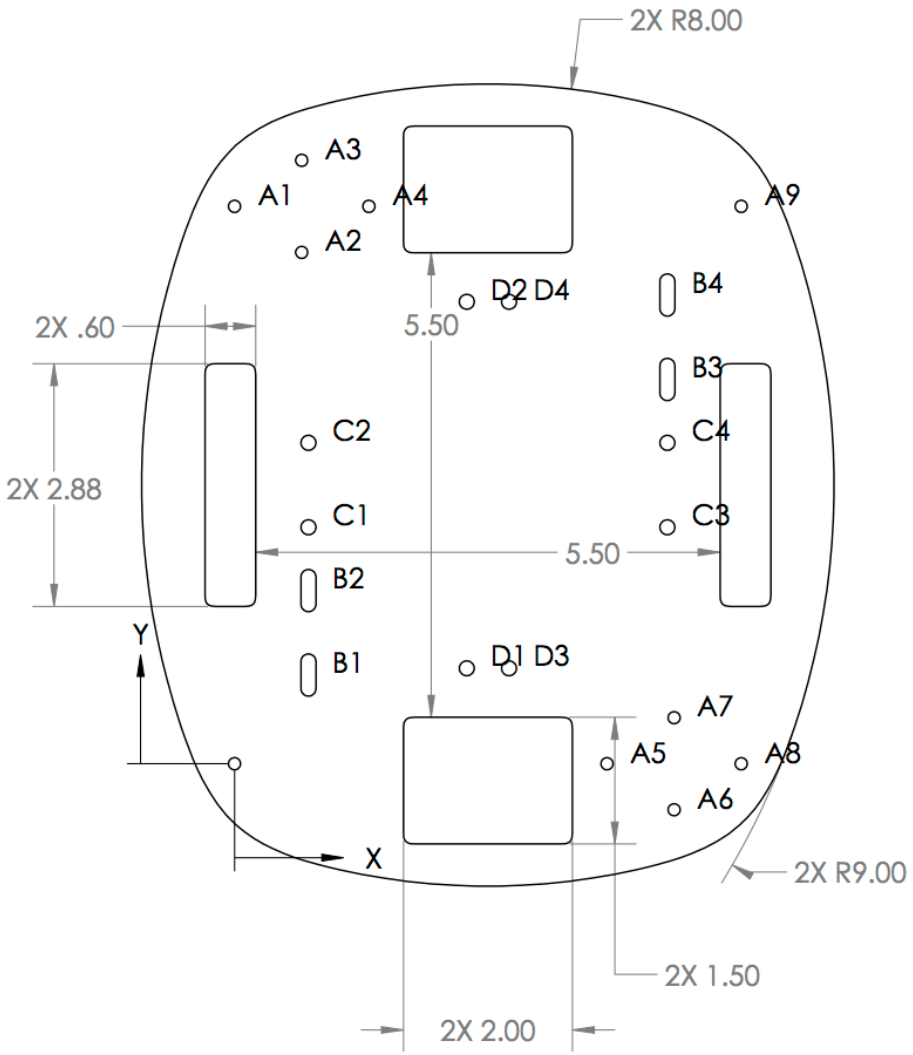

Date: 6/4/2018
TITLE:
Motor Bracket
SIZE
A
Class:
ENGS 147
REV
SCALE: 1:1
WEIGHT:
SHEET 2 OF 2
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
0.14 TYP
0.18 TYP
R0.13 TYP
2X 0.63
0.25 TYP
0.44 TYP
0.34 TYP
1.50
0.75
0.34
2X 0.88
0.65
0.18
1.13
1.38
0.13
1.25
1.25
0.25


B
A
1
2
2X ⌀ .107 THRU
6-32 UNC THRU
⌵ ⌀ .157 X 90°, FAR SIDE
2X .219
.125
.250
.375
.500
⌀.438
⌀.255
⌀.112
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
Date: 6/4/2018
TITLE:
D-Shaft Coupler
SIZE
A
Class:
ENGS 147
REV
SCALE: 4:1
WEIGHT:
SHEET 1 OF 1


B
A
1
2
0.13
0.88
0.75
0.50
0.50
0.88
0.75
0.25
2X 0.56
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
Date: 6/4/2018
TITLE:
Omni Bracket
SIZE
A
Class:
ENGS 147
REV
SCALE: 2:1
WEIGHT:
SHEET 1 OF 1


1.000
.500
1.750
1.900
1.188
.188
.500
1.563
1.750
R.125 TYP
1.683
1.500
1.317
2X .350
2X ∅ .199 THRU ALL
2X ∅ .079 THRU ALL
3-48 UNC THRU ALL
∅ .377 THRU
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
Date: 6/4/2018
TITLE:
Tower Bushing Mount
SIZE A
Class: ENGS 147
REV
SCALE: 1:1
WEIGHT:
SHEET 1 OF 1


2X ⌀ .199 THRU ALL
2X ⌀ .071 THRU ALL
⌴ ⌀ .138 ▽ .063
⌀ .150 THRU
1.900
1.550
.580
.400
.220
.250
.500
.750
1.000
1/8 in Aluminum Plate
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
Date: 6/4/2018
TITLE:
Tower Motor Mount
SIZE A
Class: ENGS 147
REV
SCALE: 2:1
WEIGHT:
SHEET 1 OF 1



Note: All large cutouts are rotationally symetric about center point

1/8th inch aluminum plate

TAG	X LOC	Y LOC	SIZE
A1	.00	6.60	∅.14 THRU
A2	.80	6.06	∅.14 THRU
A3	.80	7.15	∅.14 THRU
A4	1.59	6.60	∅.14 THRU
A5	4.41	.00	∅.14 THRU
A6	5.21	-.55	∅.14 THRU
A7	5.21	.55	∅.14 THRU
A8	6.00	.00	∅.14 THRU
A9	6.00	6.60	∅.14 THRU
B1	.88	1.05	.18 X .50 THRU ALL
B2	.88	2.05	.18 X .50 THRU ALL
B3	5.13	4.55	.18 X .50 THRU ALL
B4	5.13	5.55	.18 X .50 THRU ALL
C1	.88	2.80	∅ .18 THRU ALL
C2	.88	3.80	∅ .18 THRU ALL
C3	5.13	2.80	∅ .18 THRU ALL
C4	5.13	3.80	∅ .18 THRU ALL
D1	2.75	1.13	∅ .18 THRU
D2	2.75	5.47	∅ .18 THRU
D3	3.25	1.13	∅ .18 THRU
D4	3.25	5.47	∅ .18 THRU

UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES

TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001

COMMENTS:

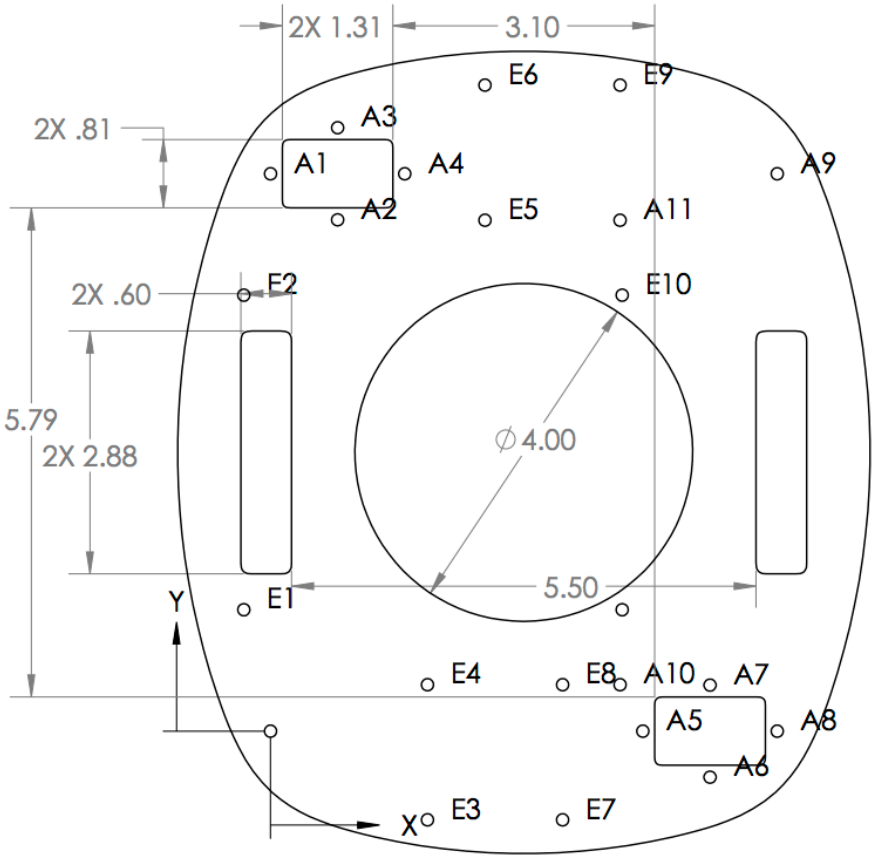
Date: 6/4/2018

TITLE: Chassis Base

SIZE A | Class: ENGS 147 | REV

SCALE: 1:2 | WEIGHT: | SHEET 1 OF 3

TAG	X LOC	Y LOC	SIZE
A1	.00	6.60	∅.14 THRU
A2	.80	6.06	∅.14 THRU
A3	.80	7.15	∅.14 THRU
A4	1.59	6.60	∅.14 THRU
A5	4.41	.00	∅.14 THRU
A6	5.21	-.55	∅.14 THRU
A7	5.21	.55	∅.14 THRU
A8	6.00	.00	∅.14 THRU
A9	6.00	6.60	∅.14 THRU
A10	4.14	.55	∅.14 THRU
A11	4.14	6.05	∅.14 THRU
E1	-.32	1.44	∅ .14 THRU ALL
E2	-.32	5.16	∅ .14 THRU ALL
E3	1.86	-1.05	∅ .14 THRU ALL
E4	1.86	.55	∅ .14 THRU ALL
E5	2.54	6.05	∅ .14 THRU ALL
E6	2.54	7.65	∅ .14 THRU ALL
E7	3.46	-1.05	∅ .14 THRU ALL
E8	3.46	.55	∅ .14 THRU ALL
E9	4.14	7.65	∅ .14 THRU ALL
E10	4.17	5.16	∅ .14 THRU ALL



UNLESS OTHERWISE SPECIFIED:

DIMENSIONS ARE IN INCHES

TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001

COMMENTS:

Date: 6/4/2018

TITLE:

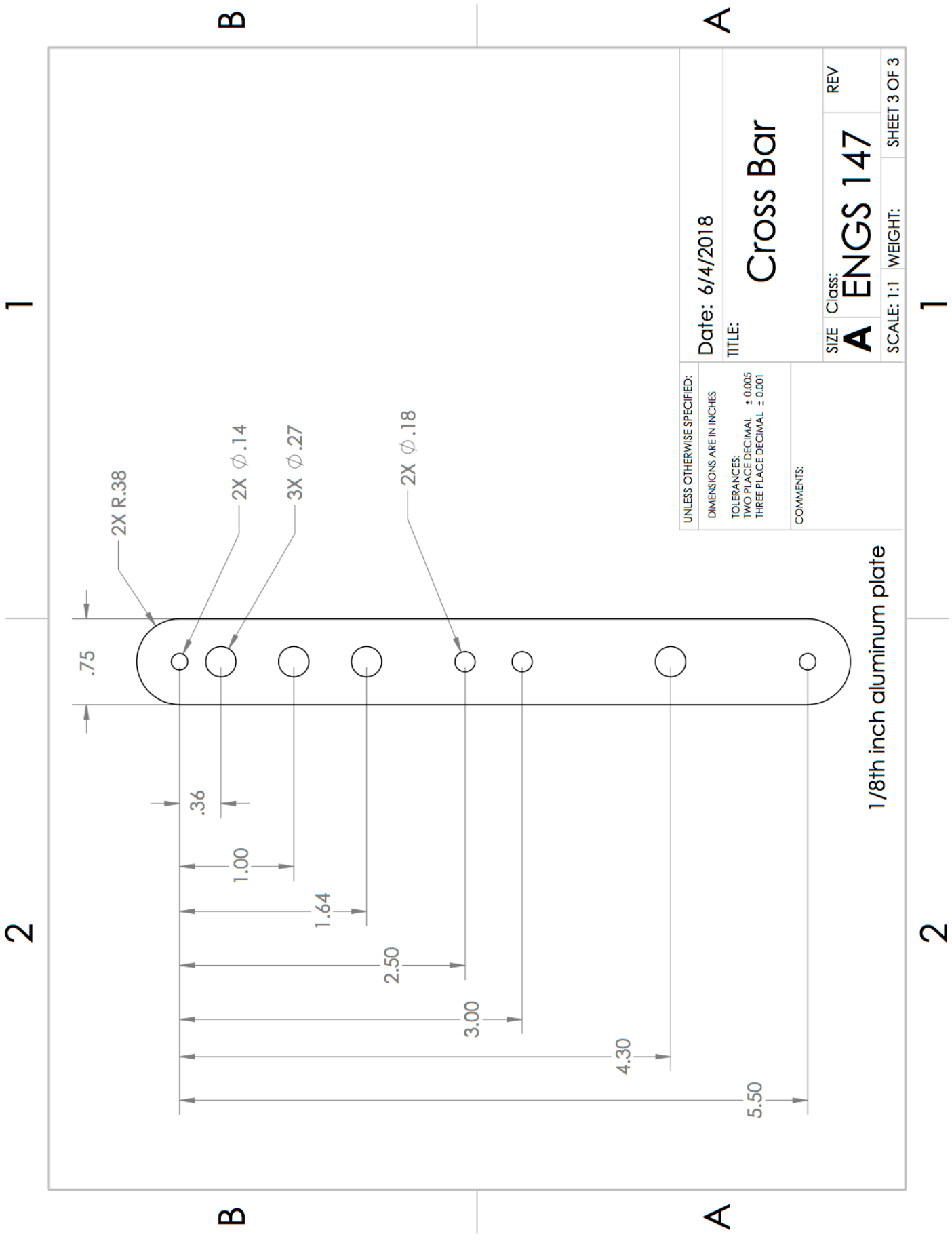
Upper Plate

SIZE	Class:	REV
A	ENGS 147	

SCALE: 1:2 | WEIGHT: | SHEET 2 OF 3

Note: All large cutouts are rotationally symetric about center point

1/8th inch acrylic plate


2X R.38
2X ϕ.14
3X ϕ.27
2X ϕ.18
.75
.36
1.00
1.64
2.50
3.00
4.30
5.50
1/8th inch aluminum plate
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN INCHES
TOLERANCES:
TWO PLACE DECIMAL ± 0.005
THREE PLACE DECIMAL ± 0.001
COMMENTS:
Date: 6/4/2018
TITLE:
Cross Bar
SIZE
A
Class:
ENGS 147
REV
SCALE: 1:1
WEIGHT:
SHEET 3 OF 3

Appendix B – Motor Analysis MATLAB Code and Outputs

```
%Mechatronics motor selection
clear all
close all
clc
format compact

%% Specifications:
%Number of motors
n_m = 2;
%Calculation factor of safety
s = 1.5;
%Robot Mass (kg)
m = 2.2487;
%Desired Max speed (m/s)
v = 0.4;
%Desired acceleration time (s)
ta = 0.5;
%Wheel Radius (m)
r = 0.073/2;
%Gearbox efficiency
ng = 0.8;
%Gearbox gear ratio
rg = 66;
%Belt drive efficiency
nb = 0.85;
%Rolling resistance constant
krr = 0.035;
%Coefficient of friction
kf = 0.8
%Gravitational constant (N/kg)
g = 9.81;

%% Power Requirements
wheel_angular_velocity = v/r

%Wheel RPM
wheel_rpm = v*60/(2*pi*r)

%Rolling resistance force and torque (newtons and newton*meters)
Frr = m*g*krr;
Trr = Frr*r/n_m

%Acceleration force and torque (newtons and newton*meters)
Fac = m*(v/ta);
Tac = Fac*r/n_m

%Continuous Motor Power required (watts)
p_rq = Trr*wheel_angular_velocity*s/(ng*nb)

%% Motor Specifications
%no load speed
w_nl = (8100/60)*2*pi;%rad/s
%stall torque
t_stall = 0.08;%Nm
%stall current
ki = 75;%amps/(N*m)
%no load current
```

```
i_nl = 0.075;%amps
%Operating voltage
V = 12;

%% Calculations & Plots
%speed torque relationship
np = 1000;
t = linspace(0, t_stall, np);
w = -(w_nl/t_stall).*t+w_nl;

figure('position', [700, 400, 1000, 600]);
subplot(2, 2, 1)
plot(t, w)
hold on
title('Speed v. Torque')
xlabel('Torque (Nm)');
ylabel('Speed (rad/s)');
grid on
grid minor

%power torque relationship
p = w.*t;
subplot(2, 2, 2)
plot(t, p)
hold on
title('Power v. Torque')
xlabel('Torque (Nm)');
ylabel('Power (Watts)');
grid on
grid minor

%current torque
%i = ((i_stall-i_nl)/t_stall).*t+i_nl;
i = ki.*t+i_nl;

subplot(2, 2, 3)
plot(t, i)
hold on
title('Current v. Torque')
xlabel('Torque (Nm)');
ylabel('Current (A)');
grid on
grid minor

%Efficiency
p_in = i.*V;
n = p./p_in*100;
subplot(2, 2, 4)
plot(t, n)
hold on
title('Efficiency v. Torque')
xlabel('Torque (Nm)');
ylabel('Efficiency (%)');
grid on
grid minor

%% Interpolation
t_rq = interp1(p(1:(length(p)/2)), t(1:(length(p)/2)), p_rq)
w_rq = interp1(t, w, t_rq)
```

```
rpm_rq = w_rq*60/(2*pi)

%% Plot Operating point
i_rq = interp1(t, i, t_rq)
n_rq = interp1(t, n, t_rq)
subplot(2, 2, 1)
plot(t_rq, w_rq, 'O')
subplot(2, 2, 2)
plot(t_rq, p_rq, 'O')
subplot(2, 2, 3)
plot(t_rq, i_rq, 'O')
subplot(2, 2, 4)
plot(t_rq, n_rq, 'O')

%% Gearing Selection
total_gear_ratio = rpm_rq/wheel_rpm
belt_ratio = total_gear_ratio/rg

%% Acceleration force and friction
max_t = max(t)*rg*1*ng*nb
max_f = max_t/r
mass_rq = n_m*max_f/kf/g

Outputs:
kf =
    0.8000
wheel_angular_velocity =
   10.9589
wheel_rpm =
  104.6498
Trr =
    0.0141
Tac =
    0.0328
p_rq =
    0.3406
t_rq =
   4.0361e-04
w_rq =
  843.9505
rpm_rq =
   8.0591e+03
i_rq =
    0.1053
n_rq =
   26.9565
total_gear_ratio =
   77.0105
belt_ratio =
    1.1668
max_t =
    3.5904
max_f =
   98.3671
mass_rq =
   25.0681
```

Appendix C – Sensor Modelling MATLAB Code

```
% Engs 147 - Final Project
% Sensor Model Comparisson with experimental and actual distance values

close all;
clear all;

M = csvread('resultsEdit.csv');
volts = M(:,1);
actDist = M(:,2);
dist1 = M(:,3);
dist2 = M(:,4);
dist3 = M(:,5);
dist4 = M(:,6);
dist5 = M(:,7);
dist6 = M(:,8);
dist7 = M(:,9);

% d = linspace(0, 150, 151);
figure(1);
plot(actDist, volts,'LineWidth',1.5);
hold on;
plot(dist1, volts);
hold on;
plot(dist2, volts);
hold on;
plot(dist3, volts);
hold on;
plot(dist4, volts);
hold on;
plot(dist5, volts);
hold on;
plot(dist6, volts);
hold on;
plot(dist7, volts);
hold on;

title('IR Sensor Model Comparison');
xlabel("distance in cm");
ylabel("IR sensor reading in volts");
grid on;
grid minor;
legend('actual', 'm1', 'm2', 'm3', 'm4', 'm5', 'm6', 'm7');

% error calculations for all models
for i = 3:9
    disp(i);
    d=actDist-M(:,i);
    SSE = norm(d,2)^2
end
figure(2);
plot(actDist, volts,'LineWidth',1.5);
hold on;
plot(dist2, volts);
hold on;
title('IR Sensor Model Comparison');
xlabel("distance in cm");
ylabel("IR sensor reading in volts");
legend('actual', 'm2');
```

```
grid on;
grid minor;
```

Appendix D – Sensor Modelling C Code

```
//Force functions to be compiled to extended memory.  Helps when the
// program gets large
#memmap xmem

//Import Library for BL2600
//#use STDIO.LIB
#use BL26XX.lib

#define DATA   0x00ff
#define RD     8
#define WR     9
#define CS1    10
#define CS2    11
#define YX     12
#define CD     13

#define BP_RESET        0X01        // reset byte pointer
#define EFLAG_RESET     0X86        // reset E bit of flag register
#define CNT             0x01       // access control register
#define DAT             0x00       // access data register
#define BP_RESETB       0X81        // reset byte pointer (x and y)
#define CLOCK_DATA      2         // FCK frequency divider
#define CLOCK_SETUP     0X98      // transfer PR0 to PSC (x and y)
#define INPUT_SETUP        0XC7     // enable inputs A and B (x and y)
                                    // set indexing function to load OL and
                                    // A and B enable gate TWP 4/2/12
#define QUAD_X1            0XA8     // quadrature multiplier to 1 (x and y)
#define QUAD_X2            0XB0     // quadrature multiplier to 2 (x and y)
#define QUAD_X4            0XB8     // quadrature multiplier to 4 (x and y)
#define CNTR_RESET         0X02     // reset counter
#define CNTR_RESETB        0X82     // reset counter (x and y)
#define TRSFRPR_CNTR       0X08     // transfer preset register to counter
#define TRSFRCNTR_OL       0X90     // transfer CNTR to OL (x and y)
#define XYIDR_SETUP        0XE1     // set index cntrl register to active low
                                    // input. index input is pulled up to +5V
                                    // in hardware to disable index functions
                                    // TWP 4/2/2012
#define HI_Z_STATE         0xFF
#define num_samples 200
#define M_PI 3.14159265358979323846

/**** FUNCTION DECLERATIONS ****/
void DispStr(int x, int y, char *s);

int EncRead(int channel, int reg);
void EncWrite(int channel, int data, int reg);
float getDistance1(float v_in);
float getDistance2(float v_in);
float getDistance3(float v_in);
float getDistance4(float v_in);
float getDistance5(float v_in);
float getDistance6(float v_in);
float getDistance7(float v_in);
```

```
void main(void){

  /**** VARIABLES ****/
  int m, s;
  int done;
  float calcDist;
  float dist1, dist2, dist3, dist4, dist5, dist6, dist7;
  float curr_sens_v;
  float v_samples[num_samples];
  float actDist_samples[num_samples];
  float dist_samples[num_samples];

  auto float actDist;
  auto char tmpbuf[128];

  /**** INITIALIZE SUBSYSTEMS ****/
  brdInit();

  digOutConfig(0xff00);
  digOut(CS1,1);
  digOut(CS2,1);
  digOut(RD,1);
  digOut(WR,1);

  // Configure channel 0 for IR sensor voltage input
  anaInConfig(0, 0);    // config for single ended unipolar

  //Enable BL2600 power supply to drive D/A channels
  anaOutPwr(1);


  /**** INITIALIZE VARIABLES ****/
  curr_sens_v = 0;
  done = 0;


  // clear sample arrays
  for(m = 0; m < num_samples; m++)
{    v_samples[m] = 0.0;
    actDist_samples[m] = 0.0;
    dist_samples[m] = 0.0;
  }


  // continue prompting user for measurements
  //printf("ENTER the actual distance, enter 0 to exit: ");

 // FILE *fp = fopen ("results.txt", "w+");
  while(!done) {

    // prompt user for actual distance
    printf("ENTER the actual distance, enter 0 to exit: ");
    actDist = atof(gets(tmpbuf));

    // exit loop is '0' is entered
    if (actDist == 0) {
```

```
      done = 1;
    }
    // store actual distance
    //printf("The value you entered is: %f\n", actDist);

    // get the sensor voltage and store
    curr_sens_v = anaInVolts(0, 1);
    curr_sens_v = curr_sens_v + anaInVolts(0, 1);
    curr_sens_v = curr_sens_v + anaInVolts(0, 1);
    curr_sens_v = curr_sens_v/(float)3;

    //printf("The current sensor reading is: %f\n", curr_sens_v);

    // dist1 = getDistance1(curr_sens_v);
    dist2 = getDistance2(curr_sens_v);
    // dist3 = getDistance3(curr_sens_v);
    // dist4 = getDistance4(curr_sens_v);
    // dist5 = getDistance5(curr_sens_v);
    // dist6 = getDistance6(curr_sens_v);
    // dist7 = getDistance7(curr_sens_v);

//    fprintf(fp, "%f, %f, %f, %f, %f, %f, %f, %f, %f\n", curr_sens_v, actDist, dist1,
dist2, dist3, dist4, dist5, dist6, dist7 );
    //printf("%f, %f, %f, %f, %f, %f, %f, %f, %f\n", curr_sens_v, actDist, dist1, dist2,
dist3, dist4, dist5, dist6, dist7 );

  }
  //fclose(fp);

}

float getDistance1(float v_in) {
  float dist;
  float p1, p2, p3, q1;   // model 1: rational fit


  // model 1: using actual dist, volts w rational fit (num: 2 deg, den: 1 deg)
  p1 = -4.039;
  p2 = 17.9;
  p3 = 12.15;
  q1 = -0.5868;

  dist = (p1*pow(v_in, (float)2.0) + p2*v_in + p3) / (v_in + q1);

  return dist;
}

float getDistance2(float v_in) {
  float dist;
  float p1, p2, p3, p4, p5;   // model 2: poly fit

  // get sensor voltage
  v_in = (float)1/v_in;

  // model 2: using actual dist,  inv volts w poly fit (deg 4, bisquare)
  p1 = 152.4;
```

```
  p2 = -381.3;
  p3 = 399.8;
  p4 = -134.4;
  p5 = 26.98;

  dist = p1*pow(v_in, (float)4.0) + p2*pow(v_in, (float)3.0) + p3*pow(v_in, (float)2.0) +
p4*v_in + p5;

  return dist;
}


float getDistance3(float v_in) {
  float dist;
  float a, b, c, d;   // model 3: exponential fit

  // model 3: using inv dist, actual volts w exponential fit (2 terms)
  a = 0.1355;
  b = 0.04664;
  c = -0.1553;
  d = -0.2055;

  dist = a*exp(b*v_in) + c*exp(d*v_in);
  dist = (float)1/dist;

  return dist;
}

float getDistance4(float v_in) {
  float dist;
  float a, b, c;   // model 4: linear fit

  // model 4: using inv dist, actual volts w linear fit
  a = -0.0008729;
  b = -0.001781;
  c = 0.1591;
  dist = a*(sin(v_in-M_PI)) + b*(pow(v_in-(float)10, (float)2.0)) + c;
  dist = (float)1/dist;

  return dist;
}

float getDistance5(float v_in) {
  float dist;
  float p1, p2, p3, p4;   // model 5: poly fit

  // model 5: using inv dist, actual volts w poly fit (deg 3)
  p1 = 0.0006422;
  p2 = -0.005483;
  p3 = 0.04199;
  p4 = -0.02149;

  dist = p1*pow(v_in, (float)3.0) + p2*pow(v_in, (float)2.0) + p3*v_in + p4 ;
  dist = (float)1/dist;

  return dist;
```

```
}


float getDistance6(float v_in) {
  float dist;
  float a, b, c;   // model 6: power fit

  // model 6: using inv dist, actual volts w powwer fit (2 terms)
  a = 0.04293;
  b = 0.7657;
  c = -0.02727;

  dist = a*pow(v_in, b)+c;
  dist = (float)1/dist;

  return dist;
}


float getDistance7(float v_in) {
  float p1, p2, p3, q1;   // model 7: rational fit
  float dist;

  // model 7: using inv dist, actual volts w rational fit (num:2, den: 1)
  p1 = 0.02327;
  p2 = 0.04265;
  p3 = -0.03095;
  q1 = 1.229;

  dist = (p1*pow(v_in, (float)2.0) + p2*v_in + p3) / (v_in + q1);
  dist = (float)1/dist;

  return dist;
}



/************ Necessary Functions ************/
int init(void)
{

   int fail;
   int i,j,k,delayvar;
   fail = 0;

   for (i = 0; i<4; i++)
   {
      EncWrite(i, XYIDR_SETUP,CNT);     // Disable Index
      EncWrite(i,EFLAG_RESET,CNT);

      EncWrite(i,BP_RESETB,CNT);

      EncWrite(i,CLOCK_DATA,DAT);
      EncWrite(i,CLOCK_SETUP,CNT);
      EncWrite(i,INPUT_SETUP,CNT);
      EncWrite(i,QUAD_X4,CNT);
```

```
      EncWrite(i,BP_RESETB,CNT);
      EncWrite(i,0x12,DAT);
      EncWrite(i,0x34,DAT);
      EncWrite(i,0x56,DAT);

      EncWrite(i,TRSFRPR_CNTR,CNT);
      EncWrite(i,TRSFRCNTR_OL,CNT);

      EncWrite(i,BP_RESETB,CNT);
      printf("written = %d, read = %d\n",0x12,EncRead(i,DAT));
      printf("written = %d, read = %d\n",0x34,EncRead(i,DAT));
      printf("written = %d, read = %d\n",0x56,EncRead(i,DAT));

    // Reset the counter now so that starting position is 0  TWP 4/2/12
     EncWrite(i,CNTR_RESET,CNT);
   }
   return fail;
}
```

Appendix E – Matlab Model and Animation of Vector Based Potential Navigation

```
%Path Finding and differential drive robot simulator
%ENGS 147 Mechatronics
%for designing and tuning robot motion controllers
%05/27/18

clear all
close all

%Set up field
figure('position', [1000, 50, 900, 900]);
axis([0, 400, 0, 400]);
axis square
grid on
grid minor
hold on

%User defined constants
XSTART = 30;
YSTART = 30;
TSTART = pi/4;
XEND = 370;
YEND = 370;
ROW = 80;
COL = 80;
BOXWIDTH = 5;
BOXHEIGHT = 5;
mapgrid = zeros(ROW, COL);


%Well
for i = 40:60
    mapgrid(i,100-i) = 1;
end
for i = 30:50
    mapgrid(i-10,i+10) = 1;
end
for i = 30:50
    mapgrid(i+10,i-10) = 1;
end

for i = 20:25
    mapgrid(i, 60-i) = 1;
end

for i = 35:40
    mapgrid(i, 60-i) = 1;
end


% Field Walls
for i = 1:ROW-10
    mapgrid(i, COL) = 1;
end

for i = 1:ROW
    mapgrid(i, 1) = 1;
end

for i = 1:COL
```

```
    mapgrid(1, i) = 1;
end

for i = 1:COL-10
    mapgrid(ROW, i) = 1;
end

%% Plot All Obstacles
for i = 1:ROW
    for j = 1:COL
        if mapgrid(i, j)>0
            x = (i-1)*BOXWIDTH;
            y = (j-1)*BOXHEIGHT;
            rectangle('position', [x, y, BOXWIDTH, BOXHEIGHT]);
        end
    end
end

%Plot End Point
plot(XEND, YEND, 'rx', 'markersize', 10, 'linewidth', 3);
%% Robot Simulator Parameters
%simulation time step
ts = 0.1;%seconds

%Robot Start Position
xbot = XSTART;
ybot = YSTART;
tbot = TSTART;

%Robot Velocity
v = 1;%in encoder ticks per time tick
vr = v;
vl = v;

%Robot size parameters
wheelbase = 15.25;%cm

%Robot Box Size (for plotting image on map)
wid = 17;
len = 20;
r = sqrt((wid/2)^2 + (len/2)^2);
tcorn = [atan2(wid, len), atan2(wid, -len), atan2(-wid, -len), atan2(-wid, len),
atan2(wid, len)];
rx = zeros(1, 5);
ry = zeros(1, 5);
%pre-plot to enable animation
b = plot(rx, ry, 'linewidth', 2, 'color', 'm');
xvect = [0 0];
yvect = [0 0];
c = plot(xvect, yvect, 'c', 'linewidth', 3);

%Navigation controller parameters
kt = 1*v;   %theta gain
ka = 5;     %attractive gain
km = 15;    %momentum gain
kr = 12000; %repulsive gain
kdiff = 0.15;   %windup torque gain
dzero = 30;     %zero effect distance from walls
searchradius = 10;  %boxes to search in all directions for vector addition
lookahead = 5;  %when to stop at end point
```

```
%% Simulator

%initialize simulation variables
tcom = 0;
tcomold = 0;
tcor = 0;
tend = 0;
while(1)
    %% Navigation Contoller

    %End location attraction
    vxend = XEND - xbot;
    vyend = YEND - ybot;
    dend = sqrt(vxend^2+vyend^2);
    xatt = vxend/dend;
    yatt = vyend/dend;

    %Unitized Momentum
    xmom = cos(tbot);
    ymom = sin(tbot);

    %Discrete bot location for outward search
    xi = floor(xbot/BOXWIDTH)+1;
    yi = floor(ybot/BOXHEIGHT)+1;

    %Search for and sum all local repulsive vectors
    xstart = xi-searchradius;
    if xstart<1
        xstart = 1;
    end
    ystart = yi-searchradius;
    if ystart<1
        ystart = 1;
    end
    xend = xi+searchradius;
    if xend>ROW
        xend = ROW;
    end
    yend = yi+searchradius;
    if yend>COL
        yend = COL;
    end

    xrep = 0;
    yrep = 0;
    for i = xstart:xend
        for j = ystart:yend
            if mapgrid(i, j)>0
                x = (i-0.5)*BOXWIDTH;
                y = (j-0.5)*BOXHEIGHT;
                vxobs = (xbot+10*cos(tbot)) - x;
                vyobs = (ybot+10*sin(tbot)) - y;
                dobs = sqrt(vxobs^2+vyobs^2);
                if dobs<dzero
                    xrep = xrep+(((1/dobs)-(1/dzero))*vxobs/dobs^2);
                    yrep = yrep+(((1/dobs)-(1/dzero))*vyobs/dobs^2);
                end
            end
        end
    end
```

```
    %Totol comand vectors and command angle
    xtot = ka*xatt+kr*xrep+km*xmom;
    ytot = ka*yatt+kr*yrep+km*ymom;
    dtot = xtot^2+ytot^2;   %measurement used in real bot to select speed
    tcom = atan2(ytot, xtot);

    %Fix atan2 for rolling theta errors
    if tcom-tcomold+tcor > pi
        tcor = tcor - 2*pi;
    elseif tcom-tcomold+tcor < -pi
        tcor = tcor + 2*pi;
    end
    tcom = tcom+tcor;

    %Display total command vector
    xvect = [xbot, xbot+xtot];
    yvect = [ybot, ybot+ytot];
    c.XData = xvect;
    c.YData = yvect;

%     %Reversals might be cool to do, but could make us get stuck
%     if tcom-tbot>pi/1.5
%         tbot = tbot+pi;
%     elseif tbot-tcom>pi/1.5
%         tbot = tbot-pi;
%     end

    %Direction of endpoint
    tend  = atan2(vyend, vxend);
    %Difference between bot angle and endpoint angle
    tdiff = (tend-tbot);

    %Windup torque = soft wall following out of local minima
    if tdiff>3*pi/2
        tdiff = 3*pi/2;
    elseif tdiff<-3*pi/2
        tdiff = -3*pi/2;
    end

    %Motor control requests
    dv = kt*(tcom-tbot) + kdiff*(tdiff);
    vr = v+dv;
    vl = v-dv;

    %When to stop the ride
    if dend<lookahead
        vr = 0;
        vl = 0;
        break
    end

    %Necessary Update for rolling theta correction
    tcomold = tcom;
    %% Robot position update using simplified kinematics equations

    %Update Variables
    tbotold = tbot;
    %Compute position and angle
    tbot = tbotold + (16.384*(vr-vl)*ts/wheelbase);
```

```
    dtbot = tbot-tbotold;
    tavg = (tbot+tbotold)/2;
    vavg = 16.384*(vl+vr)/2;
    d = vavg*ts;
%   Exact term (innacurate on the bot at varying speed - )
%     if dtbot ~= 0
%         R = d/abs(dtbot);
%         d = R*sqrt(2*(1-cos(dtbot)));
%     end
    xbot = xbot + d*cos(tavg);
    ybot = ybot + d*sin(tavg);

    %% Robot drawing

    curtcorn = tcorn+tbot;
    for k = 1:5
        rx(k) = xbot+r*cos(curtcorn(k));
        ry(k) = ybot+r*sin(curtcorn(k));
    end
    b.XData = rx;
    b.YData = ry;

    %% Pause

    pause(ts);
end
```

Appendix F – Final C Code

```
/*
 * Robotver5.c - Program that runs the Engs 147 robot
 *
 * Dan Magoon, Jennifer Jain, Jonah Sternthal, Joe Leonor
 *
 * Spring 2018
 * June 4th 2018
 */

#memmap xmem    //Force compilation to extended memory when the program gets large
#use BL26XX.lib   //Import Library for BL2600

#define DATA   0x00ff
#define RD     8
#define WR     9
#define CS1    10
#define CS2    11
#define YX     12
#define CD     13
#define BP_RESET        0X01        // reset byte pointer
#define EFLAG_RESET     0X86        // reset E bit of flag register
#define CNT             0x01       // access control register
#define DAT             0x00       // access data register
#define BP_RESETB       0X81        // reset byte pointer (x and y)
#define CLOCK_DATA      2         // FCK frequency divider
#define CLOCK_SETUP     0X98     // transfer PR0 to PSC (x and y)
#define INPUT_SETUP        0XC7     // enable inputs A and B (x and y)
                                   // set indexing function to load OL and
                                   // A and B enable gate TWP 4/2/12
#define QUAD_X1            0XA8     // quadrature multiplier to 1 (x and y)
#define QUAD_X2            0XB0     // quadrature multiplier to 2 (x and y)
#define QUAD_X4            0XB8     // quadrature multiplier to 4 (x and y)
#define CNTR_RESET         0X02     // reset counter
#define CNTR_RESETB        0X82     // reset counter (x and y)
#define TRSFRPR_CNTR       0X08     // transfer preset register to counter
#define TRSFRCNTR_OL       0X90     // transfer CNTR to OL (x and y)
#define XYIDR_SETUP        0XE1     // set index cntrl register to active low
                                   // input. index input is pulled up to +5V
                                   // in hardware to disable index functions
                                   // TWP 4/2/2012
#define HI_Z_STATE  0xFF

/*************** CONSTANT DECLERATIONS ***************/
#define LEFT        1     // left wheel motor encoder ID
#define RIGHT       2     // right wheel motor encoder ID
#define TOP         3     // IR sensor motor encoder ID

// Robot Specifications
#define WHEELBASE 15.4     // cm wheelbase of bot
#define STEP 0.0160     //cm per encoder tick of wheel

// Mathematical Constants
#define PIE 3.14159265
#define HALFPIE 1.57079633
```

```
// Mapping Constants
#define BOXES 77     // initialize grid size to BOXES x BOXES
#define MAPSIZE 5929     // = BOXES*BOXES
#define BOXLENGTH 5     // cm length per box on grid

#define MAPFILTER 100     // points-per-box required for an obstacle to be live on map
#define REQWEIGHT 4     // points-per-box threashold for an obstacle and neighbors
#define DEWEIGHT 0     // amount below mapfilter to push down values of neighborless
boxes
#define NEIGHBORS 2     // amount of neighbors required to be obstacle
#define LOOKAHEAD 10     // cm distance to lead bot in calculations

// Driving Speeds
#define VHIGH 1     // max vavg encoder ticks per time tick
#define VLOW 0.4     // min vavg encoder ticks per time tick
#define VNE 1.3     // velocity Never Exceed foreward or reverse on each wheel
#define RAMP 0.1     // ramp factor per time cycle

// Costate Timing Constants
#define CTRLTICKS 100     // length of sample time (1 tick = 1/1024 secs)
#define IRTICKS 15

// IR Sensor Parameters
#define IRW 2.2     // volts to IR sensor motor (proportional to sensor rotation speed)
#define MAXSIGHT 100     // max sight distance in cm at which sensor looks ahead
#define SENSORSPIN 300

// Start and End Positions
#define XSTART 20     //cm
#define YSTART 20     //cm
#define TSTART 0.7853981     //rad from x axis
#define XEND 355     //cm
#define YEND 355     //cm


/*************** GLOBAL TYPES ***************/
// stores the information regarding the current state of the bot
typedef struct botState {
  // bot position and velocity
  float lvel;     // left wheel velocity (encoder ticks/time ticks)
  float rvel;     // right wheel velocity (encoder ticks/time ticks)
  float xpos;     // robot x position (cm)
  float ypos;     // robot y position (cm)
  float theta;     // robot heading angle (rad)
  long lpos;     // left wheel encoder position (encoder ticks)
  long rpos;     // right wheel encoder position (encoder ticks)
  float lerrtot;     // left wheel velocity error sum
  float rerrtot;     // right wheel velocity error sum
  float lcmd;     // velocity command to left wheel (encoder ticks/time ticks)
  float rcmd;     // velocity command to right wheel (encoder ticks/time ticks)
  float lreq;     // velocity request to left wheel (encoder ticks/time ticks)
  float rreq;     // velocity request to right wheel (encoder ticks/time ticks)

  // sensor update
  long IRstartPos;     // sensor start position (ticks)
  long IRcurrPos;     // sensor current position (ticks)
```

```
  float IRcurrDist;     // obstacle distance calculated by sensor (cm)
} botState_t;

// stores an x,y float value
typedef struct vertex {
  float x;
  float y;
} vertex_t;

// stores an x,y int value
typedef struct point {
  int x;
  int y;
} point_t;


/*************** GLOBAL VARIABLES ***************/
 botState_t curr_state, *curr_statep;
 botState_t IR_state, *IR_statep;
 point_t checkpts[8];
 vertex_t ir_point, *IRvertex;
 vertex_t obstacle, *obstaclep;
 float tcomold;
 float tcomcor;
 float tendcor;
 float vavg;
 int irdir;


/*************** FUNCTION DECLERATIONS ***************/
// BL2600 necessary functions
int EncRead(int channel, int reg);
void EncWrite(int channel, int data, int reg);
int init(void);

// Robot Position and Velocity State Functions
void stateInit(botState_t *botState);
void checkInit(point_t *arr);
void stateUpdate(botState_t *botState);
void controlLoop(botState_t *botState);
void reqVel(botState_t *botState);
long getPosition(int j);

// Robot Sensor State Functions
void IRxy(botState_t *botState, vertex_t *IRvertex);
void avoidObs(botState_t *botState, vertex_t *obstaclep);
void obsVect(botState_t *botState, vertex_t *obstaclep, long arr);
void sensorInit(botState_t *botState);
void sensorReverse(botState_t *botState);
float getDistance(void);

// Mapping Functions and Conversions
long initIntArray(long sz, int initVal);
int getIndex(int x, int y);
void insertElement(long arr, int idx, int value);
int getElement(long arr, int idx);
```

```
void incElement(long arr, int idx);
float idxToAvgPosX(int node);
float idxToAvgPosY(int node);
int idxToMapPosX(int node);
int idxToMapPosY(int node);
void IRMap(long map, vertex_t *IRvertex);
void printMap(long arr);
void parseMap(botState_t *botState, long arr, point_t *checks);


/*
 *  Main - Initializes subsystems, sets up map array, switches and runs
 *       the controller
 */
void main(){
  int breakout, i, j;
  long T0, T1, T2;
  long temptime;
  long map;

  /******* initialize subsystems *******/
  brdInit();
  anaOutConfig(1, DAC_SYNC);
  anaInConfig(0, 0);    // config for single ended unipolar
  anaInConfig(3, 0);    // config for single ended unipolar

  //Enable BL2600 power supply to drive D/A channels
  anaOutPwr(1);

  // send 0 volts to DAC to prevent indeterminate bot state
  anaOutVolts(TOP, 0);
  anaOutStrobe();
  anaOutVolts(LEFT, 0);
  anaOutStrobe();
  anaOutVolts(RIGHT, 0);

  digOutConfig(0xff00);
  digOut(CS1, 1);
  digOut(CS2, 1);
  digOut(RD, 1);
  digOut(WR, 1);
  init();


  while(1){

    printf("INITIALIZING\n");

    /******* initialize global variables *******/
    irdir = 1;
    tcomold = 0;
    tcomcor = 0;
    tendcor = 0;
    obstacle.x = 0;
    obstacle.y = 0;
```

```
breakout = 0;   // non global variable

/******* initialize map *******/
//Set up map array with repulsive border
map = initIntArray(MAPSIZE, MAPFILTER);
for(j = 1; j < BOXES-1; j++){
  for(i = 1; i < BOXES-1; i++){
    insertElement(map, BOXES * j + i, 0);
  }
}
//Make end box non-repulsive
for(j = 69; j < BOXES; j++){
  for(i = 69; i < BOXES; i++){
    insertElement(map, BOXES * j + i, 0);
  }
}


/******* wait switch *******/
if(anaInVolts(6, 0) < 3){
  while(anaInVolts(6, 0) < 3){
    anaOutVolts(LEFT, 0);
    anaOutVolts(RIGHT,0);
    anaOutVolts(TOP, 0);
    anaOutStrobe();
  }
  printf("STARTING\n");
}

curr_statep = &curr_state;
IR_statep = &IR_state;
IRvertex = &ir_point;
obstaclep = &obstacle;
stateInit(curr_statep);
sensorInit(curr_statep);     // orients sensor
checkInit(checkpts);
stateUpdate(curr_statep);


while(1){

  /******* costate for robot control and obstacle avoidance *******/
  costate{
    T0 = TICK_TIMER;
    stateUpdate(curr_statep);
    obsVect(curr_statep, obstaclep, map);
    avoidObs(curr_statep, obstaclep);
    reqVel(curr_statep);
    controlLoop(curr_statep);
    IR_state = curr_state;
    waitfor(TICK_TIMER-T0 >= CTRLTICKS);
  }

  /******* costate for IR sensor readings and map update *******/
  costate{
    if(TICK_TIMER-T0 >= CTRLTICKS){
```

```
        yield;
      }
      T1 = TICK_TIMER;
      stateUpdate(IR_statep);
      IRxy(IR_statep, IRvertex);
      IRMap(map, IRvertex);
      sensorReverse(IR_statep);
      waitfor(TICK_TIMER-T1 >= IRTICKS);
    }

    /******* costate for parsing map *******/
    costate{
      if(TICK_TIMER-T0 >= CTRLTICKS){
        yield;
      }
      T2 = TICK_TIMER;
      parseMap(curr_statep, map, checkpts);
      waitfor(TICK_TIMER-T2 >= CTRLTICKS);
    }

    /******* costate for switch that pauses and prints a map *******/
    costate{
      if(TICK_TIMER-T0 >= CTRLTICKS){
        yield;
      }
      if (anaInVolts(6, 0)<3){
        printf("PAUSED\n");
        while (anaInVolts(6, 0)<3){
          anaOutVolts(LEFT, 0);
          anaOutVolts(RIGHT,0);
          anaOutVolts(TOP,0);
          anaOutStrobe();
          if(anaInVolts(7, 0) > 3){
            printf("BREAKING\n");
            //PRINT OUT THE MAP FOR PLOTTING IN MATLAB
            printMap(map);
            //END OF MAP PRINTING
            breakout = 1;
            break;
          }
        }
        if(breakout == 0){
          printf("RESTARTING\n");
        }
      }
      if(breakout != 0){
        break;
      }
    }
  }
}


/*
 *  reqVel - Request a velocity to the left and right wheels, sends a command
```

```
 *      to ramp up or ramp down
 */
void reqVel(botState_t *botState){

  // limit velocity cmd to left wheel
  if(botState->lreq > VNE){
    botState->lreq = VNE;
  } else if(botState->lreq < -VNE){
    botState->lreq = -VNE;
  }

  // limit velocity cmd to right wheel
  if(botState->rreq > VNE){
    botState->rreq = VNE;
  } else if(botState->rreq < -VNE){
    botState->rreq = -VNE;
  }

  // changes the command velocity to the left wheel
  if(botState->lreq > botState->lcmd + RAMP){
    botState->lcmd += RAMP;
  } else if(botState->lreq < botState->lcmd - RAMP){
    botState->lcmd -= RAMP;
  } else {
    botState->lcmd = botState->lreq;
  }

  // changes the command velocity to the right wheel
  if (botState->rreq > botState->rcmd + RAMP){
    botState->rcmd += RAMP;
  } else if (botState->rreq < botState->rcmd - RAMP){
    botState->rcmd -= RAMP;
  } else {
    botState->rcmd = botState->rreq;
  }

}


/*
 *  controlLoop - Sets the velocity in both the right and left wheels based on
 *      control effort
 */
void controlLoop(botState_t *botState){
  float lP, lI, rP, rI, lerr, rerr;
  float luk, ruk;

  // controller gains
  lP = 3;
  lI = 1;
  rP = 3;
  rI = 1;

  // generate left wheel control effort
  lerr = botState->lcmd - (float)botState->lvel;
  botState->lerrtot += lerr;
```

```
  luk = lP * botState->lcmd + lI * botState->lerrtot;

  // generate right wheel control effort
  rerr = botState->rcmd - (float)botState->rvel;
  botState->rerrtot += rerr;
  ruk = rP * botState->rcmd + rI * botState->rerrtot;

  // update voltage sent to wheels
  anaOutVolts(LEFT, luk);
  anaOutStrobe();
  anaOutVolts(RIGHT, ruk);
  anaOutStrobe();
}

/*
 *  getPosition - gets the position of encoder j in ticks
 */
long getPosition(int j) {
  int asb, bsb, csb;
  long position;

  EncWrite(j, TRSFRCNTR_OL, CNT);
  EncWrite(j,BP_RESETB,CNT);
  asb = EncRead(j,DAT);
  bsb = EncRead(j,DAT);
  csb = EncRead(j,DAT);

  position  = (long)asb;          // least significant byte
  position += (long)(bsb << 8);
  position += (long)(csb <<16);
  return position;
}

/*
 *  stateUpdate - Updates the current state of the robot and sensor, updates
 *      the botState struct
 */
void stateUpdate(botState_t *botState){
  long curr_l, curr_r, n_l, n_r;
  float d_theta, dx, dy, avg_theta, n_avg;
  float step;
  float base;
  float r;

  //sensor variables
  long sens_pos, start_pos;
  float sens_dist;

  /******* bot position update *******/
  curr_l = getPosition(LEFT);   // get position of left wheel
  curr_r = -1 * getPosition(RIGHT);  // get position of right wheel
  step = STEP;
  base = WHEELBASE;
  n_l = (curr_l - botState->lpos);  // difference in left curr and prev pos
  n_r = (curr_r - botState->rpos);  // difference in right curr and prev pos
```

```
  // rollover check
  if(labs(n_l)>5000){
    n_l = (long)(botState->lvel*CTRLTICKS);
  }
  if(labs(n_r)>5000){
    n_r = (long)(botState->rvel*CTRLTICKS);
  }

  n_avg = (n_l + n_r)/2.0;
  d_theta = (float)(n_r - n_l) * step/base;
  avg_theta = botState->theta + (d_theta/2.0);

  // find change in x and y positions
  dx = step * n_avg * cos(avg_theta);
  dy = step * n_avg * sin(avg_theta);

  /******* sensor readings update *******/
  sens_pos = getPosition(TOP) - botState->IRstartPos;
  sens_dist = getDistance();

  /******* bookkeeping into botState struct *******/
  botState->theta += d_theta;
  botState->xpos += dx;
  botState->ypos += dy;

  botState->lpos = curr_l;
  botState->rpos = curr_r;

  botState->lvel = (float)n_l/CTRLTICKS;
  botState->rvel = (float)n_r/CTRLTICKS;

  botState->IRcurrPos = sens_pos;
  botState->IRcurrDist = sens_dist;
}

/*
 *  sensorReverse - Changes the sign of the voltage sent to the sensor based
 *      on the desired sensor spin angle
 */
void sensorReverse(botState_t *botState){
  long sens_pos, start_pos;
  sens_pos = botState->IRcurrPos;
  start_pos = botState->IRstartPos;

  // change the sign of the voltage sent based on current sensor position
  if(sens_pos >= SENSORSPIN){
    irdir = -1;
  }else if(sens_pos <= -SENSORSPIN){
    irdir = 1;
  }

  anaOutVolts(TOP, IRW*irdir);
  anaOutStrobe();
}

/*
```

```
 *  stateInit - Initializes the state of the robot upon program start
 */
void stateInit(botState_t *botState){
  botState->lvel = 0;
  botState->rvel = 0;
  botState->xpos = XSTART;
  botState->ypos = YSTART;
  botState->theta = TSTART;
  botState->lerrtot = 0;
  botState->rerrtot = 0;
  botState->lcmd = 0;
  botState->rcmd = 0;
  botState->lreq = 0;
  botState->rreq = 0;
  botState->lpos = getPosition(LEFT);
  botState->rpos = -1 * getPosition(RIGHT);

  // for sensor
  botState->IRcurrPos = getPosition(TOP);
  botState->IRcurrDist = getDistance();
}

/*
 *  sensorInit - Initializes the state of the robot sensor upon program start
 */
void sensorInit(botState_t *botState){
  long start_pos;

  start_pos = getPosition(TOP);
  botState->IRstartPos = start_pos;
}

/*
 *  avoidObs - Generates a potential navigation field by summing up attractive
 *      and repulsive forces
 */
void avoidObs(botState_t *botState, vertex_t *obstaclep){
  float xbot, ybot, xrep, yrep, kt, lookahead, tline, dv, vr, vl, l;
  float tbot, tcom, vxcom, vycom, ka, km, kr;
  float vxend, vyend, dend, xatt, yatt, xmom, ymom, tend, tdiff, kdiff, drep, dcom;

  // avoidance gains
  kt = 1*vavg;    // theta gain
  ka = 5;    // attractive gain
  km = 15;    // momentum gain
  kr = 12000;   // repulsive gain (10000)
  kdiff = 0.14;    // windup gain
  lookahead = 5;    //cm distance at which to stop for endpoint

  // get current bot position
  xbot = botState->xpos;
  ybot = botState->ypos;
  tbot = botState->theta;

  // get obstacle positions to generate repulsive vectors
  xrep = obstaclep->x;
```

```
yrep = obstaclep->y;

// generate an attractive vector for the endpoint
vxend = XEND - xbot;
vyend = YEND - ybot;
dend = sqrt(pow(vxend, 2.0) + pow(vyend, 2.0));
xatt = vxend/dend;
yatt = vyend/dend;

// theta windup torque
tend = atan2(vyend, vxend)+tendcor;
tdiff = tend - tbot;

if(tdiff > 3 * HALFPIE) {
  tdiff = 3 * HALFPIE;
} else if (tdiff < -3 * HALFPIE) {
  tdiff = -3 * HALFPIE;
}

// avoid wall following by turning to goal after following obstacle to wall
if(xbot < 30 || ybot > 370) {
    if(tdiff < -HALFPIE){
        tendcor = tendcor + 2 * PIE;
    }
} else if(xbot > 370 || ybot < 30) {
    if(tdiff > HALFPIE){
        tendcor = tendcor - 2 * PIE;
    }
}

// generate an attractive vector based on current robot heading (momentum)
xmom = cos(tbot);
ymom = sin(tbot);

// command vector
vxcom = ka * xatt + km * xmom + kr * xrep;
vycom = ka * yatt + km * ymom + kr * yrep;
tcom = atan2(vycom, vxcom);
dcom = pow(vxcom, 2.0) + pow(vycom, 2.0);

// speed control based on total vector magnitude
if(dcom > 300){
  vavg = VHIGH;
} else {
  vavg = VLOW;
}

// correct command theta to allow for multiple rotations
if(tcom - tcomold + tcomcor > PIE){
  tcomcor = tcomcor - 2 * PIE;
} else if (tcom - tcomold + tcomcor < -PIE){
  tcomcor = tcomcor + 2 * PIE;
}
tcom = tcom + tcomcor;
tcomold = tcom;
```

```
  // control effort (dv = right velocity increase = left velocity decrease)
  dv = kt * (tcom - tbot) + kdiff * tdiff;
  botState->rreq = vavg + dv; //right velocity request
  botState->lreq = vavg - dv; //left velocity request

  // compute distance to end point
  l = sqrt(pow((XEND - xbot), 2.0) + pow((YEND - ybot), 2.0));

  // ramp down velocity when close to end point
  if (l<lookahead){
        botState->rreq = 0;   //right velocity request
        botState->lreq = 0;   //left velocity request
      }
}

/*
 *  obsVect - Outward search from robot to check whether there are obstacles
 *      around it. It generates a repulsive vetor sum based on the obstacles and
 *      stores the x and y repulsive vector values
 */
void obsVect(botState_t *botState, vertex_t *obstaclep, long arr){
  int xbotcoord, ybotcoord, searchradius, xstart, ystart, xend, yend, i, j;
  int index;
  float xbot, ybot, tbot, dobs, dobs2, dzero, vxobs, vyobs, xrep, yrep;

  searchradius = 7;   //Squares to search in all directions
  dzero = 30;   // distance threashold for no effect

  // get the current state of bot
  tbot = botState->theta;
  xbot = botState->xpos + LOOKAHEAD * cos(tbot);
  ybot = botState->ypos + LOOKAHEAD * sin(tbot);

  xbotcoord = (int)floor(xbot/BOXLENGTH);
  ybotcoord = (int)floor(ybot/BOXLENGTH);

  // map a search radius around the bot of 7 units
  xstart = xbotcoord - searchradius;
  if(xstart < 0){
    xstart = 0;
  }
  ystart = ybotcoord - searchradius;
  if(ystart < 0){
    ystart = 0;
  }
  xend = xbotcoord + searchradius;
  if(xend >= BOXES){
    xend = BOXES - 1;
  }
  yend = ybotcoord + searchradius;
  if(yend >= BOXES){
    yend = BOXES - 1;
  }

  xrep = 0;
  yrep = 0;
```

```
  // generate repulsive vectors by doing an outward search from the bot
  for(j = ystart; j <= yend; j++){
    for(i = xstart; i <= xend; i++){
      index = (BOXES * j + i);
      if(getElement(arr, index)>=MAPFILTER){
        vxobs = xbot - idxToAvgPosX(index);
        vyobs = ybot - idxToAvgPosY(index);
        dobs=sqrt(pow(vxobs, 2.0) + pow(vyobs, 2.0));
        if(dobs <= dzero){
        dobs2 = pow(dobs, 2.0);
        xrep = xrep + (((1/dobs) - (1/dzero)) * (vxobs/dobs2));
        yrep = yrep + (((1/dobs) - (1/dzero)) * (vyobs/dobs2));
        }
      }
    }
  }

  // store repulsive vectors
  obstaclep->x = xrep;
  obstaclep->y = yrep;
}

/*
 *  parseMap - Filters the map around the bot so false IR readings do not
 *      inhibit the robot's path
 */
void parseMap(botState_t *botState, long arr, point_t *checks){
  int xbotcoord, ybotcoord, searchradius, xstart, ystart, xend, yend, i, j, k;
  int index, neighbors;
  float xbot, ybot, tbot;
  searchradius = 10; //Squares to search in all directions

  // get current state of the bot
  tbot = botState->theta;
  xbot = botState->xpos + LOOKAHEAD * cos(tbot);
  ybot = botState->ypos + LOOKAHEAD * sin(tbot);

  xbotcoord = (int)floor(xbot/BOXLENGTH);
  ybotcoord = (int)floor(ybot/BOXLENGTH);

  // let the search radius avoid any squares with walls
  xstart = xbotcoord - searchradius;
  if(xstart < 1){
    xstart = 1;
  }
  ystart = ybotcoord - searchradius;
  if(ystart < 1){
    ystart = 1;
  }
  xend = xbotcoord + searchradius;
  if(xend > BOXES - 2){
    xend = BOXES - 2;
  }
  yend = ybotcoord + searchradius;
  if(yend > BOXES - 2){
```

```
    yend = BOXES - 2;
  }

  // iterate through the search area
  for(j = ystart; j <= yend; j++){
    for(i = xstart; i <= xend; i++){

      // get an index value for map coord x and map coord y
      index = (BOXES * j + i);

      // only if the element has at least REQWEIGHT IR points in it and is less
      // than MAPFILTER thresh then we want to search the neighbors to see if
      // they have the REQWEIGHT number of points if there are at least
      // NEIGHBORS # of live neighbors then set the value at the respective node
      // to MAPFILTER

      if((getElement(arr, index) >= REQWEIGHT) && (getElement(arr, index) < MAPFILTER)){
        neighbors = 0;
        for (k = 0; k < 8; k++) {
          if(getElement(arr, BOXES * (j + checks[k].y) + (i + checks[k].x)) >=
REQWEIGHT){
            neighbors = neighbors + 1;
            if(neighbors >= NEIGHBORS){
              insertElement(arr, index, MAPFILTER);
              break;
            }
          }
        }
        // we want to filter for false points, so if the point does not have enough
        // neighbors then make sure it isn't a live point
        if(!neighbors){
          insertElement(arr, index, REQWEIGHT-DEWEIGHT);
        }
      }
    }
  }
}

/*
 *  checkInit - initialize points to search for neighbor searching
 */
void checkInit(point_t *arr){
    arr[0].x = 1;
    arr[0].y = 0;
    arr[1].x = 1;
    arr[1].y = 1;
    arr[2].x = 0;
    arr[2].y = 1;
    arr[3].x = -1;
    arr[3].y = 1;
    arr[4].x = -1;
    arr[4].y = 0;
    arr[5].x = -1;
    arr[5].y = -1;
    arr[6].x = 0;
    arr[6].y = -1;
```

```
    arr[7].x = 1;
    arr[7].y = -1;
}

/*
 *  IRxy - Turns an IR reading into an x, y coordinate in cm
 */
void IRxy(botState_t *botState, vertex_t *IRvertex){
  float x, y, irtheta, sumtheta;
  if(botState->IRcurrDist < MAXSIGHT){
    irtheta = (float) 2 * PIE * (botState->IRcurrPos/1440.0);
    sumtheta = irtheta + botState->theta;
    IRvertex->x = botState->IRcurrDist * cos(sumtheta) + botState->xpos;
    IRvertex->y = botState->IRcurrDist * sin(sumtheta) + botState->ypos;
  }
}

/*
 *  getDistance - IR sensor reading model to return where an obstacle is in cm
 */
float getDistance(void) {
  float dist, v_in;
  float p1, p2, p3, p4, p5;   // poly fit model

  // get sensor voltage
  v_in = anaInVolts(0, 1);
  v_in = (float)1/v_in;

  // model: using actual dist, inv volts w poly fit (deg 4, bisquare)
  p1 = 152.4;
  p2 = -381.3;
  p3 = 399.8;
  p4 = -134.4;
  p5 = 26.98;

  dist = p1 * pow(v_in, (float)4.0) + p2 * pow(v_in, (float)3.0) + p3 * pow(v_in,
(float)2.0) + p4 * v_in + p5;

  return dist;
}


/*************** MAPPING FUNCTIONS AND CONVERSIONS ***************/

/*
 *  initIntArray - Create an integer array in extended memory
 *    sz - size of requested array
 *    initVal - value for the array to be initialized to
 *    returns the address of the first element of the array in extened memory
 */
long initIntArray(long sz, int initVal) {
  long arr;
  int i;
  int element;
  sz = sz * sizeof(int);
  arr = xalloc(sz);
```

```
  for(i = 0; i < (sz/sizeof(int)); i++){
    xsetint((arr + i * sizeof(int)), initVal);
  }
  return arr;
}

/*
 *  getIndex - gets node value of x, y grid point on map
 */
int getIndex(int x, int y){
  int index;
  if (x < 0 || y < 0 ){
    index = -8;
  }
  else if (x >= BOXES || y >= BOXES){
    index = -8;
  }
  else {
    index = BOXES * y + x;
  }
  return index;
}

/*
 *  getElement - gets an element from an array stored in extended memory
 *    arr - address of array to access
 *    idx - index value of array
 *    returns an integer element value stored at the index
 */
int getElement(long arr, int idx){
  int element;
  element = xgetint(arr + idx * sizeof(int));
  return element;
}

/*
 *  insertElement - set the value of an element in extended memeory array
 *    arr - address of array to access
 *    idx - index value to set
 *    value - integer value to store
 */
void insertElement(long arr, int idx, int value){
  xsetint(arr + idx * sizeof(int), value);
}

/*
 *  incElement - increment value at a particular index in an extended array by +1
 *    arr - address of array to access
 *    idx - index value to increment
 */
void incElement(long arr, int idx){
  int value;
    value = getElement(arr, idx);
    value = value + 1;
    insertElement(arr, idx, value);
  }
```

```
/*
 *  IRMap - maps ir sensor (x, y) struct to increment a square on the map
 *    map - address of map array to access
 *    IRvertex - struct with x, y location of IR point
 */
void IRMap(long map, vertex_t *IRvertex){
  int xcoord, ycoord, index;
  xcoord = (int)floor(IRvertex->x/BOXLENGTH);
  ycoord = (int)floor(IRvertex->y/BOXLENGTH);
  index = getIndex(xcoord, ycoord);
  if (index != -8){
    incElement(map, index);
  }
}

/*
 *  idxToAvgPosX - get the center position x cm value of a certain node
 *    node - unique index value associated with a point on the grid
 *    returns center from 0,0 of the node in cm (x coord)
 */
float idxToAvgPosX(int node) {
  int map_pos_x;
  float avg_x;
  map_pos_x = idxToMapPosX(node);
  avg_x = BOXLENGTH * (map_pos_x + 1.0) - BOXLENGTH/2.0;
  return avg_x;
}

/*
 *  idxToAvgPosY - get the center position y cm value of a certain node
 *    node - unique index value associated with a point on the grid
 *    returns center from 0,0 of the node in cm (y coord)
 */
float idxToAvgPosY(int node) {
  int map_pos_y;
  float avg_y;
  map_pos_y = idxToMapPosY(node);
  avg_y = BOXLENGTH * (map_pos_y + 1.0) - BOXLENGTH/2.0;
  return avg_y;
}

/*
 *  idxToMapPosX - given unique node value, get the x map coord
 *    node - unique index value associated with a point on the grid
 *    returns map x coord (int)
 */
int idxToMapPosX(int node) {
  int map_pos_x;
  map_pos_x = (int) (float)node % BOXES;
  return map_pos_x;
}

/*
 *  idxToMapPosY - given unique node value, get the y map coord
 *    node - unique index value associated with a point on the grid
```

```
 *    returns map y coord (int)
 */
int idxToMapPosY(int node) {
  int map_pos_y;
  map_pos_y = (int)floor(node/BOXES);
  return map_pos_y;

}

/*
 *  printMap - prints all the obstacles that were mapped
 */
void printMap(long arr) {
  int i, j, index;
  float x, y;
  for(j = 1; j < BOXES-1; j++){
    for(i = 1; i < BOXES-1; i++){
      index = BOXES * j + i;
      if(getElement(arr, index) >= MAPFILTER){
        x = idxToAvgPosX(index);
        y = idxToAvgPosY(index);
        printf("%f, %f\n", x, y);
      }
    }
  }
}


/*************** NECESSARY BL2600 FUNCTIONS ***************/

int init(void)
{
  int fail;
  int i,j,k,delayvar;
  fail = 0;

  for (i = 0; i<4; i++)
  {
     EncWrite(i, XYIDR_SETUP,CNT);     // Disable Index
     EncWrite(i,EFLAG_RESET,CNT);

     EncWrite(i,BP_RESETB,CNT);

     EncWrite(i,CLOCK_DATA,DAT);
     EncWrite(i,CLOCK_SETUP,CNT);
     EncWrite(i,INPUT_SETUP,CNT);
     EncWrite(i,QUAD_X4,CNT);

     EncWrite(i,BP_RESETB,CNT);
     EncWrite(i,0x12,DAT);
     EncWrite(i,0x34,DAT);
     EncWrite(i,0x56,DAT);

     EncWrite(i,TRSFRPR_CNTR,CNT);
     EncWrite(i,TRSFRCNTR_OL,CNT);
```

```
      EncWrite(i,BP_RESETB,CNT);

    }
    return fail;
}

// channel is an int from 0 to 3 indicating which encoder
// reg is an int which is 1 or 0 indicating whether control or data is desired
int EncRead(int channel, int reg)
{
    int EncData;
    int i, delayvar;
    EncData = 0;

    digOutConfig(0xff00); // set data lines as inputs and everything else as outputs

    // select which chip
    if (channel <= 1)
       digOut(CS1,0);
    else
       digOut(CS2,0);

     // Select control or data register
     digOut(CD,reg);

    // select which channel, X or Y
    if ((channel == 0) | (channel == 3) )
       digOut(YX,0);
   else
       digOut(YX,1);
      // assert Read low
    digOut(RD,0);

    EncData = digInBank(0);      // read the data from the data lines

     //deassert read reads the data.  Deassert, delay to allow rise
    // then deselect chips
    digOut(RD,1);

    digOut(CS1,1);
    digOut(CS2,1);

    return EncData;
}

void EncWrite(int channel, int data, int reg)
{
    int i, delayvar;

    // select which chip - channel 0 & 1 are chip 1 and channel 2 & 3 are chip 2
    if (channel <= 1)
       digOut(CS1,0);
    else
       digOut(CS2,0);
    // select which channel, X or Y  X = 0 and 2, Y = 1 and 3
```

```
  digOut(CD,reg);

  if ((channel == 0) | (channel == 3) )
     digOut(YX,0);
 else
   digOut(YX,1);
   // assert write
   digOut(WR,0);    //First assert WR before driving outputs to avoid bus
                    //contention with encoder board  TWP 4/2/12

   digOutConfig(0xffff);// set all digI/O lines as outputs
   digOutBank((char)0,(char)data);

  // deassert write
  digOut(WR,1);

  // deselect chip
  digOut(CS1,1);
  digOut(CS2,1);
  //Set all outputs to 1 so that open collector transistor is off
  digOutBank((char)0,(char)HI_Z_STATE);
  digOutConfig(0xff00);
}
```