

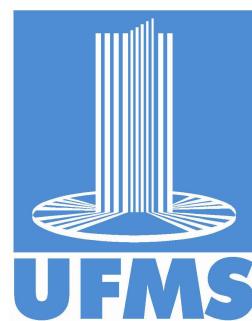
Plano de Trabalho

Controle de velocidade utilizando P.I.D.

Kelvim Rodrigues de Oliveira

Orientação: Prof. Doc. Gedson Faria

Bacharelado em Sistemas de Informação



Universidade Federal de Mato Grosso do Sul
Campus de Coxim

13 de Maio de 2017

Controle de velocidade utilizando P.I.D.

Plano de Trabalho

Coxim, 13 de Maio de 2017.

Capítulo 1

Introdução

Bla bla bla

1.1 Justificativa

Bla bla bla

1.2 Objetivos

1.3 Objeto Geral

O objetivo deste trabalho é estudar e implementar controle de velocidade entre os motores esquerdo e direito do robô jogador.

1.4 Objetivos Específicos

- Criar uma interface simples e intuitiva para que qualquer pessoa sem alto nível de conhecimento no pacote *Vaucanson-G* possa utilizá-lo.
- Desenvolver o sistema de transição livre.
- Desenvolver o sistema de transição dupla.

1.5 Organização da Monografia

Capítulo 2

Revisão da Literatura

Capítulo 3

Futebol de robôs

Este capítulo destina-se alguns conceitos, informações e história do futebol de robôs, suas principais modalidades, dinâmica de jogo e movimentos.

3.1 Uma breve história sobre futebol de robôs

Com duas organizações de referência mundial para o estudo e organização quanto ao futebol de robôs, são elas a FIRA, sigla em inglês para *Federation of International Robot-soccer Association* e a RoboCup associações que tem como objetivo promover pesquisas nas áreas de robótica e inteligência artificial, colaborando na divulgação científica através atividades afim de estimular o interesse dos participantes a resolver problemas.

A FIRA, a qual foi fundada no ano de 1995, pelo professor Kim Jong-Hwan, realizou seu primeiro campeonato mundial na Coréia, em 1997, dois anos após sua fundação, com o objetivo de levar aos leigos e as gerações jovens o espírito da ciência e tecnologia aplicada à robótica. Desde então a FIRA vem realizando diversos eventos em vários países do mundo, incluindo nossa pátria, o que fez com que o futebol de robôs obtivesse reconhecimento mundial. Então o Brasil, que teve a oportunidade de sediar a Copa do Mundo em agosto de 1999, realizada entre os dias 4 a 8, na cidade Campinas no estado de São Paulo.

A RoboCup Criada no Japão por meio da iniciativa de um grupo de pesquisadores que tinham como objetivo em comum, o jogo de futebol, afim de promover o crescimento da ciência e tecnologia. De forma independente, os professores Minoru Asada, da Universidade de Osaka, e a Professora Manuela Veloso junto a seu aluno Peter Stone da Universidade Carnegie Mellon, EUA, estavam também trabalhando em robôs jogadores de futebol. O primeiro campeonato da Robocup junto a primeira conferência, foram realizados em 1997. Onde compareceram mais de 40 equipes e mais de 5.000 espectadores. Infelizmente no Brasil

até o momento não foram realizados eventos da Robocup com âmbito mundial.

3.2 Modalidade

Assim como o futebol jogado por humanos, o futebol de robôs também tem suas regras de jogo bem definidas. A FIRA Cup é um evento organizado com competições várias categorias, destacam-se as categorias *Micro-Robot Soccer Tournament* (MiroSot) e a *Humanoid Robot Soccer Tournament* (HuroSot), ambas são disputadas sob o olhar de um árbitro humano, já a categoria *Simulated Robot Soccer Tournament* (SimuroSot), que como o próprio nome sugere, trata-se de uma modalidade de simulação, quando não se tem a disposição de os recursos de hardware referentes aos robôs.

Este trabalho tem como foco o estudo das velocidades dos motores da categoria MiroSot. O acrônimo MiroSot, vem da denominação *Micro Robot Soccer Tournament*, cuja categoria é voltada para partidas com pequenos robôs que se movimentam através de dois motor, um motor para cada roda, realizando movimentos rápidos, enviados a partir de um computador disposto com uma inteligencia artificial com suas estratégias. Sendo este o ambiente principal do trabalho. A partida deve ser jogada por duas equipes, com três robôs cada, um robô pode atuar como goleiro de acordo com as estratégias, já fora de campo o time contem três membros humanos, o “gerente”, o “técnico” e um “instrutor” os quais só tem acesso ao palco. Cada equipe possui seu computador o qual é principalmente dedicado ao processamento de visão, identificação e de localização. O tamanho dos robôs desta categoria é limitado a 7,5 cm x 7,5 cm x 7,5 cm, altura da antena não deve ser considerada na medição do tamanho de um robô, é utilizado uma bola de golfe de cor laranja afim de facilitar o processo de visão computacional. Esta categoria conta com duas ligas, a “*Large league*”(Liga grande) considerado grande pois o seu campo possui as dimensões de 400cm x 280cm e a “*Middle league*”(Liga média) que tem seu campo menor com as dimensões de 220cm x 180cm.

===== INSERIR IMAGEM DAS DIMENSÕES DO CAMPO =====

Capítulo 4

O robô

O time de futebol de robôs do campus de Coxim é participante da modalidade MiroSot conforme a seção 3.2 esta limitado as dimensões de 7,5 cm x 7,5 cm x 7,5 cm. Dotado de um Arduino nano o qual realiza o controle de todo o robô, um bluetooth do modelo **HC-06** 4.6 que possui apenas o modo *slave*(apenas recebe conexão, não busca a mesma) cujo função principal é estabelecer a conexão entre o robô e o computador da equipe, um driver para motor de corrente continua do modelo **TB6612FNG** 4.2 capais de controlar dois motores, dos quais são utilizados para realizar os movimentos, dois sensores ópticos do modelo **TCRT1000** 4.3 os quais são utilizados em conjunto com as rodas 4.4 então este conjunto trabalha como um encoder, utilizado para determinar a velocidade de rotação das rodas/motor.

4.1 Arduino

Segundo o livro Arduino em ação[1] todas as versões são baseadas em um microprocessador Atmel AVR de 8 bits de arquitetura RISC. O Arduino Nano versão 3.0 utilizado neste projeto, possui um processador ATmega328 provido de uma memória flash de 32 KB no total sendo 2 KB utilizados pelo *bootloader* deixando disponível 30 KB, com uma velocidade de clock de 16 MHz. Contendo uma porta mini-USB integrada a qual serve para troca de dados e também como fornecimento de energia. O Arduino Nano tem a opção de receber energia também pelo pino 30(VIN) em tensões de 6 a 20 volts e/ou pelo pino 27(5V) uma tensão regulada de 5,5 volts, caso estejam as três fontes de energias conectadas a placa seleciona a de maior tensão para seu funcionamento. Componente de grande utilidade, capacidade, de pequenas dimensões -18 mm x 45 mm- e pesando apenas 7 gramas o tornam ideal para um robô jogador de futebol.

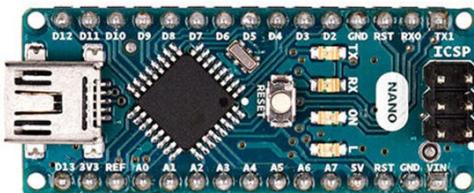


Figura 4.1: Vista superior de um Arduino Nano. Fonte:store.arduino.cc

4.2 Controlador para motores TB6612FNG

Segundo o *datasheet* disponibilizado pela empresa Toshiba, o **TB6612FNG** é um driver que pode controlar até dois motores de corrente continua. Com o total de 16 pinos, dois pinos são entrada, sendo eles **IN1** e **IN2** dos quais podem ser usados para controlar uma dentre as quatro modos de operação !!CW, CCW, short-brake, and stop!!.. Quatro pinos de saída -**A01**, **A02**, **B02** e **B01**- para os motores, podendo ser controlados separadamente. A velocidade de cada motor pode ser controlado por um sinal de **PWM** nos seus respectivos pinos **PWMA** e **PWMB**, o driver possui um pino **STBY**, abreviação do inglês *standby* que pode ser traduzido para espera, ou seja, modo de espera, para retira-lo do modo de espera deve-se colocar a tensão deste pino em alta (*HIGH*). O pino **VCC** é destinado a suprir a demanda de energia do driver, sendo operacional com tensões de 2,7 até 5,5 volts de corrente continua, já o pino **VM** é destinado a suprir a demanda de energia dos motores, tendo seu limite máximo de 15 volts em corrente continua.

!!Board comes with all components installed as shown. Decoupling capacitors are included on both supply lines. All pins of the TB6612FNG are broken out to two 0.1" pitch headers; the pins are arranged such that input pins are on one side and output pins are on the other. Note: If you are looking for the SparkFun Motor Driver with headers, it can be found here or in the Recommended Products below.!!

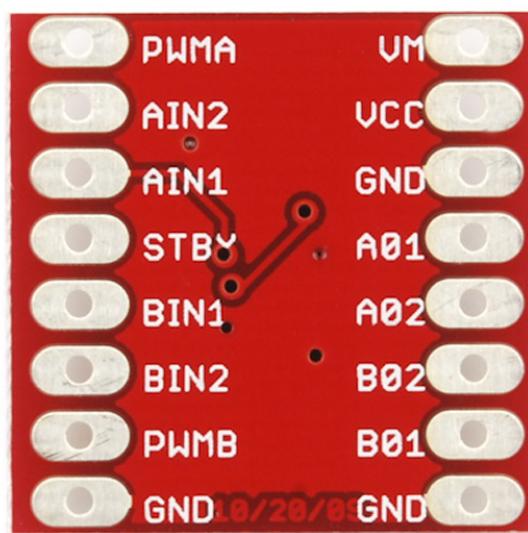


Figura 4.2: Vista inferior do driver TB6612FNG com os rótulos de seus respectivos pinos.
Fonte:www.sparkfun.com

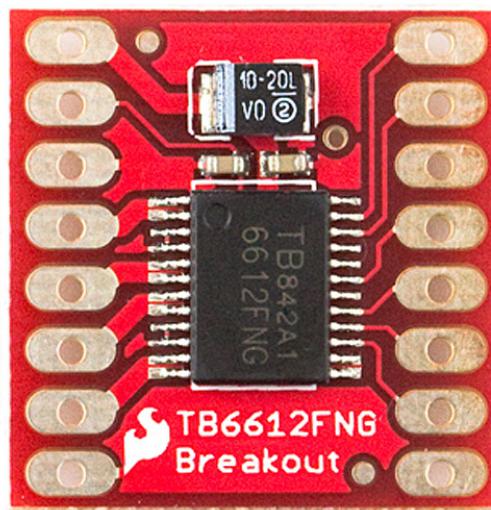


Figura 4.3: Vista superior do driver TB6612FNG. Fonte:www.sparkfun.com

4.3 Sensor Óptico TCRT1000

O sensor óptico utilizado neste trabalho é do modelo **TCRT1000**, segundo a documentação da empresa *Vishay* obtidas no sitio www.vishay.com, o sensor refletivo é um conjunto de, um emissor infravermelho com um fototransistor, inseridos em um invólucro de

chumbo o qual bloqueia a luz visível evitando erros nas leituras no fototransistor, um sensor de pequenas dimensões medindo apenas 7mm x 4mm x 2.5mm e de boa precisão. Como pode ser visto na imagem 4.5 o robô possui dois sensores para cada roda, mas neste trabalho foi usado apenas um sensor para cada roda -os conectados aos pinos com interrupção 7.1-, sendo os sensores mais externos da roda.

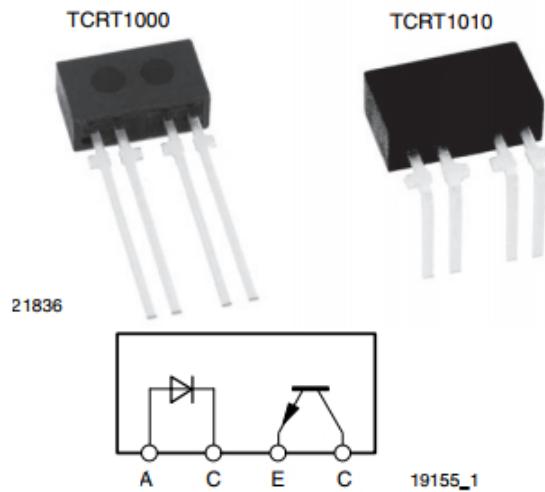


Figura 4.4: Sensor óptico TCRT1000. Fonte:www.vishay.com

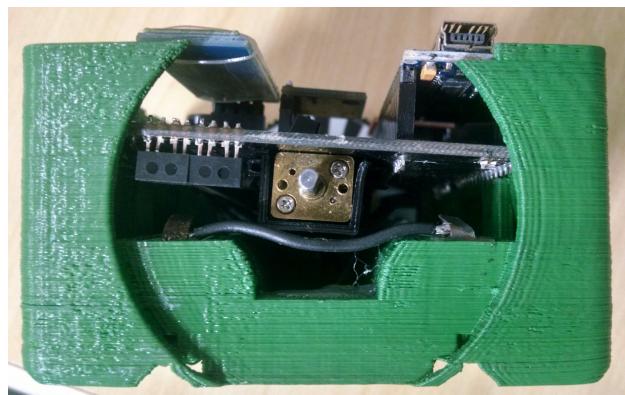


Figura 4.5: Vista lateral do robô Modu sem a roda, carcaça verde feita em impressora 3D, dois sensores ópticos e eixo de motor

4.4 Roda

O robô Modu possui apenas um par de rodas de !!XX mm!! e uma faixa de marcadores, utilizados juntamente com os sensores ópticos 4.3 transformando todo o conjunto em um encoder 4.5 descrito na próxima sessão, tendo 17 marcadores brancos e 17 marcadores preto totalizando 34 marcadores como pode ser visto na figura 4.7.



Figura 4.6: Roda com fita de 32 marcadores



Figura 4.7: Roda com fita de 34 marcadores



Figura 4.8: Roda com fita de 38 marcadores

4.5 Encoder

Segundo *Victor Adriano Turchetti* em seu trabalho de conclusão de curso pela universidade estadual de Campinas (UNICAMP) em 2007 um encoder é um componente que pode converter movimento linear ou angular em sinais digitais, dos quais geralmente são utilizados para determinar a posição, velocidade e em alguns casos dependendo da configuração do encoder pode-se também determinar a direção de rotação do sistema. O funcionamento do encoder empregado neste trabalho é a de configuração mais simples apenas um sensor óptico, o qual realiza apenas a contagem de marcadores da fita que esta colada junto a roda conforme a figura 4.7, convertendo o movimento angular das rodas em sinais digitais, estes sinais são utilizados como sinais de entrada para as interrupções 7.1, realizando a contagem do numero de pulsos em determinado tempo.

4.6 Bluetooth HC-06

O modulo foi utilizado para comunicação sem fio entre o robô e o computador, as informações são trocadas utilizando do protocolo *Serial*, o alcance do módulo segue o padrão da comunicação bluetooth, aproximadamente 10 metros, o suficiente, visto que o campo não passa pouco dos dois metros. Uma característica deste modelo (HC-06) possui apenas o modo *slave* neste modo é apenas permitido receber conexão de outros dispositivos, ou seja, não permite que este modelo busque e solicite conexão com outros dispositivos, assim como

ocorre com fones de ouvido e caixas de som.

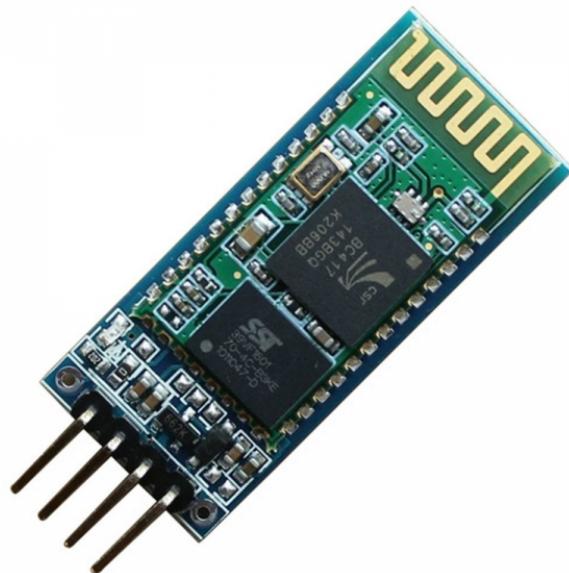


Figura 4.9: Vista superior de um componente bluetooth modelo HC-06.
Fonte:<http://buildbot.com.br>

Capítulo 5

Ambiente de desenvolvimento

Este capítulo aborda as ferramentas utilizadas para o desenvolvimento da programação dos robôs jogadores. O sistema operacional escolhido foi o Ubuntu Gnome 17.04 64-bit, por ser uma distribuição linux o Ubuntu já possui os drivers USB para comunicação a placas de teste, a utilizada no projeto foi a placa Arduino Nano 328, com o driver USB, como o FT232RL. Sera descrito também neste capítulo a instalação e configuração dos softwares: IDE Arduino, IDE Clion 2017.1 e PlatformIO.

5.1 Arduino IDE

===== <https://www.arduino.cc/en/main/software> === O Arduino 1.8.3(IDE) é um software open-source para desenvolvimento de programas para Arduino. A utilização deste software torna fácil escrever códigos e subir(upload) para a placa. O Arduino IDE tem suporte para Windows, Mac OS e para Linux. Ambiente foi desenvolvido em Java baseado no Processing e outros softwares open-source, com a utilização da IDE qualquer placa Arduino pode ser utilizada com poucas configurações.

5.1.1 Intalação

Na pagina oficial de download <https://www.arduino.cc/Main/Software> foi realizado o download conforme o sistema operacional, neste caso, Linux x64, ao selecionar o download é possível fazer uma doação para os desenvolvedores da IDE, visto que é um projeto open-source e software livre, não tendo fins lucrativos, o mesmo depende dessa colaboração da comunidade para permanecer ativo e continuar expandido. Então realizado o download da versão atual, neste momento a versão 1.6.10 para linux.

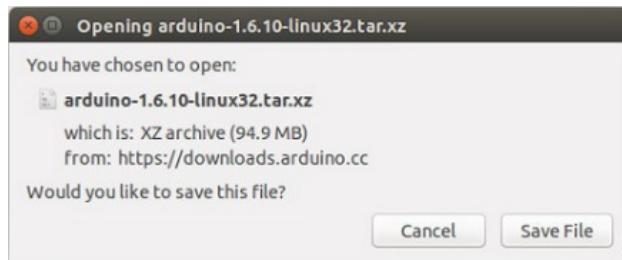


Figura 5.1: Título.

Ao finalizar o download do arquivo compactado, será necessário descompactar em um diretório desejado, este ainda não será o diretório de instalação.

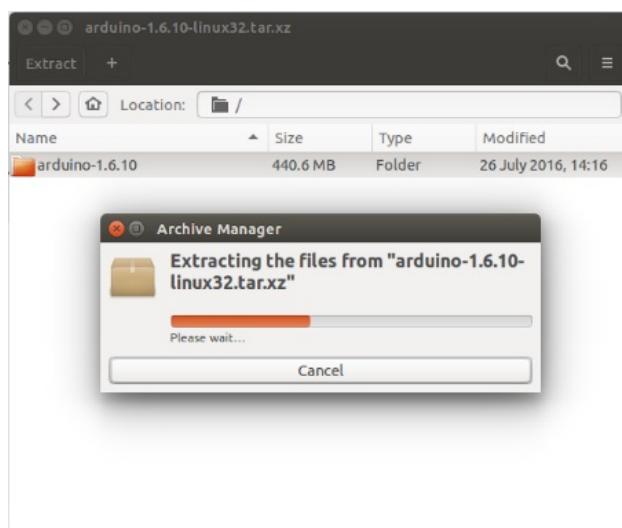


Figura 5.2: Título.

Abrindo o terminal navegue pelos diretórios utilizando o comando ‘CD’ até o diretório arduino-1.6.x o qual foi criado ao descompactar o arquivo. Execute então o arquivo ‘intall.sh’ utilizando o comando:

```
1 ./install.sh
```

O processo de instalação é rápido, um novo ícone foi criado na área de trabalho.

```
osboxes@osboxes:~/Downloads/arduino-1.6.10
osboxes@osboxes:~$ ls
Arduino  Documents  examples.desktop  Pictures  Templates
Desktop  Downloads  Music          Public    Videos
osboxes@osboxes:~$ cd Downloads
osboxes@osboxes:~/Downloads$ cd arduino-1.6.10
osboxes@osboxes:~/Downloads/arduino-1.6.10$ ./install.sh
Adding desktop shortcut, menu item and file associations for Arduino IDE... done!
!
osboxes@osboxes:~/Downloads/arduino-1.6.10$
```

Figura 5.3: Titulo.

5.2 Clion 2017.1

===== https://www.jetbrains.com/clion/ ===== CLion é uma IDE multi-plataforma, criada pela empresa JetBrains para desenvolvimento de softwares nas linguagens C e C++, uma ferramenta poderosa. Suporte as linguagens nativas C e C++, incluindo C++11, C++14, libc++ e mais. A ferramenta desenvolvimento conta com ótimos recursos tais como: navegação, geração de código, gerenciamento de bibliotecas entre outros.

5.2.1 Instalação

Na página oficial de download do Clion “https://www.jetbrains.com/clion/download/#linux” foi realizado o download do arquivo “CLion-*.tar.gz” da versão compatível com o sistema operacional. Descompacte o arquivo “CLion-*.tar.gz”, neste projeto foi descompactado no diretório /opt, sendo assim foi utilizado o seguinte comando para descompactar direto no diretório desejado.

```
1 sudo tar xf CLion-*.tar.gz -C /opt/
```

Ao finalizar a descompactação, novamente navegue até o diretório /bin, diretório este que se encontra junto aos arquivos descompactados, utilizando o comando:

```
1 cd /opt/CLion-*/bin
```

Execute o arquivo clion.sh que se encontra dentro do subdiretório /bin, com o comando:

```
1 ./clion.sh
```

5.3 PlatformIO

Diferente microcontroladores normalmente exigem diferentes ferramentas de desenvolvimento, para o Arduino temos a Arduino IDE. Outros usuários preferem ferramentas com auxílios. Tais ambientes de desenvolvimentos como o Eclipse, o qual trás recursos que ajudam a gerenciar seus projetos, bibliotecas e funções de autocompletar. As vezes é um pouco difícil manter-se na linha com diferentes microcontroladores e ferramentas. Então o ecossistema (como é chamado pelos seus desenvolvedores) open-source do PlatformIO junta tudo em uma única ferramenta. Sendo uma ferramenta multiplataforma podendo ser instalada nos sistemas operacionais Linux, Windows e MAC, dá suporte ao compilação para mais de 200 placas de teste, com mais de 15 plataformas de desenvolvimento e 10 frameworks, cobrindo então as placas mais populares do mercado, fazendo o trabalho duro de organização de centenas de projeto e bibliotecas que podem ser incluídas no projeto.

5.3.1 Instalação

A instalação ou atualização no MAC e Linux é feita pelo terminal de um modo muito fácil, apenas utilizando o comando(sudo pode ser requerido):

```
1 python -c "$(curl -fsSL https://raw.githubusercontent.com/platformio/platformio/master/scripts/get-platformio.py)"
```

5.4 Iniciando um novo projeto

Para criar um novo projeto primeiramente devemos abrir a IDE Clion, deve-se criar um novo projeto na linguagem C++ clicando no File, New Project então em Create. Ao finalizar a criação do novo projeto, abra as opções de configurações utilizando o atalho de teclado “Ctrl + Alt + S” clique na aba “Plugins” no campo de texto procure por Arduino, conforme a figura abaixo. Caso não seja localizado no repositório local clique em “Search in repositories” para procurar nos repositórios da JetBrains.

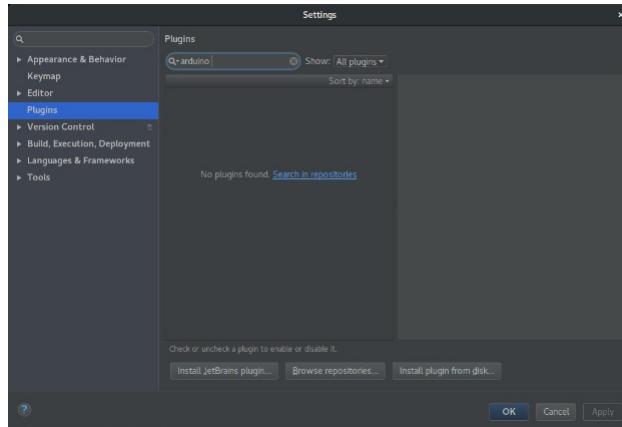


Figura 5.4: Título.

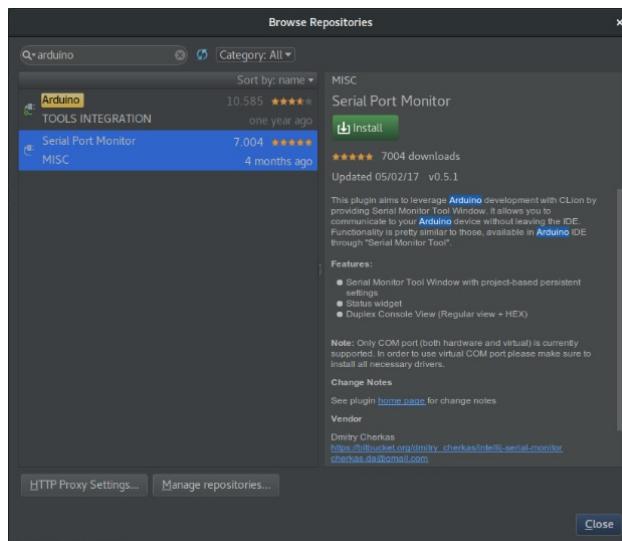


Figura 5.5: Título.

Após o final da instalação dos dois plugins reinicie o CLion para os plugins venham funcionar. Com a IDE aberta, no terminal, navegue com o comando “CD” até o diretório do projeto o qual acabou de ser criado. Neste projeto foi criado a pasta TCCTerceira no diretório de projetos do CLion o qual se encontra na home do linux.

```
1 cd /home/kelvimiro/CLionProjects/TCCTerceira
```

Ainda no terminal é utilizado o comando “platformio” com a opção “boards”, este comando lista as placas de teste suportadas pelo PlatformIO.

```
1 platformio boards
```

Ao localizar a placa desejada na lista, Arduino Nano processador Atmel 328, a qual foi escolhida para ser utilizada no desenvolvimento dos robôs de futebol do time UFMS-CPCX.

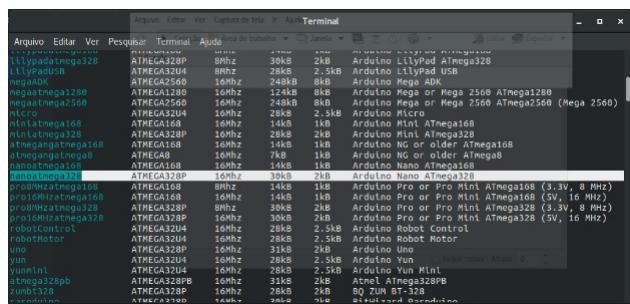


Figura 5.6: Titulo.

Localizado e copiado o nome da placa “nanoatmega328”, utilize o comando “platformio” com as opções de “init” para iniciar/instalar os recursos necessários no projeto, a opção “ide” deve-se ser utilizada o parâmetro “clion”indicando que utilizamos o framework para trabalhar junto a IDE do Clion, sendo assim o comando completo utilizado foi:

```
platformio init --ide clion --board nanoatmega328
```

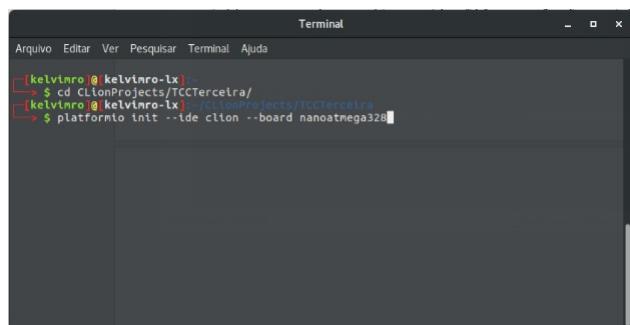


Figura 5.7: Titulo.

Finalizando a inicialização do PlatformIO será necessário recarregar o arquivo CMake. Para isso clique com o botão direito do mouse em cima do nome do projeto(TCCTerceira) e em seguida em “Reload CMake Project”.

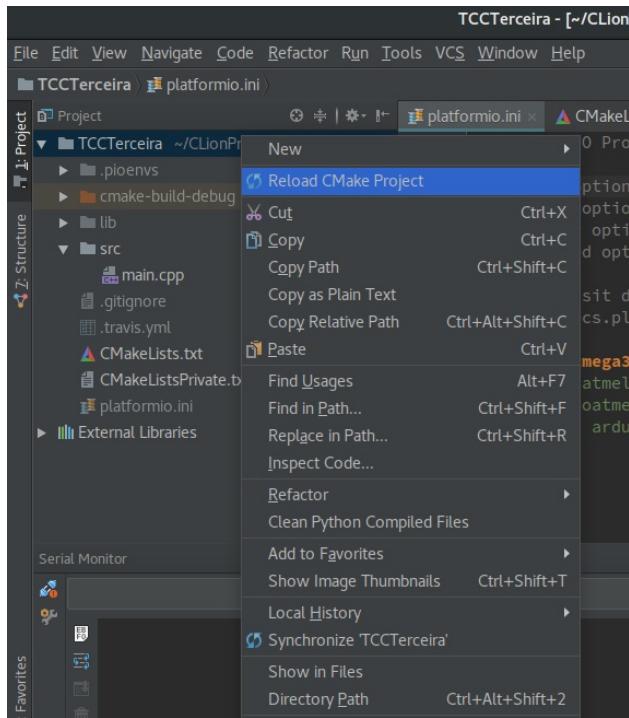


Figura 5.8: Titulo.

O PlatformIO requer que as classes, assim como a Main.cpp, deve estar dentro da sub-pasta “src” pois a ferramenta é configurada nativamente para buscar e compilar os códigos que ali estão. Na classe Main.cpp deve-se fazer o importe da biblioteca “Arduino.h” por meio do “#include” a qual é a sintaxe para importar em C.

```
1 #include <Arduino.h>
```

O código a seguir foi utilizado para teste de compilação e funcionamento do ambiente como um todo no fim da configuração. O programa tem a função básica, de piscar o led embutido na placa do arduino (LED_BUILTIN) ou seja o pino da porta 13, ficando 1 segundo acesso e 1 segundo apagado.

```
1 #include <Arduino.h>
2 void setup() {
3 // inicializa o pino digital LED_BUILTIN como saída.
4 pinMode(LED_BUILTIN, OUTPUT);
5 }
6
7 // Função loop do arduino
8 void loop() {
9 digitalWrite(LED_BUILTIN, HIGH);      // liga o LED
10 delay(1000);                      // espera um segundo
```

```

11 digitalWrite(LED_BUILTIN, LOW);      // desliga o LED
12 delay(1000);                      // espera um segundo
13 }

```

Compilando o código utilizando Ctrl + F9 ou clicando no ícone no canto superior direito

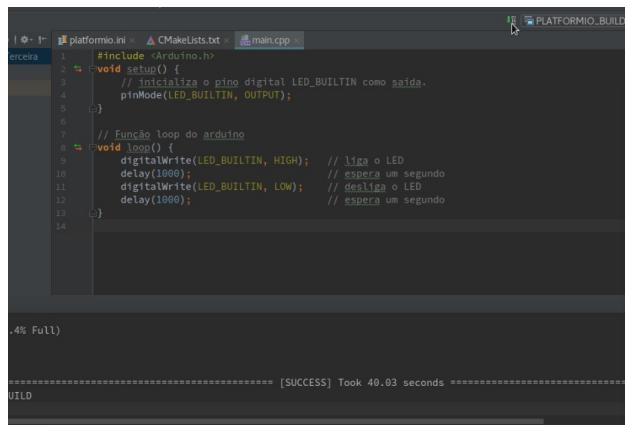


Figura 5.9: Titulo.

Ao receber a mensagem de “SUCCESS” na aba “Messages”, indica que o “build” ou seja, a compilação do programa foi concluída com sucesso. No fim a compilação (build) é hora de subir(upload) do código para a placa arduino mudando para opção PLATFORMIO_UPLOAD (na caixa de seleção no canto superior direito) e clicando no ícone “PLAY” verde.

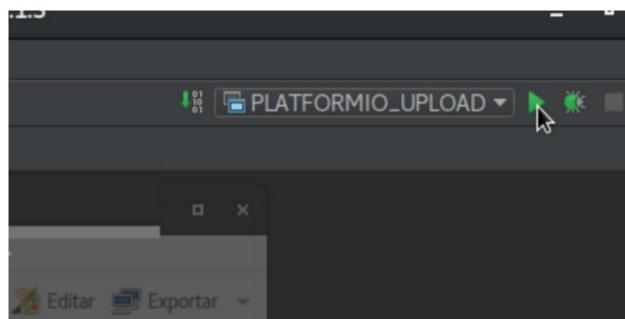


Figura 5.10: Titulo.

Aguarde a próxima mensagem de sucesso, a qual indicando que o programa já foi completamente enviado a placa, então o código já rodando. ATENÇÃO: Algumas distribuições podem solicitar a permissão de administrador para ter acesso ao dispositivo USB antes de fazer o upload do código. O privilégio pode ser concedido com a utilização do comando:

```
1 sudo chmod 777 /dev/ttyUSB0
```

Ps.: O numero “0” em ttyUSB0 deve ser trocado conforme a porta ao qual o dispositivo foi conectado.

Ps2.: Em algumas distribuições as permissões devem ser dadas sempre o dispositivo for conectado/reconectando. Finalizando assim as configurações do ambiente de desenvolvimento, que foi utilizado para gerenciar e editar os códigos neste trabalho de conclusão.

Capítulo 6

Controle Bluetooth

Um simples projeto foi desenvolvido também em Arduino. O projeto tem a finalidade de ser um controle BT(Bluetooth) com um Joystick, utilizado para controlar as direções do robô. O joystick, um dos componentes do controle, nada mais é que dois potenciômetros, um potenciômetro para os valores do eixo X e outro para os valores do eixo Y. O controle possui um componente BT para estabelecer uma troca de dados junto ao robô, o qual também possui seu componente BT, o qual será explicado no capítulo 6.1. O código atualizado pode ser encontrado no link: <https://github.com/kelvimro/TCCTerceira/blob/master/lib/ControleBT/controleBT.cpp>. O controle tem o intuito de simular os comandos que são gerados pela inteligência artificial no software que é responsável pelas táticas e movimento dos jogadores durante uma partida. Os comandos podem variar entre -100 até 100, tendo um valor para cada motor, ou seja cada computação realizada pela inteligência artificial é enviado dois números inteiros, onde o primeiro número é destinado ao motor esquerdo e o segundo ao direito. Utilizando-se da função map() a qual proporciona os valores do potenciômetro - dos quais variam entre 0 a 1023 - aos valores válidos para o robô.

```
1 if (cmdX >= 0 && cmdX <= 501) cmdX = map(cmdX, 0, 501, -100, 0);  
2 else cmdX = map(cmdX, 502, 1023, 0, 100);  
3 if (cmdY >= 0 && cmdY <= 509) cmdY = map(cmdY, 0, 509, -100, 0);  
4 else cmdY = map(cmdY, 510, 1023, 0, 100);
```

O potenciômetro Y, o qual vai representar os valores de potência, em posição neutra (sem ação do controlador) encontra-se com valor analógico de 509, então mapeado para 0, ou seja, não há potência requerida. Ao movimentar o joystick para frente, aumenta-se o valor analógico do potenciômetro até o máximo de 1023 que após ter seu valor mapeado representa o valor 100. Movimentando o joystick para trás, os valores analógicos lido diminuem até 0, denotando o valor -100.

```
1 if (cmdY >= 0 && cmdY <= 509) cmdY = map(cmdY, 0, 509, -100, 0);  
2 else cmdY = map(cmdY, 510, 1023, 0, 100);
```

Valores positivos no eixo Y indicam que o movimento solicitado é para frente, enquanto os valores negativos representam movimento para trás. O potenciômetros X, o qual vai representar os valores da diferença entre os motores. Em sua posição neutra tem o valor analógico lido de 502, representando 0% de diferença entre os motores. Afim de facilitar a compreensão chamaremos de A o motor do lado esquerdo e de B o motor do lado direito. Então ao movimentar o joystick para a esquerda o valor analógico lido diminui até 0, resultando no valor de -100, movimentando o joystick para a direita aumentando o valor analógico lido até 1023 que representa o valor 100.

```
1 if (cmdX >= 0 && cmdX <= 501) cmdX = map(cmdX, 0, 501, -100, 0);  
2 else cmdX = map(cmdX, 502, 1023, 0, 100);
```

No eixo X os valores negativos indicam que o motor A tem de se movimentar X% mais lento em relação ao motor B, fazendo o movimento de curva para o lado de A. Já com os valores positivos de X indica que o motor B deve ser X% mais lento q o A fazendo a curva para o lado B.

6.1 Bluetooth HC-05

Capítulo 7

Interrupções

O processador do Arduino Nano, o qual foi utilizado neste projeto é do modelo Atmega328 e nesta seção discorreremos sobre o funcionamento da interrupção neste processador, utilização no projeto, exemplos e !!efeitos colaterais!!.. Em sua maioria os processadores possuem interrupções dos quais permitem executar algo após determinado evento externo, sendo chamado de interrupções externas. Nas referencias online presentes no site oficial do Arduino, para mais informações a cerca de interrupções é indicado as notas de Nick Gammon!!(<https://gammon.com.au/interrupts>)!!, de um modo didático e simples ele faz um paralelo ao cotidiano, ao fazer o jantar e ter de cozinhar as batatas, é colocado um despertador com o tempo de 20 minutos, indo faze outra atividade durante o tempo de cozimento, ao despertar o alarme vc “interrompe” sua atividade e então termina o jantar. Apesar do exemplo utilizado ser quase uma mudança de estado, não é recomendado invocar funções grandes e complexas pois além de sair da linha principal de execução, as funções não contem argumentos, utilizamos então variáveis do tipo voláteis. Logo recomenda-se as interrupções para trocas de *flag*, disparo de funções emergenciais, contagens precisas entre outros a interrupção é o “botão vermelho”, veremos a seguir que *reset* é a interrupção de maior prioridade.

7.1 Rotinas de serviço de interrupção (ISR)

As rotinas de serviço de interrupção conhecidas pela sigla ISR que vem do inglês *Interruption Service Rotines* é uma rotinha a qual se encontra fora da função **loop**, ou seja, é uma função a qual será executada quando ocorrer uma interrupção, interrompendo a execução do código **loop** onde quer que ele esteja, retornando ao ponto de “pausa” ao fim da ISR. Algo de suma importância é citado por Simon Monk em seu livro, Programação com Arduino vol. 2

“Quando uma ISR está sendo executada, as interrupções são automaticamente desligadas. Isso evita a confusão que poderia ser causada se as ISRs interrompessem umas as outras.”

Ou seja, enquanto uma ISR estiver ativa outras não serão executadas, exceto uma interrupção de maior prioridade como o *reset*.

7.2 Modos de Interrupção

Os chamados modos de interrupção é como gatilho, irá disparar a ISR, ou seja, uma ISR é uma função a ser executada sempre quando o gatilho é acionado. O Atmega328 possui quatro modos de interrupção, são eles:

LOW - Dispara uma interrupção continuamente enquanto o pino selecionado estiver em nível baixo (*low*).

RISING - Dispara uma interrupção sempre que o pino passar de baixo (*low*) para alto (*high*).

FALLING - Dispara uma interrupção sempre que o pino passar de alto (*high*) para baixo (*low*).

CHANGE - Dispara uma interrupção sempre que o pino mudar de nível, em ambos os sentidos.

Vale ressaltar que o modelo Arduino Due tem um quinto modo de interrupção, o *HIGH*.

HIGH - Dispara uma interrupção continuamente enquanto o pino estiver em nível alto (*high*), assim como o modo *LOW*.

7.3 Interrupção no Atmega328

Assim como em outros processadores o Atmega328 possui sua lista de interrupções, logo a baixo podemos ve-la em ordem de prioridade, tendo em primeiro lugar -maior prioridade- a interrupção de *reset*.

Como podemos na tabela acima as interrupções externas nomeadas de **INT0_vect** e **INT1_vect** que correspondem aos pinos digitais 2 e 3 respectivamente, no Arduino Nano(os pinos de interrupção podem mudar de acordo com a placa) sendo as maiores prioridades seguidas do *reset*, aproveitamos para observar as interrupções de 7 a 17, as quais são interrupções de tempos e a interrupção de prioridade 25 a qual é a interrupção para uso da interface serial.

1	Reset	
2	External Interrupt Request 0 (pin D2)	(INT0_vect)
3	External Interrupt Request 1 (pin D3)	(INT1_vect)
4	Pin Change Interrupt Request 0 (pins D8 to D13)	(PCINT0_vect)
5	Pin Change Interrupt Request 1 (pins A0 to A5)	(PCINT1_vect)
6	Pin Change Interrupt Request 2 (pins D0 to D7)	(PCINT2_vect)
7	Watchdog Time-out Interrupt	(WDT_vect)
8	Timer/Counter2 Compare Match A	(TIMER2_COMPA_vect)
9	Timer/Counter2 Compare Match B	(TIMER2_COMPB_vect)
10	Timer/Counter2 Overflow	(TIMER2_OVF_vect)
11	Timer/Counter1 Capture Event	(TIMER1_CAPT_vect)
12	Timer/Counter1 Compare Match A	(TIMER1_COMPA_vect)
13	Timer/Counter1 Compare Match B	(TIMER1_COMPB_vect)
14	Timer/Counter1 Overflow	(TIMER1_OVF_vect)
15	Timer/Counter0 Compare Match A	(TIMER0_COMPA_vect)
16	Timer/Counter0 Compare Match B	(TIMER0_COMPB_vect)
17	Timer/Counter0 Overflow	(TIMER0_OVF_vect)
18	SPI Serial Transfer Complete	(SPI_STC_vect)
19	USART Rx Complete	(USART_RX_vect)
20	USART, Data Register Empty	(USART_UDRE_vect)
21	USART, Tx Complete	(USART_TX_vect)
22	ADC Conversion Complete	(ADC_vect)
23	EEPROM Ready	(EE_READY_vect)
24	Analog Comparator	(ANALOG_COMP_vect)
25	2-wire Serial Interface (I2C)	(TWI_vect)
26	Store Program Memory Ready	(SPM_READY_vect)

Tabela 7.1: Lista de interrupções para o chip processador Atmega328. Fonte: gammmon.com.au/interrupts

7.4 Utilizando interrupção no Arduino

Nesta seção foi como exemplos os códigos utilizados no robô no desenvolvimento do trabalho, conforme visto na seção anterior 7.3 os pinos de interrupção são os pinos digitais 2 e 3. Declaramos então no código as variáveis das quais irá representar os pinos, utilizando o código.

```
1 const int encodPinA1 = 2; // encoder A pino 2
```

```
2 const int encodPinB1 = 3; // encoder B pino 3
```

Dentro da função *setup* !!intanciamos!! os pinos como *INPUT* pois estes pinos farão as entradas de informação ao pino de interrupção.

```
1 pinMode(encodPinA1, INPUT);
2 pinMode(encodPinB1, INPUT);
```

Segundo as refeccias disposta na documentação do site oficial do Arduino temos três sintaxes diferentes para acoplar/iniciar, porem dentre elas apenas uma é recomendada pelos desenvolvedores, sendo assim foi utilizado a sintaxe recomendada.

```
1 attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

A função **digitalPinToInterrupt(pin)** é utilizada para traduzir o pino digital para o numero específico da interrupção. ISR 7.1 é o nome da função a ser chamada quando a interrupção ocorrer, lembrando que esta função não deve conter argumentos. *Mode* define o modo de quando a interrupção deve disparar, conforme descrito na seção 7.2. Seguindo então a sintaxe recomendada para iniciar nossa interrupção com o pino e função desejada.

```
1 attachInterrupt(digitalPinToInterrupt(encodPinA1), rencoderA, CHANGE);
2 attachInterrupt(digitalPinToInterrupt(encodPinB1), rencoderB, CHANGE);
```

Como pode ser visto no código acima, a função com nome de **rencoderA** (ISR) será executada sempre que seu valor do pino **encodPinA1** mudar, pois seu modo foi definido como *CHANGE*, o mesmo acontece com o pino **encodPinB1** chamando sua respectiva ISR nomeada como **rencoderB**.

```
1 void rencoderA() {
2     countA++;
3 }
4 void rencoderB() {
5     countB++;
6 }
```

Do mesmo modo que podemos acoplar uma interrupção a um pino específico, também podemos desacoplar-lo utilizando o código a seguir sempre que achar necessário.

```
1 detachInterrupt(digitalPinToInterrupt(encodPinA1));
2 detachInterrupt(digitalPinToInterrupt(encodPinB1));
```

O código exemplificado acima tem a função de desativar a interrupção do pino **encodPinA1** e do pino **encodPinB1** exclusivamente.

Existe um modo alternativo de ativar e ativar/desativar as interrupções, mas estas funções ativam/desativam todas as interrupções - exceto *rest* -

```
1  interrupts();      // Ativa interrupções  
2  noInterrupts();    // Desativa interrupções
```

Lembrando que ao desativando as interrupções utilizando-se da função acima citada, as interrupções citadas na tabela 7.1 como a *Serial* também são desativadas. Sendo muito útil quando nos trechos de códigos sensíveis ao tempo, no código a seguir desativamos as interrupções para evitar que os valores da variável contadora sejam alterados durante a execução, evitando erros de leitura quanto as velocidades das rodas.

```
1  noInterrupts();      // Desarma interrupts  
2  calibA.push(countA); // Adiciona a ultima contagem a fila de amostras  
3  countA = 0;          // Zera contadores de marcos do encoder  
4  calibMillis = millis(); // Zera o contador de tempo  
5  interrupts();        // Arma interrupsts
```

Capítulo 8

Controlador PID

== <https://www.citisystems.com.br/controle-pid/> == Neste capítulo, explanarei sobre o controle PID -que é abreviatura para Proporcional, Integral e Derivativo- e como utiliza-lo em aplicações de controle que requerem um controle de precisão. Começaremos pelo exemplo onde um carro tenta manter uma distância X do carro da frente.

8.1 P - Proporcional

Ao tentar manter uma distância X o carro de trás deve acelerar proporcionalmente quando o carro da frente acelerar e começar a distanciar, tentando alcançar o mesmo. No entanto, caso o carro de trás venha acelerar mais que o carro da frente a distância será menor que a desejada, X, então a nova correção é frear para aumentar a distância buscando a distância desejada de X. Novamente caso a correção, que desta vez é a frenagem(desaceleração), for muito, a sua distância se será superior a distância desejada de X, então novamente terá de acelerar. Isso ocorreria indefinitivamente caso a aceleração não proporcional não esteja correta, ou seja, acelerando de mais e freando de menos e não atingindo o ponto alvo(*set point*). Então esta aceleração é chapada de ganho ou proporcional(P) em um controle PID.

8.2 I - Integral

A integral, neste exemplo dos carros, seria como recuperar a distância X a cada instante, caso o veículo da frente venha a acelerar, o veículo de trás acelera gradualmente afim de atingir a distância X com mais suavidade, este procedimento pode ser comparado a integral de um controle PID.

8.3 D - Derivativo

A derivada é utilizada para eliminar erros acumulados na integral, neste exemplo dos carros, seria quando o carro de trás percebe que a distância desce ou decresce muito rapidamente em relação a desejada X, deixando as correções da integral ainda mais suaves, diminuindo a oscilação das correções em volta do Setpoint.

8.4 Utilização do PID

O controle do tipo PID foi empregado para controlar a velocidade da roda mais rápida em relação a mais lenta, como pode ser visto até o então, o controle PID se utiliza de dados obtidos durante a sua execução, no caso deste trabalho o tempo de um ciclo foi de 200 milissegundos(ms), ou seja, a cada 200 ms o controle obtém a velocidade de rotação de ambas as rodas por meio do encoder 4.5 utilizando a velocidade do motor mais lento como *set point* do motor mais rápido, calculando a proporcional, a integral e derivando, utilizando o resultado como o novo valor do PWM do motor mais rápido o deixando com a velocidade de rotação mais próxima do motor de referência.

8.4.1 Problemas no ciclo

Durante o emprego do controle PID para equalizar as velocidades dos motores foi percebido que o tempo de ciclo para o controle se tornar eficiente é superior o intervalo de comandos recebidos pelo computador de controle o qual envia novos comandos a cada 200ms. O controle do tipo PID não consegue equalizar as velocidades na primeira correção, então ao se aproximar da segunda ou terceira correção o robô recebe a próxima instrução mudando todas as referências de velocidades.

8.4.2 Problemas devido a interrupções

Como pode ser visto o encoder 4.5 é o conjunto da roda 4.4 provida de marcadores e dos sensores ópticos ??, para realizar a contagem de marcadores os sensores ópticos estão conectados nos pinos digitais 2 e 3, se utilizando da função de interrupção para contagem dos marcadores e determinar as velocidades, veremos no próximo capítulo, mais precisamente na seção de rotinas de interrupção 7.1, que ao gerar uma interrupção todo o sistema “para” temporariamente, trazendo discrepância nas contagens devido a sobreposição de interrupções. Concluindo então que a utilização de controle do tipo PID do modo implementado

não é recomendado.

Capítulo 9

Controle por pesos

Um controle por peso é de simples funcionamento, cada motor tem seu peso o qual é multiplicado pelo seu PWM solicitado pelo computador controlador dos robôs, com a diferença dos pesos os motores mesmo com potencia real diferente tendem a rodarem com uma velocidade parecida. Para atribuir valores aos pesos, foram feito os !!X!! passos começando com coleta de amostras, calculando a média das amostras adquiridas e então calculando os pesos.

9.1 Coleta de amostras - `getAmostras(int _PWM)`

Foi desenvolvido um método chamado `getAmostras(int _PWM)` tendo como argumento o PWM desejado para coleta de amostras. O método então coleta o numero de amostras determinado na variável **NUM_AMOSTRA** de escopo global como pode ser visto na linha 2 do código abaixo, também podemos verificar a variável **CALIBMILLIS** a qual determina intervalo de tempo para cada amostras. Afim de evitar erros na contagem devido a interrupções 7.1 as contagens são executadas separadamente, ou seja, primeiro é colhido as amostras do motor A e depois colhida as amostras do motor B.

```
1 // Numero de amostras para a media movel
2 int NUM_AMOSTRA = 15;
3 // Tempo de coleta de amostras
4 const int CALIBMILLIS = 1000;
```

Sendo usado a configuração acima, a contagem de impulsos gerados pelo encoder 4.5 é realizada durante um intervalo de 1000 milissegundos, gerando então uma amostra. Logo abaixo o código da função na integra, a variável *print* é do tipo *String* e de escopo global,

utilizada para armazenar informações a serem impressas posteriormente.

```

1 void getAmostras(int _PWM) {
2     print = "PWM =\t";
3     print += _PWM;
4                                     // https://www.arduino.cc/en/Reference/NoInterrupts
5     noInterrupts();                      // Desativa interrupções
6     print += "\nA:\n";
7     PWM_valA = _PWM;                    // PWM_valA é o PWM do motor A de 0 a 255
8     PWM_valB = 1;                      // PWM_valA é o PWM do motor A de 0 a 255
9     motorRefresh();
10    countB = countA = 0;
11    static double calibMillis;          // calibMillis timer de loop
12    calibMillis = millis();
13    interrupts();                     // Reativa intrrupções
14    while (PWM_valA >= 5) {
15        for (int i = 0; i < NUM_AMOSTRA; ++i) { // Controla numero de amostras
16            // Enquanto encoder conta durante CALIBMILLIS, faça nada ;
17            while ((millis() - calibMillis) <= CALIBMILLIS) {}
18            noInterrupts();                  // Desarma interrupts
19            calibA.push(countA); // Adiciona a ultima contagem a fila de amostras
20            print += countA;
21            print += "\n";
22            countA = 0;                   // Zera contadores de marcos do encoder
23            calibMillis = millis();       // Zera o contador de tempo
24            interrupts();               // Arma interrupt
25        }
26        PWM_valA = PWM_valB = 2;         // Reduz vel. mas mantem sentido
27        motorRefresh();
28    }
29    noInterrupts();                  // Desativa interrupções
30    // Set PWM - config inicial
31    print += "\nB:\n";
32    PWM_valA = 1;
33    PWM_valB = _PWM;                // PWM_valA é o PWM do motor A de 0 a 255
34    motorRefresh();
35    countB = countA = 0;
36    calibMillis = millis();
37    interrupts();
38    while (PWM_valB >= 5) {
39        for (int i = 0; i < NUM_AMOSTRA; ++i) {

```

```

40           // Enquanto conta durante CALIBMILLIS, faça nada :)
41   while ((millis() - calibMillis) <= CALIBMILLIS) {}
42       noInterrupts();                                // Desarma interrupts
43       calibB.push(countB);
44       print += countB;
45       print += "\n";
46       countB = 0;                                     // Zera contadores de marcos do encoder
47       calibMillis = millis();
48       interrupts();                                 // Arma interrupt
49   }
50   PWM_valA = PWM_valB = 2;
51   motorRefresh();
52 }
53 Serial.print(print);                            // Imprime o conteudo da variavel print
54 print = " ";                                    // Esvazia variavel print
55 if (processMedia()) {} // Invoca o metodo responsavel por calcular a media
56 }
```

Como podemos ver no código acima a linha 1 é a declaração do método de retorno *void*, sem retorno. Excluindo as linhas da variável *print* da explicação visto que a mesma é apenas impressão de informações, podemos observar nas linhas 5, 18, 29 e 42 o comando **noInterrupts()**; e nas linhas 13, 24, 37 e 48 o comando **interrupts()**; ambos explicados na sessão 7.4. As linhas 12, 23, 36 e 47 pode ser visto o comando **millis()**; este tem como retorno o numero em milissegundos dês de que a placa Arduino começou a rodar o programa, sendo assim nas linhas citadas é como se marcasse o horário da execução naquele instante e gravados na variável **calibMillis**. Os PWM são utilizados como critérios de parada da função *while* nas 14 e 38, sendo alterados apenas quando o *for* da linha 15 obter o numero de amostras determinados em **NUM_AMOSTRA**. As duas estruturas de repetição *while*, nas linhas 17 e 41 é utilizadas como uma espera, a fim de evitar a função **delay()**; a qual se utiliza de interrupções 7.1 para executar a espera, sendo assim não há execução de nenhum código até que o tempo atual **millis()** subtraindo o ultimo tempo atual -anotado na linha 12- seja maior que o tempo de coleta determinado na variável **CALIBMILLIS**, após decorrido tempo de **CALIBMILLIS** em milissegundos, é desativado as interrupções na linha 18 afim de não aumentar a contagem de marcadores, esta contagem a qual é salva na variável **contA**, após desativar as interrupções é adicionado a uma estrutura de fila chamada **calibA** a ultima contagem de **contA** conforme a linha 19 logo após salvar a ultima contagem este contador é zerado na linha 22, anotando-se o novo tempo atual na linha 23 e reativando as interrupções na linha 24, este processo será executado até obter o numero desejado de amostras devido a função *for* da linha 15, ao fim, obtendo o numero total de amostras os PWMs do motor

A e B são setados para 2, afim de para-los então chamado o metodo **motorRefresh()**; o qual simplesmente atualiza os valores de PWM nos motores como pode ser visto no código abaixo, finalizando a coleta de amostras para o motor A.

```

1 void motorRefresh() {
2     PWM_valA *= pesoA;
3     PWM_valB *= pesoB;
4     PWM_valA = int(PWM_valA);
5     PWM_valB = int(PWM_valB);
6     analogWrite(PWMA, PWM_valA);
7     analogWrite(PWMB, PWM_valB);
8 }
```

O mesmo processo ocorre para aquisição de amostras para o motor B entre as linhas 29 e 52, desativando interrupções, determinando o valor do PWM de B para o PWM a ser testado, zerando os contadores, salvando o tempo atual, fazendo as esperas no *while* assim como ocorreu no motor A. Ao final da coleta das amostras de ambos os motores, antes de retornar a função, um método chamado **processMedia()** é invocado para calcular a média das 15 amostras de cada motor.

9.2 Média de amostras - processMedia()

A função cujo tem por finalidade computar a média para cada roda, sempre será executada que a função **getAmostras(int)** for invocada, utilizando-se das amostras já obtidas conforme a sessão 9.1.

```

1 boolean processMedia() {
2     static double _soma;
3     _soma = 0;
4     static double _med;
5     _med = 0;
6     static int _mark;
7     _mark = 0;
8     print = "\nMédia\t";
9     while (!calibA.isEmpty()) {
10         _soma += calibA.pop();
11         _mark++;
12     }
13     _med = _soma / _mark;
```

```

14     mediaA.push(_med);
15     print += _med;                                     // Imprime média de A
16     _soma = 0;
17     _med = 0;
18     _mark = 0;
19     while (!calibB.isEmpty()) {
20         _soma += calibB.pop();
21         _mark++;
22     }
23     _med = _soma / _mark;
24     mediaB.push(_med);
25     print += "\t";                                     // Imprime indicadores e média de B
26     print += _med;
27     delayMicroseconds(999);
28     Serial.println(print);
29     return true;
30 }
```

A função começa declarando três (`_soma`, `_med` e `_mark`) variáveis locais, pois são utilizadas apenas dentro da função, as variáveis são utilizadas para auxiliar nos cálculos, declaradas como estáticas afim de economizar memória, utilizando-se da mesma variável sempre que a função for chamada. Ao estanciar as variáveis nas linhas 2, 4 e 6 e zeradas nas linhas 3, 5 e 7 inicia-se o processo de calculo da média propriamente dita na linha 9 onde o critério de parada da estrutura de repetição `while` é a função `.isEmpty()` uma função disponível da estrutura de dados da variável **calibA** e **cablibB** que são do tipo fila, esta função `.isEmpty()`- retorna verdadeiro (`true`) caso a fila esteja vazia, sendo assim enquanto a fila **calibA** não estiver vazia soma-se o atual valor de `_soma` ao valor retirado da fila utilizando a função `.pop()` conforme a linha 10, então incrementando 1 o valor de `_mark` na linha 11, a variável `_mark` tem por finalidade contar o numero exato de amostras retiradas da fila, ou seja, sempre uma amostra for retirada da fila é incrementado em um ao marcador. Assim que todos as amostras forem retiradas da fila e somadas a variável responsável por armazenar temporariamente o valor da média a variável `_med` recebe a divisão da `_soma` pelo `_mark`, o numero de elementos contados, tendo então a média das amostras obtidas, com o valor computado em mãos este é inserido em outra fila chamada de **mediaA** por meio da função `.push()` passando `_med` como argumento, conforme a linha 14. Ao finalizar o calculo da média das amostras do motor A o mesmo ocorre com as amostras do motor B, para isso inicia-se zerando as três variáveis auxiliares locais nas linhas 16, 17 e 18, passando pelo mesmo processo de retirada da fila, soma, divisão e adicionando a pilha de médias de B, como pode ser visto entre as linhas 19 e 24 tendo como diferença apenas as variáveis responsareis para o calculo do motor

B.

9.3 Determinação dos Pesos - finalMedia()

A função chamada **finalMedia()** é o último passo a ser executado na calibragem dos pesos para os motores. Esta tem como principal objetivo determinar qual o motor mais lento e usa-lo como referência dando-lhe peso 1, sendo assim o motor de maior velocidade terá o peso com o valor inferior ou igual a um (no caso da diferença entre os motores seja irrelevante), isto é feito utilizando o !!!nome do conteúdo matemática(media alguma coisa)!!! onde a velocidade do motor maior(**mtMaior**) multiplicado por um valor **X** é igual a velocidade do motor mais lento **mtMenor**:

$$mtMaior \cdot X = mtMenor$$

$$X = \frac{mtMenor}{mtMaior}$$

Assim determinamos o fator multiplicador que aproxima o valor superior do valor inferior, um peso a ser multiplicado pela potência solicitada. O seu código segue abaixo para melhor entendimento.

```

1  boolean finalMedia() {
2      static double _soma;
3      _soma = 0;
4      static double _med;
5      _med = 0;
6      static int _mark;
7      _mark = 0;
8      while (!mediaA.isEmpty()) {
9          _soma += mediaA.pop();
10         _mark++;
11     }
12     _med = _soma / _mark;
13     pesoA = _med;
14     _soma = _med = 0;
15     _mark = 0;
16     while (!mediaB.isEmpty()) {
17         _soma += mediaB.pop();
18         _mark++;
19     }
20     _med = _soma / _mark;
21     pesoB = _med;

```

```

22
23     if (pesoA > pesoB) {
24         pesoA = pesoB / pesoA;
25         pesoB = 1;
26         return true;
27     } else if (pesoA < pesoB) {
28         pesoB = pesoA / pesoB;
29         pesoA = 1;
30         return true;
31     } else if (pesoA == pesoB) {
32         pesoA = pesoB = 1;
33         return true;
34     } else {
35         // Log de erro em caso de cmd não localizado
36         Serial.println("#### PESOS não mapeados #####");
37         Serial.print(pesoA);
38         Serial.print(" <- pesoA pesoB -> ");
39         Serial.print(pesoB);
40         return false;
41     }
42 }
```

Ao fim da execução desta função as variáveis globais **pesoA** e **pesoB** estarão com seus respectivos pesos, deixando os motores menores diferenças de velocidade entre si.

9.4 Função de teste - getPercent()

A função de teste é muito similar a função **finalMedia()** tendo apenas duas diferenças básicas. A primeira diferença é a divisão não será a menor velocidade dividido pela maior e sim a velocidade do motor A dividido pela velocidade do motor B, sendo assim, sempre será calculado a proporcionalidade da velocidade do motor A em relação a velocidade do motor B, imprimindo seu valor proporcional em percentagem(%), tendo um valor superior a 100% quando a velocidade do motor A for mais rápida que a velocidade do motor B e inferior a 100% quando a velocidade menor. A segunda diferença é que o resultado dos cálculos são apenas apresentados para o estudo das diferenças, ou seja, os valores não são salvos nas variáveis de peso e sim salvas nas variáveis locais e temporárias **_pA** e **_pB**.

```

1 boolean getPercent() {
2     static double _soma;
```

```
3     _soma = 0;
4     static double _med;
5     _med = 0;
6     static int _mark;
7     _mark = 0;
8     while (!mediaA.isEmpty()) {
9         _soma += mediaA.pop();
10        _mark++;
11    }
12    _med = _soma / _mark;
13    static double _pA;
14    _pA = _med;
15    _soma = _med = 0;
16    _mark = 0;
17    while (!mediaB.isEmpty()) {
18        _soma += mediaB.pop();
19        _mark++;
20    }
21    _med = _soma / _mark;
22    static double _pB;
23    _pB = _med;
24    delayMicroseconds(2000);
25    print = "\nPerc\t";
26    print += (_pA / _pB * 100);
27    print += "%";
28    Serial.println(print);
29 }
```

Conforme visto a cima, a função não altera nenhuma variável com influencia nas configurações dos motores, apresentando apenas uma proporção entre os motores.

Capítulo 10

Utilizando as funções

Neste capítulo será explanado como as funções **getAmostras(int) ??**, **processMedia()** 9.2, **finalMedia()** 9.3 e **getPercent()** 9.4 foram utilizadas para obter os resultados apresentados neste trabalho.

10.1 Diferença de motores

Com o intuito de confirmar a necessidade de uma calibragem para os motores e comprovar que os motores são tem diferentes velocidades mesmo sendo produzidos pelo mesmo fabricante e de modelos iguais, foi utilizado a função **getAmostras(int) ??** com a configuração de 15 amostras sendo 1 segundo de exposição cada amostra e então utilizado a função **processMedia()** 9.2 para obter uma média para cada motor, sem utilizar a função **finalMedia()** 9.3 (a função responsável por atualizar os pesos) foi utilizado diretamente a função **getPercent()** 9.4 a fim de determinar em % a diferença entre os motores sem qualquer tipo de controle, resultando em um valor em percentagem(%) e plotando um único ponto no gráfico. Este processo comparativo foi executado 15 vezes em ambos os motores em cada valor de PWM e os valores de PWM testados foram 100, 150, 200 e 255, gerando o gráfico abaixo, onde cada linha colorida representa um valor de PWM, cada ponto deste gráfico tem em sua coordenada (X,Y) sendo X o valor percentual da velocidade do motor A em relação ao motor B e o valor de Y o numero do teste iniciando no teste de numero 1 até o de numero 15.

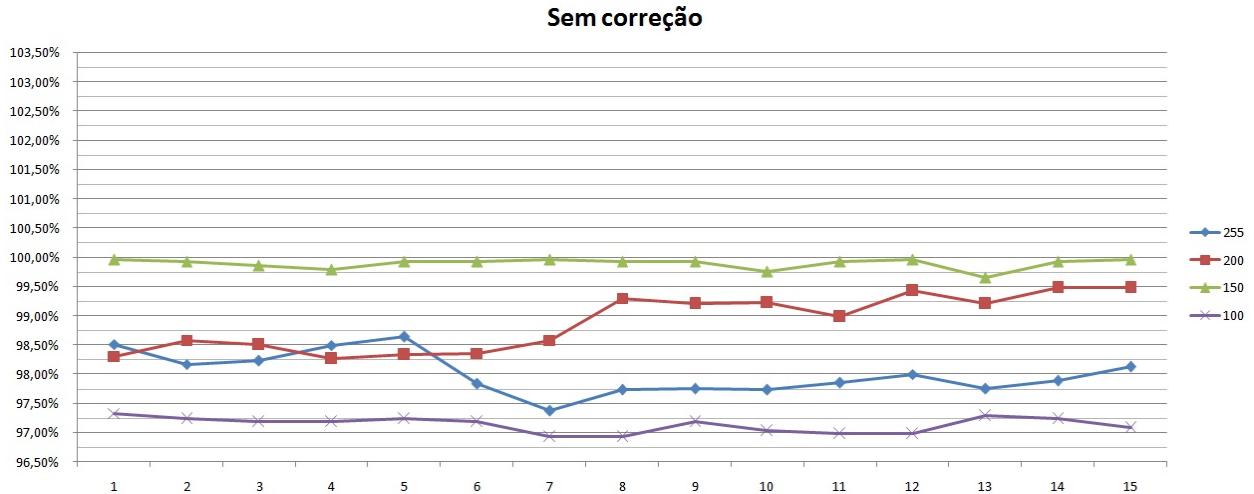


Figura 10.1: Velocidades do motor A em relação ao motor B em %, quinze comparações para cada valor de PWM. Valores de PWM testados: 100, 150, 200 e 255

Como podemos ver no gráfico acima, todos os valores mesmo os mais próximos a 100% estão abaixo de 100%, ou seja, em todos os testes e nas quatro potências (PWMs) testadas o motor A permanece com uma velocidade inferior a velocidade do motor B, comprovando a necessidade de controle e também prova a diferença entre os motores mesmo sendo “iguais”. Também podemos observar que as diferenças no PWM se mostram maiores.

Com os resultados acima 10.1 apresentados foi proposto dois métodos para o cálculo dos pesos. O primeiro 10.1.1 utilizando apenas o PWM mais discrepante na obtenção de amostras 9.1, neste trabalho o PWM 100, o segundo método 10.1.2 é a utilização dos quatro valores de PWM no momento das coletas de amostras 9.1.

10.1.1 Pesos com PWM 100

Para calcular os pesos utilizando apenas como referência o PWM 100, foi utilizado a função `getAmostras(100)` quatro vezes, tendo como resultado a seguinte tabela 10.1. Lem-

Amostra #	Motor A	Motor B
1	127.87	131.93
2	128.73	132.27
3	128.80	132.53
4	129.07	132.60

Tabela 10.1: Tabela com a média de quatro amostras coletadas utilizando a função `getAmostras(100)`.

brando que a função `getAmostras(int _PWM)` 9.1 utiliza a função `processMedia()` 9.2

ao fim de cada execução, ou seja, a função geradora de média é executada quatro vezes gerando os as médias acima 10.1 apresentadas. Ao fim das coletas passamos para o passo ao qual efetivamente calcula-se os pesos utilizando-se da função **finalMedia()** 9.3, como explicado a função determina o somando-se as médias de velocidade obtidas do motor A dividindo-a pelo numero de amostras(quatro), somando-se as médias de velocidade do motor B e também dividindo pelo numero de amostras, tendo as duas médias do somatório da média é realizado uma proporção com uma regra de três básica, assumindo o menor valor como o valor máximo(100%), então obtemos os pesos, conforme os dados do estudo o peso do motor A, representado pela variável **pesoA** o valor de 1 (100%) e a variável **pesoB** representando o valor do peso do motor B com o valor de 0.97 (97%).

```
1 void motorRefresh() {  
2     PWM_valA *= pesoA;  
3     PWM_valB *= pesoB;  
4     PWM_valA = int(PWM_valA);  
5     PWM_valB = int(PWM_valB);  
6     analogWrite(PWMA, PWM_valA);  
7     analogWrite(PWMB, PWM_valB);  
8 }
```

Com os pesos determinados sempre o programa fizer uma alteração nos valores de PWM dos motores, utilizando a função **motorRefresh()** acima descrita, este será multiplicado pelo seu respectivo peso antes de alterar efetivamente a velocidade. Sendo assim o robô calibrado com as configurações de peso acima, ao selecionar o PWM de valor 100 para ambos os motores o valor determinado para o motor A será de 100 porem o valor do motor B será de 97.

Resultados calibragem de unico PWM

Ao utilizar a calibragem acima citada, a qual utiliza apenas o PWM de valor 100 na tomada de amostras para calculo dos pesos, teve-se o seguinte resultado para os quatro PWMs (100, 150, 200 e 255).

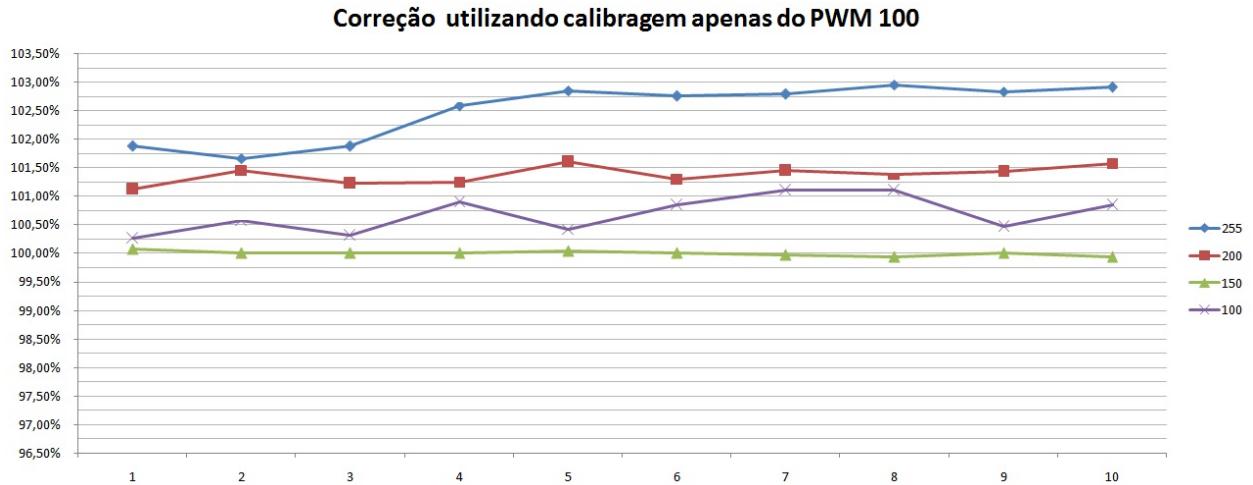


Figura 10.2: Velocidades do motor A em relação ao motor B em %, quinze comparações para cada valor de PWM. Valores de PWM testados: 100, 150, 200 e 255

No gráfico acima 10.2 podemos ver claramente que o motor A passou a ser sempre mais rápido que o moto B, apenas mundo o erro de lado, o PWM de valor 150 passa a ser o mais preciso.

10.1.2 Pesos com quatro PWMs

A fim de aumentar a precisão das correções, foi utilizado quatro valores de PWM na fase de coleta de amostras 9.1, tendo como resultado a seguinte tabela 10.2. Ao fim da coleta

Amostra #	PWM	Motor A	Motor B
1	255	347.40	355.33
2	200	270.13	274.67
3	150	198.87	202.80
4	100	129.73	133.00

Tabela 10.2: Tabela com a média de quatro amostras coletadas utilizando a função `getAmostras(100); getAmostras(150); getAmostras(200); getAmostras(255)`.

de amostras utilizando os quatro valores de PWM citados e ao fim da execução da função `finalMedia()` 9.3 temos os valores de 1(100%) para o `pesoA` e o valor de 0.98(98%) para o `pesoB`. Assim como citado na sessão anterior os valores de PWM solicitados ao robô é multiplicado pelo seu peso antes de ser aplicado.

Resultado calibragem com quatro valores PWMs

Utilizando-se da configuração acima citada a qual foi utilizado as quatro amostragens para o calculo do peso temos o seguinte resultado.

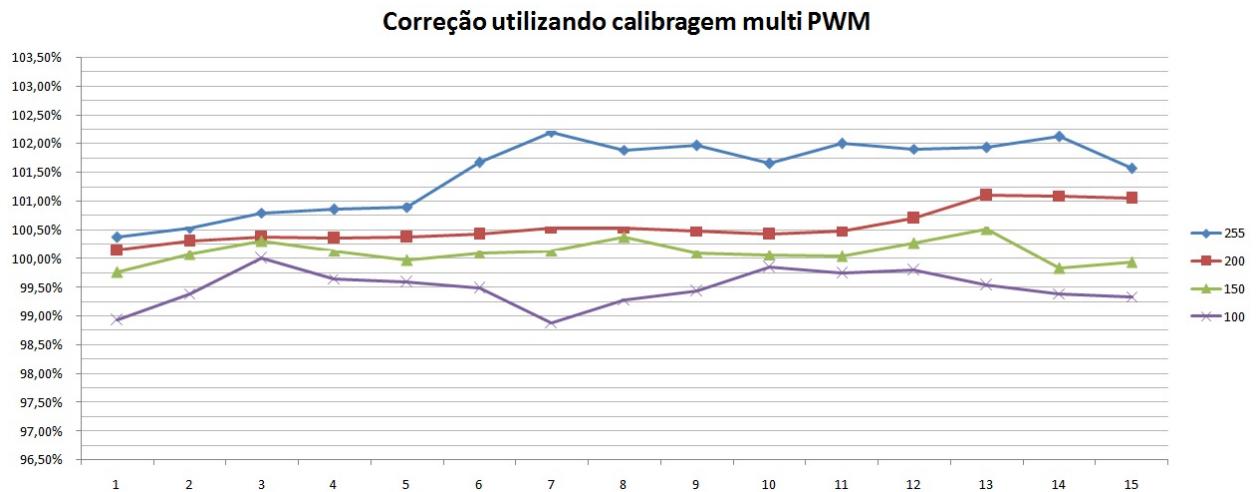


Figura 10.3: Velocidades do motor A em relação ao motor B em %, quinze comparações para cada valor de PWM. Valores de PWM testados: 100, 150, 200 e 255

Como podemos ver no gráfico acima o PWM 150 permanece mais estável que os demais valores tendo alguns testes abaixo de 100% e outros acima. Notamos também que o robô se tornou mais equilibrado, tendo alguns valores de PWM com o motor A mais rápido e outros valores tendo o motor B como mais rápido, deixando assim o robô mais equilibrado.

Capítulo 11

Conclusão

Com base nos estudo realizados até o momento é fato que os motores necessitam de uma pequena correção para trabalharem de forma equivalente em relação as suas velocidades. Podemos também concluir pela analise do gráfico 10.1 que cada regime de potencia requer uma correção diferente e também que os regimes de grande potencia (PWMs acima dos 200, sabe-se que o maior valor é 255) se mostram menos estáveis. Analisemos então os quatro regimes de potencia (PWMs) separadamente, começando com o PWM de valor 100.

Nos gráficos a seguir a linha azul (\tilde{N} Calib \diamond) refere-se aos valores obtidos nos testes 10.1 sem calibragem, a linha vermelha (Mono Calib \square) aos valores obtidos nos testes 10.2 utilizando apenas o PWM 100 para calibragem e por ultimo a linha verde (Multi Calib \triangle) representando os valores nos testes 10.3 o qual foi utilizado quatro valores de PWM para a calibragem.

11.1 Analise do PWM 100

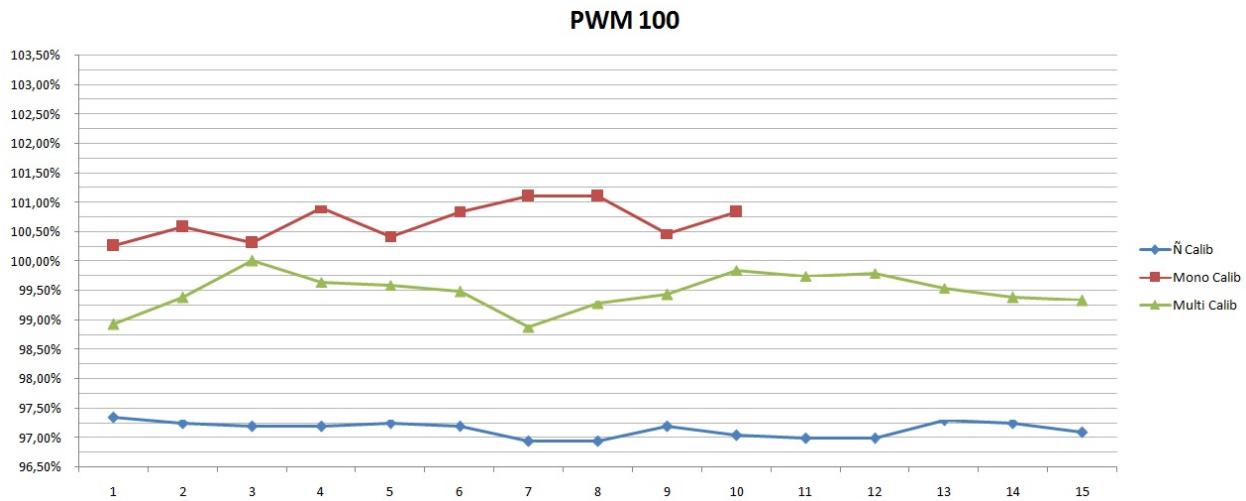


Figura 11.1: Velocidades do motor A em relação ao motor B em %, quinze comparações para o valor de PWM 100.

Com os dados deste gráfico podemos notar que o PWM 100 se mostra estável tendo em sua maior variação em apenas 1% entre o teste 1 e 3, pode-se ver que as correções com um(linha vermelha) ou quatro valores de PWM(linha verde) na calibragem são eficientes, para equalização das velocidades dos motores tendo diferença de apenas 1% para o motor A ou para o motor B, o qual sem correção (linha azul) apresentou um erro superior a 2%.

11.2 Analise do PWM 150

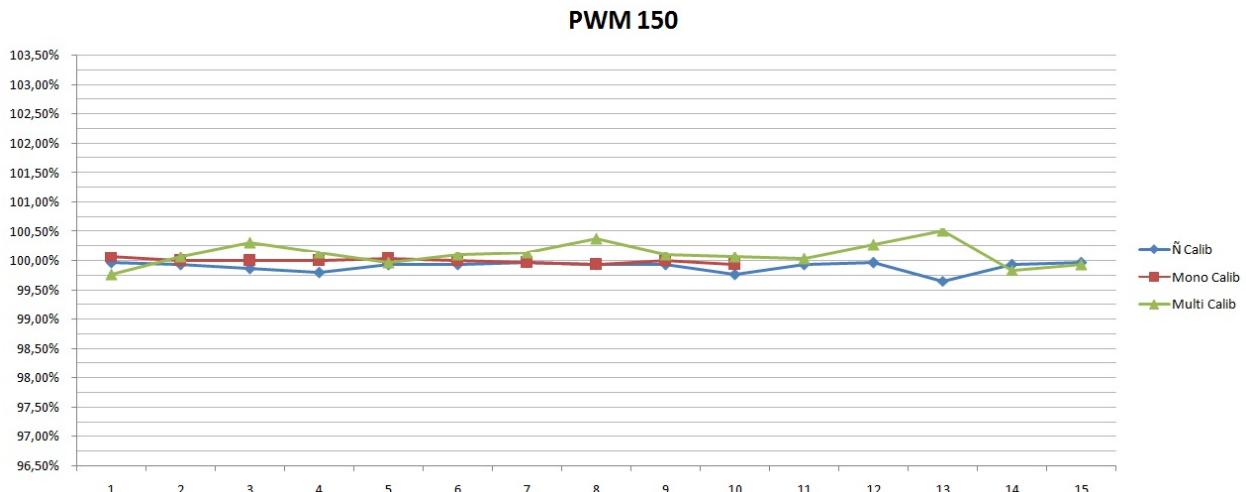


Figura 11.2: Velocidades do motor A em relação ao motor B em %, quinze comparações para o valor de PWM 150.

Analisando separadamente os regimes de potencia podemos notar que o valor de PWM 150 se mostra o a potencia mais estável quando comparado as velocidades, no gráfico 11.2 podemos ver claramente que em nos três casos a diferença de velocidades dos motores não foi superior a 0,5% para mais ou para menos, tornando a correção quase desnecessária para este valor de PWM ou valores próximos a ele.

11.3 Analise do PWM 200

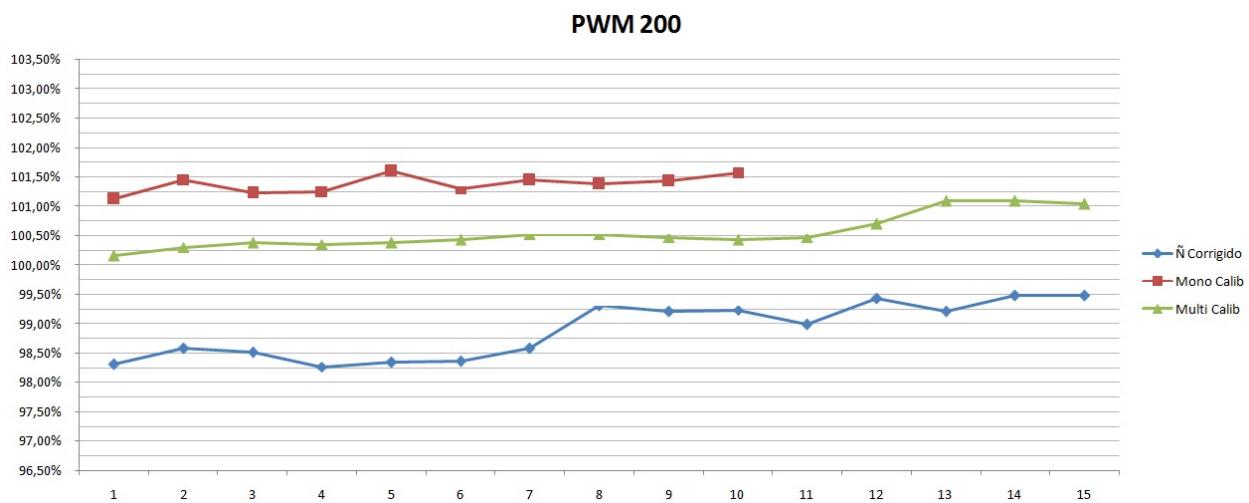


Figura 11.3: Velocidades do motor A em relação ao motor B em %, quinze comparações para o valor de PWM 200.

O gráfico acima 11.3 o qual projeta os três testes realizados com o valor de 200 para o PWM (equivalente a aproximadamente 78,43% de toda a potencia disponível) podemos notar a variação de 0,5% ou mais entre os testes. A correção mais eficiente para este regime de potencia foi o de calibragem múltipla 10.3.

11.4 Analise do PWM 255

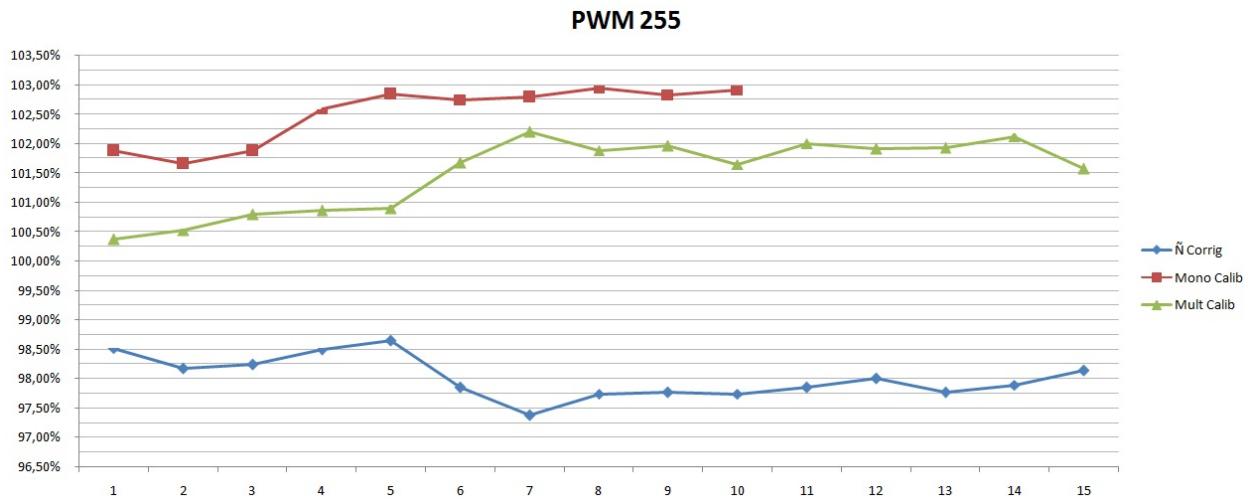


Figura 11.4: Velocidades do motor A em relação ao motor B em %, quinze comparações para o valor de PWM 255.

O regime de máxima potencia se demonstrou o mais instável tendo variação superiores a 1% entre os testes, acredita-se que por trata-se de máxima potencia os motores não mantém uma consistência em seu desempenho, deixando-o mais sensível a fatores externos ou variação da corrente disponível vinda das baterias, mesmo que a correção utilizando quatro valores de PWM 10.3 se mostrando mais eficiente em manter as velocidades parecidas, ainda sim demonstra variações inesperadas. Tais variações -inconstâncias- podem ser notadas nas figuras 10.1 e 10.3 com o PWM 255 (linhas azuis \diamond) e com o PWM de 200(linhas vermelhas \square)11.3.

Capítulo 12

Metodologia

- (A) Desenvolvimento e entrega do plano de trabalho;
- (B) Escrita da introdução, levantamentos dos dados teóricos e bibliográficos.
- (C) Revisão da literatura.
- (D) Implementar a aplicação
- (E) Escritas do trabalho de conclusão do curso.

Capítulo 13

Considerações Finais

Prof. Doc. Gedson Faria
Orientador

Kelvim Rodrigues de Oliveira
Acadêmico

Referências Bibliográficas

- [1] MARTIN EVANS, JOSHUA NOBLE, J. H. *Arduino em ação*. Novatec.