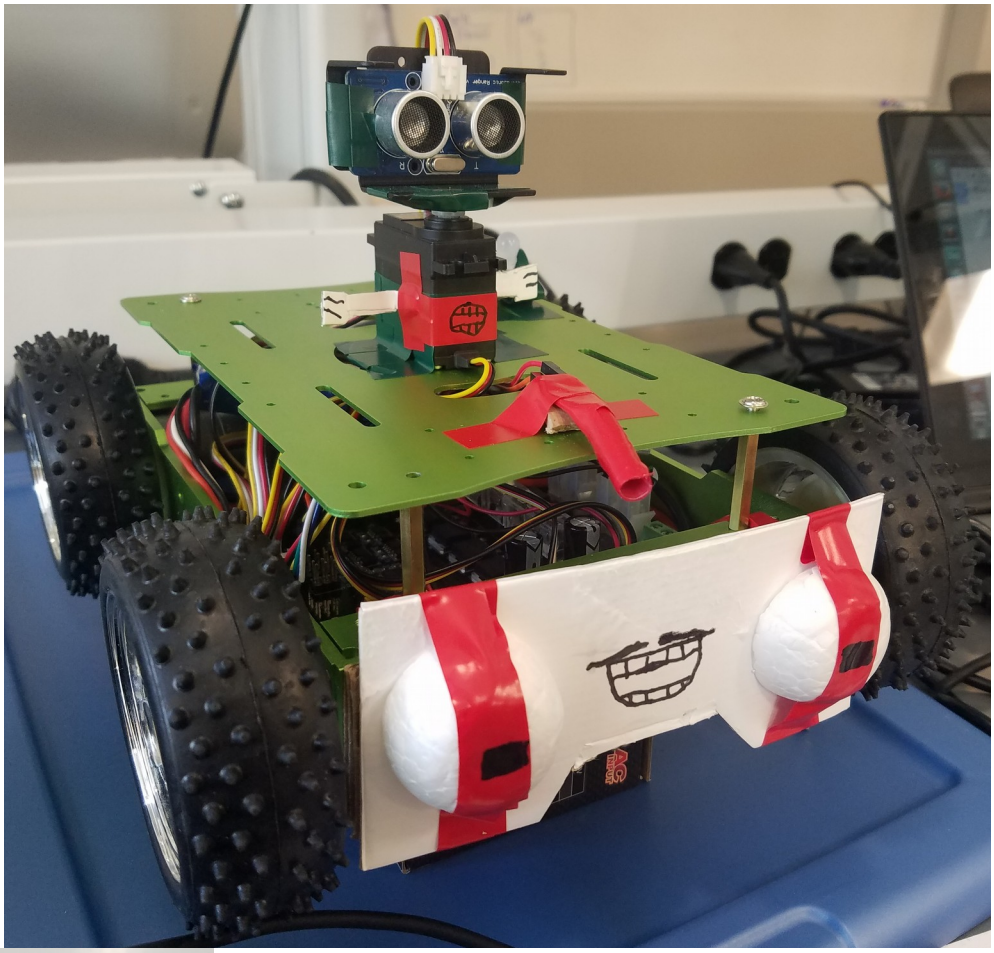
Nicolas Fredrickson

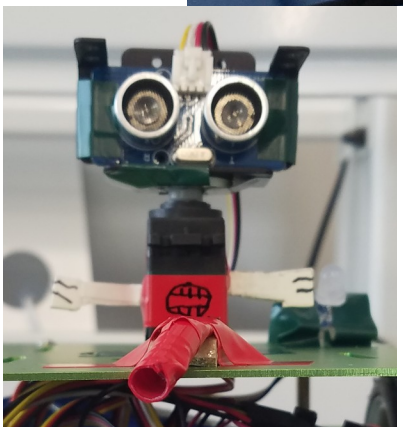
May 16th, 2017

Robotics Final Report

Our goal with this robot was to have it autonomously navigate to a certain goal point, then navigate back to its starting position, and then climb an incline using a light sensor. The robot was navigating through a 5x5 grid, with obstacles placed to occupy entire sectors. The incline was placed on the edge of one sector.

The starting position, our goal position, and our incline position were all deterministic – that is, they were determined ahead of time. The positions of the obstacles, however, were not. Thus, my robot – named **Slick** – had to navigate the grid while determining where our obstacles were.





Above: A picture of Slick, showing off the HS-422 Servo, its mounted Ultrasonic Ranger, the light-sensing Diode, the decorative bumper (which conceals the button) and the body of the robot. Also in view (albeit hard to see) is the RGB LED, just to the right of the servo, above the Master Blaster arm.

Left: A frontal view of the Servo, Ultrasonic Ranger, Diode, and RGB LED (just right of the Master Blaster arm).

The Robot

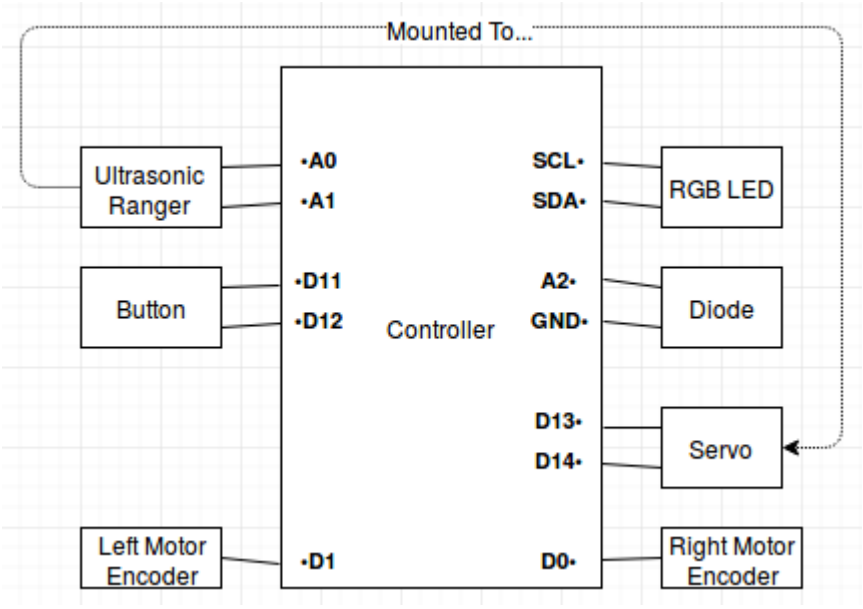
Our robot was composed of multiple hardware components, which were broadly grouped together in the code as different subsystems. When discussing the robot, I tend to break up the subsystems as the Motors, the Button, the Servo, the Ultrasonic Ranger, the Diode, and the RGB LED. The motors were simply the motors that came with the robot; they were attached to interrupt pins in order to count the number of encoder ticks, and don't merit much discussion beyond that.

The Button was placed on the front of the robot in order to prevent damage to the motors. If the robot was told to go forward and were to collide with an object, it would still attempt to spin the motors and ultimately wear down their machinery. However, since the button didn't extend past the wheels when mounted on the front, I created a bumper. The bumper sat over the button and was held loosely in place with electrical tape, to allow for the bumper to be compressed. In the end, the bumper was used not only to stop the robot after a collision, but to use that occurrence as a method of detection – transforming Slick into a bumpbot.

The Servo and the Ultrasonic Ranger, while separate subsystems, were very closely related. The Ultrasonic Ranger's purpose was to detect obstacles from a distance. In order to get the best coverage, the Ranger would need to face different directions; however, turning the robot would result in accumulating error. The Ranger was mounted onto the Servo so that it could face to the left, straight ahead, and to the right without the robot having to change it's orientation. Unfortunately, the Servo's range was constricted to roughly 180 degrees, which means we couldn't check behind the robot – not a problem though, as the only time we have to back up is after going forward. Also, the Ranger was meant to check for obstacles ahead of the robot while it was moving, and to stop the robot if a obstacle was detected. This proved to be wildly inaccurate and led to strange behavior, so it was removed.

The Diode was used to detect light during the Incline section of the robot's behavior. In order to give the diode a focus in one specific direction, it was placed in a custom sheath, composed of a straw and several layers of electrical tape. The Diode was also pointed downward, in order to track the guiding lights on the incline.

The RGB LED was placed on top of the robot purely for feedback. It was often difficult to gauge the robot's state when testing, so I needed someway for the robot to visually communicate its state. It had rather inconsistent behavior, owing to it typically being more of a debug tool than a necessary component of the robot.



Behaviors

The machine's behaviors were collected as a set of states – Wander, Wander Out, Go Home, Incline, and Done. It also kept track of it's movement state – Forward, Reverse, Stopped, Turn Left, Turn Right. In practice, only the first three movement states were relevant. The machine's default state was Stopped, in the Wander state.

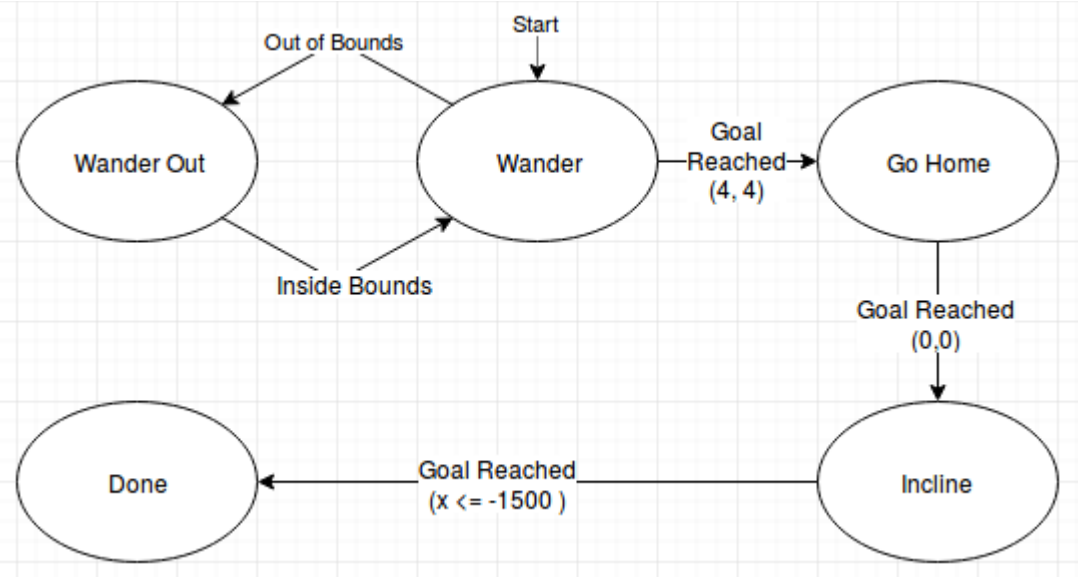
Whenever the robot was stopped, the Wander state would use the Ranger to investigate left, forward, and right. It would then try and plot a path to the goal, using a (mostly working) version of the A* algorithm. The algorithm selected the cheapest path, scoring each path using a heuristic – paths that introduced more error (i.e. turning) cost more. While we did find a complete path to the goal, we were actually only interested in the first action on the path, and the first subsequent action where we would stop the robot (either turning or straight-up stopping). We would take the first action (maybe going forward, going reverse, going left, or going right) and would consider that 'subsequent action' as our sub-target (for when we want to stop).

If we were going forward or reverse, and we went outside the boundary, we would enter the Wander Out state. We would stop, the immediately switch to the opposite direction. When we had re-entered the boundary, the vehicle would stop and re-enter Wander mode.

When going forward, if we collided with an obstacle, we would: stop, register the appropriate sector as blocked, set our sub-target to the current sector, and then reverse. If we were going forward or reverse, and we reached our sub-target, we would stop. Determining where to stop in a particular sector changed with our direction (forward or backward) and our heading.

Now, once we had reached our target we transition to the Go Home state. This state is literally just the Wander state with a several changes. First, the goal has been changed from (4, 4) to (0, 0). Second, we no longer check for hazards with our Ranger, or our Button, our even checking if we've gone over the boundry. Third, the robot now moves at a faster speed going forward or reversing. Finally, the path algorithm now gives a higher cost to sectors we haven't passed through before.

After we reach Go Home's goal, the robot turns to align itself with the ramp and pauses, to give me a chance to properly place it on the ramp. The robot the progresses up the ramp by alternating between going slightly left and going slightly right. Whatever state it is in, it flips when it detects that it has gone over the LED strip found on the incline. This keeps it roughly centered until it reaches its goal – having an x value of less than (or equal to) -1500, a measured value that typically stopped the robot as soon as it left the incline.



The Results

Wander Time	Go Home Time	Incline	Style, Verve
1:07 1 Mark	1:19 1 Mark	1:32 No marks	5 + 1

During the Wander phase, my robot ended up perceiving a blockage where there was none. I thought that would be the end of things, but to my surprise, my robot actually navigated out of that perceived dead end and proceeded to reach the goal. I found it quite impressive – even if it netted me the absolute longest time, even without the marks.

The error – which was an error I had seen before – seemed to happen when the Ranger was investigating a sector between two boxes. Typically when the robot ended up in these situations, it would end with an infinite loop of moving forward and backwards – a victim of bad path finding. Seemed that didn't happen this time – though why, I'm not too sure.

Still, after the Wander state, it only took 25 extra seconds to finish. My run was one of those with the least marked, and I had the most style and verve. Honestly, I feel like that's reflective of my overall design philosophy concerning the robot. I was trying to aim for consistency with the turning and navigation. It was exactly what I expected, but maybe just a bit slower than I wanted.

```
 1  #include "motordriver_4wd.h"
 2  #include "structs.h"
 3  #include <seeed_pwm.h>
 4  #include <StackArray.h>
 5
 6  //#define DEBUG 1
 7  //#define DEBUG_PING_CHECK 1
 8  //#define DEBUG_PATH 1
 9  //#define DELAY_START 1
10
11  enum {FORWARD, BACKWARD, TURN_LEFT, TURN_RIGHT, STOPPED} moveState;
12  enum {WANDER, WANDEROUT, GOHOME, INCLINE, DONE} machineState;
13
14  //Calibration function prototypes
15  void inline checkTicks();
16  void inline defaultPan();
17  void inline tickStopper();
18  void inline initialTest();
19  void inline lightInit();
20  void inline colorLoop();
21  void inline farthest();
22
23  void setup()
24  {
25    //Individually initialize our components
26    servoInit();
27    moveInit();
28    buttonInit();
29    lightInit();
30    diodeInit();
31    worldNavInit();
32
33    initialTest();
34
35    clavInit();
36
37    #ifdef DELAY_START
38    lightCycle(250);
39    #endif
40
41    //Initialize the serial communications:
42    Serial.begin(9600);
43  }
44
45  void loop()
46  {
47    //forward(20);
48    clavBrain();
49    //farthest();
50    //Serial.println( getDiodeVal() );
51  }
```

```
 1
 2  #define COLOR_HUE_FORWARD   0.0
 3  #define COLOR_HUE_LEFT   0.25
 4  #define COLOR_HUE_BACKWARD 0.5
 5  #define COLOR_HUE_RIGHT  0.75
 6
 7  #define CLAV_BRAIN_TURN_SPEED_LEFT  40
 8  #define NINETY_DEGREE_LEFT  69//67  //Right ticks to turn left
 9
10  #define CLAV_BRAIN_TURN_SPEED_RIGHT 40
11  #define NINETY_DEGREE_RIGHT 69  //Left ticks to turn right //66 //70
12
13  #define PING_TARGET_CM_STOP   60 //We can see two boxes away!
14  #define PING_TARGET_CM_MOVE   24
15  //#define PING_TARGET_MM   500
16
17  #define INVESTIGATE_ITERATES    2
18
19  #define WANDER_TARGET_COL  4
20  #define WANDER_TARGET_ROW  4
21
22  #define GO_HOME_TARGET_COL  0
23  #define GO_HOME_TARGET_ROW  0
24
25  #define INCLINE_TARGET_COL  -5
26  #define INCLINE_TARGET_ROW  0
27
28  #define INCLINE_TARGET_THETA  ONE_PI
29
30  #define INCLINE_SPEED_FAST 16
31  #define INCLINE_SPEED_SLOW 8
32  #define COLLISION_SPEED   5
33  #define WANDER_SPEED      10
34  #define KNOWN_SPEED       15
35
36  int targetCol, targetRow;
37  boolean parity;
38
39  //=========================================================================================
40  // Initializer
41  //=========================================================================================
42  inline void clavInit(){
43    tickCountLeft = tickCountRight = 0;
44    moveState = STOPPED;
45    updateDirection();
46
47    specificColor(COLOR_HUE_FORWARD);
48
49    dTheta = 0;
50    dX = 0;
51    dY = 0;
52
53    x = EXCESS / 2; //0;
54    y = EXCESS / 2; //0;//200;//50.0;//80.0;
55    thetaState = PI_O2;
56    theta = radianToValue(thetaState);
57    addShift(thetaState);
58
59    targetCol = WANDER_TARGET_COL;
60    targetRow = WANDER_TARGET_ROW;
61
62    radianPanTo( RANGER_FORWARD );
63    delay(50);
64    parity = false;
65    //anglePanTo(90);
66    //Straight( 10, 1 );
67  }
68
69  //=========================================================================================
```

```
 70   // Loop
 71   //========================================================================================
 72   inline void clavBrain()
 73   {
 74
 75     //if(moveState == FORWARD)
 76       //pingCheck();
 77
 78     //delay(100);
 79
 80     //---- update robot config (x,y,theta)
 81     dX = PI * WHEEL_RADIUS * cos(theta) * ((double)(tickCountLeft + tickCountRight) /
     TICK_PER_ROT);
 82     xNoSkew = xNoSkew + dX;
 83     x = x + dX;
 84
 85     dY = PI * WHEEL_RADIUS * sin(theta) * ((double)(tickCountLeft + tickCountRight) /
     TICK_PER_ROT);
 86     yNoSkew = yNoSkew + dY;
 87     y = y + dY;
 88
 89
 90     tickCountLeft = tickCountRight = 0;
 91
 92     markCurrentPass();
 93
 94     #ifdef DEBUG
 95     Serial.print(x); Serial.print(" :x || y: "); Serial.println(y); Serial.print("\t");
 96     Serial.print(valueToSectorMM( x )); Serial.print(" :col || row: "); Serial.println
     (valueToSectorMM( y ));
 97     #endif
 98
 99     //Change our light
100     if(parity)
101       radianToColor( radianToValue( moveEffectTheta() ) );
102     else
103       radianToColor( theta );
104
105     parity = !parity;
106
107     if(machineState == WANDER || machineState == WANDEROUT){
108       clavGoPlace(true);
109       //---- check if we're completely in the
110       if ( posInTarget(targetCol, targetRow) )
111       {
112         ClavStop();
113
114         #ifdef DEBUG
115         Serial.println("======================");
116         Serial.println("\t Wander Target Reached!");
117         Serial.println("======================");
118         #endif
119
120         //---- update state
121         radianPanTo( RANGER_FORWARD );
122         delay(100);
123
124         machineState = GOHOME;
125         targetCol = GO_HOME_TARGET_COL;
126         targetRow = GO_HOME_TARGET_ROW;
127       }
128     }
129     else if (machineState == GOHOME)
130     {
131       //---- use map to go home
132       clavGoPlace(false);
133
134       //---- check if we're completely in the
135       if ( posInTarget(targetCol, targetRow) )
```

```
136       {
137         ClavStop();
138
139         #ifdef DEBUG
140         Serial.println("======================");
141         Serial.println("\t Go Home - Target Reached!");
142         Serial.println("======================");
143         #endif
144
145         machineState = INCLINE;
146         targetCol = INCLINE_TARGET_COL;
147         targetRow = INCLINE_TARGET_ROW;
148
149         turnToTheta(ONE_PI);
150
151         lightCycle(50);
152       }
153     }
154     else if (machineState == INCLINE) {
155       if ( getDiodeVal() > 100) {
156         if (lightState == LEFT) {
157           //Set Right
158           MOTOR.setSpeedDir1(INCLINE_SPEED_FAST, DIRF);
159           //Set Left
160           MOTOR.setSpeedDir2(INCLINE_SPEED_SLOW, DIRR);
161           lightState = RIGHT;
162           delay(100);
163         }
164         else if (lightState == RIGHT){
165           //Set Right
166           MOTOR.setSpeedDir1(INCLINE_SPEED_SLOW, DIRF);
167           //Set Left
168           MOTOR.setSpeedDir2(INCLINE_SPEED_FAST, DIRR);
169           lightState = LEFT;
170           delay(100);
171         }
172       }
173
174       //---- check if we're completely in the
175       if ( x <= -1550 )
176       {
177         ClavStop();
178
179         #ifdef DEBUG
180         Serial.println("======================");
181         Serial.println("\t Incline - Target Reached!");
182         Serial.println("======================");
183         #endif
184
185         machineState = DONE;
186       }
187     }
188     else if (machineState == DONE){
189       lightCycle(100);
190     }
191   }
192
193   //~~~~~~~~~~~~~
194   // Sub-brain: Wander (for wandering, and the like)
195   //~~~~~~~~~~~~~
196   void clavGoPlace(boolean investigate){
197
198     //If we've stopped
199     if( moveState == STOPPED ){
200       int speedWeDo;
201       //Investigate the surrounding area
202       if(investigate){
203         investigatePing();
204         speedWeDo = WANDER_SPEED;
```

```
205       }
206       else
207         speedWeDo = KNOWN_SPEED;
208
209       #ifdef DEBUG
210       printPassGrid();
211       delay(100);
212       printStatGrid();
213       delay(100);
214       #endif
215
216       //Formulate a plan
217       pathFindFromCurrent(targetCol, targetRow);
218
219       tickCountLeft = tickCountRight = 0;
220
221       if(investigate){
222         radianPanTo( RANGER_FORWARD ); delay(50);
223       }
224
225       #ifdef DEBUG
226       Serial.println( getStartAction() );
227       #endif
228
229       //Follow the plan
230       switch( getStartAction() ){
231         default:
232         case ACTION_STOP:
233         case ACTION_NONE:
234           break;
235
236         case ACTION_FORWARD:
237           ClavForward(speedWeDo);break;
238
239         case ACTION_REVERSE:
240           ClavReverse(speedWeDo); break;
241
242         case ACTION_LEFT_FORWARD:
243           TurnLeft90(); ClavForward(speedWeDo); break;
244
245         case ACTION_RIGHT_FORWARD:
246           TurnRight90(); ClavForward(speedWeDo); break;
247       }
248     }
249     //If we've reached our subtarget
250     else if( posInSubTarget() ){
251       //Stop!
252       ClavStop();
253       #ifdef DEBUG
254       Serial.println("======================");
255       Serial.println("\t Subtarget Reached!");
256       Serial.println("======================");
257       #endif
258     }
259     // If we're moving and we're suddenly attacked by a wild box
260     else if( moveState == FORWARD && digitalRead(buttonPin) )
261     {
262       //---- stop
263       ClavStop();
264       //---- mark that we had a collision, adjust our sights
265       blockCollision();
266
267       tickCountLeft = tickCountRight = 0;
268
269       ClavReverse(COLLISION_SPEED);
270
271       #ifdef DEBUG
272       Serial.println("======================");
273       Serial.println("\t Button Pressed!");
```

```
274       Serial.println("======================");
275       #endif
276     }
277     //otherwise, if we're moving forward and we go out of bounds
278     else if( (moveState == FORWARD || moveState == BACKWARD) && outOfBoundry() &&
      machineState == WANDER )//(OUT_OF_BOUNDRY) )
279     {
280       int oldState = moveState;
281       //---- stop
282       ClavStop();
283
284       //Mark that we've gone out of bounds
285       machineState = WANDEROUT;
286
287       #ifdef DEBUG
288       Serial.println("======================");
289       Serial.println("\tOut of Bounds!");
290       Serial.println("======================");
291       #endif
292
293       if(oldState == FORWARD)
294         ClavReverse(WANDER_SPEED);
295       else if(oldState == BACKWARD)
296         ClavForward(WANDER_SPEED);
297     }
298     else if ( machineState == WANDEROUT && inBoundry() )//(IN_BOUNDRY))
299     {
300       //---- stop
301       ClavStop();
302       machineState = WANDER;
303     }
304     /*
305     else if(machineState == WANDER || machineState == WANDEROUT){
306       projectionCheck();
307     }
308     */
309   }
310
311   //============================================================================================
312   // Initiate the forward - this code is duplicated so much that we might as well
313   //============================================================================================
314   inline void ClavForward(int inSpeed) {
315     //---- update state
316     moveState = FORWARD;
317     updateDirection();
318     //---- go forward
319     forward(inSpeed);
320     //---- R E S T    A E S T H E T I C A L L Y
321     delay(100);
322   }
323
324   //============================================================================================
325   // Initiate the reverse - this code is duplicated so much that we might as well
326   //============================================================================================
327   inline void ClavReverse(int inSpeed) {
328     //---- update state
329     moveState = BACKWARD;
330     updateDirection();
331     //---- back up
332     backward(inSpeed);
333     //---- R E S T    A E S T H E T I C A L L Y
334     delay(100);
335   }
336
337   //============================================================================================
338   // Stop the car - this code is duplicated so much that we might as well
339   //============================================================================================
340   inline void ClavStop() {
341     //---- stop
```

```
342     stopMotion();
343     //---- R E S T    A E S T H E T I C A L L Y
344     delay(100);
345     //---- update state
346     moveState = STOPPED;
347     updateDirection();
348   }
349
350   //=============================================================================================
351   // TurnLeft90
352   //=============================================================================================
353   void TurnLeft90() {
354       //Set up for turning left
355       specificColor(COLOR_HUE_LEFT);
356       moveState = TURN_LEFT;
357       updateDirection();
358       tickCountLeft = tickCountRight = 0;
359
360       //begin the turn
361       turnInPlaceLeft(CLAV_BRAIN_TURN_SPEED_LEFT);
362
363       //while loop to turn as we please
364       while (tickCountRight < NINETY_DEGREE_LEFT)
365       {
366         delayMicroseconds(1);
367       }
368
369       //Stop the turn
370       ClavStop();
371
372       #ifdef DEBUG
373       Serial.println("======================");
374       Serial.println("\t Turning Left!");
375       Serial.println("======================");
376       #endif
377
378       tickCountLeft = tickCountRight = 0;
379
380       removeShift(thetaState);
381
382       //Update our theta (positively)
383       incrementTheta();
384
385       addShift(thetaState);
386
387       //R E S T    A E S T H E T I C A L L Y
388       delay(100);
389   }
390
391   //=============================================================================================
392   // TurnRight90
393   // dirn is 1 for right, -1 for left
394   //=============================================================================================
395   void TurnRight90() {
396       //Set up for turning right
397       specificColor(COLOR_HUE_RIGHT);
398       moveState = TURN_RIGHT;
399       updateDirection();
400       tickCountLeft = tickCountRight = 0;
401
402       //begin the turn
403       turnInPlaceRight(CLAV_BRAIN_TURN_SPEED_RIGHT);
404
405       //while loop to turn as we please
406       while (tickCountLeft < NINETY_DEGREE_RIGHT)
407       {
408         delayMicroseconds(1);
409       }
410
```

```
411       //Stop the turn
412       ClavStop();
413
414       #ifdef DEBUG
415       Serial.println("======================");
416       Serial.println("\t Turning Right!");
417       Serial.println("======================");
418       #endif
419
420       tickCountLeft = tickCountRight = 0;
421
422       removeShift(thetaState);
423
424       //Update our theta (negatively)
425       decrementTheta();
426
427       addShift(thetaState);
428
429       //R E S T    A E S T H E T I C A L L Y
430       delay(100);
431   }
432
433   //================================================================================
434   // TurnToTheta
435   //================================================================================
436   void turnToTheta(int target){
437     switch( target - thetaState ){
438       case -1: TurnRight90();
439         break;
440       case 1: TurnLeft90();
441         break;
442       case 2:
443       case -2: TurnRight90(); TurnRight90();
444         break;
445       default:
446       case 0:
447         break;
448     }
449   }
450
451   //Take a look around, see?
452   void investigatePing(){
453     //start left
454     int angle = RANGER_LEFT;
455
456     //for each angle
457     for(int i = 0; i < 3; i++){
458       //Pan to that angle
459       radianPanTo(angle);
460       delay(50);
461
462       //If whatever we're scanning isn't automatically going to be out of bounds
463       if( inSectors(valueToSectorMM( x ) + projectColPan(), valueToSectorMM( y ) +
      projectRowPan()) ){
464         //Then ping X times
465         for(int j = 0; j < INVESTIGATE_ITERATES; j++){
466           pingCheck();
467           delay(100);
468         }
469       }
470       angle = radianLeft(angle);
471     }
472     evalPanTheta();
473   }
474
475   inline void pingCheck(){
476     boolean test;
477     int dist_cm = pingCM();
478     double objX, objY;
```

```
479     int objRow, objCol;
480
481     if( pingMoveCheck(dist_cm) || pingStopCheck(dist_cm) ) {
482       objX = (x / 10) + cos( evalPanTheta() ) * dist_cm;
483       objY = (y / 10) + sin( evalPanTheta() ) * dist_cm;
484
485       objCol = valueToSectorCM( objX );
486       objRow = valueToSectorCM( objY );
487
488       if( objCol == valueToSectorMM( x ) && objRow == valueToSectorMM( y ) ){
489           objCol += projectColPan();
490           objRow += projectRowPan();
491       }
492
493       if( inSectors( objCol, objRow ) )
494       {
495         if( getSectorStat( objCol, objRow ) <= 0.0 )
496           setSectorStat( objCol, objRow, 0.5);
497
498         incrementSectorStat( objCol, objRow, ( 1.0 - getSectorStat( objCol, objRow ) ) / 2.0
    );
499       }
500     }
501   }
502
503   inline boolean pingMoveCheck(int dist_cm){
504     return (moveState == FORWARD || moveState == BACKWARD) && dist_cm < PING_TARGET_CM_MOVE;
505   }
506
507   inline boolean pingStopCheck(int dist_cm){
508     return moveState == STOPPED && dist_cm < PING_TARGET_CM_STOP;
509   }
```

```
1   //================================================================================
2   // Path Finder!
3   //================================================================================
4   boolean pathFindFromCurrent(int targCol, int targRow) {
5
6     pathTargetCol = targCol;
7     pathTargetRow = targRow;
8
9     boxForgive = false;
10
11    //(SectorPath){ nextPoint.col, nextPoint.row, firstPoint.inHeading,
    firstPoint.outAction };
12    subTarget = (SectorPath) {
13      valueToSectorMM( x ), valueToSectorMM( y ), thetaState, ACTION_NONE
14    };
15
16    boolean success = pathFindRecursive( &subTarget );
17    worldPassReset();
18
19    if (!success) {
20      boxForgive = true;
21      subTarget = (SectorPath) {
22        valueToSectorMM( x ), valueToSectorMM( y ), thetaState, ACTION_NONE
23      };
24      success = pathFindRecursive( &subTarget );
25      worldPassReset();
26    }
27
28    Serial.print("Subtarget: "); Serial.print(subTarget.col); Serial.println( subTarget.row);
29
30    if (subTarget.col > 5)
31      subTarget.col = 4;
32    else if (subTarget.col < 0)
33      subTarget.col = 0;
34
35    if (subTarget.row > 5)
36      subTarget.row = 4;
37    else if (subTarget.row < 0)
38      subTarget.row = 0;
39
40    Serial.print("\t Modified Subtarget: "); Serial.print(subTarget.col); Serial.println
    (subTarget.row);
41
42    return success;
43  }
44
45  boolean pathFindRecursive(SectorPath *thisSector) {
46
47    //Check if this is even a valid position
48    if ( !inSectors(thisSector->col, thisSector->row) || getSectorPass(thisSector->col,
    thisSector->row) == PASSED_BLOCKED ) {
49      resetSectorPassPlan(thisSector->col, thisSector->row);
50      return false;
51    }
52
53    //Check if we hit the target
54    if ( thisSector->col == pathTargetCol && thisSector->row == pathTargetRow) {
55      //If we did, set this action as STOP
56      thisSector->outAction = ACTION_STOP;
57      //Set our subtarget to this
58      subTarget.col = thisSector->col;
59      subTarget.row = thisSector->row;
60      resetSectorPassPlan(thisSector->col, thisSector->row);
61      //Spread the word!
62      return true;
63    }
64
65    //Mark our postion on the pass grid
66    setSectorPassPlan( thisSector->col, thisSector->row);
```

```
 67
 68     boolean success = false;
 69
 70     //These track the direction indicies
 71     //Since we came from one direction,
 72     //it naturally follows that there are only three we can take
 73     //Thus, three variables
 74     byte headings[3] = {255, 255, 255};
 75
 76     //And we only need to track the low and mid scores
 77     //Since we only care about their positions relative to each other;
 78     byte scores[3] = {255, 255, 255};
 79     byte genScore = 255;
 80
 81     //Now, probe our four options:
 82     for (int probeHead = 0; probeHead < THETA_INCREMENTS; probeHead++) {
 83
 84       genScore = headingScore(
 85                    thisSector->col + projectCol(probeHead), thisSector->row + projectRow
    (probeHead),
 86                    thisSector->col, thisSector->row, thisSector->inHeading, probeHead
 87                  );
 88
 89       //if the score is valid
 90       if ( genScore != 255 ) {
 91         //if it's lower than the lowest, or equal
 92         if ( genScore == scores[0] || min(genScore, scores[0]) == genScore ) {
 93           //boot the middle to the highest
 94           headings[2] = headings[1]; scores[2] = scores[1];
 95           //boot the lowest to middle
 96           headings[1] = headings[0]; scores[1] = scores[0];
 97           //set the lowest
 98           headings[0] = probeHead; scores[0] = genScore;
 99         }
100         //else, if it's lower than the middle, or equal
101         else if ( genScore == scores[1] || min(genScore, scores[1]) == genScore ) {
102           //boot the middle to the highest
103           headings[2] = headings[1]; scores[2] = scores[1];
104           //set the middle
105           headings[1] = probeHead; scores[1] = genScore;
106         }
107         //else, if it's lower than the highest, or equal
108         else if ( genScore == scores[2] || min(genScore, scores[2]) == genScore )
109           //set the highest
110           headings[2] = probeHead; scores[2] = genScore;
111       }
112     }
113
114     SectorPath nextSector = (SectorPath) {
115       0, 0, 0, ACTION_NONE
116     };
117
118     for (int i = 0; i < 3 && !success; i++) {
119       if (headings[i] != 255 && scores[i] != 255) {
120         thisSector->outAction = headingsToAction( thisSector->inHeading, headings[i] );
121
122         nextSector.col = thisSector->col + projectCol( headings[i] );
123         nextSector.row = thisSector->row + projectRow( headings[i] );
124         nextSector.inHeading = actionToNewHeading( thisSector->inHeading, thisSector-
    >outAction );
125         nextSector.outAction = ACTION_NONE;
126
127         success = pathFindRecursive( &nextSector );
128       }
129     }
130
131     //If we succeeded
132     if (success)
133       //And this sector's action is a waypoint action
```

```
134       if (thisSector->outAction >= ACTION_STOP)
135         //And its not the start sector
136         if ( thisSector->col != valueToSectorMM( x ) || thisSector->row != valueToSectorMM
      ( y ) ) {
137           #ifdef DEBUG
138           Serial.println("subtargetReassign"); Serial.print(thisSector->col); Serial.print
      (thisSector->row);
139           #endif
140           subTarget.col = thisSector->col; subTarget.row = thisSector->row;
141         }
142
143     //Finally, undo the marking so we can use the pass grid later
144     resetSectorPassPlan(thisSector->col, thisSector->row);
145
146     return success;
147   }
148
149   byte headingScore(int col, int row,  int currCol, int currRow, int oldHeading, int
      newHeading) {
150     //If a path isn't possible, reject it;
151     if ( !isPathPossible( col, row ) )
152       return 255;
153
154     //if we are considering high probability squares dangerous, and this is high prob,
      reject.
155     if ( !boxForgive && statGrid[ row ][ col ] >= 0.50)
156       return 255;
157
158     //Base score
159     byte score = 1;
160
161     //If the parities of the heading don't match, that means we have to turn - that's worth
      two points.
162     if ( (oldHeading % 2) != (newHeading % 2) )
163       score += 2;
164
165     //If the sector is an unknown, we multiply the blockage certainty by 10 and add it to
      the score
166     if ( boxForgive && passGrid[ row ][ col ] == PASSED_UNSURE )
167       score += byte( 10 * statGrid[ row ][ col ] );
168
169     //If the sector is an unknown, and we are in GoHome or GoIncline, unknown spaces cost
      more
170     if ( machineState == GOHOME && passGrid[ row ][ col ] != PASSED_OPENED )
171       score += 3;
172
173     //If, relative to our last position, we're moving away from the goal, we add 4.
174     if ( abs(pathTargetCol - currCol) - abs(pathTargetCol - col ) < 0  ||
175          abs(pathTargetRow - currRow) - abs(pathTargetRow - row ) < 0 )
176       score += 4;
177
178     return score;
179   }
```