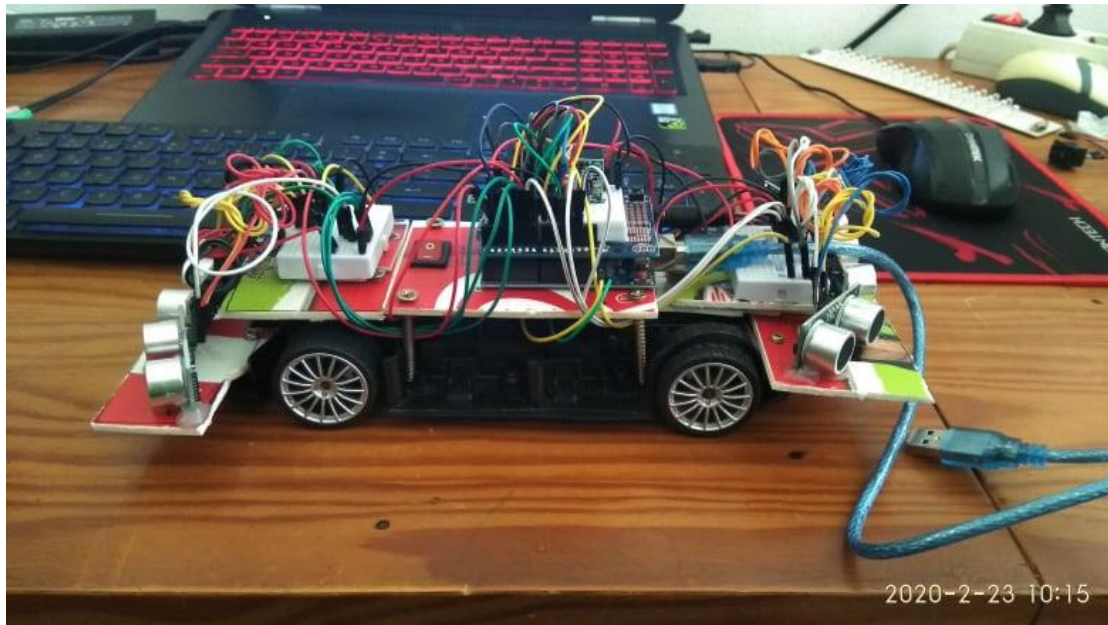


ad



Relatório Carro Autónomo

PROJETO DE FÍSICA APLICADA Á COMPUTAÇÃO

Vasco Gomes | Física Aplicada á Computação | 09/03/2020

Índice

Introdução	3
A base do projeto	4
O Arduino	10
Os sensores.....	11
A ponte h	15
O giroscópio	19
O código	30
Defines	30
Declaração de Pinos	31
protótipos	32
função setup	33
delclaração de variáveis globais.....	34
função loop.....	35
função decisionmaker()	36
função middlesensorfront()	39
função speedcontroller().....	41
função goleft()	42
função goright()	43
função goforward() e goback().....	43
função stopEngine()	44
função stopturnigleft() e stopturningright()	45
função returnRoute()	45
função calculateError()	46
função setup_mpu_6050_registers()	47
função read_mpu_6050_data().....	48
função showgirodata()	49
Conclusão	52
Webgrafia	53

Introdução

Este trabalho, tem o propósito de demonstrar os princípios da física, num projeto.

Eu decidi realizar um projeto, pois embora já tivesse trabalhado com Arduino no passado, nunca tinha realizado nenhum projeto utilizando essa base, e sabendo agora aquilo que dá para fazer com o Arduino, fiquei interessado, em realizar o projeto, até porque tinha curiosidade se daria para fazer ou não.

Fiquei curioso e quis aprender mais, do que propriamente o que demos nas aulas teórico práticas, por isso propus-me a pelo menos fazer um carro autónomo, que tem por base um carro telecomandado, cuja board não funcionava, mas os motores sim, e utilizar o Arduino, assim como variados sensores que vão ser mostrados no decorrer do relatório.

A base do projeto

Para começar, este projeto foi desenvolvido numa base de um carro telecomandado, em que a *board*, deixou de trabalhar, mas os motores ainda funcionavam, por isso decidi aproveitar e fazer com essa base, tal como mostra a figura.

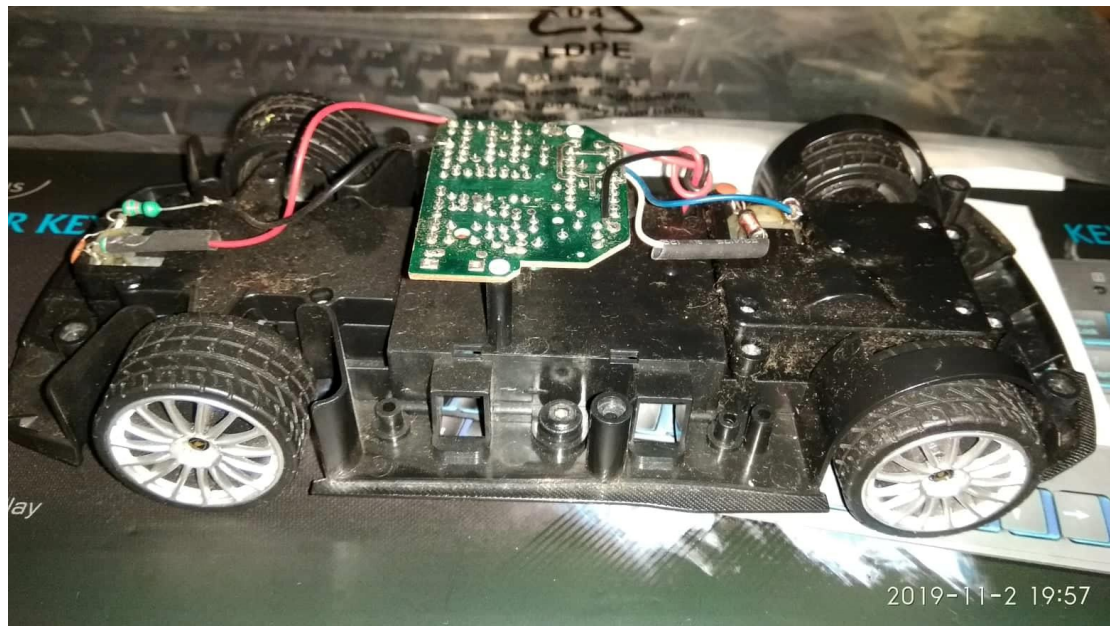


Figura 1 - Base do projeto do carro autónomo

Na figura acima, podemos visualizar como era o *chassis* do carro antes de lhe começar a colocar o Arduino por cima.

No início do projeto, eu achei que poderia alimentar o carro e o Arduino com a bateria original do carro, que era de apenas 4,5 volts, e que mais tarde, percebi que necessitava de mais energia para o projeto, pois esta bateria mal alimentava o Arduino, até porque se formos a ver bem o Arduino recebe de uma porta *USB* normal 5,5 volts, logo 4,5 volts não iriam chegar para alimentar os motores.

Após ter-me apercebido disso e o professor ter chamado à atenção para esse facto, decidimos que o melhor era ligar o Arduino a uma pilha de 9 volts, e os motores também, mas para que isso funcionasse estas tinham que ficar montadas em paralelo, quer isto dizer que em vez de aumentar a voltagem, como acontece quando se ligam 2 pilhas em série, aumenta aos amperes, ou seja basicamente aumenta a capacidade.

Como podemos ver na seguinte figura:



Figura 2 - 2 Pilhas de 9 volts ligadas em paralelo

Ver vídeo relativo a ligar o Arduino com a bateria original do carro.

Após ter a base para o projeto, era hora de começar a montar o Arduino em cima do carro, e para isso, decidi fazer moldes em cartão para saber como ficaria o Arduino lá posicionado, e que depois iriam ser trocados por pequenas placas de PVC, que seriam futuramente aparafusadas á base do carro, e seguras com mais 4 parafusos para adicionar estabilidade ao projeto em si.

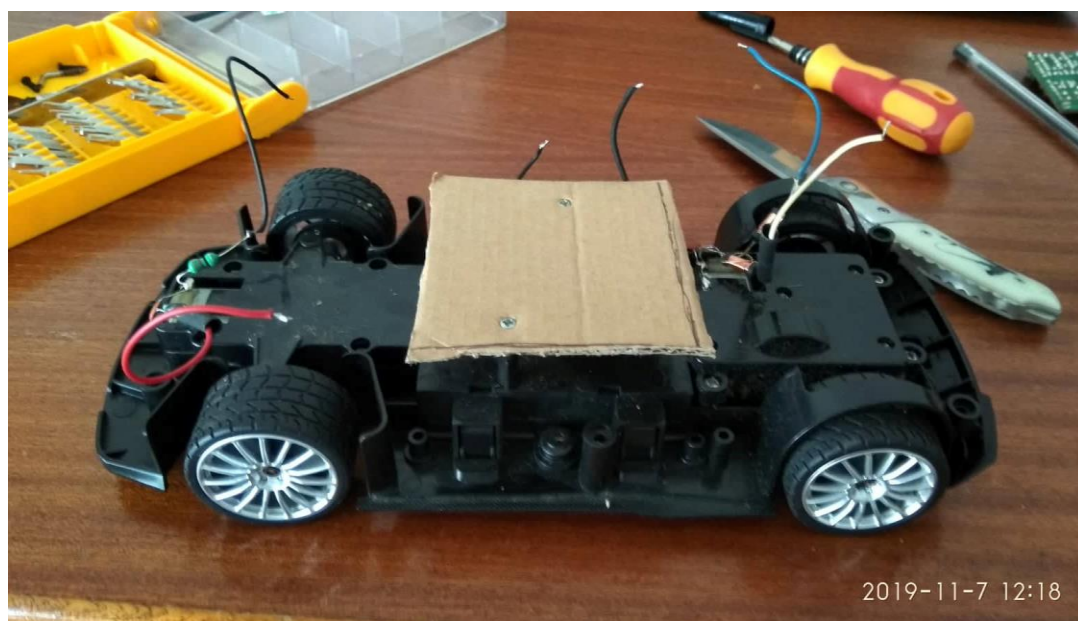


Figura 3 - Base experimental de cartão

Após ter concluído como ficaria, passei para uma placa de *PVC*, e comecei a fazer as soldas para a alimentação do Arduino, em que o professor deu um cabo de ligação de pilhas de 9 volts para o Arduino já cortado que eu poderia descarnar e utilizar no meu projeto como se pode observar na seguinte figura.

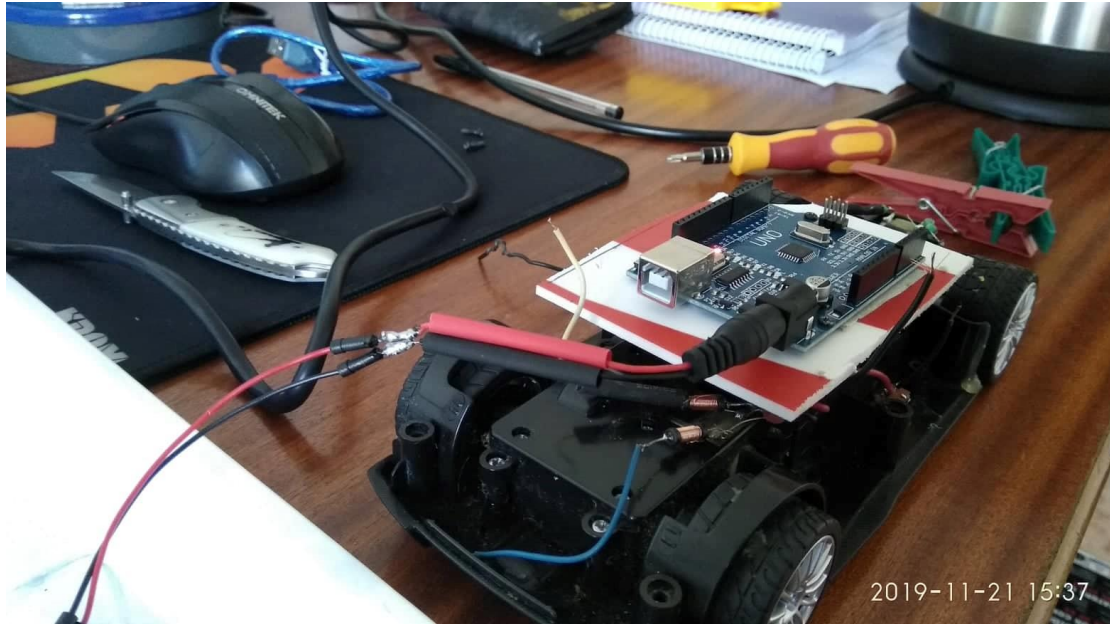


Figura 4 - Base em PVC

Após ter posicionado e testado o Arduino no carro, tentei perceber onde encaixar todo o resto do material que iria utilizar, assim como os sensores de ultrassons, a ponte h, e o giroscópio.

A primeira coisa que decidi colocar a funcionar no carro, foi a *ponte h*, pois se havia coisa que queria testar desde que comecei o projeto, foi se conseguia, e como conseguia controlar os motores elétricos do carro de modo a que ele se movimentasse, escusado será dizer que na 1ª vez que o meti a andar, bateu na parede, porque calculei mal o tempo que demoraria a chegar até lá.

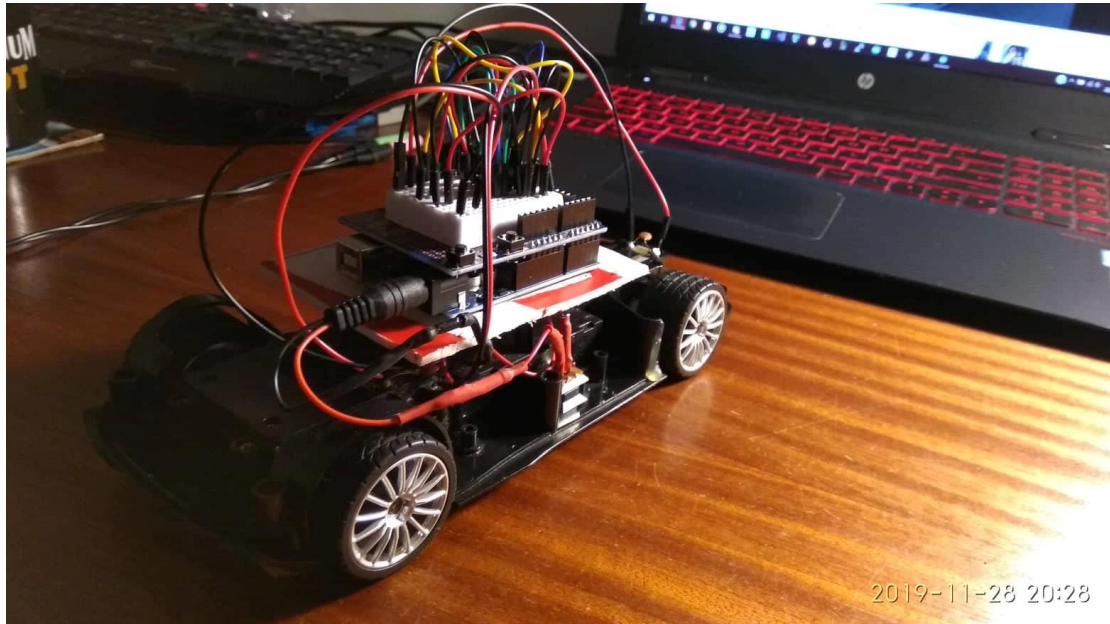


Figura 5 - Ponte h, no carro autónomo

Depois, passei para a implementação dos sensores, vale sempre a pena salientar, que uma vez que estou a usar sensores que comprei ao preço do “*Vom Mijona*”, não funcionem da maneira que funcionam, sensores de ultrassons mais caros, mas regra geral não fazem mau trabalho para um simples protótipo.

Vale a pena salientar que os sensores estão presos no “para-choques”, com cola quente, e que foi a melhor maneira que encontrei de os fixar sem ter que andar a usar outro tipo de material.

Ver os 3 testes relativos á ponte h.

Na fase de implementação dos sensores frontais, decidi colocar os sensores frontais, no sítio onde aparafusa a capa do carro à frente, como mostra a seguinte figura.

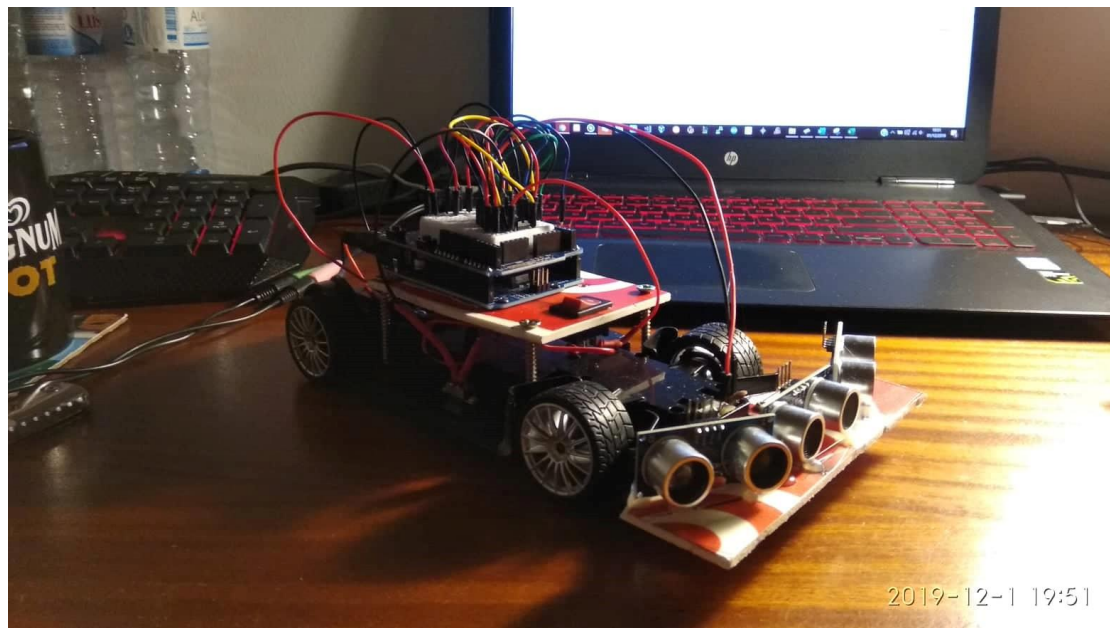


Figura 6 - Implementação dos sensores HC-SR04 frontais no carro

Ver os 3 testes do funcionamento dos sensores de ultrassons.

Após ter colocado os sensores, agreguei no código os comandos para parar o carro quando os sensores se encontrassem a uma certa distância de uma parede, o carro teve de bater umas quantas vezes para que conseguisse minimamente saber qual a distância que iria usar, pois tudo abaixo de 30 centímetros os sensores e o os motores não conseguiam parar o carro a tempo.

Depois dos frontais seguiram-se os traseiros, ficando como mostra a figura seguinte:



Figura 7 - Sensores Traseiros

Para além de adicionar todos os sensores, percebi depois que tinha de organizar os cabos de alguma forma, e nada melhor do que 2 breadboards para esse efeito, 1 na parte da frente, e outra na parte de trás, para espalhar todos os cabos do veículo.

Após isso, liguei tudo como estava e tentei ao máximo fazer com que o carro ficasse “limpo de cabos”, uma tarefa impossível, mas colocá-los de forma a ser mais organizado, e por fim, coloquei ainda o giroscópio no carro, tal como mostra a figura.

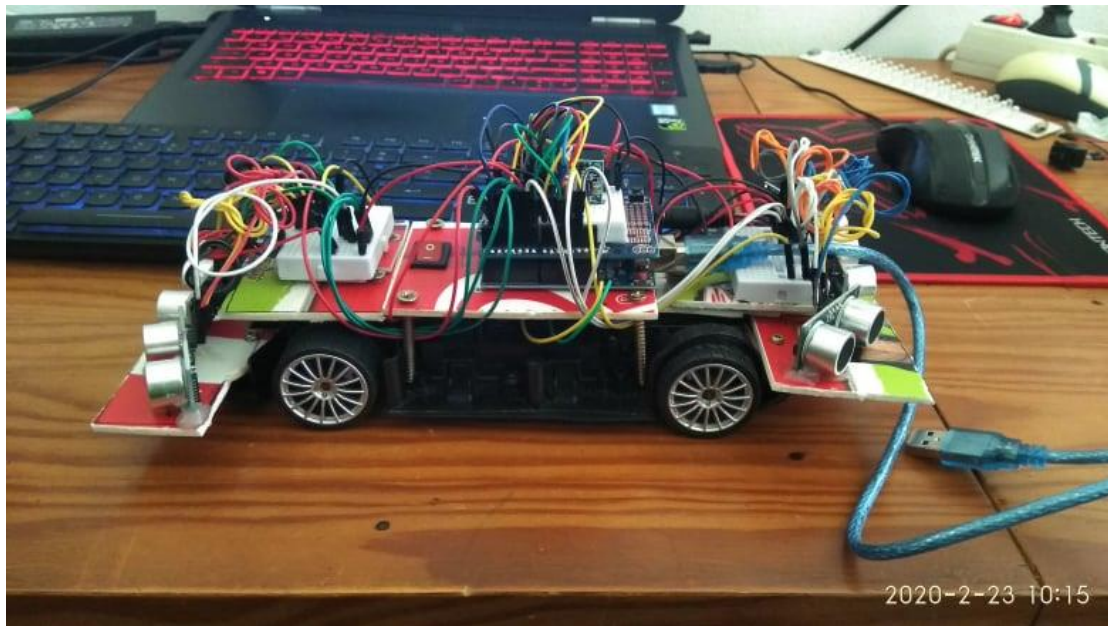


Figura 8 - Cable Management do carro

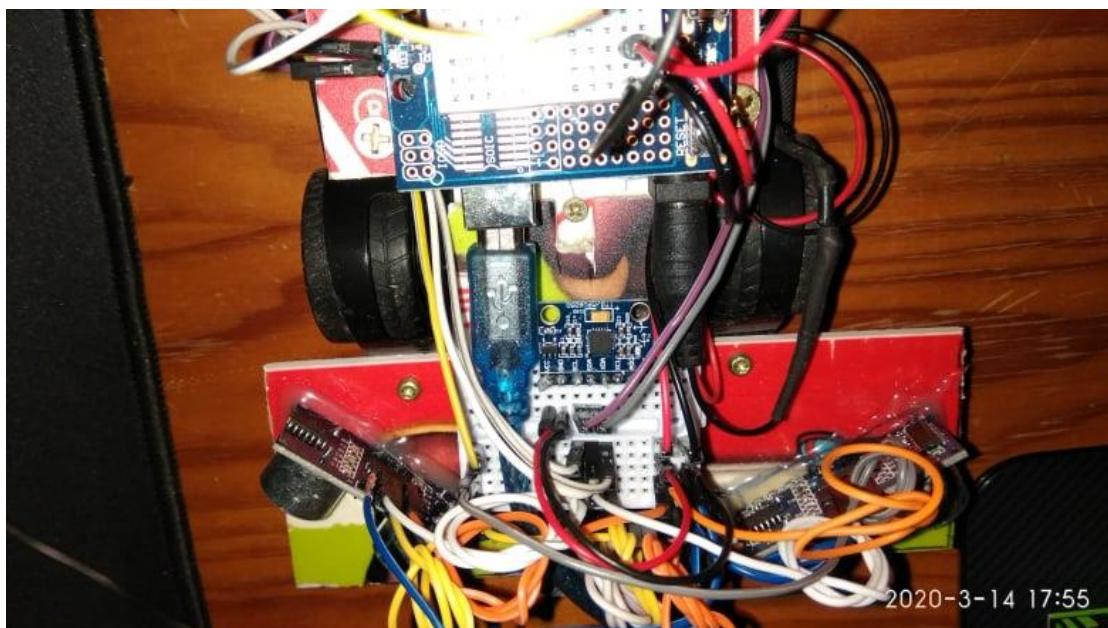


Figura 9 - Giroscópio colocado ao pé da ponte h

O Arduino

Neste projeto, utilizei um Arduino Uno R3, que é basicamente nada mais nada menos do que uma placa de circuitos com um chip central e pinos de entradas e saídas de dados, que é programável desde os programas mais simples como o *Hello World*, até um “Carro Autônomo” em fase protótipo.



Figura 10 - Arduino Uno R3

Este pequeno pedaço de tecnologia tem capacidade para 14 pinos de input e output, uma SRAM de 2KB, um CLOCK speed de 16MHz, e ainda 13 leds embutidos na própria placa.

Esta pequena maravilha da engenharia conta ainda com proporções bastante acessíveis e quase que cabe num bolso, com largura de 68.6 milímetros, e comprimento de 53.4 milímetros, este pesa ainda 25 gramas.

Mas colocando todas essas especificações à parte, o meu Arduino, vai ter ligado 6 sensores HC-SR04, que serão 3 sensores colocados na parte da frente do veículo, e outros 3 na parte de trás, 2 motores, que vão ser comandados por uma ponte h, que contém um chip L293D, e para finalizar 1 giroscópio.

Os sensores

Os sensores que estou a utilizar para o projeto, foram sugeridos pelo professor orientador neste caso os sensores HC-SR04, como podemos observar na figura seguinte.

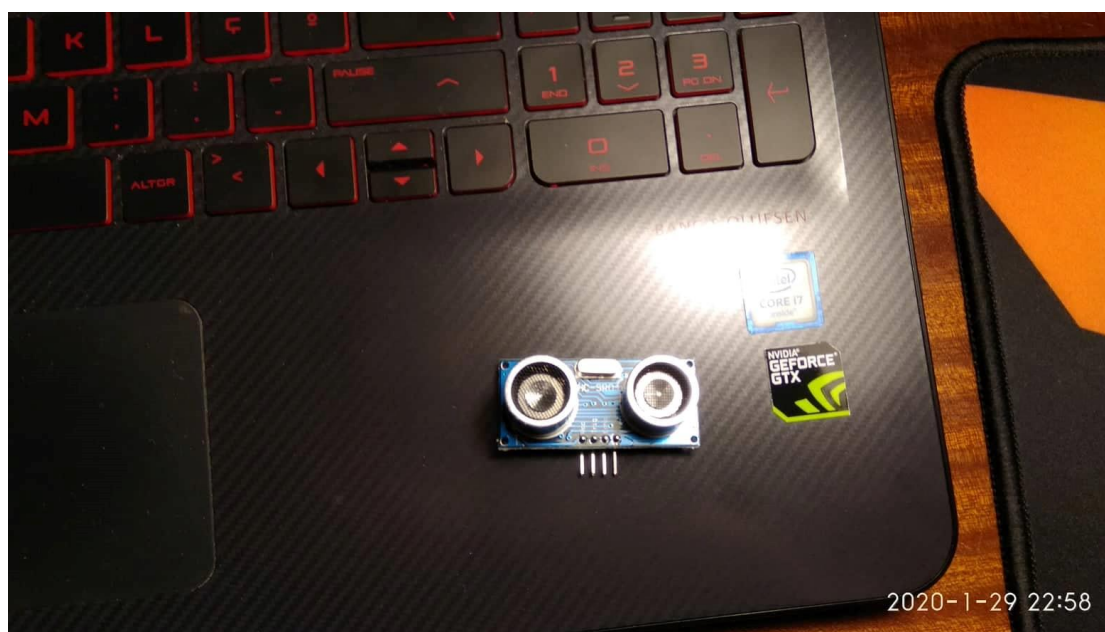


Figura 11 - Sensor HC-SR04

Estes sensores, são sensores de ultrassons em que 1 destes “olhos” é o transmissor de ultrassons, e o outro é o recetor, estes são ligados no Arduino a 5 volts e a um *ground*.

Antes de os ter começado a utilizar, testei 1 que o professor me emprestou para testes, e funcionou, com as suas limitações, mas produz o efeito desejado.

Estes sensores no código de teste tem um *delay*, para que consiga determinar uma distância precisa, pois se tirarmos o *delay*, a distância que este está a medir não é a real, que isto dizer que sem os *delay*, ele atrapalha-se todo, pois uma onda é projetada e é recebida, e o Arduino calcula a distância com base numa fórmula e o tempo que esse ultrassom demora a voltar para o recetor, ora se o *delay* não estiver lá ele dispara 1 atrás do outro, e como 1 deles demora mais tempo a chegar este dá distâncias diferentes.

No projeto decidi utilizar 6 sensores de ultrassons, divididos em grupos de 3, 3 para a frente e os outros 3 para trás, em que tenho 1 ao meio do carro a apontar para a frente e os restantes ficam nas pontas a apontar 1 para cada lado, isto para que consiga cobrir uma maior área, e porque o carro desvia-se e anda para trás não faria sentido ter só à frente, logo com os 6 consigo cobrir toda a distância.

No início do projeto, como foi referido, utilizei um sensor de testes, emprestado pelo professor, e para saber se o sensor estaria a funcionar em condições, fiz um código de testes para testar o sensor, como se pode verificar nas seguintes figuras.

Nas primeiras linhas do código, temos os defines, estes servem para definir os pinos que vão ser usados, neste caso estamos a usar o pino 4 como pino de *Trig*, e o pino de *Echo*, como pino número 3 do Arduino.

Após isso, são declaradas as variáveis globais, e seguidamente vem a função *Setup*, em que temos o *Serial.begin()*, logo na 1ª linha e seguidamente, dizemos ao Arduino que função é que os pinos declarados nos defines vão ter, neste caso o pino *Trig*, tem a função de *output*, pois é este pino que vai mandar os sinais ultrassónicas e o pino *Echo*, que tem a função de *input*, pois é este que vai receber as ondas ultrassónicas.

```
//The sensor HC-SR04 is connectted with arduino through:
//Arduino Pin 10-Trig
//Arduino Pin 13 Echo
//declares the pins
#define trigPin 4
#define echoPin 3

//declares 2 global variables
float duration=0;
float distance=0;
float distanceCentimeters=0;
void setup() {
    Serial.begin(9600);
    //defines the pins that arduino is using to output and input
    //the trig pin is the output from the arduino
    pinMode(trigPin, OUTPUT);
    //the echo pin is the input to the arduino
    pinMode(echoPin, INPUT);
}
```

Figura 12 - Declaração de pinos e função Setup

Após a função *Setup*, vem a função *Loop*, e é nesta função que todos os cálculos vão ser feitos e apresentados no *Serial Monitor*.

Para começar, temos de ver os valores que vêm dos sensores, e isso é feito com o comando *digitalWrite()*, em que primeiramente colocamos o pino *Trig* a *Low*, seguidamente, após um comando *delayMicroseconds()*, de 2 microssegundos, colocamos o mesmo pino a *High*, e por fim ao fim de 2 microssegundos, colocamos o mesmo pino de volta a *Low*.

Por fim é guardada na variável *duration* o valor que é recebido do pino *Echo*, quando este está a *High*, através do comando *pulseIn*.

```
void loop() {  
  //Write a pulse to the HC-SR04 Trigger Pin  
  digitalWrite(trigPin, LOW);  
  delayMicroseconds(2);  
  digitalWrite(trigPin, HIGH);  
  delayMicroseconds(10);  
  digitalWrite(trigPin, LOW);  
  
  //Measure the response from the sensor  
  //in this case we are going to measure the duration with the echo pin is HIGH  
  
  duration = pulseIn(echoPin, HIGH);  
}
```

Figura 13 - Função loop do ficheiro de testes ao sensor de ultrassons

Para se realizar o cálculo da distância, a distância obtida pelo sensor é dividida por 2, porque o valor que está na variável antes da operação de divisão corresponde à distância de ida, bateu no obstáculo, e volta, e nós apenas precisamos da distância compreendida entre as 2 primeiras.

Após isso, temos de multiplicar por 0.39370, que corresponde á velocidade do som, para obtermos a distância em centímetros. Caso essa distância for menor do que 2 mostra a mensagem no *Serial Monitor* “Too Close”, e se for superior a 400, mostra “Out of Range”.

Para finalizar, a distância é mostrada no *Serial Monitor*, através de um *Serial.print()*, como é mostrado na figura seguinte.

```

//Measure the response from the sensor
//in this case we are going to measure the duration with the echo pin is HIGH

duration = pulseIn(echoPinMid, HIGH);

//Determine distance from duration
//We are using 343 meter per second as speed of sound
distanceCentimeters = (duration / 2) * 0.0343;
//return distance;
//Send the Results to the Serial Monitor
Serial.print(F("Distance Front= "));
if (distanceCentimeters >= 400)
{
    Serial.println(F("Out of Range"));
}
else if (distanceCentimeters <= 2 )
{
    Serial.println(F("Too Close!!!"));
}
else

```

Figura 14 - Finalização do Loop e apresentação de resultados

Após de tudo programado, é hora de ligar e testar, o modo como o testei foi simples, liguei o sensor ao Arduino, e por fim corri o código, como mostra a figura seguinte.

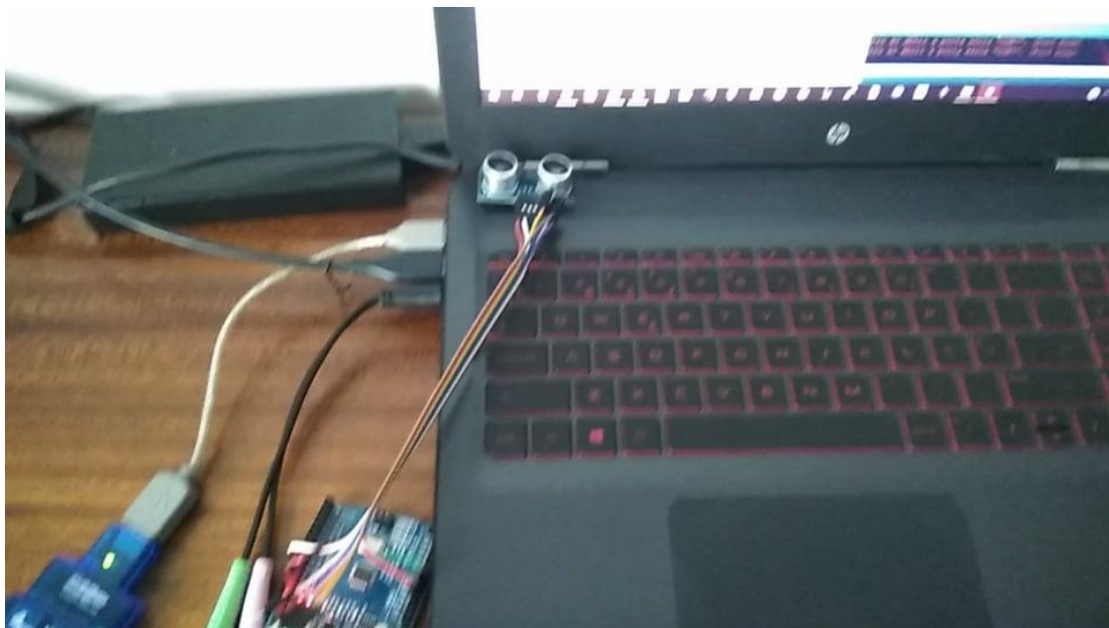


Figura 15 - Teste do sensor HC-SR04

A ponte h

A existência da ponte h neste projeto, deve-se ao facto de uma vez que eu tenho não 1 mas 2 motores para controlar, não que fizesse muita diferença se tivesse apenas 1, pois iria precisar de 1 na mesma, e para controlar os motores, eu necessito de um chip *L293D*, que basicamente eu consigo controlar 2 motores com apenas 1 chip, e até mesmo alterar a velocidade dos mesmos, embora o único que me interessa alterar a velocidade é o motor traseiro.

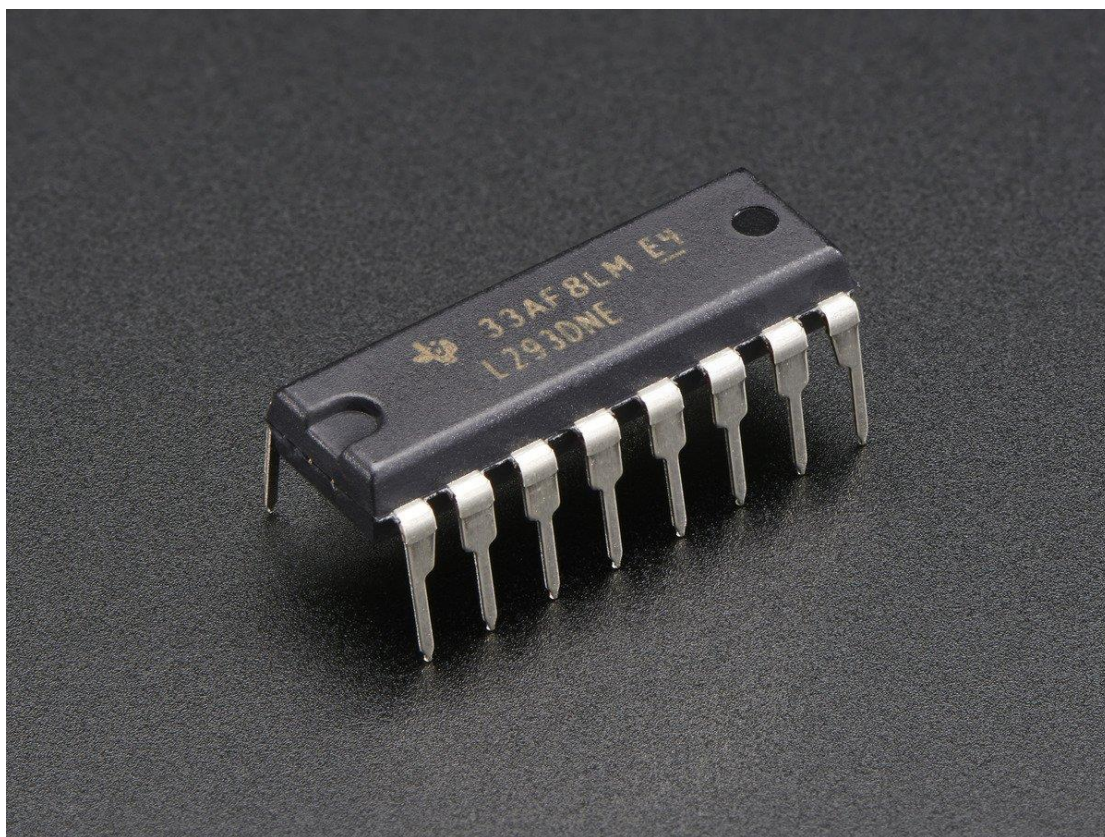


Figura 16 - Chip que equipa a ponte h

Com este pequeno chip montado numa *breadboard*, que por sua vez está colada numa *shield* que está ligada directamente no Arduino, posso fazer mover o carro para trás e para a frente, assim como travar, desligar os motores, e ainda neste caso para o motor frontal virar.

Podemos observar toda a montagem da ponte h, no Arduino *Shield*, que foi feito utilizando o programa Fritzing, através da figura seguinte:

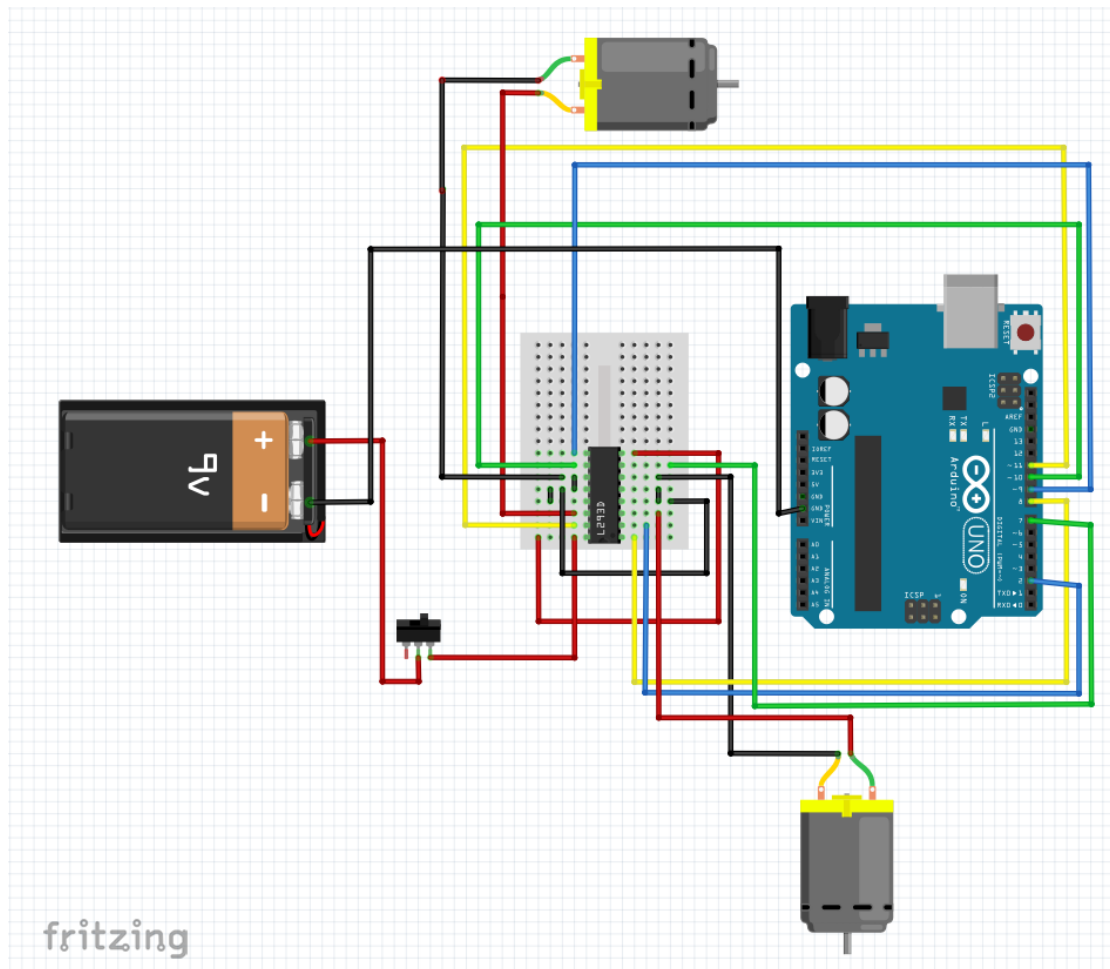


Figura 17 - Esquema ponte h desenhado em Fritzing

Para testar a ponte h, uma vez que fiz um ficheiro de testes para os sensores de ultrassons, os HC-SR04, fez-me sentido fazer também para testar a ponte h, assim como um motor que veio no kit do Arduino, e uma vez que o motor era igual ao do carro, era ainda melhor, pois caso não conseguisse fazer o motor trabalhar, também não iria fazer o carro andar.

O código, pode ser demonstrado nas figuras seguintes, nas primeiras 3 linhas de código temos os defines para os pinos do motor, sendo que os 2 primeiros são relativos aos pinos do motor individualmente, e o último é uma junção dos 2 pinos, em que caso queira desligar o motor, posso fazê-lo desativando o pino 9.


```
// DC motor 2 control
#define P3A 10 // define pin 10 as for P3A
#define P4A 11 // define pin 13 as for P4A
#define EN34 9 // define pin 9 as for EN3 and EN4 enable
```

Figura 18 - Definição dos pinos do motor traseiro do carro

Temos na função `goFoward()`, como colocar o motor a funcionar, em que para isso basta utilizar o comando `digitalWrite()`, para ativar ambos os pinos através da variável `EN34`, e colocando o pino 3 a *High*, e o pino 4 a *Low*, para que este ande para a frente, pois caso fosse ao contrário, este andaria para trás.

```
void goFoward()
{
    //Go Foward
    Serial.println(" Rotating CW");
    digitalWrite(EN34 , HIGH); // Enable 1A and 2A
    digitalWrite(P3A, HIGH); // send + or HIGH singal to P1A
    digitalWrite(P4A, LOW); // send - or LOW singal to P2A

    //digitalWrite(EN34 ,LOW); // Disable 1A and 2A
}
```

Figura 19 - Função que faz o carro andar para a frente

E uma vez tudo programado, podemos ver também a forma em que realizei os testes sem o carro, apenas com metade da ponte h ligada, e um motor, que por sua vez a ponte h estava ligada aos respetivos pinos do Arduino, como se pode ver na seguinte figura.

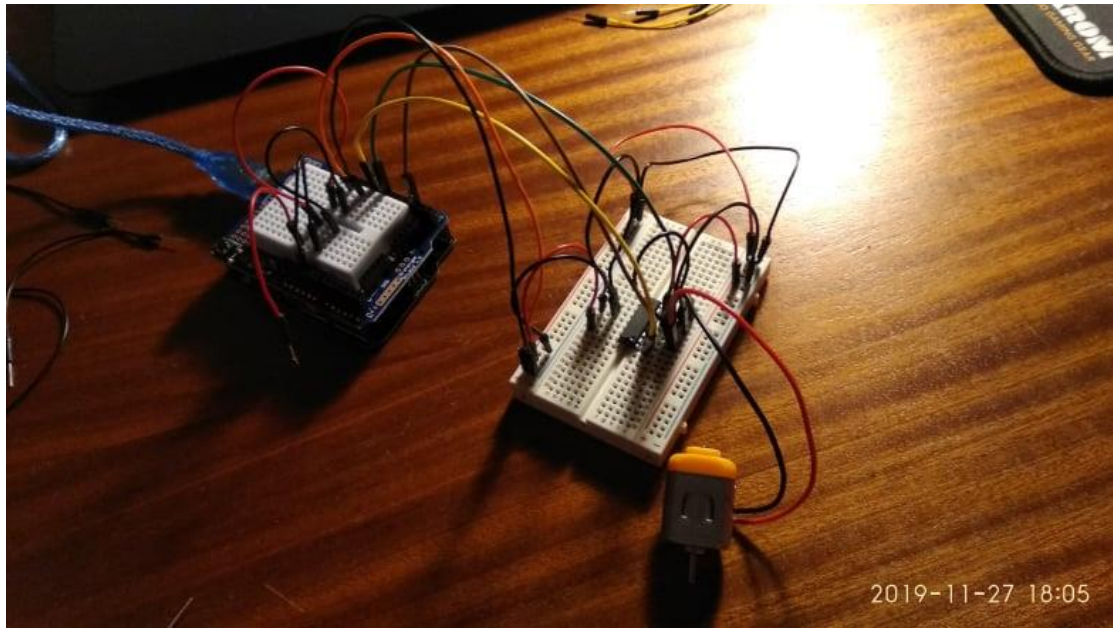


Figura 20 - Teste da ponte h com um motor

O giroscópio

O giroscópio, é um instrumento que me permite medir ângulos, e a aceleração do mesmo, este modelo, vem ainda equipado com um sensor de temperatura.

O modelo que eu vou por a uso, que me foi recomendado pela parte do professor orientador da disciplina, é o giroscópio gy-521, ou mais conhecido ainda como giroscópio de 6 eixos, uma vez que este mede 3 eixos do acelerómetro, e 3 eixos do giroscópio.

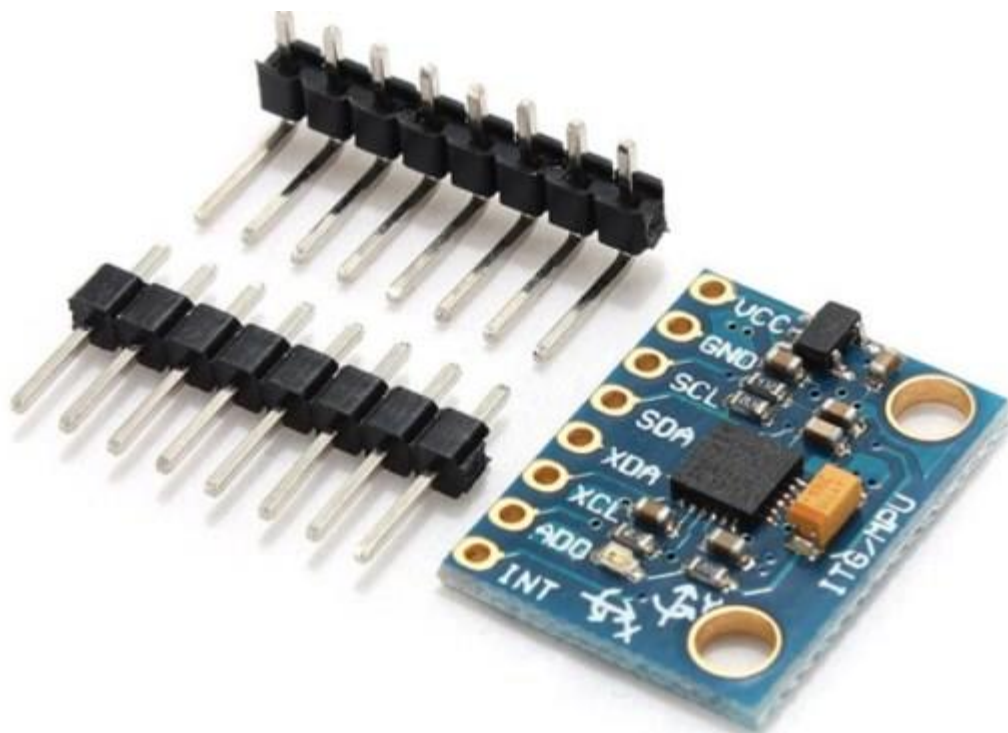


Figura 21 - Giroscópio gy-521

Para que o gy-521, consiga ler esta quantidade de eixos todos este modelo vem equipado com o chip MPU-6050, e para o ligar, utilizamos 1 pino de 5 volts, 1 pino ground, e as ligações de dados são feitas através do protocolo de comunicação I2C, e que os pinos de dados são ligados precisamente no pino SCL(*Serial Clock*) e SDA(*Serial Data*), para que haja a comunicação.

Após ter efetuado pesquisas sobre o giroscópio, e os nomes de *pitch*, *roll*, e assim como *yaw*, entendi o que significavam e para que serviam, como se pode ver na figura seguinte, estes eixos são os principais eixos de um avião, em que por exemplo quando o piloto desce o nariz do avião, vai interferir com o *pitch axis*, se este fizer uma curva para a esquerda, utilizando o leme, irá rodar sobre o *roll axis*, e por fim, mas não menos importante o *yaw axis*, que é controlado por um *airfoil*, que se situa na parte de trás do avião, ou mais especificamente na sua cauda, que faz com que o avião se desvie para a esquerda e para a direita conforme o estipulado pelo piloto.

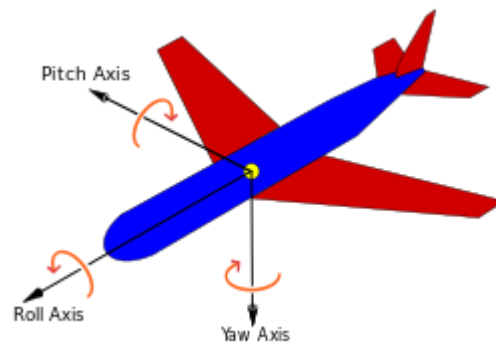


Figura 22 - Representação gráfica de eixos de avião

Agora que sabemos um pouco mais sobre este giroscópio, e do problema que é fugir dos valores reais no que se trata ao *yaw*, vamos perceber como é que ele funciona.

Este giroscópio, lê dos 3 eixos do giroscópio, velocidade rotacional, que é basicamente o quão rápido um certo dispositivo gira em torno do seu próprio eixo, que neste caso pode ser traduzida por velocidade angular. E os 3 eixos acelerómetro, mede a velocidade gravitacional, que depois são aplicadas fórmulas matemáticas, que convertem a velocidade em ângulos, através de trigonometria.

E uma vez combinado todos esses valores, dá origem a um quatião, que não é nada mais nada menos do que 1 única orientação, fundindo todos os valores do acelerómetro e do giroscópio, podemos depois determinar ângulos, como os que se verificam na figura seguinte.

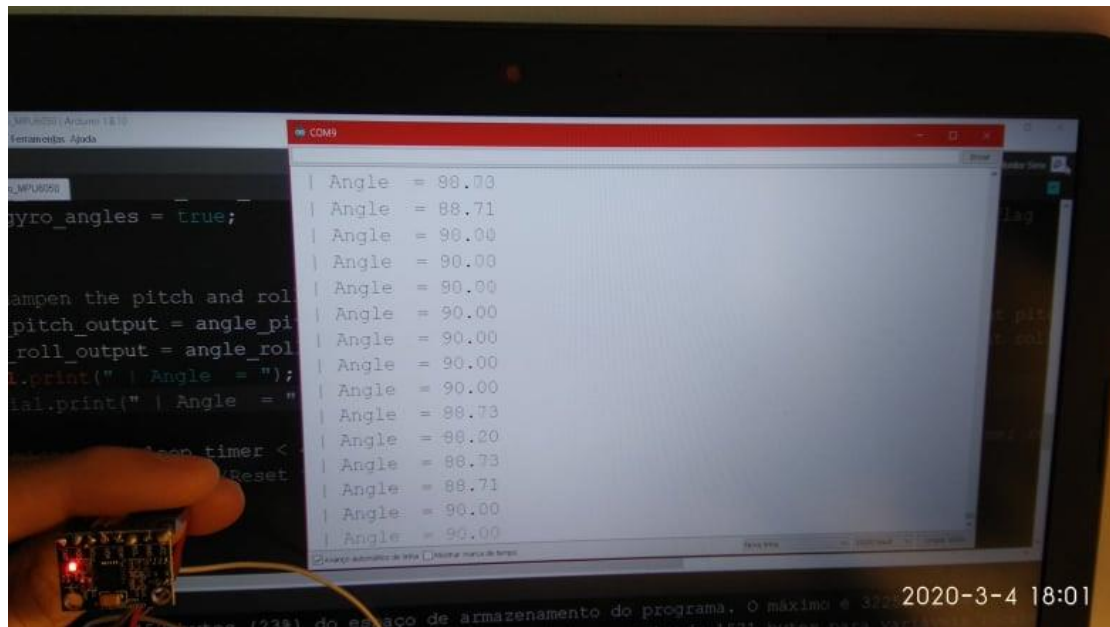


Figura 23 - Teste do giroscópio a 1 ângulo de 90°

No caso da figura anterior, estou a mexer com o *pitch axis*.

Quando o giroscópio é posto na posição que se vê na figura seguinte, e se tenta extrair o ângulo do *pitch axis*, neste caso, o que sai do giroscópio não é o ângulo, mas sim a velocidade angular.

Quer isto dizer que posso estar a tentar descrever uma curva de 90° graus com o giroscópio, nesta posição, em que rodando devagar obtenho valor na casa dos 30°/s, e se rodar mais depressa, obtenho valores acima disso.

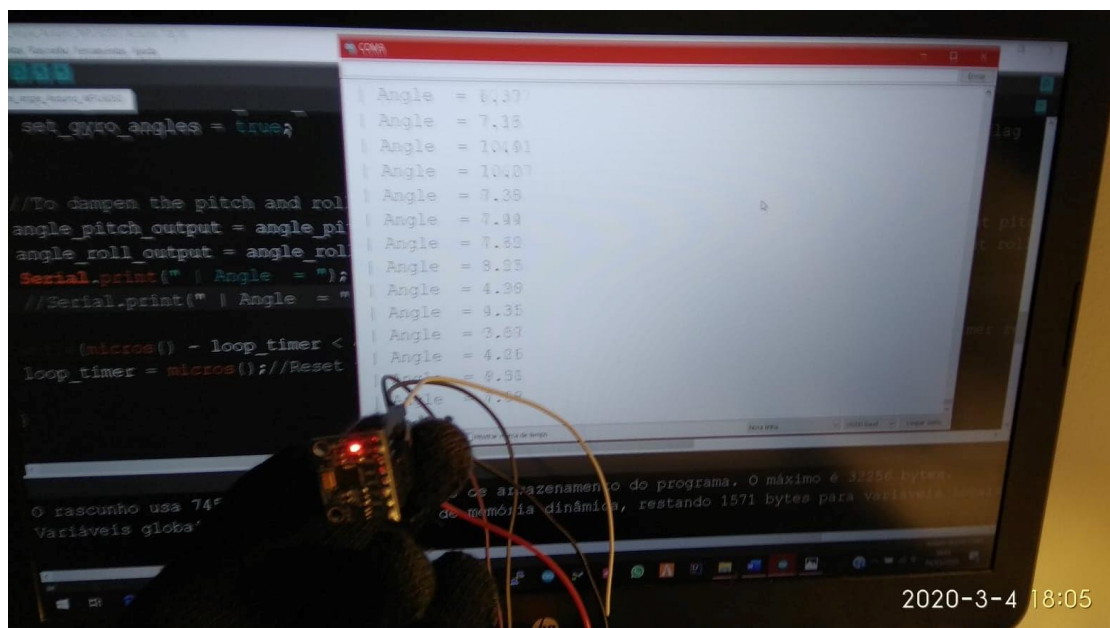


Figura 24 - Giroscópio a dar a velocidade angular

Agora, vamos ver como é que tudo isso é feito no código.

Logo na 1ª linha de código temos o define da biblioteca *Wire.h*, que é a biblioteca responsável pela comunicação I2C do Arduino, e em seguida, temos a declaração de todas as variáveis globais.

```
#include <Wire.h>
//Declaring some global variables
int gyro_x, gyro_y, gyro_z;
long gyro_x_cal, gyro_y_cal, gyro_z_cal;
boolean set_gyro_angles;

long acc_x, acc_y, acc_z, acc_total_vector;
float angle_roll_acc, angle_pitch_acc;

float angle_pitch, angle_roll;
int angle_pitch_buffer, angle_roll_buffer;
float angle_pitch_output, angle_roll_output;

long loop_timer;
int temp;
```

Figura 25 - Declaração de variáveis globais para uso do programa

Seguidamente, temos a função *Setup*, onde é usada a biblioteca *Wire*, que utiliza o protocolo de comunicação por I2C, seguidamente, existe uma chamada à função *setup_mpu_6050_registers()*, em que esta inicializa as comunicações com o MPU-6050, e após isso, temos ainda 3 variáveis que foram globalmente declaradas anteriormente, em que estão a ser incrementadas com os valores respetivos do giroscópio, referente a cada eixo, X, Y, e Z, todo este código está dentro de um ciclo *for* que começa em 0, e faz 1000 leituras para calibração do giroscópio.

Para finalizar, temos ainda um *delay* de 3 segundos antes de avançar para o *loop*.

```

void setup() {

    Wire.begin();
    setup_mpu_6050_registers();
    for (int cal_int = 0; cal_int < 1000 ; cal_int ++){
        read_mpu_6050_data();
        gyro_x_cal += gyro_x;
        gyro_y_cal += gyro_y;
        gyro_z_cal += gyro_z;
        delay(3);
    }
}

```

Figura 26 - Função Setup

Mas antes de irmos para o *loop*, vamos passar pela função `setup_mpu_6050_registers()`, e descobrir como esta funciona.

A função `setup_mpu_6050_registers()`, inicia o giroscópio, e com os comandos `Wire.write()`, mete os apontadores de memória a apontar para os bits de energia do giroscópio, como se pode ver na figura seguinte, na penúltima linha da tabela.


		MPU-6000/MPU-6050 Register Map and Descriptions						Document Number: RM-MPU-6000A-00 Revision: 4.2 Release Date: 08/19/2013			
Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
67	103	I2C_MST_DELAY_CT RL	R/W	DELAY_ES _SHADOW	-	-	I2C_SLV4 _DLY_EN	I2C_SLV3 _DLY_EN	I2C_SLV2 _DLY_EN	I2C_SLV1 _DLY_EN	I2C_SLV0 _DLY_EN
68	104	SIGNAL_PATH_RES ET	R/W	-	-	-	-	-	GYRO _RESET	ACCEL _RESET	TEMP _RESET
6A	106	USER_CTRL	R/W	-	FIFO_EN	I2C_MST _EN	I2C_IF _DIS	-	FIFO _RESET	I2C_MST _RESET	SIG_COND _RESET
6B	107	PWR_MGMT_1	R/W	DEVICE _RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
6C	108	PWR_MGMT_2	R/W	LP_WAKE_CTRL[1:0]		STBY_XA	STBY_YA	STBY_ZA	STBY_XG	STBY_YG	STBY_ZG

Figura 27 - Datasheet gy-521 registo 6B referente ao power mangment

```
// divide by 1000 to get average offset
gyro_x_cal /= 1000;
gyro_y_cal /= 1000;
gyro_z_cal /= 1000;
Serial.begin(19200); // setup Serial Monitor
loop_timer = micros();
}
```

Figura 28 - Finalização da função setup

Para finalizar a função *Setup*, uma vez que foram feitas 1000 leituras dos sensores, dividimos os valores por 1000, de cada um dos eixos do giroscópio para obtermos um valor médio. Temos seguidamente o comando *Serial.begin* para mostrar os dados no *Serial Monitor*, e por fim, temos uma variável *loop timer*, que vai receber o valor da função *micros()*. As variáveis presentes na figura anterior, contém a média do “erro” produzido pelo giroscópio.

No código essa mesma inicialização do sensor está feita após as primeiras 3 linhas da função, sendo que a próxima linha de código já se direciona ao registo 6B do giroscópio.



```

void setup_mpu_6050_registers(){
    //Activate the MPU-6050
    Wire.beginTransmission(0x68);
    Wire.write(0x6B);
    Wire.write(0x00);
    Wire.endTransmission();
    //Configure the accelerometer (+/-8g)
    Wire.beginTransmission(0x68);
    Wire.write(0x1C);
    Wire.write(0x10);
    Wire.endTransmission();
    //Configure the gyro (500dps full scale)
    Wire.beginTransmission(0x68);
    Wire.write(0x1B);
    Wire.write(0x08);
    Wire.endTransmission();
}

```

Figura 29 - Função `setup_mpu_6050_registers()`

Após isso, temos a configuração do giroscópio, em que acede aos registos 1B, e 1C, respetivamente, que se encontram nas últimas 2 linhas da tabela, da figura seguinte.

	MPU-6000/MPU-6050 Register Map and Descriptions	Document Number: RM-MPU-6000A-00 Revision: 4.2 Release Date: 08/19/2013
---	--	---

3 Register Map

The register map for the MPU-60X0 is listed below.

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0D	13	SELF_TEST_X	R/W	XA_TEST[4-2]			XG_TEST[4-0]				
0E	14	SELF_TEST_Y	R/W	YA_TEST[4-2]			YG_TEST[4-0]				
0F	15	SELF_TEST_Z	R/W	ZA_TEST[4-2]			ZG_TEST[4-0]				
10	16	SELF_TEST_A	R/W	RESERVED		XA_TEST[1-0]		YA_TEST[1-0]		ZA_TEST[1-0]	
19	25	SMP_LRT_DIV	R/W	SMP_LRT_DIV[7:0]							
1A	26	CONFIG	R/W	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
1B	27	GYRO_CONFIG	R/W	-	-	-	FS_SEL[1:0]		-	-	-
1C	28	ACCEL_CONFIG	R/W	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]				

Figura 30 - Datasheet gy-521 configuração do Giro, e do Acelerómetro

Nas figuras seguintes, temos os dados que o giroscópio e o acelerómetro dispõem, após fazerem o teste de resposta a eles mesmos, que consiste numa fórmula matemática, como se vê na figura para calcular os seguintes valores.

Estes dados, são no caso da figura 1 os limites para cada eixo do giroscópio.

The self-test response is defined as follows:

Self-test response = Sensor output with self-test enabled – Sensor output without self-test enabled

Figura 31 Datasheet gy-521 self-test response

FS_SEL	Full Scale Range
0	$\pm 250\text{ }^{\circ}/\text{s}$
1	$\pm 500\text{ }^{\circ}/\text{s}$
2	$\pm 1000\text{ }^{\circ}/\text{s}$
3	$\pm 2000\text{ }^{\circ}/\text{s}$

Figura 31 - Dados de configuração do giroscópio

E na figura que se segue, temos os dados resultantes do self-test do acelerómetro.

AFS_SEL	Full Scale Range
0	$\pm 2g$
1	$\pm 4g$
2	$\pm 8g$
3	$\pm 16g$

Figura 32 - Dados de configuração do acelerómetro

Caso os valores dos outputs não estiverem dentro do range dos valores das tabelas, tanto mínimo, como máximo, então o *self-test* falhou, e pode ter havido algum problema na configuração do giroscópio ou do acelerómetro.

Passando para o *loop*, na função principal do Arduino, temos a chamada á função `read_mpu_6050_data()`, que vai ler os valores dados pelo chip MPU-6050.

Seguidamente, temos as variáveis relativas a cada eixo do giroscópio, a que estão a ser decrementadas os valores dos erros, que foram calculados na função *Setup*, no ciclo *for*.

Após isso, são feitos os cálculos dos ângulos *pitch*, e *roll*, em que por exemplo no caso do ângulo *pitch*, é incrementado o valor do *gyro_x* e multiplicado por 0.0000611, estes são os cálculos para se extrair o ângulo, mas existe um problema, os ângulos não estão a ser mostrados em graus.

Para resolver esse problema, tanto o valor adquirido anteriormente, tem de ser convertido para graus, $0.000001066 = 0.0000611 * (3.142(\text{PI}) / 180\text{degr})$, que é depois

feito quando no caso do ângulo *pitch* é incrementado o valor do *angle_roll*, e este é multiplicado pelo seno do *gyro_z*, e este por sua vez pelo valor 0.000001066.

```
void loop(){

    read_mpu_6050_data();
    //Subtract the offset values from the raw gyro values
    gyro_x -= gyro_x_cal;
    gyro_y -= gyro_y_cal;
    gyro_z -= gyro_z_cal;

    //Gyro angle calculations . Note 0.0000611 = 1 / (250Hz x 6
    angle_pitch += gyro_x * 0.0000611;
    angle_roll += gyro_y * 0.0000611;
    //0.000001066 = 0.0000611 * (3.142(PI) / 180degr) The Ardui
    angle_pitch += angle_roll * sin(gyro_z * 0.000001066);
    angle_roll -= angle_pitch * sin(gyro_z * 0.000001066);
```

Figura 33 - Função loop

Seguidamente, faz-se a mesma coisa, mas para os valores do acelerómetro, como se pode demonstrar na imagem seguinte:

```
//Accelerometer angle calculations
acc_total_vector = sqrt((acc_x*acc_x)+(acc_y*acc_y)+(acc_z*acc_z));
//57.296 = 1 / (3.142 / 180) The Arduino asin function is in radians
angle_pitch_acc = asin((float)acc_y/acc_total_vector)* 57.296;
angle_roll_acc = asin((float)acc_x/acc_total_vector)* -57.296;

angle_pitch_acc -= 0.0;
angle_roll_acc -= 0.0;
```

Figura 34 - Continuação da função loop

Para finalizar a função loop, temos um if, em que caso a variável *set_gyro_angles* tenha valores, os ângulos *pitch*, e *roll*, são calculados da seguinte maneira, em que se junta 96% da parte do giroscópio, e os restantes 4% do acelerómetro, isto porque o giroscópio é bom a curto prazo, pois com o passar do tempo, os valores do mesmo sofrem um desvio, e por isso fundimo-los com os dados provenientes do acelerómetro, que estes sim são bons a longo prazo, que depois dão os ângulos em que o giroscópio se encontra.

Caso a variável não se encontre preenchida, as variáveis *angle_pitch*, e *angle_roll*, passam a ter os valores das variáveis *angle_pitch_acc*, e *angle_roll_acc*, provenientes do acelerómetro respetivamente.

Para finalizar as variáveis de output respetivas a cada ângulo passam a ser iguais a 0.9 do seu mesmo *output*, e apenas a 0.1 das respetivas variáveis, após isso são mostradas no *Serial Monitor*, após isso temos a conclusão do ciclo *while*, e por fim o *reset* da variável *loop_timer* que age como um contador.

```
if(set_gyro_angles){
    angle_pitch = angle_pitch * 0.9996 + angle_pitch_acc * 0.0004;
    angle_roll = angle_roll * 0.9996 + angle_roll_acc * 0.0004;
}
else{
    angle_pitch = angle_pitch_acc;
    angle_roll = angle_roll_acc;
    set_gyro_angles = true;
}

//To dampen the pitch and roll angles a complementary filter is used
angle_pitch_output = angle_pitch_output * 0.9 + angle_pitch * 0.1;
angle_roll_output = angle_roll_output * 0.9 + angle_roll * 0.1;
Serial.print(" | Angle = "); Serial.println(angle_roll_acc);
//Serial.print(" | Angle = "); Serial.println(angle_pitch_acc);

while(micros() - loop_timer < 4000);
loop_timer = micros();//Reset the loop timer
```

Figura 35 - Finalização da função *loop*

E para finalizar o giroscópio, temos ainda a função *read_mpu_6050_data()*, como se pode ver na figura seguinte, esta função foi chamada no início da função *loop*, e esta função como o próprio nome indica é a que extrai os dados do giroscópio acendendo aos endereços de memória corretos.

À semelhança da função *setup_mpu_6050_registers()*, esta também vai aceder aos registos 68, e 3B, para que possa dar energia, e vai fazer um *request* ou seja um pedido de informações ao chip, neste caso o *request* vai até ao bit 14, que faz os 15 bits de dados do 0 ao 14.

Após isso é feito um ciclo *while*, em que enquanto tiverem posições por percorrer, ou seja enquanto não chegarem ao registo 14, vão extraíndo os dados numa sequência de 8 bits e guardados nas suas respetivas, e como podemos ver este giroscópio, tem ainda um sensor de temperatura, para além de nos dar os valores do acelerómetro e do giroscópio, como podemos ver na figura seguinte.

```
void read_mpu_6050_data(){
    Wire.beginTransaction(0x68);
    Wire.write(0x3B);
    Wire.endTransmission();
    Wire.requestFrom(0x68,14);
    while(Wire.available() < 14);
    acc_x = Wire.read()<<8|Wire.read();
    acc_y = Wire.read()<<8|Wire.read();
    acc_z = Wire.read()<<8|Wire.read();
    temp = Wire.read()<<8|Wire.read();
    gyro_x = Wire.read()<<8|Wire.read();
    gyro_y = Wire.read()<<8|Wire.read();
    gyro_z = Wire.read()<<8|Wire.read();
}
```

Figura 36 - Função read_mpu_6050_data(), que lê os dados do MPU-6050

O código

O código para o projeto, foi desenvolvido em c++, uma vez que essa é a linguagem para se programar no Arduino.

DEFINES

Nas primeiras linhas de código, temos os *#defines* para definir os pinos que vamos utilizar no projeto, primeiramente, defini os pinos para os sensores de ultrassons frontais, como mostra a seguinte figura.

```
////////FRONT SENSORES////////  
  
//SR-04 FRONT MIDDLE SENSOR  
//Arduino Pin 3-Trig  
//Arduino Pin 2 Echo  
//declares the pins  
int trigPin = A1;  
//the trig pin is the same pin for the 3 sensores  
#define echoPinMid 4  
  
//SR-04 FRONT LEFT SENSOR  
#define echoPinLeft 3  
  
//SR-04 FRONT RIGHT SENSOR  
#define echoPinRight 5
```

Figura 37 - Declaração sensor HC-SR04

Para os sensores HC-SR04, ou seja, os sensores de ultrassons trabalharem, para além de terem de estar ligados a um *vcc* de 5 volts e a um *ground*, para conseguirem fazer alguma coisa tem de estar ligados ao Arduino, e neste caso os pinos escolhidos foram o pino A1, como pino de *Trig*, e os pinos 3, 4, 5, como pinos de *Echo*.

Neste caso nos sensores apenas temos um pino *Trig*, pois eu quero que os ultrassons que saem dos sensores sejam todos disparados ao mesmo tempo, tendo apenas necessidade de mais 3 cabos para ligar os pinos *Echo*, que recebem esse sinal, é também uma forma de tentar reduzir ao máximo o uso de cabos excessivos, pois o meu Arduino está esgotado de pinos.

DECLARAÇÃO DE PINOS

Seguidamente, após termos definido os pinos para os sensores, defino os pinos para os motores, como se pode demonstrar na figura seguinte.

```
// DC motor 1 control
#define P1A 2 // define pin 2 as for P1A
#define P2A 7 // define pin 7 as for P2A
#define EN12 8 // define pin 8 as for 1,2EN enable
```

Figura 38 - Declaração dos pinos utilizados no 1º motor

Mas os pinos dos sensores, e os dos motores já são ligeiramente diferentes, pois, o motor, tem um pino *ground* e outro *vcc*, e esse mesmo pino *vcc*, vai ligar ao pino que definimos no Arduino, e quando é ativado, pela ponte h, o motor começa a rodar.

```
// DC motor 2 control
#define P3A 10 // define pin 10 as for P3A
#define P4A 11 // define pin 13 as for P4A
#define EN34 9 // define pin 9 as for EN3 and EN4 enable
```

Figura 39 - Declaração dos pinos utilizados no 2º motor

Para finalizar a declaração de pinos e variáveis, acabamos com as variáveis globais para o giroscópio.

```
//GYRO LYBRARIE FOR I2C COMUNICATIONS
#include <MPU6050_tockn.h>
#include <Wire.h>
MPU6050 mpu6050(Wire);
```

Figura 40 - Declaração de variáveis usadas no Giroscópio

PROTÓTIPOS

Antes da função *setup*, temos os protótipos das funções isto apenas serve para saber quantas funções temos, que tipo de parâmetros as funções têm, como mostra a figura seguinte.

```
// function prototypes
void decisionMaker(int, int, int, int, int, int, float);
int middleSensorFront();
int leftSensorFront();
int rightSensorFront();
int middleSensorBack();
int leftSensorBack();
int rightSensorBack();
void speedController();
void goLeft(int, int);
void goRight(int, int);
void goForward();
void goBack();
void stopEngine(float);
void stopTurningLeft();
void stopTurningRight();
void initializeGiro();
float showGyroData();
```

Figura 41 - Protótipos das funções

Após isso estar feito, vem a função *setup*, que no Arduino apenas é executada uma vez, e neste caso esta tem a especificação de para o que é que os pinos que foram definidos anteriormente servem, e como podemos ver, por exemplo o pino *Trig*, foi definido como *output*, ou seja estes pino que atua como “atirador” das ondas de ultrassons, e o pino *Echo*, que foi definido como *input*, que atua como recetor das ondas de ultrassons, como podemos verificar na figura seguinte.

FUNÇÃO SETUP

Na mesma função *setup*, estão ainda declarados os pinos para os 2 motores do carro, sendo que todos os pinos estão definidos como *output*, uma vez que os motores não necessitam de mandar dados, mas sim de transmitir movimento, não há necessidade de haver pinos de *input* para os motores, como podemos observar nas 3 figuras seguintes.

```
void setup() {  
  //defines the pins that arduino is using to output and input  
  //the trig pin is the output from the arduino  
  
  //SR-04 FRONT MIDLE SENSOR  
  pinMode(trigPin, OUTPUT);  
  //the echo pin is the input to the arduino  
  pinMode(echoPinMid, INPUT);  
  
  //SR-04 FRONT LEFT SENSOR  
  pinMode(echoPinLeft, INPUT);  
  
  //SR-04 FRONT Right SENSOR  
  pinMode(echoPinRight, INPUT);
```

Figura 42 - Função setup

```
//SR-04 BACK MIDLE SENSOR  
pinMode(trigPinBack, OUTPUT);  
//the echo pin is the input to the arduino  
pinMode(echoPinMidBack, INPUT);  
  
//SR-04 BACK LEFT SENSOR  
pinMode(echoPinLeftBack, INPUT);  
  
//SR-04 BACK Right SENSOR  
pinMode(echoPinRightBack, INPUT);
```

Figura 43 - Continuação função setup

```

// DC MOTOR PINS
Serial.begin(19200); // setup Serial Monitor to display information
pinMode(P1A, OUTPUT); // define pin as OUTPUT for P1A
pinMode(P2A, OUTPUT); // define pin as OUTPUT for P2A
pinMode(EN12, OUTPUT); // define pin as OUTPUT for 1,2EN

// DC MOTOR 2 PINS
pinMode(P3A, OUTPUT); // define pin as OUTPUT for P3A
pinMode(P4A, OUTPUT); // define pin as OUTPUT for P4A
pinMode(EN34, OUTPUT); // define pin as OUTPUT for 3,4EN

//GIRO
initializeGiro();

```

Figura 44 -Finalização da função setup

DECLARAÇÃO DE VARIÁVEIS GLOBAIS

Ainda antes de passarmos para o *loop*, temos ainda algumas variáveis que foram declaradas como variáveis globais para serem usadas no *loop*, e pelas suas funções adjacentes.

```

//global variables declared for the loop
int distanceToStop = 30;
int distanceToSpeedDown = 40;
int distanceMidFront = 0;
int distanceLeftFront = 0;
int distanceRightFront = 0;
int distanceMiddleBack = 0;
int distanceLeftBack = 0;
int distanceRightBack = 0;
int distanceAvoidObstacle = 50;
float acceleration = 0;
float angle=0;
float angleToTurn=0;
float maxAngle=0;
float minAngle=0;
int angleTurn=15;
int simetricAngleToTurn=-15;

```

Figura 45 - Declaração de variáveis globais

FUNÇÃO LOOP

Vamos agora entrar na função loop, que é a função principal do nosso programa, e que vai repetir o código que lá estiver dentro, até que o número 1 seja igual ao número 2, e todos nós sabem que isso não vai acontecer, logo ele repete o código infinitamente.

```
void loop() {  
  
    distanceMidFront = middleSensorFront();  
    distanceLeftFront = leftSensorFront();  
    distanceRightFront = rightSensorFront();  
    distanceMiddleBack = middleSensorBack();  
    distanceLeftBack = leftSensorBack();  
    distanceRightBack = rightSensorBack();  
    angle = showGyroData();  
  
    //function that determines to where the car goes at  
    decisionMaker(distanceMidFront, distanceLeftFront,
```

Figura 46 - Função loop do Arduino

Como podemos ver na figura, no loop, estou a utilizar todas as variáveis que foram declaradas como globais em cima do mesmo e estou a atribuir-lhes valores, igualando á função correspondente no momento da declaração das mesmas.

FUNÇÃO DECISIONMAKER()

Segue-se a função *decisionMaker()*, que tem 7 variáveis a entrar como parâmetro da função, esta função está carregada de *if's*, e *else's*, que irão tomar as decisões do carro, com base nos valores dos sensores e das variáveis.

Para começar na função *decisionMaker()*, como o próprio nome indica, é a função que toma as decisões do carro, e por isso é que tem 7 variáveis de entrada, esta função tem de agregar todos os dados das outras funções todas, e por fim juntá-los e decidir o que fazer.

Na 1ª figura desta função, temos uma verificação em que se o valor do sensor frontal que está no meio, for superior ao valor predefinido para evitar obstáculos, o carro vai em frente, e após o carro estar a andar, as distâncias vão ser diferentes, então a comparação é feita de novo, e caso a distância seja inferior ou igual á distância para evitar obstáculos, este verifica os 2 sensores ao lado, e o sensor que tiver mais distância disponível, é para onde o carro se move.

```
void decisionMaker(int distanceMidFront, int distanceLeftFront, int distanceRightFront, int distanceM
{
    //if the distance to the front is bigger that the preset to speed down then goes front
    if(distanceMidFront > distanceAvoidObstacle)
    {
        goFoward();
    }
    //if the distance in front is smaller then the preset to avoid obstacles the car turns
    if(distanceMidFront <= distanceAvoidObstacle)
    {
        //checks the other sensores and decide where to go
        if((distanceLeftFront > distanceAvoidObstacle) && (distanceRightFront <= distanceAvoidObstacle))
        {
            //goes to the left
            goLeft(distanceLeftFront, distanceAvoidObstacle);
        }
    }
}
```

Figura 47 - Função *decisionMaker()*

Na 2ª figura desta função, vemos que é feita uma outra comparação, dentro do *if* que determinou o carro para virar à esquerda, em que se o valor do sensor da esquerda for menor ou igual do que a distância predefinida para parar, o carro para.

Existe ainda outro caso que é se a distância á direita for maior do que a distância à esquerda o carro vira para a direita, e caso o valor do sensor direito for menor ou igual do que o valor do sensor predefinido para parar, o carro para.

```

//if the distance from the left sensor gets minor, the car stops
if(distanceLeftFront <= distanceToStop)
{
    stopEngine(acceleration);
}
}
else
    if((distanceLeftFront <= distanceAvoidObstacle) && (distanceRightFront > distanceAvoidObstacle))
    {
        //goes to the right
        goRight(distanceRightFront, distanceAvoidObstacle);

        //if the distance from the right sensor gets minor, the car stops
        if(distanceRightFront <= distanceToStop)
        {
            stopEngine(acceleration);
        }
    }
}

```

Figura 48 - Continuação função decisionMaker()

Nesta 3ª figura, podemos ver que temos ainda mais 3 verificações, em que a 1ª, serve para se o carro se chegar muito perto de uma parede, este pare, e as 2 últimas é para que quando todos os sensores da frente estão com uma distância inferior ou igual á distância de paragem, o Arduino verifica o sensor traseiro do meio, e caso a distância seja maior do que a distância para evitar obstáculos o carro anda para trás.

Caso contrário, este fica no sítio, pois não consegue ir para mais lado nenhum, apenas se nós por exemplo tivermos um pé atrás do carro e o removermos, este começa a andar para trás.

```

else
    //speedController();
    if(distanceMidFront <= distanceToStop)
    {
        stopEngine(acceleration);
    }
    if((distanceMidFront <= distanceToStop) && (distanceLeftFront <= distanceToStop) && (d
    {
        //checks the back sensores
        if(distanceMiddleBack > distanceAvoidObstacle)
        {
            goBack();
        }
    }
}

```

Figura 49 - Continuação função decisionMaker()

Na 4ª figura da função decisionMaker(), temos basicamente a mesma coisa que foi feita para os sensores da frente, em que se encontrar um obstáculo na traseira do carro e o sensor da frente ainda não tiver distância para seguir em frente, o Arduino, poder manobrar o carro, de modo a que evite os obstáculos.

```

//if the distance in back is smaller then the presetted to avoid obstacles the car turns
if(distanceMiddleBack <= distanceAvoidObstacle)
{
    //checks the other sensores and decide where to go
    if((distanceLeftBack > distanceAvoidObstacle) && (distanceRightBack <= distanceAvoidObstacle))
    {
        //goes to the left
        goLeft(distanceLeftBack, distanceAvoidObstacle);

        //if the distance from the left sensor gets minor, the car stops
        if(distanceLeftFront <= distanceToStop)
        {
            stopEngine(acceleration);
        }
    }
}

```

Figura 50 - Continuação função decisionMaker()

Na 5ª e última figura da função decisionMaker(), é basicamente a continuação do código da figura anterior, mas esta apresenta um *delay* em microssegundos, para que haja qualquer coisa que atrase um pouco o funcionamento da função, pois os sensores por exemplo trabalham ao seu ritmo e também tem um *delay*, caso não tenham não vão ter a mesma precisão, e nesta função é só mesmo um compasso de espera pelas leituras.

```

    else
    {
        if((distanceLeftBack <= distanceAvoidObstacle) && (distanceRightBack > distanceAvoidObstacle))
        {
            //goes to the right
            goRight(distanceRightBack, distanceAvoidObstacle);

            //if the distance from the right sensor gets minor, the car stops
            if(distanceRightFront <= distanceToStop)
            {
                stopEngine(acceleration);
            }
        }
    }
    else
    {
        if(distanceMiddleBack <= distanceToStop)
        {
            stopEngine(acceleration);
        }
    }
    delayMicroseconds(25);
}

```

Figura 51 - Finalização da função decisionMaker()

FUNÇÃO MIDDLESENSORFRONT()

Após a finalização da função, temos as 6 funções seguidas que dão os dados dos sensores, e uma vez que as funções são todas iguais, e só muda o nome, apenas vou colocar no relatório 1 delas.

Na 1ª figura da função `middleSensorFront()`, temos a declaração e inicialização das variáveis, e seguidamente, temos a ativação do pino *Trig*, que basicamente as 5 linhas de código lançam o sinal ultrassónico durante 2 microssegundos e depois desligam

```
int middleSensorFront()
{
    //variable declaration
    float duration = 0;
    float distance = 0;
    float distanceCentimeters = 0;

    //Write a pulse to the HC-SR04 Trigger Pin
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);
```

Figura 52 – Função middleSensorFront()

Na 2ª figura da função, o pino *Echo*, é ativado, e a distância que o sensor depois dá, (que é mostrada na figura seguinte), é calculada com base na fórmula = (tempo/2) * 0.0343, em que o tempo que ele demorou a voltar ao sensor é o tempo de ida e volta, e eu apenas necessito do tempo de ida, logo divido isso por 2, e depois multiplico pelo valor da velocidade do som que é de 343 metros por segundo, e isso dá-me a distância em centímetros de que o sensor se encontra do obstáculo.

Caso a distância seja maior que 400, ele dá *out of range*, pois, o sensor não consegue medir mais que isso, e quando a distância é menor do que 2, este dá *too close*, pois este não consegue medir distâncias menores que 2 centímetros.


```

//Measure the response from the sensor
//in this case we are going to measure the duration with the echo pin is HIGH

duration = pulseIn(echoPinMid, HIGH);

//Determine distance from duration
//We are using 343 meter per second as speed of sound
distanceCentimeters = (duration / 2) * 0.0343;
//return distance;
//Send the Results to the Serial Monitor
Serial.print(F("Distance Front= "));
if (distanceCentimeters >= 400)
{
    Serial.println(F("Out of Range"));
}
else if (distanceCentimeters <= 2 )
{
    Serial.println(F("Too Close!!!"));
}
else

```

Figura 53 - Continuação função middleSensorFront()



The screenshot shows a serial monitor window titled 'COM3' with a red header bar. The window displays the output of an Arduino program, showing distance measurements from six sensors. The data is organized into two groups, each preceded by a time range and the text 'Rotating CW'. The first group (0.05-1.15) shows distances for Front, Left, Right, Middle Back, Left Back, and Right Back sensors. The second group (0.06-1.26) shows distances for Front and Left sensors. The bottom of the window features a status bar with checkboxes for 'Avanço automático de linha' and 'Mostrar marca de tempo', and dropdown menus for 'Nova linha' and '19200 baud'. A 'Limpar saída' button is also present.

```

0.05-1.15
Rotating CW
Distance Front= 93.42 centimeters

Distance Left= 133.31 centimeters

Distance Right= 11.90 centimeters

Distance Middle Back= 33.49 centimeters

Distance Left Back= 27.71 centimeters

Distance Right Back= 22.11 centimeters

0.06-1.26
Rotating CW
Distance Front= 95.53 centimeters

Distance Left= 133.31 centimeters

```

Figura 54 - Output de todas as distâncias dos 6 sensores

Para finalizar, a função, esta apresenta uma mensagem no *Serial Monitor*, da distância a que o obstáculo se encontra do sensor, e devolve esse mesmo valor.

```
{  
    Serial.print(distanceCentimeters);  
    Serial.print(F(" centimeters"));  
    Serial.println();  
    Serial.println();  
    //delay(500);  
}  
//delay(500);  
return distanceCentimeters;  
}
```

Figura 55 - Finalização função *middleSensorFront()*

FUNÇÃO SPEEDCONTROLLER()

Depois de todas as funções para obter os resultados dos sensores, segue-se a função que faz o controlo de velocidade, como mostra a seguinte figura.

Esta função utiliza a ponte h, uma vez que se trata de trabalhar com 1 motor, e vai aplicar no pino do motor traseiro a velocidade do motor, velocidade essa que vai de 0 a 255, que são 8 bits, posteriormente revela uma mensagem para o utilizador no *Serial Monitor*.

```
void speedController()  
{  
    //uses pulse with modulation to control the velocity of the motor, that goes from 0 to 255  
    analogWrite(P3A, 100);  
    Serial.println("The speed of the engine has been reduced");  
}
```

Figura 56 - Função *speedController()*

FUNÇÃO GOLEFT()

A próxima função da lista é a função `goLeft()`, que faz nada mais nada menos do que virar as rodas da frente para a esquerda.

Para que isso aconteça, a ponte h manda sinais elétricos para o motor, neste caso a 1ª coisa que faz é colocar o pino EN12 a *High*, que por sua vez desbloqueia o pino 1A e o pino 2ª, seguidamente com o mesmo sinal manda neste caso para o pino 1A o valor *High*, que significa que está a passar corrente, e para o pino 2A mete a *Low*, e isso faz com que o carro vire para a esquerda.

Por fim, verifica se a distancia do sensor esquerdo, é menor ou igual à distância para evitar obstáculos, deixa de virar para a esquerda, e uma vez que a função está a correr num loop, provavelmente, como detetou um obstáculo á esquerda, e depois deixou de virar, é provável que vire para a direita para se desviar.

```
void goLeft(int distanceLeft, int distanceAvoidObstacle)
{
    Serial.println(" Turns Left");
    digitalWrite(EN12 , HIGH); // Enable 1A and 2A
    digitalWrite(P1A, HIGH); // send + or HIGH signal to P1A
    digitalWrite(P2A, LOW); // send - or LOW signal to P2A
    Serial.println("Going Left");
    if(distanceLeft <= distanceAvoidObstacle)
    {
        stopTurningLeft();
    }
    //digitalWrite(EN12 ,LOW); // Disable 1A and 2A
}
```

Figura 57 - Função goLeft()

FUNÇÃO GORIGHT()

A função `goRight()`, é basicamente uma cópia da função `goLeft`, em que só altera, qual dos pinos fica a *High*, e qual deles fica a *Low*, neste caso para virar para a direita, o pino 1A fica *Low*, e o pino 2A fica *High*.

E por fim a última mudança é mesmo no sensor em que a comparação do `if` está a ser feita, em que no anterior era o esquerdo, e agora é o direito.

```
void goRight(int distanceRight, int distanceAvoidObstacle)
{
    Serial.println(" Turns Right");
    digitalWrite(EN12 , HIGH); // Enable 1A and 2A
    digitalWrite(P1A, LOW); // send + or HIGH singal to P1A
    digitalWrite(P2A, HIGH); // send - or LOW singal to P2A
    Serial.println("Going Right");
    if(distanceRight <= distanceAvoidObstacle)
    {
        stopTurningRight();
    }
    //digitalWrite(EN12 ,LOW);// Disable 1A and 2A
}
```

Figura 58 - Função `goRight()`

FUNÇÃO GOFOWARD() E GOBACK()

Seguidamente, temos as funções `goFoward()`, e a função `goBack()`, em que estas são bastante similares às funções `goLeft()`, e `goRight()`.

A função `goFoward` como o próprio nome indica, movimenta o veículo para a frente.

```
void goFoward()
{
    //Go Foward
    Serial.println(" Rotating CW");
    digitalWrite(EN34 , HIGH); // Enable 1A and 2A
    digitalWrite(P3A, HIGH); // send + or HIGH singal to P1A
    digitalWrite(P4A, LOW); // send - or LOW singal to P2A

    //digitalWrite(EN34 ,LOW);// Disable 1A and 2A
}
```

Figura 59 - Função `goFoward()`

A função `goBack()`, por sua vez, movimenta o veículo para trás.

```
void goBack()
{
    //Go Back
    Serial.println(" Rotating CCW");
    digitalWrite(EN34 , HIGH); // Enable 1A and 2A
    digitalWrite(P3A, LOW); // send + or HIGH singal to P1A
    digitalWrite(P4A, HIGH); // send - or LOW singal to P2A

    //digitalWrite(EN34 ,LOW); // Disable 1A and 2A
}
```

Figura 60 - Função `goBack()`

FUNÇÃO `STOPENGINE()`

A função `stopEngine()`, tem como objetivo, parar o carro, e funciona com o valor do acelerómetro. Nesta função entra 1 valor do tipo *float* como parâmetro, que diz respeito à aceleração, em que se andar para a frente é positiva, e caso esteja a andar para trás é negativa, posto isto o *if*, atua consoante esse valor, e faz as rodas traseiras do carro girarem na direção contrária á que estão a girar no momento, por fim tem um *delay* de 80 milissegundos, e para finalizar desabilita os pinos numa só linha de código, desabilitando o pino EN34, e para finalizar envia uma mensagem ao *Serial Monitor* a dizer que o motor parou.

```
void stopEngine(float acceleration)
{
    if(acceleration > 0)
    {
        //puts the engine in reverse to stop
        digitalWrite(EN34 , HIGH); // Enable 1A and 2A
        digitalWrite(P3A, LOW); // send + or HIGH singal to P1A
        digitalWrite(P4A, HIGH); // send - or LOW singal to P2A
    }
    else
    {
        if(acceleration < 0)
        {
            //puts the engine in foward to stop
            digitalWrite(EN34 , HIGH); // Enable 1A and 2A
            digitalWrite(P3A, HIGH); // send + or HIGH singal to P1A
            digitalWrite(P4A, LOW); // send - or LOW singal to P2A
        }
        delay(80);
        //turns off the engine
        digitalWrite(EN34 , LOW); // Disable 1A and 2A
        Serial.println(" Engine Stopped");
    }
}
```

Figura 61 - Função `stopEngine()`

FUNÇÃO STOPTURNIGLEFT() E STOPTURNINGRIGHT()

As funções, funcionam da mesma forma que a função stopEngine(), uma vez que também “dizem” ao Arduino para parar de fazer qualquer coisa, e a maneira de desabilitar os pinos 1A, e 2ª da função stopTurningLeft() referentes ao motor virar para a esquerda, e os mesmos pinos para virar á direita, poupa em cada 1 das funções 2 linhas de código, pois teria de meter os 2 pinos com o valor “Low”, ou com o mesmo valor para que estes se desativem, poupamos assim espaço de memória, em termos de em vez de termos 4 linhas de código em 2 funções, apenas temos 2 linhas nas 2 funções, como mostra a figura seguinte.

```
void stopTurningLeft()
{
    digitalWrite(EN12 , LOW); // Disable 1A and 2A
}

void stopTurningRight()
{
    digitalWrite(EN12 , LOW); // Disable 1A and 2A
}
```

Figura 62 - Funções stopTurningLeft() e stopTurningRight()

FUNÇÃO RETURNROUTE()

A função returnRoute(), tem como objetivo colocar o carro na rota correta, caso ele seja desviado da rota sem que tivesse um obstáculo á sua frente.

Ou seja quer isto dizer que se o carro involuntariamente se desviasse da rota, esta função iria corrigir o problema.

Esta função utiliza os dados do giroscópio, para saber o quanto o carro se desviou da rota desejada, em que por exemplo o carro é desviado para a esquerda, ele anota qual foi o maior número que a função showGyroData retornou, e guarda, uma vez feito isso, este passa a rodar as rodas no sentido oposto, em que neste caso seria para a direita.

Como é que o carro sabe isso? O giroscópio quando inclinado num determinado ângulo dá um valor positivo, e sendo inclinado no sentido oposto dá um valor negativo, depois é fácil, este deteta se o carro se desviou para cada um dos lados através do ciclo if, guarda o maior, no caso de ter ido para a direita guarda o maior valor, e seguidamente vira as rodas para o lado contrário, e o mesmo acontece se a situação fosse inversa como demonstra a figura seguinte com o código da função.

```

void returnRoute(float angle)
{
    Serial.println(angle);
    if(angle > angleToTurn)
    {
        //saves the max value witch the car turns
        maxAngle += max(angle, angleToTurn);
        //goes back to it's original route
        goRight(distanceRightFront, distanceAvoidObstacle);
    }
    else
        if(angle < simetricAngleToTurn)
        {
            //saves the max value witch the car turns
            minAngle += min(angle, simetricAngleToTurn);
            //goes back to it's original route
            goLeft(distanceLeftFront, distanceAvoidObstacle);
        }
}

```

Figura 63 -Função returnRoute()

FUNÇÃO INITIALIZEGIRO()

Esta função é um pouco mais pequena do que a que se encontra na secção do giroscópio, devido a esta ter pouco código, e para explicar ser mais completa, logo esta corre mais rápido.

```

void initializeGiro()
{
    Wire.begin();
    mpu6050.begin();
    mpu6050.calcGyroOffsets(true);
}

```

Figura 64 -Função initializeGiro()

FUNÇÃO SHOWGIRODATA()

E para finalizar a nossa longa lista de funções, temos a função `showGyroData()`, esta função é a responsável ir buscar o ângulo Z, para que este possa ser utilizado na função `returnRoute()`.

```
float showGyroData()  
{  
    mpu6050.update();  
    Serial.print("\tangleZ : ");  
    Serial.print(mpu6050.getAngleZ());  
  
    return mpu6050.getAngleZ();  
}
```

Figura 65 - Função showGyroData()

Por último, temos a posição do giroscópio no carro, como se pode verificar na seguinte figura.

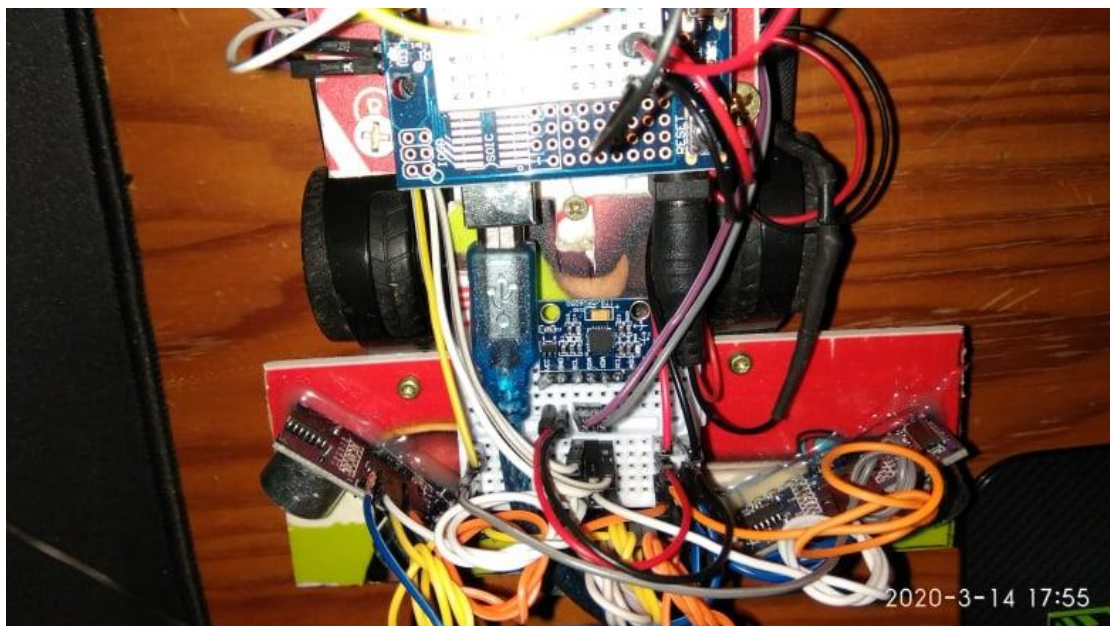


Figura 66 - Posição do giroscópio no carro

Conclusão

Ao realizar este projeto, foi possível para mim aprender a programar em Arduino, uma vez que nunca o tinha feito, o que implica ter uma noção que estamos a desenvolver código para trabalhar diretamente com o Hardware que temos, como por exemplo no caso dos sensores de ultrassons, desenvolver o código e aplicá-lo no carro.

Ao trabalhar com o Arduino, apercebi-me também que o espaço de memória que este tem é bastante limitado, e que a programação que fazemos para utilizar no Arduino, tem de ser o mais simplificado possível, e trabalhar com as bibliotecas do Arduino para que possamos libertar espaço da memória reservada para as variáveis, que acontece quando queremos imprimir uma mensagem escrita por nós no *Serial Monitor*.

Este projeto, fez-me também começar a perceber o mínimo dos mínimos da eletrónica, e ao fazer este projeto, ganhei conhecimento nesta área, aprendi a soldar cabos, a utilizar as breadboards de uma maneira eficaz a reduzir os cabos pois o carro já não tem espaço para mais nada, assim como a pesquisar as informações que necessitava e a consultar os datasheet dos sensores, especialmente do giroscópio para entender como este funcionava internamente.

Com todo o conhecimento que adquiri, enquanto fazia o projeto consegui colocar de novo a funcionar alguns equipamentos elétricos que tinha em casa, que eventualmente se estragaram, em que um dos que mais me deixou feliz, foi colocar 2 carros telecomandados que já não andavam á uns bons 10 anos a funcionar novamente, isto porque o transformador pifou, e tivemos de fazer uma gambiarra para carregar as baterias, após vermos que o problema não seria das baterias como se pensava, ficámos contentes, pois conseguimos meter os carros a trabalhar.

Para finalizar, quero agradecer ao professor Nuno Pereira, o docente da cadeira de Física aplicada á computação, onde pude aprender, e colocar os meus conhecimentos á prova tanto no projeto, como no dia a dia em casa, e por despertar o meu conhecimento pela robótica, pois eu agora tenho uns quantos projetos na cabeça que gostaria de executar por diversão.

Webgrafia

Arduino Uno R3 especificações – consultado a 29/01/2020

<https://store.arduino.cc/arduino-uno-rev3>

Código MPU-6050(Giroscópio) – consultado a 11/02/2020

<https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/>

<https://www.youtube.com/watch?v=UxABxSADZ6U>

Como declarar pino analógico no Arduino – consultado a 11/02/2020

<https://www.arduino.cc/en/tutorial/AnalogInput>

gy-521 datasheet– consultado a 11/02/2020

<https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>

O que é o protocolo I2C– consultado a 26/02/2020

<https://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/>

Função max() Arduino– consultado a 27/02/2020

<https://www.arduino.cc/reference/en/language/functions/math/max/>

Principais eixos de um avião– consultado a 04/03/2020

https://en.wikipedia.org/wiki/Aircraft_principal_axes

<https://www.grc.nasa.gov/www/k-12/airplane/yaw.html>

Velocidade rotacional– consultado a 04/03/2020

https://pt.wikipedia.org/wiki/Velocidade_rotacional