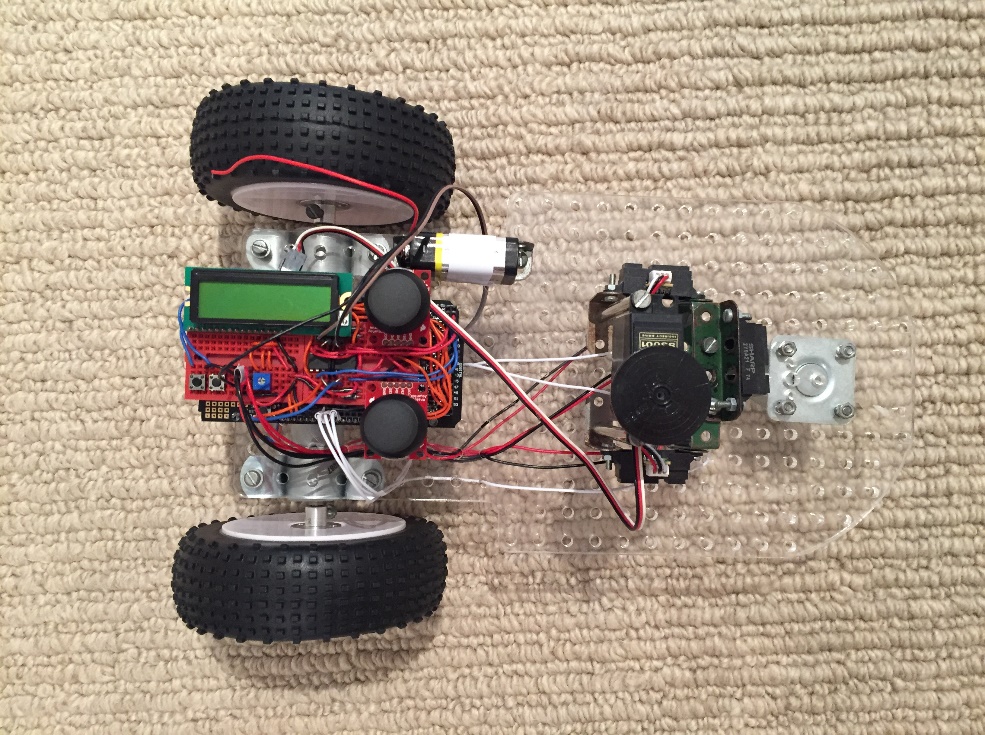
MXEN2002 REPORT

# INTRODUCTION

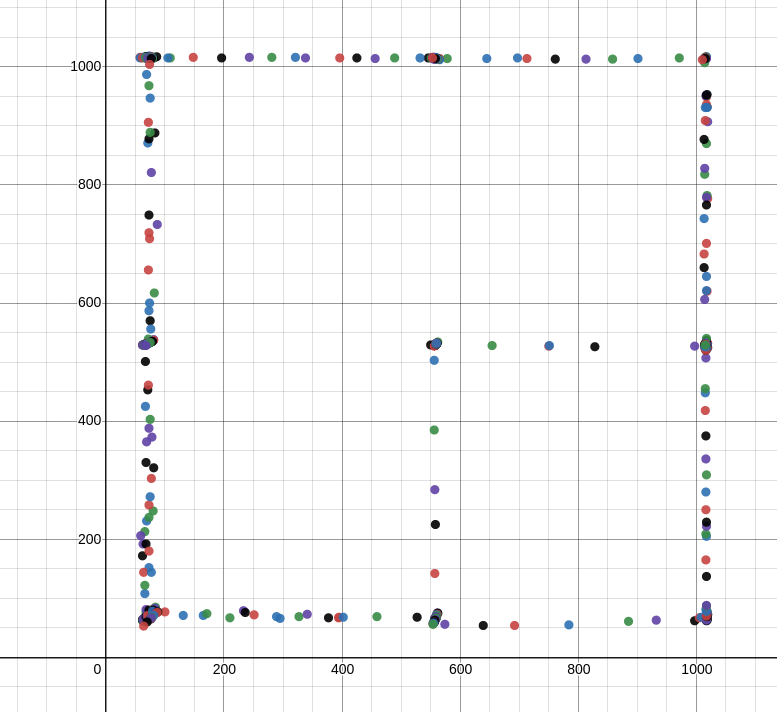
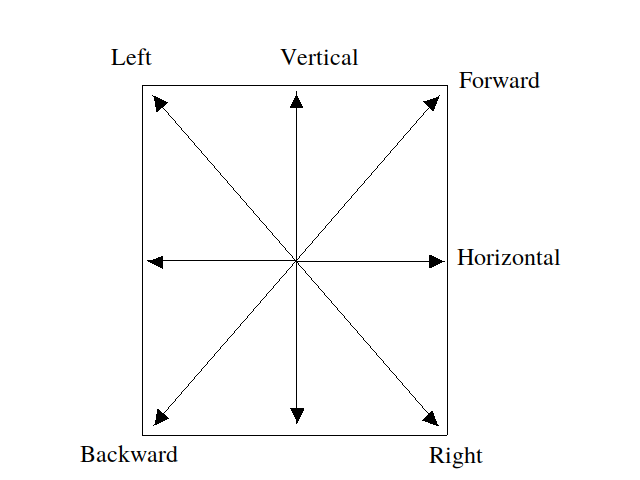
The goal of the project was to make a robot capable of both autonomous and manual control.



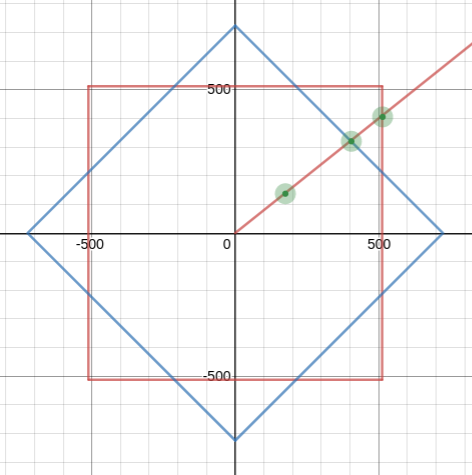
# OVERVIEW OF SUBSYSTEMS

## JOYSTICK MIXING

Sampling readings from both the horizontal and vertical channel as the joystick moves around outside of its range gives the following plot. We can also notice that if we assign the left and right motors a speed proportional to the magnitude of the difference between the horizontal/vertical channel and the mean and a direction related to which side of the mean the readings lie, then it will behave like there is joystick mixing, albeit rotated 45° from what is expected.



If we translate the input to be centred around the origin (by subtracting the mean from both channels) and treat the input as a complex number with real and imaginary components corresponding to the horizontal and vertical channels, then it would be easy to rotate the input so that it takes on the correct orientation by multiplying by another complex number with magnitude 1 and argument of 45 degrees. The issue with this is that the magnitudes would no longer be correct. One option would be to scale up the inputs after the rotation and then clamp the maximum value assigned to the motor speed. This reduces the control the user has over the speed as there is less of a difference between stopped and full speed.

An alternative to this is to calculate a scale factor for each point by finding the ratio between the line segment that connects the origin to the input before rotation and the line segment connecting the origin to the input after rotation, both passing through the point whose scale factor we are trying to calculate. An example of these points is shown in the figure below.

For some point we can find the point on the unrotated and rotated boundary respectively, where is half the width of the square.

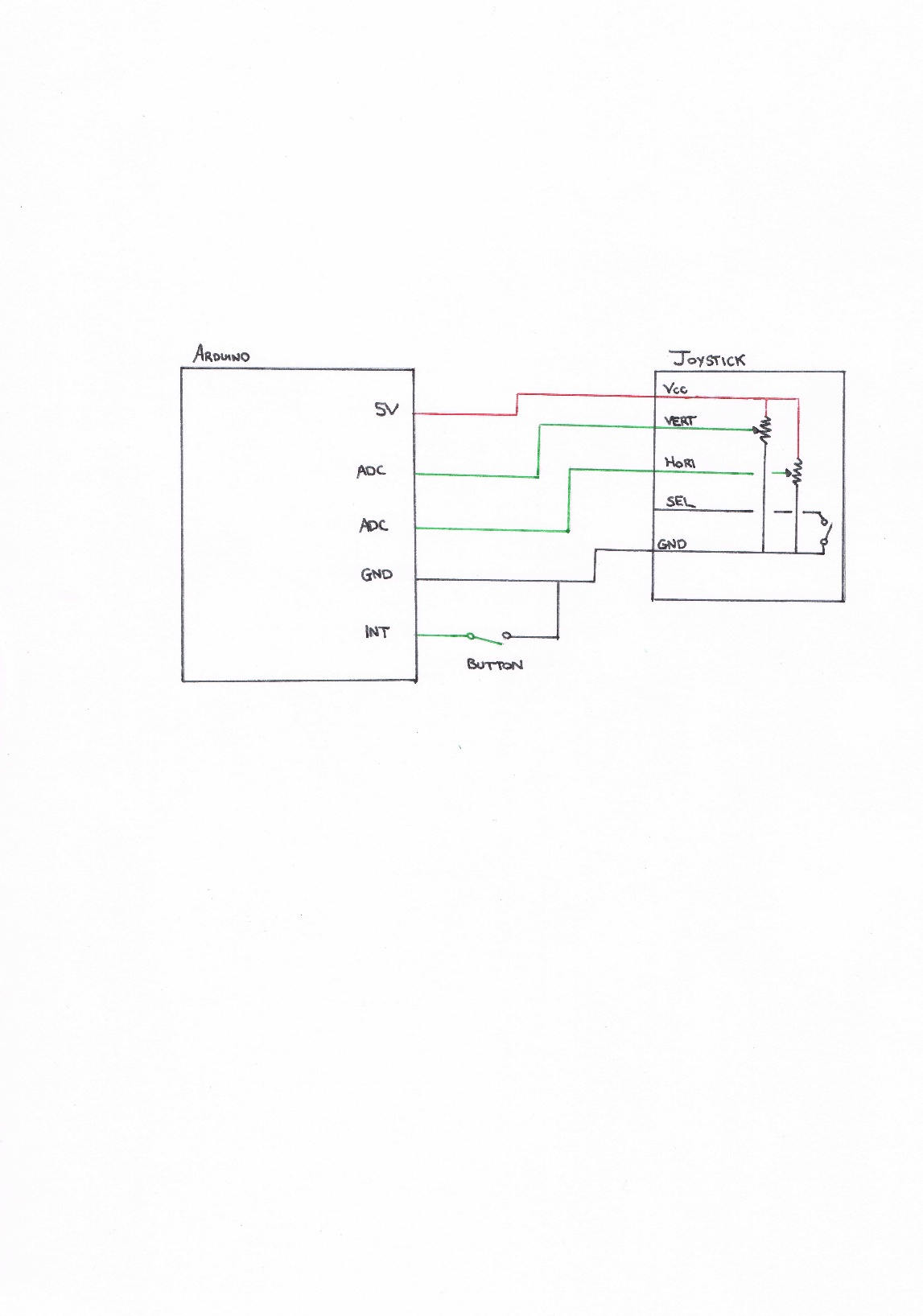
Taking the ratio between the magnitudes of the vectors from the origin to these points gives the function

Which simplified to

If we have a point in the second octant, we can use the symmetry along the line to say that

Using a similar fact for the lines and we can express the scale factor for all points with

To assign the motor speeds and directions according to this transformation, we follow this algorithm:

1. INPUT ,
2. LET
3. LET
4. LET
5. Set the left and right motor direction based on the sign of and.

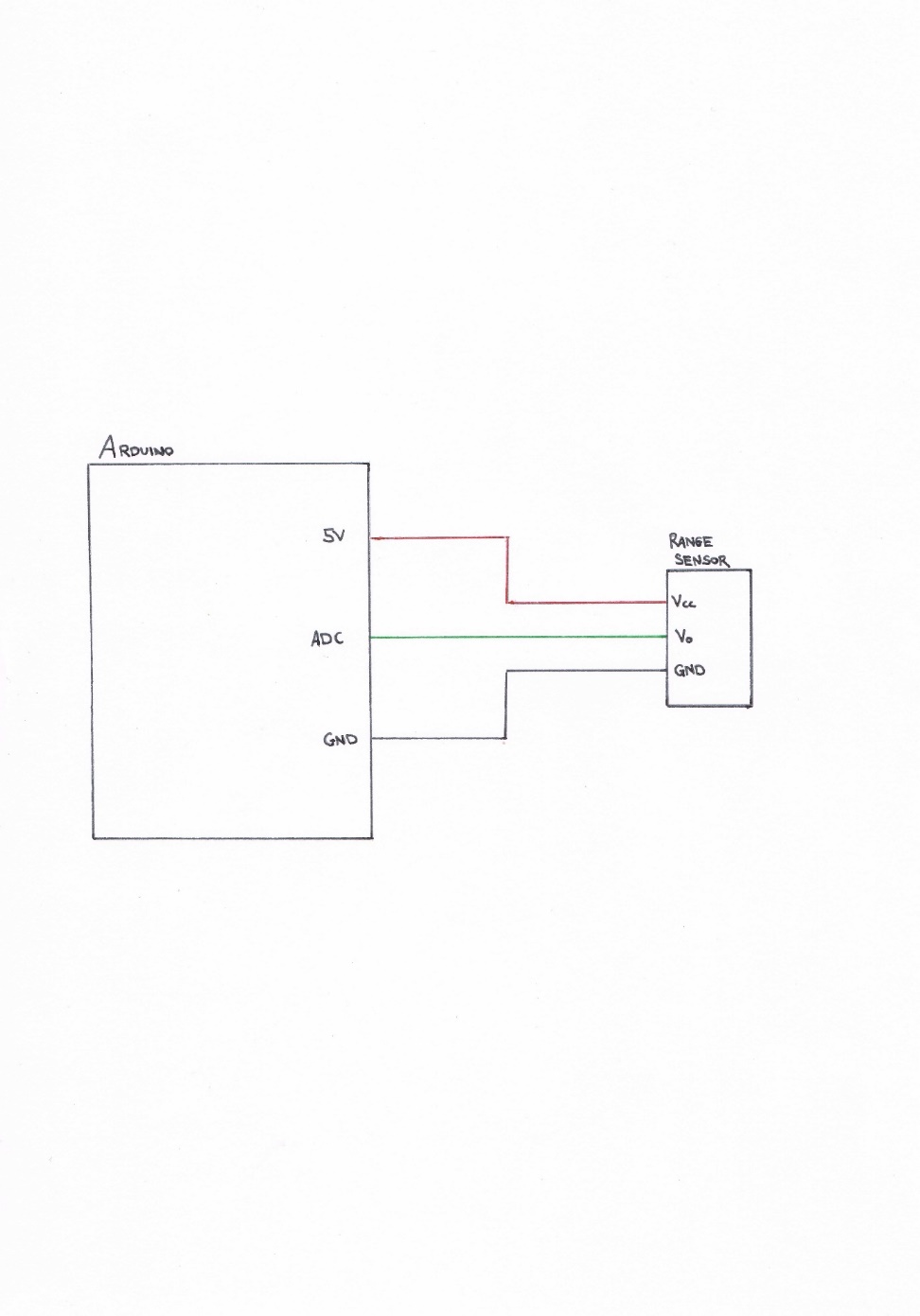
## COMMUNICATION PROTOCOL

Non-existent atm.

## RANGE SENSORS

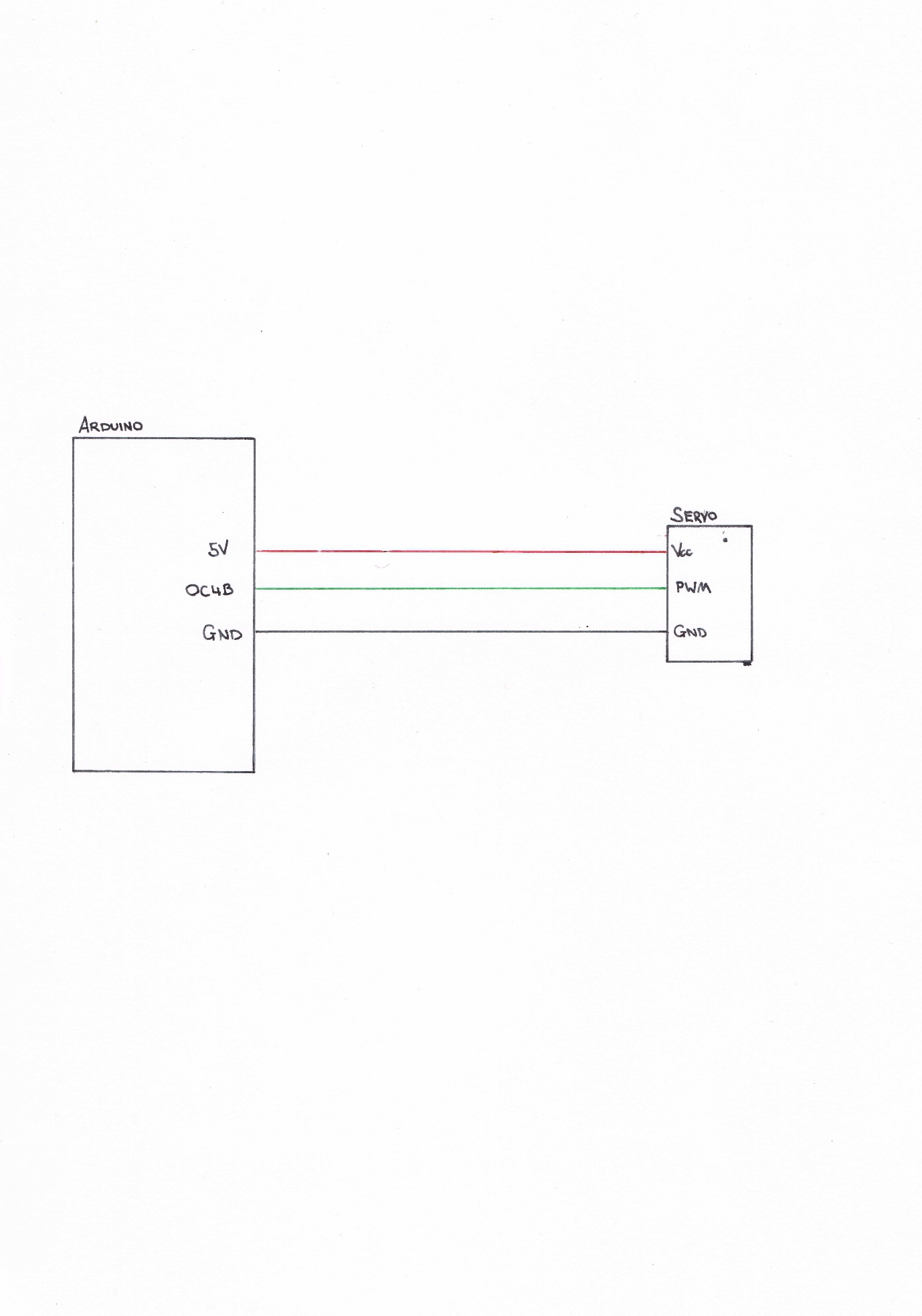
There are two types of range sensor present on the robot. The front sensor is a Sharp GP2Y0A21YK0F with a rated range of 10-80cm and the left and right sensors are both a Sharp GP2Y0A41SK0F with a rated range 4-30cm.

The front sensor was calibrated by taking the average of a minimum of 5000 samples of the adc value at varying distances both inside and outside the rated range. These were then plotted and a curve of best fit was found for a suitable range,

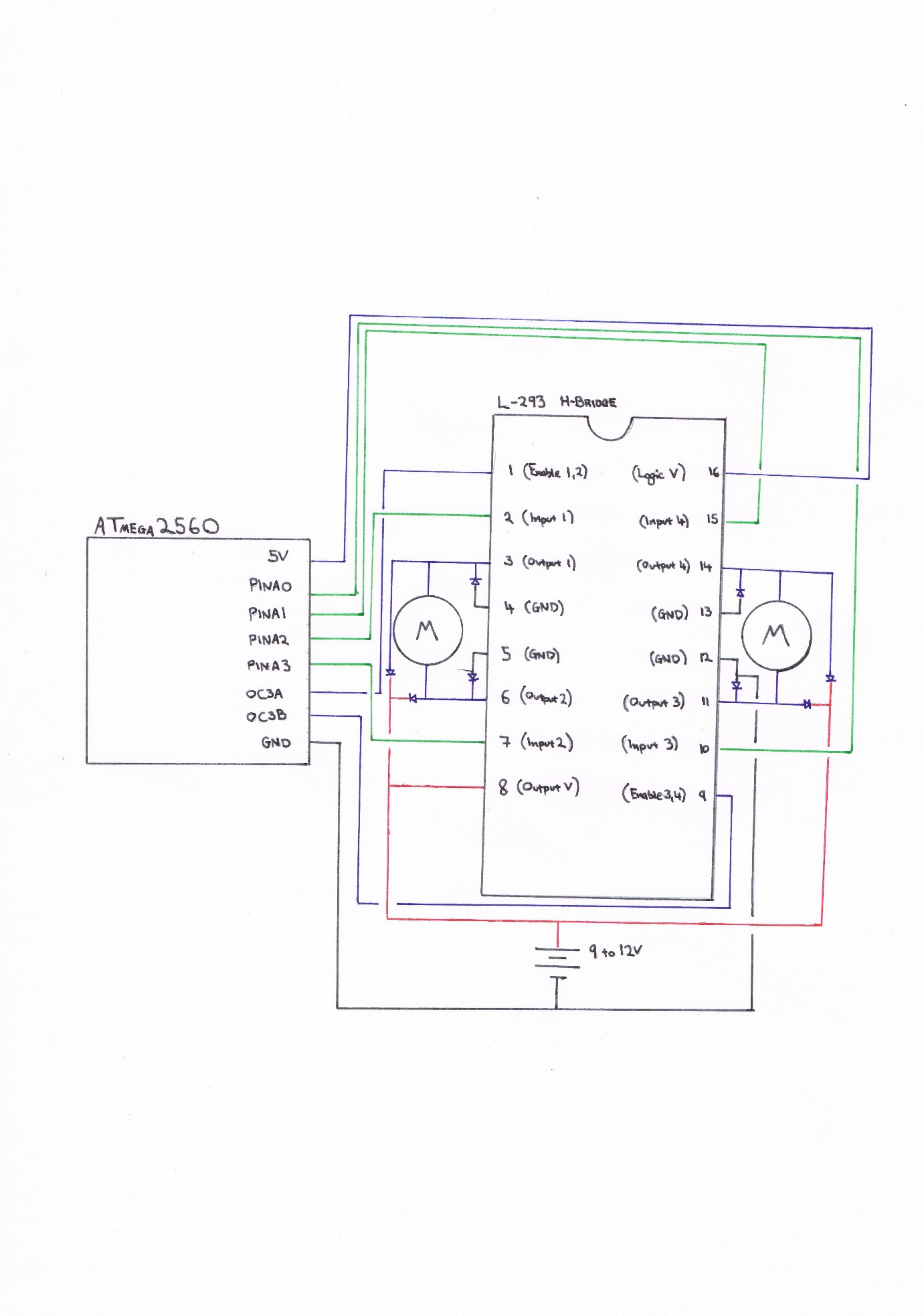
Since there was a significant amount of noise present, a circular buffer was used and a rolling average of the sensor readings was used to made decisions.

## SERVOMECHANISM

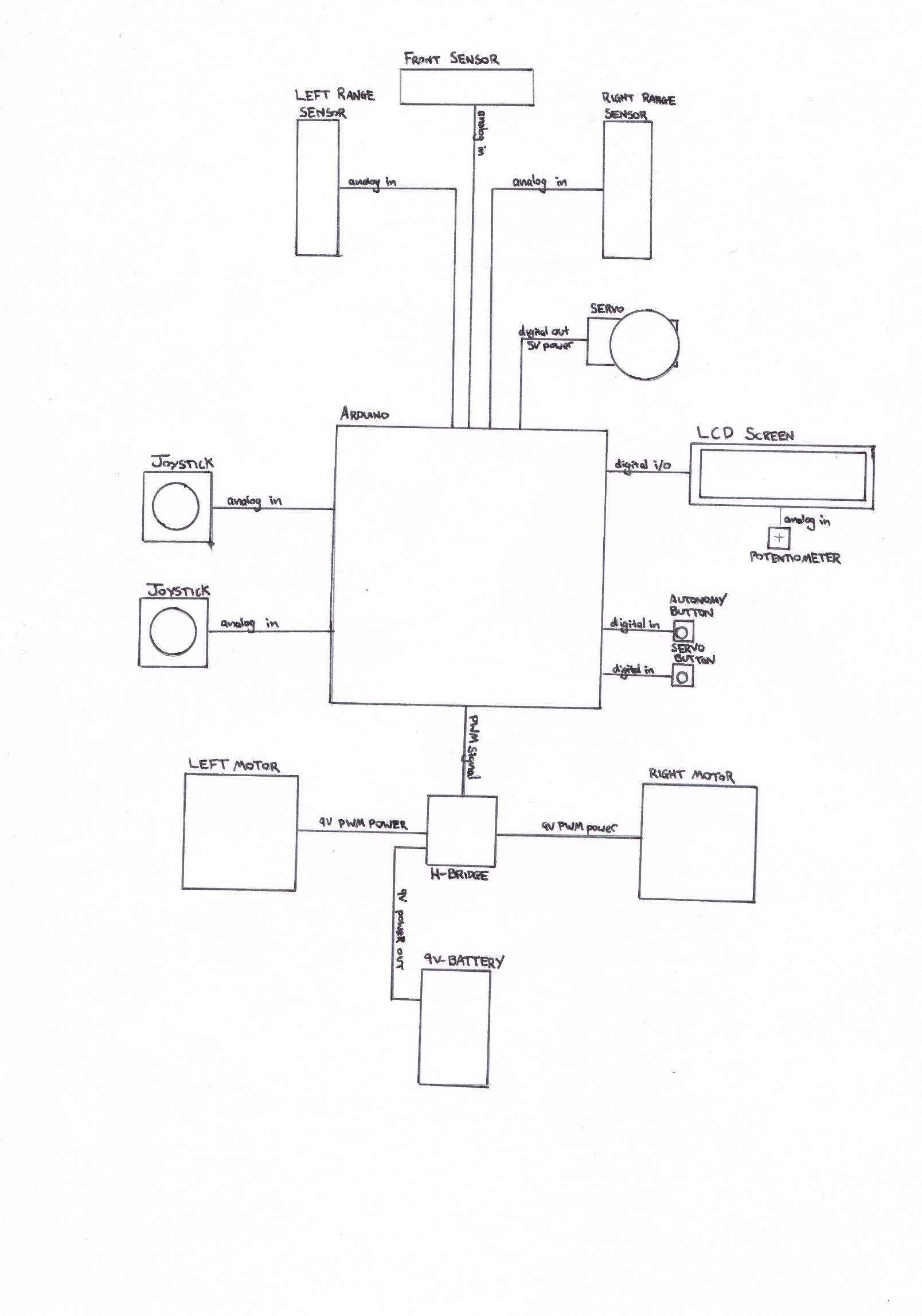
The servo used in the robot is a Futaba S3001. The base frequency used is 50Hz, correlating to a period of 20ms, and the pulse width is varied between 1ms and 2ms to set the position. The reading from the joystick was linearly interpolated to be evenly distributed in this range, allowing for the control of the position.



## H-BRIDGE

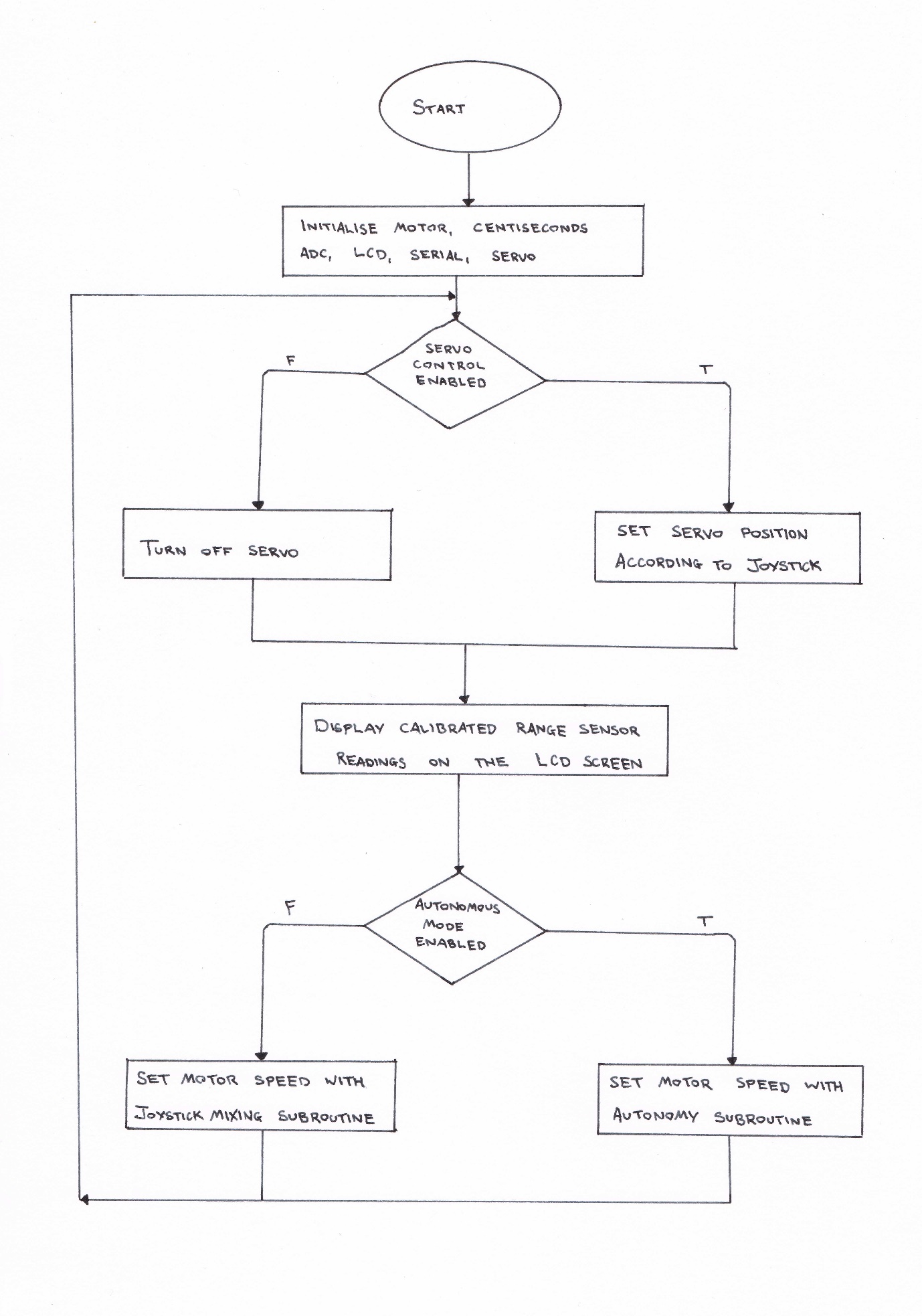
The H-Bridge used in the robot is an L-293. It allows for the control of both the motor direction and speed. The direction for one motor is controlled using the two input wires on the same side as the motor. Only one of these should be enabled at a time, or a short circuit will occur. In the program, this has been abstracted into a function call to prevent this from happening. The speed of the motor is then controlled using a Pulse Width Modulation (PWM) signal on the enable pin. A base frequency of 10kHz was used and the duty cycle is varied from 0% to 100%.

# SCHEMATIC DIAGRAM

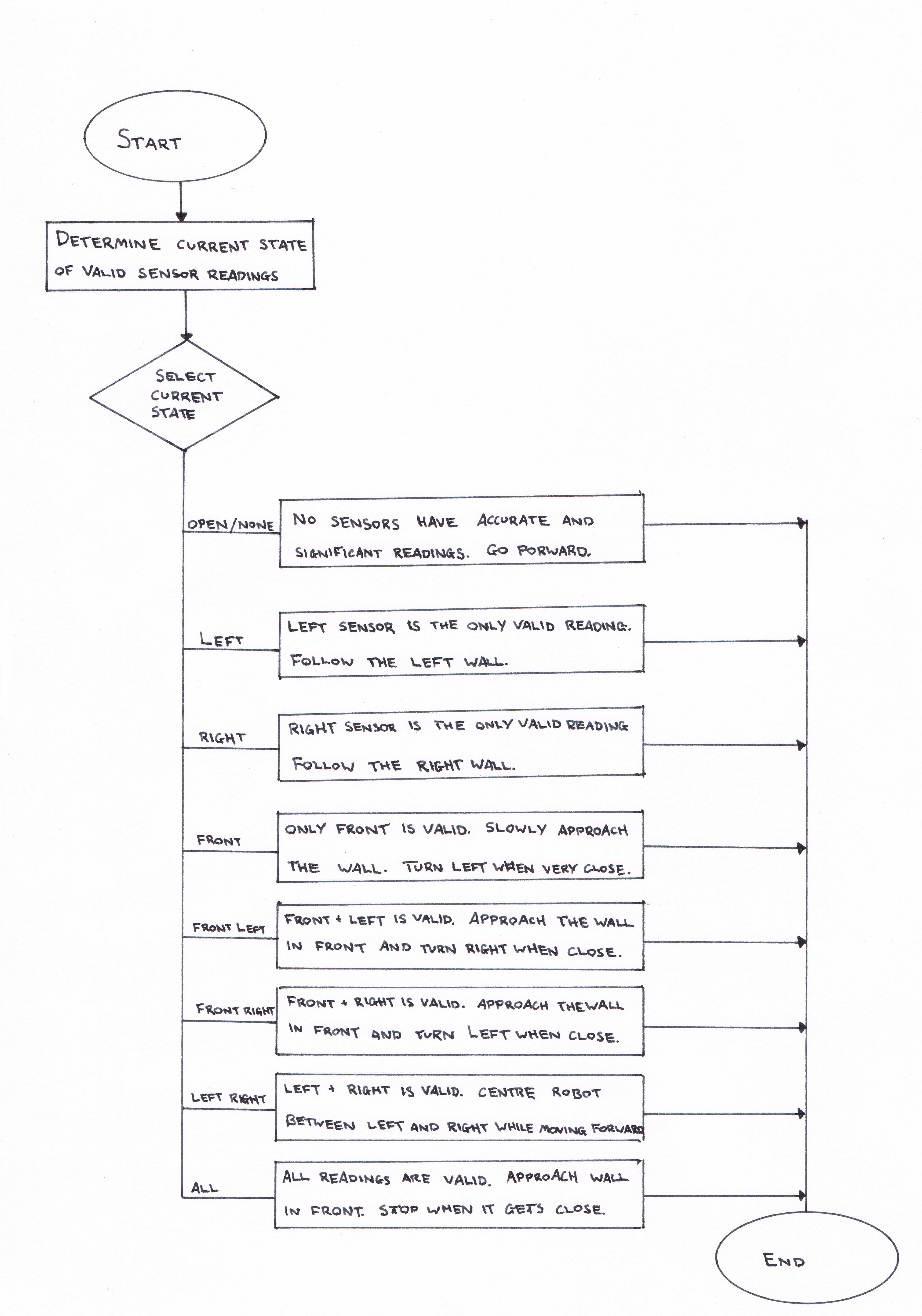


# FLOW CHARTS

## MAIN PROGRAM FLOW



## AUTONOMY



# CALCULATIONS

## PWM TOP VALUES

The servo and motor timers were configured to be in mode 10, which is a PWM phase correct mode. The timer counter has triangular waveform, hence the divide by 2 in the following formula, which was used to calculate the top value to set the base frequency.

For the motor, a prescaler of 1 was chosen and a frequency of 10kHz.

For the servo, a prescaler of 64 was chosen and a frequency of 50Hz

For the centisecond timer, CTC (Clear Timer on Compare) or mode 4 was selected. The timer counter has a ramp waveform, and so the TOP value for OCRnA was calculated using the following formula. A prescaler of 64 was selected and a frequency of 100Hz.

All of these TOP values are less than (as they are 16 bit timers) so they will not overflow. Additionally there is no truncation resulting from the integer division, so the times are all accurate.

# REFLECTIONS

The joystick mixing method currently used does not account for the region where the duty cycle of the motors is being set, but is not high enough to have any noticeable effect on the motor speed. The consequence of this is that there is a ‘dead-zone’ when moving the joystick that results in there being no movement of the robot. Remedying this would result in the user having a finer control over the speed of the wheels.

The autonomous method could be further refined to make it smoother when turning and following walls. Currently, it stops one of the wheels to orient itself, however it would be better to just slow the speed of the wheel.

# APPENDICES

## COMPLETE SOURCE CODE

### main.c

#include <avr/io.h>

#include <avr/interrupt.h>

#include <stdbool.h>

#include <stdlib.h>

#include <util/delay.h>

#include <stdio.h>

#include <math.h>

#include <inttypes.h>

#include "hd44780.h"

#include "serial.h"

#include "adc.h"

#include "servo.h"

#include "util.h" // squish, transform rolling\_average and a string buffer

#include "sensors.h" // transformations from adc to distance in mm

#include "motor.h" // defines L\_MOTOR, R\_MOTOR, FORWARD, BACKWARD and set\_motor\_dir

#include "autonomous.h" // sensors states enum and functions to print

#include "centiseconds.h" // global centisecond timer using TIMER1

#define MOVING\_AVG\_POINTS 30

// Distance from sensor to edge of robot in mm

const int16\_t CENTER\_OFFSET = 87;

const int16\_t SIDE\_OFFSET = 37;

volatile bool interrupt0 = true;

ISR(INT0\_vect)

{

// static because we want it to keep its state.

static volatile uint32\_t INT0\_invalid\_to = 0;

// Avoid reading a volatile multiple times

uint32\_t centi = centiseconds;

**if**(INT0\_invalid\_to < centi)

{

interrupt0 = !interrupt0; // toggle the interrupt0 boolean

INT0\_invalid\_to = centi + 30; // debounce for 0.3s

}

}

volatile bool interrupt1 = true;

ISR(INT1\_vect)

{

// static because we want it to keep its state.

static volatile uint32\_t INT1\_invalid\_to = 0;

// Avoid reading a volatile multiple times

uint32\_t centi = centiseconds;

**if**(INT1\_invalid\_to < centi)

{

interrupt1 = !interrupt1; // toggle the interrupt1 boolean

INT1\_invalid\_to = centi + 30; // debounce for 0.3s

}

}

void manual\_tick()

{

int16\_t adc0 = adc\_read(11) - (1024/2); // adc0 = [-512, 511]

adc0 \*= 1;

int16\_t adc1 = adc\_read(12) - (1024/2); // adc1 = [-512, 511]

adc1 \*= -1;

transform(&adc0, &adc1); // adc{0,1} = [-512, 512], but rotated 45 deg and scaled appropriately

set\_motor\_dir(L\_MOTOR, adc0 < 0); // set motor direction to the sign of adc0

set\_motor\_dir(R\_MOTOR, adc1 < 0); // set motor direction to the sign of adc1

adc0 = abs(adc0); // Now we can make them positive

adc1 = abs(adc1); // as we&apos;ve extracted the sign bit.

// transform the range [0, 512] into [0, MOTOR\_PWM\_PERIOD]

adc0 = squish(adc0, 0.0f, 512.0f, 0.0f, MOTOR\_PWM\_PERIOD);

adc1 = squish(adc1, 0.0f, 512.0f, 0.0f, MOTOR\_PWM\_PERIOD);

L\_MOTOR\_SPEED = adc0;

R\_MOTOR\_SPEED = adc1;

}

int main(void)

{

///// BUTTON INTERRUPT ///////

EICRA |= (1<<ISC11);

EICRA &= ~(1<<ISC10);

EIMSK |= (1<<INT1);

EICRA |= (1<<ISC01);

EICRA &= ~(1<<ISC00);

EIMSK |= (1<<INT0);

DDRD = 0b00;

PORTD = 0b11;

motor\_init(); // Sets up pins for controlling the H-Bridge and PWM (TIMER3)

centiseconds\_init(); // Global timer named &apos;centiseconds&apos; (Using TIMER4)

adc\_init(); // Initialise the analog to digital converter

lcd\_init(); // For displaying sensor readings on the LCD screen

serial0\_init(); // For writing to the serial monitor.

servo\_init(); // Sets up pins for controlling the servo motor (Using TIMER1)

sei(); // Globally enable interrupts

**while**(1)

{

**if**(interrupt1)

{

static int64\_t invalid\_to = 0;

int64\_t centi = centiseconds;

**if**(centi > invalid\_to)

{

servo\_pos(squish(adc\_read(14), 0, 1024, 0, 180));

invalid\_to = centi + 10;

}

}

**else**

{

servo\_off();

}

// Set joystick position

// Variables for keeping track of the rolling average of the 3 range sensors.

// Static because we want them to maintain their state.

static int64\_t l\_prev\_sum = 0, f\_prev\_sum = 0, r\_prev\_sum = 0;

static int16\_t l\_prev\_i = 0, f\_prev\_i = 0, r\_prev\_i = 0;

static int16\_t l\_points[MOVING\_AVG\_POINTS], f\_points[MOVING\_AVG\_POINTS], r\_points[MOVING\_AVG\_POINTS];

// Read data from ADC and convert it to mm.

// Translate it so that it is distance from the edge of the robot.

int16\_t left\_sensor = side\_to\_mm( adc\_read(0)) - SIDE\_OFFSET;

int16\_t front\_sensor = front\_to\_mm(adc\_read(1)) - CENTER\_OFFSET;

int16\_t right\_sensor = side\_to\_mm( adc\_read(2)) - SIDE\_OFFSET;

// Add the reading to the rolling average circular buffer and reassign left\_sensor to be a de-noised value

left\_sensor = rolling\_average(left\_sensor, &l\_prev\_sum, &l\_prev\_i, MOVING\_AVG\_POINTS, l\_points);

right\_sensor = rolling\_average(right\_sensor, &r\_prev\_sum, &r\_prev\_i, MOVING\_AVG\_POINTS, r\_points);

front\_sensor = rolling\_average(front\_sensor, &f\_prev\_sum, &f\_prev\_i, MOVING\_AVG\_POINTS, f\_points);

static int64\_t update\_invalid\_to = 0; // static to retain state

**if**(centiseconds > update\_invalid\_to) // only update the display every 4 seconds

{

lcd\_goto(0);

sprintf(buffer, "L:%4dmmR:%4dmm", left\_sensor, right\_sensor);

lcd\_puts(buffer);

lcd\_goto(0x40);

sprintf(buffer, "C: %6dmm", front\_sensor);

lcd\_puts(buffer);

update\_invalid\_to = centiseconds + 25;

}

**if**(interrupt0) // manual mode

{

manual\_tick();

}

**else** // autonomous mode

{

autonomous\_tick(left\_sensor, front\_sensor, right\_sensor);

}

}

}

### servo.h

#ifndef SERVO\_H

#define SERVO\_H

void servo\_init();

void servo\_pos(int16\_t angle\_deg);

void servo\_off();

### servo.c

#include <avr/io.h>

#include "servo.h"

#define SERVO\_PWM\_PERIOD ((int16\_t)(16000000LL / 50 / 64 / 2))

#define SERVO\_COMPARE OCR1B

#define SERVO\_MIN (SERVO\_PWM\_PERIOD / 20)

#define SERVO\_MAX (SERVO\_PWM\_PERIOD / 10)

void servo\_init()

{

TCCR1A = 0b00100010; // OCR4A Outputting + Mode 10

TCCR1B = 0b00010011; // 64 Prescaler + Mode 10

ICR1 = SERVO\_PWM\_PERIOD; // set the top value

DDRB |= 0b1000000; // set bit 6 of port B

servo\_pos(90);

}

void servo\_pos(int16\_t angle\_deg)

{

// clamp angle between 0 and 180

**if**(angle\_deg < 0)

angle\_deg = 0;

**else** **if**(angle\_deg > 180)

angle\_deg = 180;

SERVO\_COMPARE = SERVO\_MIN + (SERVO\_MAX - SERVO\_MIN) \* angle\_deg / 180;

}

void servo\_off()

{

SERVO\_COMPARE = 0;

}

### sensors.h

#ifndef SENSORS\_H

#define SENSORS\_H

#include <inttypes.h>

int16\_t front\_to\_mm(int16\_t adc);

int16\_t side\_to\_mm(int16\_t adc);

#endif

### sensors.c

#include "sensors.h"

int16\_t front\_to\_mm(int16\_t adc)

{

// Pretty much just magic numbers. Formula comes from the curve of best fit for the

// sensor readings at different distances. Check sensor data for justification.

**return** -(53227500ll\*adc-268558095607ll)/(2000\*(2500ll\*adc-78227));

}

int16\_t side\_to\_mm(int16\_t adc)

{

// Same as front\_to\_mm, formula comes from a curve of best fit made using data from

// sensors readings. See data for justification.

**return** -(61377500ll\*adc - 110758264243ll)/(1250ll\*(2500ll\*adc+68970));

}

### util.h

#ifndef UTIL\_H

#define UTIL\_H

#include <stdlib.h>

#include <inttypes.h>

#define ROOT2 1.4142135623f

extern char buffer[64]; // Large buffer that can be used for formatting strings before writing to serial.

int16\_t squish(int16\_t value, float current\_min, float current\_max, float new\_min, float new\_max);

void transform(int16\_t\* restrict int\_a, int16\_t\* restrict int\_b);

int16\_t rolling\_average(int16\_t new\_point, int64\_t\* prev\_sum, int16\_t\* prev\_index,size\_t point\_count, int16\_t points[]);

#endif

### util.c

#include "util.h"

#include <math.h>

char buffer[64]; // Large buffer that can be used for formatting strings before writing to serial.

// Please note that this function only takes 16 bit integers, while the timer

// needs a 16 bit UNSIGNED integer. If you change the frequency so that it

// requires a greater timer range, you should change these parameters

int16\_t squish(int16\_t value, float current\_min, float current\_max, float new\_min, float new\_max)

{

// the line passing through the points (current\_min, new\_min) and (current\_max, new\_max) can be

// expressed generally as y=mx+c where x is the current value and y is the new value. In matrix

// form this could be expressed in the equation [1, current\_min; 1, current\_max][c; m] = [new\_min, new\_max].

// Multiplying by the inverse gives

// c = (current\_min\*new\_max - current\_max \* new\_min)/(current\_min - current\_max)

// m = (new\_min - new\_max)/(current\_min - current\_max)

// Substituting these into the formula y=mx+c, we can easily find the new value y

float c = (current\_min\*new\_max - current\_max\*new\_min)/(current\_min - current\_max);

float m = (new\_min - new\_max)/(current\_min - current\_max);

**return** m\*value + c;

}

// Takes a pair of integers int\_a + int\_b\*I and rotates

// them 45 degrees. It then scales the magnitude along the radius

// to make an output that, for a square input, will give a square output.

void transform(int16\_t\* restrict int\_a, int16\_t\* restrict int\_b)

{

// convert int\_a and int\_b to be floating point

// numbers original a and original b

const float oa = \*int\_a, ob = \*int\_b;

// ROTATE 45 DEGREES

// a + b\*i = (root2/2)\*(1 + I) \* (oa + ob\*I)

// Rotate the inputted value by 45 degrees (clockwise)

// and store the result in the complex number a + b\*I

float a = (ROOT2 / 2.0f)\*(oa - ob);

float b = (ROOT2 / 2.0f)\*(oa + ob);

// CALCULATE THE SCALE FACTOR.

float sf; // scale factor;

// x & y are only used for solution simplification.

// First Make a complex number x + y\*I that is in the first octant 0 <= arg <= pi/4

// fabs() makes the point in the first quadrant.

float x = fabs(a), y = fabs(b);

// We then reflect if it is in the second octant

// This allows us to exploit symmetry to only have to solve for 1 case.

**if**( y > x ) { float temp = x; x = y; y = temp; }

// set the scale factor (unless x is sufficiently small, to avoid divide by 0).

sf = ( x < 0.0001 ? 1.0f : sqrt( (x+y)\*(x+y)/(2\*x\*x) ) );

// see ./Transformation/transformation equations.png for justification of formula

// SCALE

a \*= sf; // scale a by the scale factor

b \*= sf; // scale b by the scale factor

// ASSIGN

\*int\_a = a; // assign int\_a its new value

\*int\_b = b; // assign int\_b its new value

**return**;

}

int16\_t rolling\_average(int16\_t new\_point, int64\_t\* prev\_sum, int16\_t\* prev\_index, size\_t point\_count, int16\_t points[])

{

// &apos;points&apos; is a circular buffer to keep track of the last &apos;point\_count&apos; readings.

// When a value is removed from the buffer, it is subtracted from the sum. When

// it is added to the buffer it is added to the sum. This means that we don&apos;t need

// to sum the whole array each time.

\*prev\_sum -= points[\*prev\_index];

\*prev\_sum += new\_point;

points[\*prev\_index] = new\_point;

(\*prev\_index) = (\*prev\_index + 1) % point\_count;

**return** \*prev\_sum / point\_count;

}

### motor.h

#ifndef MOTOR\_H

#define MOTOR\_H

#include <stdbool.h>

#include <inttypes.h>

#define L\_MOTOR 0

#define R\_MOTOR 2

#define L\_MOTOR\_SPEED OCR3B

#define R\_MOTOR\_SPEED OCR3A

#define FORWARD false

#define BACKWARD true

#define FULL\_SPEED 512 // Since duty cycle isn&apos;t proportional to the speed

#define HALF\_SPEED 448 // of the motors, these defines can be used to

#define SLOW\_SPEED 384 // set the speed more intuitively.

#define STOP\_SPEED 0

// 10kHz base frequency

#define MOTOR\_PWM\_PERIOD ((int16\_t)(16000000UL / 1 / 10000 / 2))

void motor\_init(); // Uses TIMER3 for pwm

int set\_motor\_dir(int lsb, bool forward);

#endif

### motor.c

#include "motor.h"

#include <avr/io.h>

void motor\_init() // Uses TIMER3 for pwm

{

// Configure timer3 to be in mode 10 (spread across TCCR3{A,B})

TCCR3A = 0b10100010; // OC3A & OC3B Outputting

TCCR3B = 0b00010001; // 1 Prescaler

ICR3 = MOTOR\_PWM\_PERIOD; // Set the top value for timer3

DDRE |= 0b00011000; // PWM pins are outputting (cant clobber the whole port like with TCCRA{A..B})

R\_MOTOR\_SPEED = (uint16\_t)MOTOR\_PWM\_PERIOD \* 8 / 10; // Initially 80% duty cycle

L\_MOTOR\_SPEED = (uint16\_t)MOTOR\_PWM\_PERIOD \* 2 / 3; // Initially 66% duty cycle

////////// MOTOR CONTROL PORT A SETUP //////////

DDRA |= 0b00001111; // Set the motor control pins to output

PORTA &= ~0b00001010; // Turn OFF these bits

PORTA |= 0b00000101; // Turn ON these bits

// We can only have one in a pair set a time.

// Use set\_motor\_dir(L\_MOTOR, FORWARD) in code to set the direction.

}

int set\_motor\_dir(int lsb, bool forward)

{

PORTA &= ~(0b11 << lsb); // Turn the motor off

**if**(forward)

PORTA |= (0b01 << lsb);

**else**

PORTA |= (0b10 << lsb);

}

### centiseconds.h

#ifndef CENTISECONDS\_H

#define CENTISECONDS\_H

extern volatile uint32\_t centiseconds;

void centiseconds\_init();

#endif

### centiseconds.c

#include <inttypes.h>

#include <avr/interrupt.h>

#include <avr/io.h>

volatile uint32\_t centiseconds = 0;

// Timer tick. Increment centiseconds by 1.

ISR(TIMER4\_COMPA\_vect)

{

centiseconds += 1;

}

void centiseconds\_init()

{

TCCR4A = 0x0; // no output

TCCR4B |= 0b1011; // CTC, 64 prescaler

OCR4A = 16000000L/64L/100; // When reached, 1cs has passed

// Clear timer on compare for Output Compare A Match Register (OCR4A)

TIMSK4 = (1<<OCIE4A);

}

### autonomous.h

#ifndef AUTONOMOUS\_H

#define AUTONOMOUS\_H

#include <inttypes.h>

enum sensor\_state

{

OPEN\_FORWARD = 0b000, // nothing anywhere. Just send it

FOLLOW\_LEFT = 0b100, // Something on the left, nothing in front or on the right

FOLLOW\_RIGHT = 0b001, // Something on the right, nothing in front or to the left

APPROACH\_FRONT = 0b010, // Only something in front. Approach carefully and then turn left.

CAUTION\_FRONT\_LEFT = 0b110, // Something in front and to the left. Start turning right

CAUTION\_FRONT\_RIGHT = 0b011, // Something in front and to the right. Start turning left.

CAUTION\_LEFT\_RIGHT = 0b101, // Something on either side of the robot. Stay in the center

DEAD\_END = 0b111 // Surrounded on all sides.

};

void print\_state(enum sensor\_state state);

void follow\_left(int16\_t\* left\_speed, int16\_t\* right\_speed, int16\_t left\_sensor, int16\_t right\_sensor);

void caution\_front\_left(int16\_t\* left\_speed, int16\_t\* right\_speed,

int16\_t left\_sensor, int16\_t front\_sensor, int16\_t right\_sensor);

void autonomous\_tick(int16\_t left\_sensor, int16\_t front\_sensor, int16\_t right\_sensor);

#endif

### autonomous.c

#include <avr/io.h>

#include <stdio.h>

#include "autonomous.h"

#include "util.h"

#include "centiseconds.h"

#include "motor.h"

#include "serial.h"

#define SIGNIFICANT\_READING\_MM 300

#define SIGNIFICANT\_READING\_MM\_SIDE 300

#define FRONT\_CRITICAL\_DIST 160

#define SIDE\_CRITICAL\_DIST 80

#define TOLERANCE 30

// Used for debugging.

void print\_state(enum sensor\_state state)

{

**switch**(state)

{

**case** OPEN\_FORWARD:

serial0\_print\_string("OPEN\_FORWARD\n"); **break**;

**case** FOLLOW\_LEFT:

serial0\_print\_string("FOLLOW\_LEFT\n"); **break**;

**case** FOLLOW\_RIGHT:

serial0\_print\_string("FOLLOW\_RIGHT\n"); **break**;

**case** APPROACH\_FRONT:

serial0\_print\_string("APPROACH\_FRONT\n"); **break**;

**case** CAUTION\_FRONT\_LEFT:

serial0\_print\_string("CAUTION\_FRONT\_LEFT\n"); **break**;

**case** CAUTION\_FRONT\_RIGHT:

serial0\_print\_string("CAUTION\_FRONT\_RIGHT\n"); **break**;

**case** CAUTION\_LEFT\_RIGHT:

serial0\_print\_string("CAUTION\_LEFT\_RIGHT\n"); **break**;

**case** DEAD\_END:

serial0\_print\_string("DEAD\_END\n"); **break**;

**default**:

serial0\_print\_string("default\n"); **break**;

}

}

void follow\_left(int16\_t\* left\_speed, int16\_t\* right\_speed, int16\_t left\_sensor, int16\_t right\_sensor)

{

\*left\_speed = \*right\_speed = FULL\_SPEED;

**if**(left\_sensor < SIDE\_CRITICAL\_DIST) // if we are a bit close, stop the right wheel

\*right\_speed = STOP\_SPEED;

**else** **if**(left\_sensor > SIDE\_CRITICAL\_DIST + TOLERANCE) // if we are a bit far, stop the left wheel

\*left\_speed = STOP\_SPEED;

}

void caution\_front\_left(int16\_t\* left\_speed, int16\_t\* right\_speed, int16\_t left\_sensor, int16\_t front\_sensor, int16\_t right\_sensor)

{

**if**(front\_sensor < (FRONT\_CRITICAL\_DIST + TOLERANCE) || left\_sensor < (SIDE\_CRITICAL\_DIST + TOLERANCE))

{

\*left\_speed = \*right\_speed = FULL\_SPEED;

**if**(right\_sensor < SIDE\_CRITICAL\_DIST)

\*left\_speed = STOP\_SPEED;

**else** **if**(right\_sensor > SIDE\_CRITICAL\_DIST + TOLERANCE)

\*right\_speed = STOP\_SPEED;

}

**else**

{

\*left\_speed = FULL\_SPEED;

\*right\_speed = STOP\_SPEED;

}

}

void autonomous\_tick(int16\_t left\_sensor, int16\_t front\_sensor, int16\_t right\_sensor)

{

static enum sensor\_state current\_state, old\_state;

int16\_t left\_speed = FULL\_SPEED;

int16\_t right\_speed = FULL\_SPEED;

bool left\_significant = left\_sensor < SIGNIFICANT\_READING\_MM\_SIDE;

bool right\_significant = right\_sensor < SIGNIFICANT\_READING\_MM\_SIDE;

bool front\_significant = front\_sensor < SIGNIFICANT\_READING\_MM;

static int64\_t state\_invalid\_to = 0;

**if**(centiseconds > state\_invalid\_to) // update current state 4times a second

{

current\_state = (left\_significant << 2) | (front\_significant << 1) | (right\_significant << 0);

**if**(current\_state != old\_state)

{

print\_state(current\_state);

}

state\_invalid\_to = centiseconds + 25;

}

**switch**(current\_state)

{

**case** OPEN\_FORWARD: // there is nothing in the way

left\_speed = right\_speed = FULL\_SPEED;

**break**;

**case** FOLLOW\_LEFT: // we want to follow the left wall

follow\_left(&left\_speed, &right\_speed, left\_sensor, right\_sensor);

**break**;

**case** FOLLOW\_RIGHT: // following right is the same as left, but reflected

follow\_left(&right\_speed, &left\_speed, right\_sensor, left\_sensor);

**break**;

**case** APPROACH\_FRONT:

**if**(front\_sensor < FRONT\_CRITICAL\_DIST)

{

**if**(right\_sensor > SIDE\_CRITICAL\_DIST \* 3 && left\_sensor > SIDE\_CRITICAL\_DIST \* 3)

{

left\_speed = STOP\_SPEED;

right\_speed = FULL\_SPEED;

}

**else**

{

**if**(left\_sensor>right\_sensor + TOLERANCE)

{

left\_speed = STOP\_SPEED;

right\_speed = FULL\_SPEED;

}

**else**

{

left\_speed = STOP\_SPEED;

right\_speed = FULL\_SPEED;

}

}

}

**else**

{

left\_speed = right\_speed = HALF\_SPEED;

}

**break**;

**case** CAUTION\_FRONT\_LEFT:

// There is something to the front and left, so we must be careful as we get closer

// caution\_front\_left(&left\_speed, &right\_speed, left\_sensor, front\_sensor, right\_sensor);

**if**(front\_sensor < (FRONT\_CRITICAL\_DIST + TOLERANCE) || left\_sensor < (SIDE\_CRITICAL\_DIST + TOLERANCE))

{

left\_speed = FULL\_SPEED;

right\_speed = FULL\_SPEED;

**if**(right\_sensor < SIDE\_CRITICAL\_DIST)

left\_speed = STOP\_SPEED;

**else** **if**(right\_sensor > SIDE\_CRITICAL\_DIST + TOLERANCE)

right\_speed = STOP\_SPEED;

}

**else**

{

//left\_speed /= front\_sensor < left\_sensor ? front\_sensor - FRONT\_CRITICAL\_DIST : left\_sensor - FRONT\_CRITICAL\_DIST;

left\_speed = FULL\_SPEED;

right\_speed = left\_speed / 2;

}

**break**;

**case** CAUTION\_FRONT\_RIGHT:

// same as CAUTION\_FRONT\_LEFT, but reflected

// caution\_front\_left(&right\_speed, &left\_speed, right\_sensor, front\_sensor, left\_sensor);

**if**(front\_sensor < (FRONT\_CRITICAL\_DIST + TOLERANCE) || right\_sensor < (SIDE\_CRITICAL\_DIST+TOLERANCE) )

{

left\_speed = FULL\_SPEED;

right\_speed = FULL\_SPEED;

**if**(left\_sensor < SIDE\_CRITICAL\_DIST)

right\_speed = STOP\_SPEED;

**else** **if**(left\_sensor > SIDE\_CRITICAL\_DIST + TOLERANCE)

left\_speed = STOP\_SPEED;

}

**else**

{

//right\_speed /= front\_sensor < right\_sensor ? front\_sensor - FRONT\_CRITICAL\_DIST : right\_sensor - FRONT\_CRITICAL\_DIST;

right\_speed = FULL\_SPEED;

}

left\_speed = right\_speed / 2;

**break**;

**case** CAUTION\_LEFT\_RIGHT:

**if**(left\_sensor > right\_sensor + TOLERANCE)

{

left\_speed = STOP\_SPEED;

right\_speed = FULL\_SPEED;

}

**else** **if** (right\_sensor > left\_sensor + TOLERANCE)

{

right\_speed = STOP\_SPEED;

left\_speed = FULL\_SPEED;

}

**else**

{

left\_speed = right\_speed = FULL\_SPEED;

}

**break**;

**case** DEAD\_END:

set\_motor\_dir(L\_MOTOR, BACKWARD);

set\_motor\_dir(R\_MOTOR, BACKWARD);

L\_MOTOR\_SPEED = squish(left\_speed, 0.0f, FULL\_SPEED, 0.0f, MOTOR\_PWM\_PERIOD);

R\_MOTOR\_SPEED = squish(right\_speed, 0.0f, FULL\_SPEED, 0.0f, MOTOR\_PWM\_PERIOD);

**break**;

}

set\_motor\_dir(L\_MOTOR, FORWARD);

set\_motor\_dir(R\_MOTOR, FORWARD);

L\_MOTOR\_SPEED = squish(left\_speed, 0.0f, FULL\_SPEED, 0.0f, MOTOR\_PWM\_PERIOD);

R\_MOTOR\_SPEED = squish(right\_speed, 0.0f, FULL\_SPEED, 0.0f, MOTOR\_PWM\_PERIOD);

static int64\_t speed\_invalid\_to = 0;

old\_state = current\_state;

**if**(centiseconds > speed\_invalid\_to) // update current state 4times a second

{

sprintf(buffer, "l: %4d, r: %4d\n", left\_speed, right\_speed);

serial0\_print\_string(buffer);

speed\_invalid\_to = centiseconds + 25;

}

}