Leandro Luiz Rezende de Oliveira

Controle de Trajetória Baseado em Visão Computacional Utilizando o Framework ROS

Juiz de Fora - MG, Brasil

11 de novembro de 2013

Leandro Luiz Rezende de Oliveira

Controle de Trajetória Baseado em Visão Computacional Utilizando o Framework ROS

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica, área de concentração: Sistemas Eletrônicos, da Universidade Federal de Juiz de Fora, como requisito parcial para obtenção do grau de Mestre.

Orientador:
André Luís Marques Marcato

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Juiz de Fora - MG, Brasil

11 de novembro de 2013

Dissertação de Mestrado sob o título *"Controle de Trajetória Baseado em Visão Computacional Utilizando o Framework ROS "*, defendida por Leandro Luiz Rezende de Oliveira e aprovada em 11 de novembro de 2013, em Juiz de Fora, Estado de Minas Gerais, pela banca examinadora constituída pelos professores:

Prof. Dr. André Luís Marques Marcato
Orientador

Prof. Dr. Carlos Henrique Valério de Moraes
Universidade Federal de Itajubá

Prof. Dr. Leonardo de Mello Honório
Universidade Federal de Juiz de Fora

Resumo

O objetivo do presente trabalho é apresentar o desenvolvimento de um controle de trajetória para robôs móveis baseado em visão computacional, implementado no Framework ROS (*Robotic Operating System*). O ambiente do futebol de robôs foi utilizado como plataforma de teste da metodologia proposta. Para executar essa tarefa foi desenvolvido um algoritmo para o sistema de visão robótica, capaz de executar a calibração do sistema de captura de imagens e a identificação dos robôs no campo de acordo com a forma e a cor das etiquetas de identificação através da biblioteca *OpenCV*, a qual é integrada na estrutura fornecida pelo ROS. Os algoritmos elaborados de visão computacional, controle de alto nível e controle de baixo nível foram estruturados conforme as diretrizes do ROS, sendo assim denominados de nós. Os algoritmos de alto nível responsáveis pelo controle de trajetória, tratamento de imagem e controle são executados em um computador *desktop* ou *notebook*, ao passo que o algoritmo de baixo nível para controle é executado em uma plataforma Arduino embarcada no robô. O computador ou *notebook* e a plataforma Arduino embarcadas nos robôs trocam informações entre si de forma distribuída utilizando tópicos que interligam os nós e transmitem mensagens utilizando o padrão *Publisher/Subscriber*. Ainda é abordado o desenvolvimento do robô diferencial proposto, com seus dispositivos de controle, comunicação e tração.

Palavras-chave: Controle de Trajetória. *Framework* ROS. Visão Computacional. Robô Móvel Diferencial.

Abstract

The goal of this work is to present the development of a path control for mobile robots based on computer vision, implemented in ROS (Robotic Operating System) Framework. The robot soccer environment was used as test platform for the proposed methodology. To accomplish this task was developed an algorithm to the robotic vision system, able to perform the calibration of image capture system and the identification of robots in the field according to the shape and color identification labels through the OpenCV library, which is integrated into the structure provided by ROS. The algorithms developed for computer vision, high-level control and low-level control were structured according to the guidelines of the ROS, therefore called nodes. The algorithms responsible for the high-level path control, image processing and control are performed on a desktop computer or notebook, while the algorithm for low-level control is running on a platforms Arduino embedded in the robots. The computer or notebook and embedded platform Arduino exchange informations among themselves in a distributed manner using topics, interconnecting nodes and transmit messages using the Publisher/Subscriber default. Also is addresses the development of a differential robot proposed, with their control devices, communications and traction.

Keywords: Path Control. Framework ROS. Computer Vision. Differential Mobile Robot.

Dedicatória

Dedico este trabalho aos meus pais, Luiz e Luzia que sempre me guiaram para o crescimento intelectual e pessoal. Às minhas irmãs, Luziene Cristina e Lizandra Maria pela amizade e carinho.

Em especial à minha esposa Mariana, pelo apoio e pelas críticas e à minha filha, Maria Carolina, por ter trazido mais luz e força para a minha vida. Peço desculpas às duas pelas vezes que tive que me ausentar para dedicar mais tempo para o desenvolvimento desse trabalho.

Agradecimentos

Primeiramente venho agradecer à Deus, por ter me guiado por todos esses anos. Agradeço ao meu orientador, o Professor André Luís Marques Marcato pela confiança a mim depositada e por todo apoio e incentivo no desenvolvimento do Mestrado. Ao Professor Leonardo de Mello Honório, que juntamente ao meu orientador possibilitou a convivência em um laboratório com excelentes condições de trabalho.

Um agradecimento em especial à todos do Grupo de Robótica Inteligente - GRIn, dentre eles, a agora Professora Ana Sophia, Elias, Lucas, Exuperry, Alexandre, pelo apoio e por terem me aturado por todo esse período de estudo.

Agradeço também à Universidade Federal de Juiz de Fora, que desde a minha formação técnica vem me acolhendo e me fornecendo crescimento profissional e pessoal.

"O saber a gente aprende com os mestres e os livros.
A sabedoria, se aprende é com a vida e com os humildes."
(Cora Coralina)

Sumário

Lista de Figuras

1 Introdução p. 15

1.1 Objetivos . . . p. 17

1.2 Proposta . . . p. 17

1.3 Organização do Trabalho . . . p. 18

2 Robótica Móvel p. 19

2.1 Um Breve Histórico . . . p. 19

2.2 Definição e Classificações . . . p. 25

2.3 Sensores . . . p. 26

2.4 Cinemática do Robô Móvel . . . p. 28

2.4.1 Representação da Posição do Robô . . . p. 29

2.4.2 Modelo Cinemático do Robô . . . p. 30

2.5 Sistemas de Navegação e de Controle . . . p. 34

3 Visão Robótica p. 37

3.1 Fundamentos da Cor . . . p. 38

3.1.1 Espaço de Cor . . . p. 38

3.1.1.1 Espaço de Cor RGB . . . p. 39

3.1.1.2 Espaço de Atributos de Cor . . . p. 41

3.2 *OpenCV* . . . p. 43

3.2.1 Processamento de Imagens . . . p. 45

3.2.1.1 Filtro . . . . p. 47

3.2.1.2 Conversão entre Espaços de Cor e de Atributos . . . . p. 50

3.2.2 Visão Computacional . . . . p. 51

3.2.2.1 Subtração de Fundo . . . . p. 52

3.2.2.2 Transformada de Hough . . . . p. 53

3.3 Sistema de Captura de Imagens . . . . p. 57

4 Robotic Operating System - ROS p. 59

4.1 ROS . . . . p. 59

4.1.1 Sistema de Arquivos do ROS . . . . p. 62

4.1.1.1 Pacotes (*packages*) . . . . p. 62

4.1.1.2 Manifestos (*manifests*) . . . . p. 63

4.1.1.3 Pilhas (*stacks*) . . . . p. 63

4.1.1.4 Mensagens (*messages*) . . . . p. 64

4.1.1.5 Serviços (*services*) . . . . p. 65

4.1.2 Processamento Grafo do ROS . . . . p. 65

4.1.2.1 Nós (*nodes*) . . . . p. 66

4.1.2.2 Tópicos (*topics*) . . . . p. 67

4.1.2.3 Mensagens . . . . p. 67

4.1.2.4 Serviços . . . . p. 68

4.1.2.5 Servidor de Parâmetro (*parameter server*) . . . . p. 68

4.1.2.6 Mestre (*master*) . . . . p. 68

4.1.2.7 Bolsas (*bags*) . . . . p. 70

4.1.3 Comunidade ROS . . . . p. 70

4.1.4 Nomes de Recurso Grafo . . . . p. 71

4.1.4.1 Nomes Válidos . . . . p. 72

4.1.4.2 Deliberação . . . . p. 72

4.1.4.3 Remapeamento . . . . p. 73
4.1.5 Nomes de Recurso de Pacote . . . . p. 73
4.1.5.1 Nomes Válidos . . . . p. 74
4.1.6 Bibliotecas Cliente . . . . p. 74
4.1.7 Variável de Ambiente . . . . p. 75
4.1.8 Arquivos *launch* . . . . p. 76
4.1.9 ROS Fuerte Turtle . . . . p. 76

5 Experimentos e Resultados p. 78

5.1 Robô Diferencial Proposto . . . . p. 78
5.1.1 Arduino Nano . . . . p. 80
5.1.2 Módulo de Comunicação *XBee* . . . . p. 81
5.1.3 Módulo Ponte H . . . . p. 82
5.1.4 Módulo de tração . . . . p. 84
5.2 Sistema de Visão Robótica . . . . p. 85
5.2.1 Nó de Visão Robótica . . . . p. 91
5.3 Controle de Alto Nível do Robô . . . . p. 98
5.3.1 Nó de Controle de Alto Nível . . . . p. 98
5.4 Controle de Baixo Nível do Robô . . . . p. 100
5.4.1 Nó de Controle de Baixo Nível . . . . p. 103
5.5 Configurações Adicionais . . . . p. 105
5.6 Execução Prática . . . . p. 110

6 Conclusão p. 113

6.1 Trabalhos Futuros . . . . p. 114

Referências Bibliográficas p. 116

Anexo A – Ferramentas de Linha de Comando do ROS p. 121

Anexo B – Programas dos Nós do ROS p. 126

B.1 Código do Nó - `Find_Robot` . . . . p. 126

B.2 Código do Nó - `Robot` . . . . p. 131

B.3 Código do Nó - `serial_node` . . . . p. 138

Lista de Figuras

2.1 Robô Móvel: *Elsie*. Disponível em: <*http://www.frc.ri.cmu.edu/* > . . . . . p. 20

2.2 Robô Móvel: *Beast*. Disponível em: <*http://www.frc.ri.cmu.edu/*> . . . . . . p. 21

2.3 Robô Móvel: *Shakey*. Disponível em: <*http://www.ai.sri.com/shakey/*> . . . p. 21

2.4 Robô Móvel: *HILARE*. Disponível em: <*http://homepages.laas.fr/*> . . . . . p. 22

2.5 Sojourner. Disponível em: <*http://spacepioneers.msu.edu/*> . . . . p. 24

2.6 Humanoíde *Asimo*. Disponível em: <*http://www.honda.co.jp/ASIMO/*> . . . p. 24

2.7 Ciclo Percepção-Ação. Fonte: Baseada em [Pieri 2002]. . . . . p. 25

2.8 O *Frame* de Referência Global e o *Frame* de Referência Local do Robô. Fonte: Baseada em [Siegwart e Nourbakhsh 2004] . . . . p. 30

2.9 Robô Diferencial Não-Holonômico. Fonte: Baseada em [Dudek e Jenkin 2010] e [Siegwart e Nourbakhsh 2004] . . . . p. 32

2.10 Posição de destino sem a orientação de destino especificada. Fonte: baseada em [Novak e Seyr 2004] . . . . p. 36

3.1 Espectro Eletromagnético. Fonte: [Cattin 2012] . . . . p. 39

3.2 Círculos Representando as Cores Primárias do Sistema Aditivo. Fonte: [Cattin 2012] . . . . p. 39

3.3 Cubo de Cores RGB. Fonte: [Cattin 2012] . . . . p. 40

3.4 Imagem RGB. Fonte: [Cattin 2012] . . . . p. 41

3.5 Espaço de Atributos HSV. Fonte: [Cattin 2012] . . . . p. 42

3.6 Exemplos de Regiões de Vizinhança. Fonte: baseada em [Marengoni e Stringhini 2010] . . . . p. 47

3.7 Comportamento do Filtro *Gaussian* 1D. Fonte: [OpenCV 2012] . . . . p. 48

3.8 Mapeamento de uma linha existente no plano $x \times y$ para o plano $x \times y$. Fonte: baseada em [Jain, Kasturi e Schunck 1995] . . . . p. 54

3.9 Exemplo de geração de pontos no espaço de Hough. Fonte: baseada em [Martins 2007] . . . . p. 55

3.10 Câmera *Basler Scout scA640-120fc*. Disponível em: <http://www.baslerweb.com/> . . . . p. 57

3.11 Lente *Fujinon DV3.4x3.8SA-1*. Fonte: Manual do Fabricante. . . . . p. 58

4.1 Conceitos Básicos do ROS. Fonte: [Coleman 2013b] . . . . . . p. 66

4.2 Comunicação do Nó Publicador com o Mestre. Fonte: [Conley 2012f] . . p. 69

4.3 Comunicação do Nó Subscritor com o Mestre. Fonte: [Conley 2012f] . . p. 70

4.4 Comunicação Direta entre Nós. Fonte: [Conley 2012f] . . . . p. 70

4.5 ROS Fuerte Turtle. Fonte:[Foote 2013a] . . . . p. 77

5.1 Robô Diferencial Proposto. . . . . p. 79

5.2 Etiqueta de identificação do robô. . . . . p. 79

5.3 Arduino Nano. Disponível em: *<http://arduino.cc/en/>* . . . . p. 81

5.4 Módulo RF XBee PRO S1. Fonte: Manual do Fabricante. . . . . p. 81

5.5 Adaptador/Conversor USB - CON-USBBEE. Disponível em: *<http://multilogica-shop.com/adaptador-con-usbbee>* . . . . p. 82

5.6 Módulo Ponte H. Disponível em: *<http://www.pololu.com/catalog/product/713>* . . . . p. 82

5.7 Modulação por Largura de Pulso - PWM. Disponível em: *<http://arduino.cc/en/Tutorial/PWM>* . . . . p. 84

5.8 Sistema de Tração. Disponível em: *<http://www.pololu.com/catalog/product/1218>* . . . . p. 85

5.9 Janela de execução do Coriander. . . . . p. 86

5.10 Janela inicial do nó de calibração. . . . . p. 88

5.11 Alguns *frames* utilizados na Calibração da Câmera. . . . . p. 89

5.12 Janela do nó de calibração com índices satisfatórios. . . . . p. 89

5.13 Janela do nó de calibração sem distorções. p. 89

5.14 Representação do Processamento Grafo do ROS para a Visão Computacional. p. 91

5.15 Representação da interface ROS *OpenCV* para imagens. . . . . p. 93

5.16 Processo de Visão Robótica. . . . . p. 95

5.17 Fluxograma da Programação de Visão Robótica. . . . . p. 97

5.18 Modelo do Controlador de Alto Nível. . . . . p. 98

5.19 Simulação da Técnica de Controle de alto nível. . . . . p. 99

5.20 Modelo do Controlador PID em malha fechada. . . . . p. 101

5.21 Gráficos da saída dos motores direito e esquerdo, respectivamente. . . . . . . p. 102

5.22 Sistema de Controle obtido com o Bloco *PID Controller* do Simulink. . . . . p. 103

5.23 Parâmetros do controlador PI. . . . . p. 103

5.24 Resposta do controlador PI ao degrau unitário. . . . . p. 104

5.25 Representação do Processamento Grafo do ROS para a Visão Computacional e para os Controles de Baixo e Alto Nível. p. 104

5.26 Representação do Grafo do ROS. . . . . p. 105

5.27 Estrutura e interligações para o envio de mensagens ROS. Disponível em: [Bouchier 2013] . . . . p. 107

5.28 Trajetória percorrida pelo robô móvel proposto. . . . . p. 110

5.29 Variação da Posição no Eixo X ao longo do Tempo. . . . . p. 111

5.30 Variação da Posição no Eixo Y ao longo do Tempo. . . . . p. 111

5.31 Variação do Ângulo Global - θ. . . . . p. 112

1 *Introdução*

O desenvolvimento da tecnologia vem possibilitando, cada dia mais, o uso de equipamentos sofisticados e inovadores por parte da sociedade, de modo a fornecer maior comodidade, agilidade na resolução de tarefas, entretenimento dentre outros benefícios. Inserido nesse contexto está a robótica móvel, a qual apresentou um crescimento expressivo em suas linhas de pesquisas e implementações nas últimas décadas. Essas aplicações práticas, com robôs móveis, em diferentes atividades no cotidiano do ser humano, vem demonstrando o quão promissor é o papel desta área para o crescimento tecnológico. Algumas dessas aplicações estão presentes em soluções de diversas áreas como: residenciais - com aspiradores de pó e cortadores de grama robóticos, industriais - com transporte automatizado e veículos de carga autônomos, e militares - com sistemas de monitoramento aéreo remoto (VANTs), resgate e exploração em ambientes hostis. Esses exemplos apresenta parte da grande gama das aplicações atuais com robôs móveis, além do interesse econômico envolvido no desenvolvimento e comercialização destes.

Com a apresentação deste cenário, constata-se a grande relevância do aprimoramento desta área de pesquisa em nosso país, com a finalidade de manter-se alinhado ao pensamento mundial, que vem transformando o jeito de viver e de trabalhar dos humanos. A pesquisa e o desenvolvimento no segmento da robótica móvel se embasa em conhecimentos de diversas áreas, como a Engenharia Elétrica, a Engenharia da Computação e a Engenharia Mecânica. E essas por sua vez devem fornecer suporte com conhecimentos e com técnicas, os quais possibilitarão dotar esses robôs móveis com sistemas mais robustos, seguros, autônomos e inteligentes.

No entanto, para que tudo isso seja possível é imprescindível a formação de profissionais capacitados para atuar nesta área de grande expansão, além de disseminar a robótica e seus conceitos, de forma a torná-la mais factível e alcançável no dia-a-dia dos estudantes, seja a nível superior ou técnico. E como base para o ensino de tais conceitos, muitas entidades fazem o uso de ambientes que são tidos como problemas primários a serem dominados, como é o caso do ambiente do futebol de robôs.

A ideia de robôs jogarem futebol foi mencionada pela primeira vez com o professor Alan Mackworth, em 1992, em seu artigo intitulado "*On Seeing Robots*", apresentado no *Vision*

Interface'92, em Vancouver no Canadá [Martins 2007]. A partir dessa ideia surgiram vários *Workshops* e competições, num primeiro momento localizadas em pequenas iniciativas, mas que se estenderam a eventos internacionais, como a RoboCup (*Robot World Cup Initiative*) e a FIRA (*Federation of International Robot-soccer Association*), onde discussões e práticas sobre o assunto ganharam maiores proporções e grande importância como um meio de desafios para os pesquisadores das diversas áreas envolvidas [Kitano 1998].

A aplicação da plataforma de futebol de robôs como um meio de pesquisa e desenvolvimento requer conhecimentos que vão desde o projeto e a construção do *hardware* até o projeto e a elaboração do *software*, e um ponto que une esses extremos são os sistemas embarcados, no quais um *software* é integrado ao *hardware* de modo que seus recursos sejam controlados para a realização das tarefas pré estabelecidas.

Não se deve pensar no futebol de robôs como apenas uma brincadeira no meio acadêmico, pois sua utilização pode gerar subsídio para a criação ou o desenvolvimento de novas tecnologia, como é o caso da *IntellWheels*, uma cadeira de rodas com a capacidade de contornar objetos, executar tarefas e até comunicar com outros dispositivos presentes no ambiente em que esta está inserida. Este trabalho foi desenvolvido na Faculdade de Engenharia da Universidade do Porto (FEUP) a partir de algumas metodologias empregadas no futebol de robôs [Flores 2011].

No presente trabalho será utilizada a plataforma para a categoria *Mirosoft*, onde é necessário que sejam desenvolvidos algoritmos de controles de baixo e alto níveis, além de um algoritmo próprio de visão computacional, capaz de localizar o robô e a bola em campo.

Seguindo o pensamento de inovação e de compartilhamento do conhecimento serão utilizadas duas ferramentas para estruturação da solução proposta, bem como dos algoritmos já mencionados, sendo essas o ROS (*Robot Operating System*) e o *OpenCV*.

O ROS é um sistema de código fonte aberto que veio auxiliar o desenvolvimento de aplicações robóticas, sendo já amplamente difundido mundialmente tanto na academia quanto na indústria. Ele disponibiliza bibliotecas e ferramentas para ajudar os desenvolvedores de programas robóticos a criarem suas aplicações e integrarem os diferentes dispositivos e algoritmos em contínua evolução tecnológica. O ROS atualmente está sobre responsabilidade da *Willow Garage*, um grupo de pesquisa robótico que está fortemente comprometido com o desenvolvimento *open source* e *software* reutilizável [Garage 2011a].

No caso dos algoritmos de visão computacional, os mesmos serão desenvolvidos com base na biblioteca *OpenCV*, essa por sua vez criada com a finalidade de acelerar as pesquisas e as aplicações comerciais de visão computacional no mundo. Essa biblioteca originou-se na Intel em 1999, mas em 2008 o grupo *Willow Garage* assumiu o desenvolvimento da mesma

[Garage 2011b].

Para desempenhar o papel de interligação entre *hardware* e *software*, ou seja, do sistema embarcado, será utilizado o *Arduino*, uma plataforma de prototipagem eletrônica baseada na flexibilidade e na fácil implementação.

É importante relatar, que os estudos realizados no decorrer desta dissertação, renderam um Minicurso intitulado ”Introdução ao ROS (*Robotic Operating System*)”, apresentado no XI Simpósio Brasileiro de Automação Inteligente - SBAI, no período de 13 a 17 de outubro de 2013, em Fortaleza-CE.

1.1 Objetivos

O presente trabalho tem por objetivo apresentar o desenvolvimento de um sistema base para o controle de robôs móveis em aplicação no futebol de robôs, sendo esse sistema, em sua plenitude, construído sobre as diretrizes do *Framework* ROS (Robotic Operating System), e dividido em visão robótica, controle de alto nível e controle de baixo nível, sendo esse último, realizado a partir da integração com a placa de desenvolvimento programável - *Arduino*.

No segmento de visão robótica serão utilizadas técnicas de segmentação de imagens e visão computacional, através da biblioteca *OpenCV*, para que possa ser localizado o robô móvel no vídeo capturado. Já nos controles de alto e baixo nível serão verificados a aplicação do sistema de controle proposto para a realização das tarefas delegadas ao robô.

Além de apresentar, quase que na totalidade, a abrangência e a dinâmica que o *Framework* do ROS traz para o trabalho proposto.

1.2 Proposta

Esse trabalho propõe a utilização do ambiente de futebol de robôs como plataforma de testes para a implementação dos conhecimentos interdisciplinares na estrutura modular do ROS.

Com isso, serão apresentados e elaborados programas específicos para desempenharem as distintas funções no decorrer de todo o processo de controle do robô móvel, sendo que esses programas estarão inseridos nos nós do ROS, o quais irão capturar a imagem do campo, realizar todo o processamento necessário na visão robótica, desempenhar o papel de planejador do caminho a ser trilhado, além de controlar, em baixo nível, o *hardware* do robô móvel.

Não será proposto um novo protocolo de comunicação, mas a implementação do envio de

mensagens do ROS, que utiliza o padrão *Publisher/Subscriber*.

1.3 Organização do Trabalho

O Capítulo 2, em sua primeira Seção, é feito um breve levantamento dos fatos históricos, relativos à Robótica Móvel, apresentando os robôs que marcaram seus nomes na história, além de conceitos e teorias que até hoje são de grande importância para a implementação nesta área da Robótica. Em sua segunda Seção são apresentados definições e classificações referentes à Robótica Móvel, as quais são fundamentais para o projeto e planejamento do robô proposto neste trabalho. Já em sua terceira Seção é realizada a abordagem dos sensores empregados para a coleta de informações, tanto no ambiente quanto das condições do robô. Na quarta Seção, são apresentados conceitos da Cinemática do Robô Móvel e a modelagem necessária para controle do robô. Na quinta e última Seção são descritos o conceito de Espaço de Configuração, os Sistemas de Navegação e de controle, sendo esse de baixo e alto nível.

O Capítulo 3, traz em sua primeira Seção definições sobre os fundamentos da cor, bem como espaço de cor e espaço de atributos de cor, sendo essa de extrema importância para a apresentação das características de cada um desses espaços e fornecendo assim embasamento teórico na determinação de qual usar para se ter um sistema de visão mais robusto e tolerante à variação da intensidade luminosa do ambiente. Já em sua segunda Seção é apresentada a biblioteca *OpenCV*, a qual foi de fundamental importância para a implementação do sistema de visão, além de conceitos e técnicas referentes a Processamento de Imagens e Visão Computacional. Na última Seção, tem-se a descrição do sistema de captura de imagens utilizado na realização prática do presente trabalho.

O Capítulo 4, detalha a aplicação desenvolvida neste trabalho e, por isto, descreve as diversas características do *Framework* ROS - *Robotic Operating System*, apresentando seus níveis de conceitos: a nível de Sistemas de Arquivos, a nível de Processamento Gráfico e a nível de Comunidade. Ainda sobre o ROS, são descritos os Nomes dos Recursos Grafo e de Pacote, as bibliotecas cliente, as variáveis de ambiente e em especial a versão do ROS utilizada.

O Capítulo 5, mostra todo o desenvolvimento do trabalho prático, como a apresentação do robô diferencial proposto, com seus dispositivos de controle, comunicação e tração, além dos experimentos efetuados tanto em ambiente de simulação, para verificação e definição dos controladores de alto e baixo níveis, como em ambiente real através do robô móvel.

Por fim, o Capítulo 6, conclui esse trabalho e sugere possíveis trabalhos futuros, baseando-se nos resultados obtidos no decorrer do desenvolvimento deste.

2 Robótica Móvel

Tratando-se de robôs, várias cenas de filmes de ficção científica, sejam eles de ação, de aventura ou de animação são lembrados, como é o caso de "*Guerra nas Estrelas*", "*Eu, Robô*"e "*Wall-e*"com seus *droids*, humanoides e robôs de serviços. Todos eles demonstrando o desejo dos seres humanos de, no futuro, conviverem com robôs móveis no dia-a-dia. Ainda está distante o dia em que será possível criar máquinas autônomas, adaptáveis e inteligentes, mas com os avanços nas áreas de inteligência artificial e controle, juntamente com o desenvolvimento de novos materiais, já vem sendo viável a construção de robôs para os mais variados fins [Pieri 2002].

A robótica móvel, de uma forma geral, é a área da robótica que trabalha com mecanismos com capacidade de se locomoverem pelo ambiente em que estão inseridos, não estando restritos a uma base fixa.

2.1 Um Breve Histórico

A Robótica móvel teve início durante a 2ª Guerra Mundial como resultado das várias pesquisas realizadas em novos campos da ciência, como foi o caso da computação e da cibernética. Os primeiros robôs móveis eram, em sua grande maioria, bombas voadoras que faziam o uso de sistemas de orientação e controle por radar, afim de serem detonadas apenas a uma determinada distância do alvo. No caso dos foguetes *V1* e *V2*, eles possuíam um primitivo sistema de piloto automático e foram os predecessores dos atuais mísseis balísticos continentais. [Keramas 1998]

Com a invenção do transistor em 1948, não só a eletrônica teve seu horizonte expandido, a própria robótica se beneficiou com tal avanço, tendo seus robôs agora controlados por computadores. Dois anos após, em 1950, *Grey Walter* apresentou *Elsie* e *Elmer*, dois de seus oito robôs móveis. *Elsie* foi construído com olho de fototubo e dois amplificadores de tubo a vácuo, sendo esses responsáveis, respectivamente, por direcionar o robô e acionar seus motores. Como pode ser observado ao fundo da Figura 2.1, está visível a cobertura do robô, a qual trouxe ao mesmo sua semelhança à tartarugas. Oficialmente ele foi chamado de *Machina Speculatrix*, pois este

robô estava programado para explorar o ambiente embasado nas informações de seu olho, um sensor luminoso. Apresentando assim, um comportamento ”inteligente”, quando comparado a uma bactéria [Pieri 2002] [Pereira e Pimenta 2011].

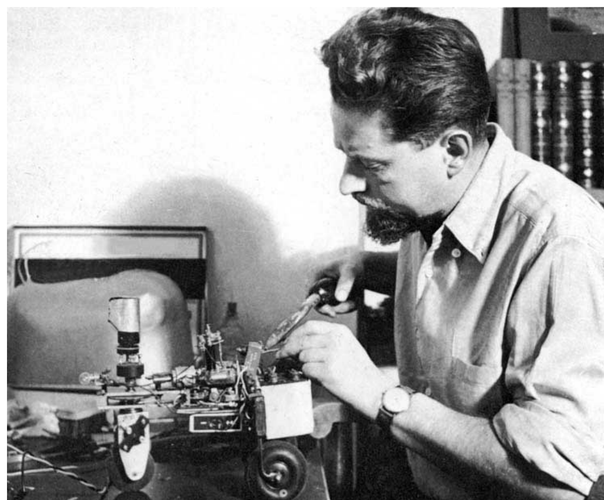


Figura 2.1: Robô Móvel: *Elsie*.
Disponível em: <*http://www.frc.ri.cmu.edu/* >

Em 1961, a Universidade John Hopkins apresentou o *Beast*, que pode ser visualizado na Figura 2.2, um robô controlado por dezenas de transistores, que fazia uso de um sonar para se guiar pelos corredores, sendo que ao detectar que suas baterias estavam ficando fracas ele procurava uma tomada elétrica na parede e realizava seu próprio recarregamento. O *Beast* era muito mais complexo do que a *Elsie*, uma vez que suas ações coordenadas se comparavam ao comportamento de caça de protozoários, como as amebas [Moravec 1999].

Alguns anos depois, em 1969, surgiu o *Mowbot*, que foi o primeiro robô cortador de grama automático. Neste mesmo período, a robótica móvel começou a fazer uso de conceitos da mecânica e da robótica fixa. A princípio, com o avanço nas áreas de sensores, processamento de imagens e inteligência artificial, equipar um robô móvel com dispositivos que lhe capacitasse para atuar em ambientes dinâmicos parecia ser algo fácil, entretanto, a robótica se deparou com a grande complexidade envolvida no desenvolvimento de sistemas móveis que fossem robustos e adaptáveis [Pieri 2002].

Já em 1969, através da publicação de [Nilsson 1969], *Nilsson* descreveu o primeiro sistema robótico móvel que utilizava *quadtree*[1] para representar o ambiente e grafos de visibilidade para o planejamento de trajetória. Após três anos de estudos e desenvolvimentos na SRI International, atual Instituto de Pesquisa de Stanford, foi apresentado o robô *Shakey*, mostrado na Figura 2.3, que apesar de seus movimentos trêmulos era capaz de ”raciocinar”sobre as suas

[1] Estrutura em forma de árvore gerada através da decomposição de um ambiente bidimensional pelo refinamento sucessivo das células.

Figura 2.2: Robô Móvel: *Beast*.
Disponível em: <*http://www.frc.ri.cmu.edu/*>

ações, interpretando os comandos dados por seus operadores e ponderando as informações coletadas do ambiente através de seus sensores, dentre eles: câmera de TV, sensor de profundidade e sensores de contato. Sendo considerado como o primeiro robô móvel controlado por inteligência artificial e juntamente com outro robô, o *Stanford Cart*, foram os primeiros robôs controlados por computadores [Pieri 2002] [Moravec 1999] [Pereira e Pimenta 2011]. Em 1973, a SRI International apresenta *WAVE*, a primeira linguagem de programação para robôs [Pereira e Pimenta 2011].

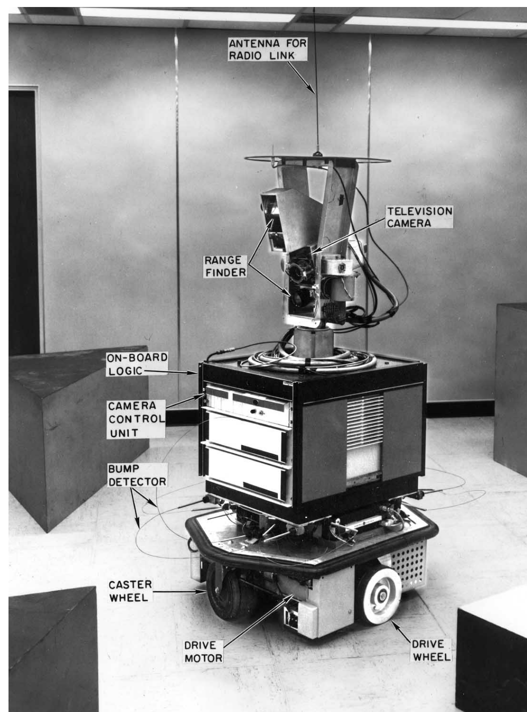


Figura 2.3: Robô Móvel: *Shakey*.
Disponível em: <*http://www.ai.sri.com/shakey/*>

Após alguns anos, a União Soviética explorou a superfície da lua com o robô móvel, *Lunokhod 1*. Sendo que em seguida, no ano de 1976, a NASA com seu Experimento *Viking* enviou duas sondas não tripuladas a Marte.

Em 1980, o *Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS)* apresentou o *HILARE*, considerado o primeiro robô móvel europeu, com sensores de hodometria, sonar e *laser*. Para cada 20 centímetros de deslocamento eram gastos 10 segundos de processamento. Na Figura 2.4 pode ser observado o *HILARE* em funcionamento.

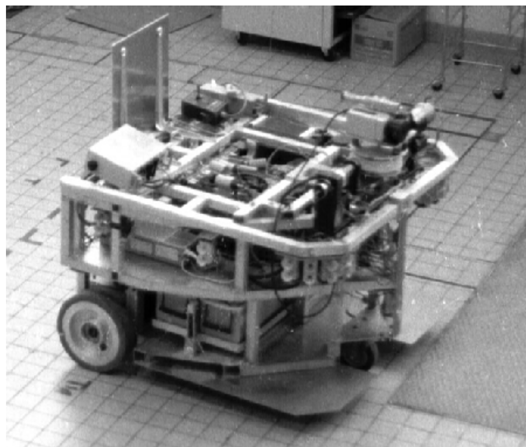


Figura 2.4: Robô Móvel: *HILARE*.
Disponível em: <*http://homepages.laas.fr/*>

Nesse período a robótica móvel já enfrentava problemas relacionados às questões de movimentos rotacionais e translacionais, os quais tornavam crítica a modelagem do ambiente. Diante desse cenário, em 1983 com a publicação de [Lozano-Pérez 1983], *Lozano-Pérez* introduz a ideia de "região de incerteza", criada através do crescimento dos obstáculos. Dessa forma, utilizando-se de grafos de visibilidade para o planejamento de trajetória, o robô poderia passar a ser tratado como um simples ponto no espaço de configuração.

Neste mesmo ano, *Brooks* descreveu o método *freeway*, embasado no conceito de cones generalizados, colocando-se como uma alternativa à modelagem do espaço livre e ao planejamento de trajetória.

Tais métodos, baseados em mapa de ambiente, classificados de arquiteturas deliberativas ou planejadas, apresentavam restrições críticas com relação a criação e manutenção do mapa do ambiente. E com o intuito de minimizar os problemas provenientes do mundo real, foi necessário realizar certas simplificações, como considerar o ambiente sendo estático e totalmente conhecido. Com a aplicação dessas simplificações os problemas foram contornados, entretanto a aplicação de robôs ficou restrita a ambientes inalterados.

Com o intuito de restaurar a capacidade dos robôs móveis de interação com os ambientes

dinâmicos e inspirado no comportamento dos insetos, *Brooks*, em 1986, introduziu uma arquitetura reativa (*arquitetura de subsunção*), na qual o robô tem suas atitudes baseadas na leitura de seus sensores [Brooks 1986]. A partir dessa arquitetura, fundada na decomposição da inteligência em comportamentos individuais, foram gerados módulos que coexistem e cooperam para a reação diante de comportamentos mais complexos. Ainda neste ano, *Khatib* introduziu o método dos campos potenciais, onde o robô passa a ser uma partícula sob a influência de campos eletromagnéticos, sendo esses provenientes dos obstáculos e do ponto objetivo [Khatib 1986].

Nos anos seguintes, *Arkin* escreveu vários artigos apresentando uma arquitetura reativa, fundamentada em esquemas motores, a qual tornou-se mais tarde uma arquitetura híbrida denominada de *AuRA - Autonomous Robot Architecture*.

Em 1990, *Kumpel* e *Sarradilla* apresentam o artigo referenciado em [Kumpel e Sarradilla 1990], descrevendo o projeto *MARIE - Mobile Autonomous Robot in an Industrial Environment*. Tal projeto integrava mapas geométricos e topológicos, além de usar um método hierárquico para a navegação, onde a partir dos mapas globais encontrava o caminho e através dos sensores realizava o desvio dos obstáculos. Neste mesmo ano, *Brooks* apresentava algumas melhorias à sua arquitetura de subsunção, juntamente com vários robôs, desenvolvidos no MIT - Instituto de Tecnologia de Massachusetts, os quais faziam uso dessa arquitetura. Também nesse ano, *Joseph Engelberger*, juntamente com alguns colegas, projetou o *Helpmate*, o primeiro robô autônomo para auxiliar nas atividades hospitalares.

Já em 1992, através de [Mataric 1992], *Mataric* demonstra a preocupação com a necessidade de se ter alguma representação do ambiente, afim de capacitar o robô móvel mais do que apenas a avegação aleatória, com isso propõe um método reativo, este embasado em um mapa atualizável, construído através de marcas conhecidas como *landmarks*, detectadas no ambiente, dentro de uma arquitetura de subsunção.

Ainda neste ano, *Zelinsky* propõe em [Zelinsky 1992], um método simples para mapeamento do ambiente, utilizando sensores de contato, durante a própria execução. Tal ambiente é mapeado em uma *quadtree*, sendo que a menor célula possui o tamanho do diâmetro do robô. A trajetória inicial é uma linha reta, mas durante a execução da mesma são colhidas as leituras dos sensores, as quais são utilizadas para atualizar a estrutura da *quadtree* que serve de base para o reestruturação do caminho.

Em julho de 1997, o robô móvel *Sojourner* da NASA, apresentado na Figura 2.5 toca o solo de Marte, que apesar de não se mover muito foi capaz de confiar em um planejamento *off-line*. Já em janeiro de 2004, outros dois robôs móveis, conhecidos como *rovers Spirit* e *Opportunity* pousam em Marte, desta vez com maior sucesso, conseguindo explorar maiores áreas devido à

melhor autonomia das baterias, sendo que o *Spirit* andou 7,7 km e o *Opportunity* já andou 26,69 km [Pereira e Pimenta 2011].

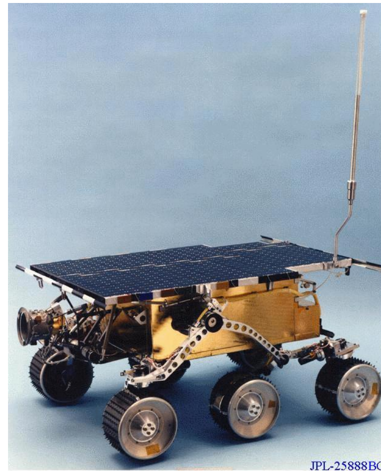


Figura 2.5: Sojourner.
Disponível em: <*http://spacepioneers.msu.edu/*>

No início do século XXI, a robótica móvel teve seu grande marco com o aparecimento dos robôs humanóides, sendo que em 2005, após quase três década de estudos e desenvolvimentos, a HONDA apresentou o *Asimo* o sucessor de *P2* e *P3*, um robô humanóide de 1,5m de altura capaz de correr e subir escadas, algo nunca antes alcançado. Na Figura 2.6 está mostrada uma foto do *Asimo*, cujo nome significa: ***A**dvanced **S**tep in **I**nnovative **Mo**bility*.

Figura 2.6: Humanoíde *Asimo*.
Disponível em: <*http://www.honda.co.jp/ASIMO/*>

2.2 Definição e Classificações

O robô móvel pode ser definido como sendo um dispositivo mecânico montado sobre uma base não fixa, que age sob o controle de um sistema computacional, o qual equipado com sensores e atuadores é capaz de interagir com o ambiente. Tal interação se dá através de *ciclos de percepção-ação*, que consiste em três fundamentos básicos [Pieri 2002]:

- Obtenção de informações do ambiente inserido, através de sensores;
- Processamento das informações coletadas e seleção de ações pertinentes;
- Execução das ações planejadas por meio do acionamento dos atuadores.

Diversos ciclos de percepção-ação, conforme apresentado na Figura 2.7, são executados pelo robô, modificando seu estado no ambiente em busca da realização das tarefas estabelecidas [Ribeiro, Costa e Romero 2001].

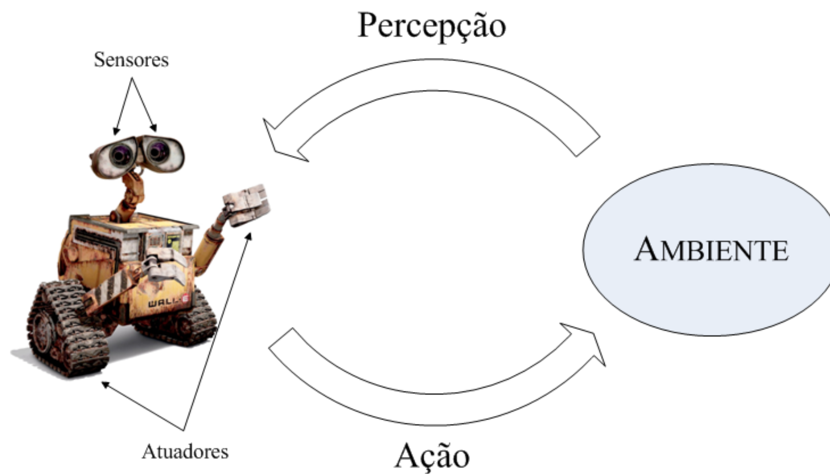


Figura 2.7: Ciclo Percepção-Ação.
Fonte: Baseada em [Pieri 2002].

O campo da robótica móvel, de uma forma geral, pode ser dividido conforme as diferentes topologias, as formas de locomoção e o tipo de controle.

Quanto à forma de locomoção, os robôs podem ser classificados em quatro tipos: terrestres, aquáticos, aéreos e espaciais [Dudek e Jenkin 2010]. Os robôs terrestres, os mais populares, são aqueles que caminham sobre o piso, podendo usar três diferentes tipos de atuadores: rodas, esteiras ou pernas. No presente trabalho é utilizado o robô com o primeiro tipo de atuador, visto que é o mais simples, não demandando um *hardware* tão complexo quanto os dois seguintes.

Já os robôs aquáticos podem operar tanto na superfície da água quanto sob a água, sendo normalmente equipados com propulsores e balões de ar, possibilitando respectivamente o deslocamento e a submersão dos mesmos. Os aéreos por sua vez, são robôs que imitam os aviões e os pássaros, utilizando de asas fixas ou móveis para se deslocarem na atmosfera. E por último, os robôs espaciais, que são projetados para funcionarem em microgravidade, ou seja, em atividades de exploração espacial.

Os robôs terrestres com rodas podem ainda ser subclassificados em holonômicos e não holonômicos. Sendo essa característica, referente ao fato do robô ser capaz ou não de alcançar determinadas posições, imediatamente adjacentes à sua posição atual, sem a necessidade de execução de manobras complexas. Na maioria das aplicações de robôs com rodas, não é verificada tal característica, sendo necessário para isso que as rodas possuam esterçamento independente. Um exemplo de veículo não-holonômico é o automóvel, pois para o mesmo ser estacionado paralelamente à calçada e entre dois outros veículos já estacionados, são necessárias algumas manobras. Visto que é impossível seu deslisamento lateral.

Quanto ao tipo de controle, os robôs podem ser separados em três grupos: os teleoperados, os semi-autônomos e os autônomos. Os primeiros realizam todos os seus movimentos de acordo com os comandos recebidos do operador. No segundo caso, os robôs recebem ordens do operador, mas possuem autonomia para executar a referida tarefa. Já os últimos, não recebem qualquer tipo de orientação, sendo capazes de executarem suas tarefas, totalmente independente, conforme os dados obtidos do ambiente em que estão inseridos.

2.3 Sensores

Um dos grandes desafios da robótica móvel nos dias atuais é a obtenção de informações sobre o ambiente onde se está inserido, afim de que, a partir desses dados possa ser realizado um conjunto de tarefas, sem que seja necessária a intervenção humana. Na tentativa de fornecer tais dados aos dispositivos robóticos, são empregados diversos tipos de sensores, os quais são capazes de quantificar ou detectar parâmetros específicos por meio de elementos transdutores[2], sendo esses ativos ou passivos. Os sensores ativos são aqueles que de alguma forma emitem sinais e analisam as pertubações sofridas nesses, durante a propagação no ambiente. Já o sensores passivos apenas realizam observações do ambiente em que estão inseridos sem promover qualquer tipo de alteração neles.

Para a robótica móvel, os sensores mais utilizados são os sensores de choque (*bumpers*),

[2]Transdutores são elementos que tem a função de converter uma dada magnitude física em outra.

sensores infravermelho, sensores do tipo sonar, sensores do tipo radar, sensores *laser*, sensores baseados em satélites (GPS – *Global Position System*), sensores baseados em acelerômetros e giroscópios (IMU - *Inertial Measurement Unit*), bússulas, sensores de hodometria e sensores baseados em visão [Dudek e Jenkin 2010].

Os sensores ainda podem ser classificados quanto à sua disposição, como sendo internos ou externos. Os sensores internos são aqueles localizados no próprio robô e devido a isso se movem juntamente com o mesmo, fornecendo informações quanto a posição, velocidade e aceleração. Já os sensores externos são fixados no ambiente, por onde se pretende locomover ou em algum outro sistema móvel, de modo a monitorar o próprio robô e disponibilizar informações quanto a realização de suas tarefas. [Pieri 2002]

Outra classificação que ainda pode ser feita dos sensores é quanto à abrangência de seus dados. Os sensores podem fornecer um conhecimento local, que é usado para a determinação das características internas do robô, como ocorre no caso dos sensores de hodometria, que podem informar velocidade e posição dos motores. Os sensores também podem fornecer um conhecimento global, o qual irá disponibilizar informações dos estados do robô em relação ao ambiente, como ocorre com a utilização dos sensores de visão, que irão fornecer a velocidade e posição do robô.

Alguns sensores de visão tem por característica fornecer uma grande quantidade de dados. Dessa forma é necessário um hardware dedicado para realizar a análise e processamento destes. Por exemplo, esses sensores podem ser do tipo CCD (*Charge-Coupled device*), capazes de realizar a capitação de 30 quadros por segundo, podendo obter imagens à cores ou em escala de cinza e com resolução variável, conforme a qualidade do CCD utilizado. Esse tipo de sensor pode ser usado com apenas um ou em configuração dupla (estereoscopia), essa segunda forma é capaz de reconstituir um ambiente em sua forma tridimensional, sendo assim possível obter a distância entre o robô e um objeto no ambiente. Todavia, esse processo de reconstrução de ambientes demanda um alto custo computacional estando sujeito a problemas de oclusão e imprecisão [Langer 2007].

Esses sensores visuais também podem ser classificados em internos ou externos. No caso da câmera ou das câmeras estarem localizadas no robô é necessário o uso de conhecimentos prévios e marcos visuais (*landmarks*) para que o mesmo consiga se locomover no ambiente [Ma, Kosecka e Sastry 1999]. As câmeras externas também são utilizadas para observar o movimento do robô e informar a sua localização, mas a principal diferença está em onde o processamento da imagem será feito. Quando das câmeras externas, o processamento da imagem não se dá na unidade central de processamento interna do robô, mas sim em um computador externo

que a partir de códigos pré-definidos toma algumas decisões e as transmite ao robô, muitas vezes denominado de agente. O controle do robô é feito de forma externa e quando não existe além do sistema visual, um processamento e sensoriamento local, este se torna uma tarefa não trivial. Atualmente, esta configuração tem sido encontrada em aplicações que se deseja obter robôs de baixo custo ou que não podem ou não conseguem conter carga [Ma, Kosecka e Sastry 1999].

O robô utilizado neste trabalho, possui sensores interno de hodometria, sendo esses do tipo codificador ótico incremental, os quais fornecem informações para que possa ser obtida a velocidade rotacional das rodas e assim controlar os erros pertinentes às diferenças construtivas dos motores.

Já com relação à câmera de vídeo, a mesma foi fixada em uma estrutura externa, com a finalidade de fornecer uma visualização completa do ambiente, no caso o campo.

2.4 Cinemática do Robô Móvel

A cinemática do robô, compreende-se como o estudo do movimento do robô com relação a um determinado sistema de referência. Tendo em vista, que é necessário entender o comportamento mecânico do robô tanto para projetá-lo em conformidade com as tarefas que irá realizar quanto para entender como deverá ser o controle [Pieri 2002].

A robótica móvel possui muitos dos conceitos em comum com a robótica de manipuladores. Dessa forma, um *workspace* que para um robô manipulador define a gama de poses possíveis de serem alcançadas pelo seu atuador, relativo ao seu ponto de apoio no ambiente. No caso do robô móvel é igualmente importante, pois define também as várias poses possíveis que podem ser alcançadas em seu ambiente. No caso da controlabilidade de um braço robótico que irá definir a maneira como serão acionados os motores para que seja possível a partir de uma pose conhecida chegar em uma nova. Já para o robô móvel, irá definir possíveis caminhos e trajetórias em seu *workspace*.

Mas a principal diferença entre as duas áreas está relacionada à estimação da posição. Pois no caso de um robô manipulador, que possui uma base fixa no ambiente, medir a posição final de seu atuador é simplesmente uma questão de entender a cinemática do referido robô e medir a posição de todas suas as articulações intermediárias (estado das juntas). Já no caso do robô móvel, ele é um autômato auto-contido, que pode mover-se totalmente em relação ao seu ambiente, não havendo nenhuma maneira direta de obter sua posição instantaneamente. Para obter a posição desse robô, é necessário integrar o movimento deste ao longo do tempo, mas ainda sim estarão presentes imprecisões de estimativa de movimento, devido à inserção de incerte-

zas, como por exemplo, a derrapagem, o que torna a definição da posição do robô uma tarefa extremamente desafiadora [Siegwart e Nourbakhsh 2004].

Para compreender os movimentos de um robô, primeiramente se faz necessário descrever a contribuição que cada roda proporciona para o movimento final, pois cada roda tem um papel na determinação do movimento do robô em geral. Além de impor restrições ao movimento do mesmo, quando por exemplo não é permitido o seu deslisamento lateral.

2.4.1 Representação da Posição do Robô

No desenvolver da análise serão consideradas as seguintes suposições:

- A estrutura da base móvel do robô é um corpo rígido sobre as rodas, operando sobre o plano horizontal.
- As dimensões do chassi sobre o plano horizontal são três: duas para a posição no plano e uma para a orientação ao longo do eixo vertical, sendo essa ortogonal ao plano.
- As rodas são rígidas e não deformáveis.
- Não há escorregamento nas rodas, ou seja, todo movimento produzido pela roda é transmitido em movimento da estrutura móvel do robô.

Para especificar a posição do veículo sobre o plano é estabelecida uma relação entre a referência global do plano e a referência local do robô, conforme mostrado na Figura 2.8. Os eixos X_w e Y_w definem uma base inercial arbitrária sobre o plano, como sendo a referência global desde a origem $O : \{X_w, Y_w\}$. Para especificar a posição do robô é escolhido um ponto P sobre o chassi, sendo este o ponto de referência.

Já a base $\{X_r, Y_r\}$ indica os dois eixos relativos ao ponto P sobre o chassi, ou seja, a referência local do robô. Sendo a posição de P na referência global especificada pelas coordenadas x e y, e a diferença angular entre as duas referências dada por θ.

Feitas essas definições, o posicionamento do robô pode ser descrito como um vetor com esses três elementos, também denominado de vetor de configurações, onde o subíndice w define que a base desta posição está referenciada globalmente.

$$q_w = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

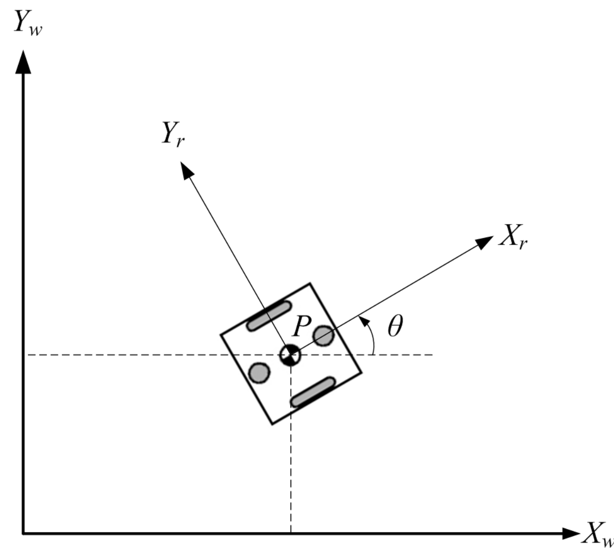


Figura 2.8: O *Frame* de Referência Global e o *Frame* de Referência Local do Robô.
Fonte: Baseada em [Siegwart e Nourbakhsh 2004]

Para descrever o movimento do robô em termos das componentes de movimento, se faz necessário o mapeamento do movimento ao longo dos eixos de referência global em relação ao movimento ao longo dos eixos de referência local. Dessa forma, o mapeamento será feito em função do posicionamento do robô, e através da matriz de rotação ortogonal a seguir, a qual é a matriz de rotação elementar em torno do eixo z:

$$R(\theta) = \begin{bmatrix} \cos\theta & sen\theta & 0 \\ -sen\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Esta matriz é usada para mapear o movimento da referência global $\{X_w, Y_w\}$ em relação à referência local $\{X_r, Y_r\}$. Sendo esta operação indicada pela Equação 2.3 e em função de θ.

$$q_r = R(\theta)q_w = \begin{bmatrix} \cos\theta & sen\theta & 0 \\ -sen\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.3}$$

2.4.2 Modelo Cinemático do Robô

Em robótica móvel, o modelo cinemático relaciona as velocidades dos atuadores, no caso em estudo - as rodas, às velocidades de um referencial fixo no robô, geralmente em seu centro de massa. Como já apresentado anteriormente na Figura 2.8, o ponto P. Da mesma forma que a cinemática dos manipuladores, o modelo cinemático dos robôs móveis relaciona as velocidades,

demonstrando que o modelo encontrado é similar ao Jacobiano do sistema.

O modelo cinemático mais simples para um robô móvel pode ser denotado por:

$$\dot{q} = G(q_r)u(t) \tag{2.4}$$

onde:

$\dot{q}$ é a derivada de q em função do tempo;
$G(q)$ é uma matriz quadrada, caso o vetor de controle tiver dimensão igual ao número de graus de liberdade do robô;
q_r é o vetor de configurações;
$u(t)$ é o vetor de controle, representado pelas velocidades de atuação do robô;

O fato da matriz $G(q)$ ser quadrada simplifica o controle do robô, entretanto para a grande maioria dos robôs móveis essa matriz não será, visto que o número de velocidades de atuação é menor que o número de graus de liberdade do sistema. Tal fato é geralmente caracterizado por restrições de movimento que não permitem o controle independente de todas as variáveis do vetor de configuração. Essas restrições são restrições não-holonômicas, as quais reduzem o número de velocidades independentes do robô, tornando o sistema subatuado. E como apresentado na Seção 2.2, os robôs com esse tipo de restrições são classificados como robôs não-holonômicos [Dudek e Jenkin 2010].

O robô utilizado neste trabalho é conhecido como robô diferencial, o qual é constituído de duas rodas controladas por motores independentes, sendo essas instaladas sobre um eixo imaginário que passa pelo ponto P, conforme mostrado na Figura 2.9.

O modelo cinemático referente a esse robô, deve relacionar as velocidade das rodas com as velocidades do referencial fixo em seu centro de massa. Sendo necessário considerar que as rodas do robô só se movem na direção perpendicular a esta, ou seja, não há derrapagem lateral da roda. Para que essa condição seja satisfeita, deve existir um ponto, em torno do qual cada uma das rodas do robô siga um caminho circular, mesmo que este ponto esteja localizado no infinito, o que implicaria em uma trajetória retilínea por parte do robô. Este ponto é chamado de Centro Instantâneo de Curvatura (*Instantaneous Center of Curvature - ICC*) ou ainda de Centro Instantâneo de Rotação (*Instantaneous Center of Rotation - ICR*) [Dudek e Jenkin 2010]. O ICC sempre estará localizado no ponto de cruzamento entre os eixos das rodas, que no caso do robô diferencial em estudo, devido aos eixos das rodas serem paralelas e coincidentes, o mesmo estará situado sobre a linha dos eixos.

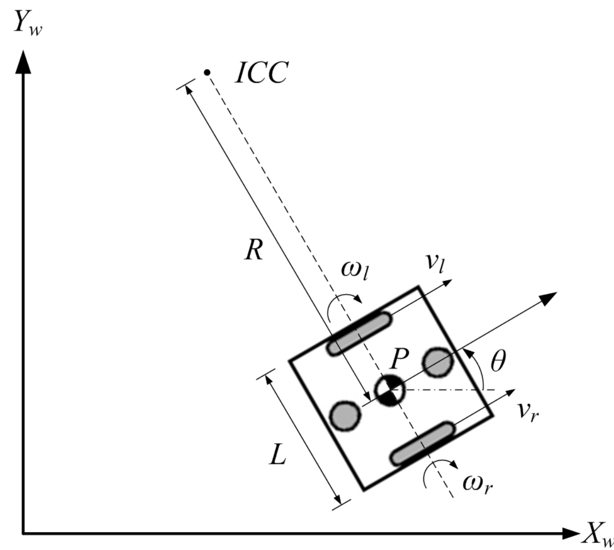


Figura 2.9: Robô Diferencial Não-Holonômico.
Fonte: Baseada em [Dudek e Jenkin 2010] e [Siegwart e Nourbakhsh 2004]

Como o robô é um corpo rígido, as rodas direita e esquerda devem se mover instantaneamente ao redor do ICC com a mesma velocidade angular - ω. Obtendo assim as seguintes relações:

$$\omega = \frac{v_r}{R - \frac{L}{2}} = \frac{r\omega_r}{R - \frac{L}{2}} \tag{2.5}$$

$$\omega = \frac{v_l}{R + \frac{L}{2}} = \frac{r\omega_l}{R + \frac{L}{2}} \tag{2.6}$$

onde:

v_r e v_l são as velocidades lineares das rodas direita e esquerda, respectivamente;
ω_r e ω_l são as velocidades angulares das rodas;
r é o raio das rodas;
L é a distância entre os centros das rodas;
R é o raio de curvatura instantânea do caminho do robô.

Observando as Equações 2.5 e 2.6, é possível obter duas relações a respeito do ICC. Se $v_r = v_l$, o raio R será infinito e o robô irá se mover em uma linha reta. Caso $v_r = -v_l$, o raio R será zero e o robô irá girar sobre um ponto central entre as duas rodas, o ponto P, ou seja, irá girar sem sair do lugar. Tal característica favorece a navegação deste tipo de robô móvel diferencial em ambientes estritos [Dudek e Jenkin 2010].

Através de manipulações matemáticas nas duas Equações anteriores é possível obter as relações em função de R e ω:

$$R = \frac{L}{2}\frac{(\omega_r + \omega_l)}{(\omega_r - \omega_l)} \tag{2.7}$$

$$\omega = \frac{r(\omega_r - \omega_l)}{L} \tag{2.8}$$

Sendo que a velocidade linear do centro de massa do robô pode ser calculada a partir do produto entre R e ω:

$$V = \frac{r(\omega_r + \omega_l)}{2} \tag{2.9}$$

Caso a velocidade linear do robô seja decomposta em suas componentes x e y e lembrando que ω é a taxa de variação de θ no tempo, é possível obter as três relações a seguir:

$$\dot{x}(t) = V(t)\cos\theta(t) \tag{2.10}$$

$$\dot{y}(t) = V(t)\sin\theta(t) \tag{2.11}$$

$$\dot{\theta}(t) = \omega(t) \tag{2.12}$$

Substituindo V e ω, descritos nas Equações 2.9 e 2.8, nas Equações 2.10, 2.11 e 2.12, obtém-se o modelo cinemático para o robô diferencial, em função das velocidades angulares das rodas direita - ω_r e esquerda - ω_l:

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} r\cos\theta(t)/2 & r\cos\theta(t)/2 \\ rsen\theta(t)/2 & rsen\theta(t)/2 \\ r/L & -r/L \end{bmatrix} \cdot \begin{bmatrix} \omega_r(t) \\ \omega_l(t) \end{bmatrix} \tag{2.13}$$

O mesmo modelo pode ser representado em função das velocidades linear e angular do robô:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta(t) & 0 \\ sen\theta(t) & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V(t) \\ \omega(t) \end{bmatrix} \tag{2.14}$$

Uma característica importante a ser observada nesse sistema é que o número de entradas de controle independentes é menor que o número de graus de liberdade devido à existência de uma restrição não-holonômica. Sendo essa restrição descrita por:

$$\dot{x}sen\theta - \dot{y}cos\theta = 0 \tag{2.15}$$

A interpretação física da Equação 2.14 é que a componente de velocidade perpendicular às rodas do robô deve ser nula, tendo como consequência a dependência entre as velocidades $\dot{x}$ e $\dot{y}$. Essa dependência é relativa às rodas de eixo fixo, pois as duas rodas esféricas utilizadas para dar maior estabilidade, as quais estão localizadas em um eixo perpendicular às anteriores, não impõe restrições diretas ao movimento. Visto que as mesmas não possuem um eixo principal, logo não existem restrições de rolamento ou deslizamento [Siegwart e Nourbakhsh 2004].

2.5 Sistemas de Navegação e de Controle

Os Sistemas de Navegação e Controle devem ser capazes de guiar o robô por caminhos, os quais resultem em posição e orientação finais definidas pelo sistema de estratégia, apresentando o mínimo de erro em relação à posição objetivo. Além de evitar colisões com os obstáculos no campo, sendo esses os robôs do time adversário ou os limites do campo.

De acordo com [Novak e Seyr 2004], o algoritmo de controle do robô é baseado em um modelo de camadas múltiplas, onde a camada inferior é responsável por controlar a mecânica do robô, a fim de garantir que a trajetória, designada pela camada de alto nível, seja seguida pelo robô. O sistema responsável por gerenciar esse controle de baixo nível será abordado no Capítulo 5, onde serão apresentados passo a passo todos os cálculos para a obtenção e a definição de qual controlador será mais eficaz, dentre os controladores P, PI, PD ou PID.

Já a camada superior, deve ser ser capaz de calcular a trajetória, processo esse, comumente denominado de planejamento de trajetória. O cálculo de uma trajetória, para o presente trabalho, é baseado em pontos de intersecção, sendo esses o resultado da resolução das tarefas geradas. A solução das tarefas será executada pela camada anterior, que é em si uma subcamada da camada de decisão.

A habilidade básica do robô proposto é ser capaz de alcançar uma posição de destino, sendo que para isso, existem diversas possibilidades de como essa posição de destino pode ser descrito. Ela pode ser definida por combinações arbitrárias dos seguintes parâmetros:

- coordenadas x e y de uma posição no plano;
- a orientação para a posição de destino;
- a velocidade para a posição de destino;
- e o intervalo de tempo necessário para atingir a posição de destino.

No trabalho aqui desenvolvido, não será atribuída uma orientação específica em que o robô deverá abordar a posição de destino, mas em trabalhos futuros e para o projeto completo do futebol de robôs, essa característica deverá ser considerada.

Sendo assim, o algoritmo deve apenas calcular um caminho de uma posição de origem até uma posição de destino, sem a atribuição da orientação. Sendo que, se a velocidade e o intervalo de tempo forem especificados ou não, isso não afetará a definição da trajetória, nesta simples abordagem.

Logo, a trajetória pode ser calculada a partir dos parâmetros de destino e da posição atual, a qual irá consistir de linhas retas, sendo as velocidades linear e angular variáveis conforme a necessidade da correção do erro, em cada um dos trechos.

Com este algoritmo, uma posição de destino sem orientação específica é calculada a partir dos parâmetros da posição de origem do robô, esses identificados por: x_R, y_R e θ, e da posição de destino, identificados por: x_T, y_T, conforme apresentado na Figura 2.10

A orientação θ deve ser ajustada de modo que o robô fique de frente para a posição de destino, conforme a Equação 2.16, sendo essa correção definida como o erro angular. É importante salientar que $\Delta\theta$ deverá ser corrigido de modo que esteja no intervalo [-π, π).

$$\Delta\theta = \theta_T - \theta \tag{2.16}$$

No caso de θ_T, que é o ângulo de direcionamento para o alvo, o mesmo será obtido em função da Equação 2.17, a qual leva em conta as coordenadas atuais do robô e da posição de destino.

$$\theta_T = tan^{-1} \frac{(y_T - y_R)}{(x_T - x_R)} \tag{2.17}$$

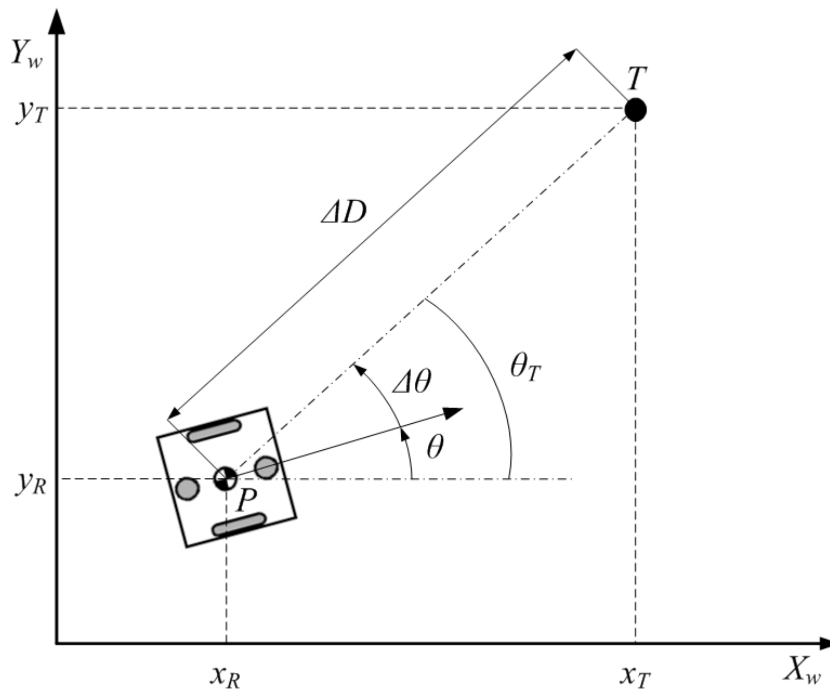


Figura 2.10: Posição de destino sem a orientação de destino especificada.
Fonte: baseada em [Novak e Seyr 2004]

Já o valor de θ poderá ser obtido conforme será apresentado na Subseção 5.1.

O módulo da diferença entre a posição de origem e a posição de destino, o qual é identificado como sendo o erro linear, deve ser calculado conforme a Equação 2.18

$$\Delta D = \sqrt{(x_T - x_R)^2 + (y_T - y_R)^2} \tag{2.18}$$

Portanto, as velocidades angular ω_R e linear V_R serão calculadas conforme o desempenho do controlador proposto, para os erros obtidos com as Equações 2.16 e 2.18, respectivamente.

3 Visão Robótica

A visão robótica envolve os estudos de diversas áreas como: o processamento de imagens, visão computacional, reconhecimento de padrões, dentre outras.

A área de processamento de imagens tem sido alvo de crescente interesse, visto que esta, viabiliza um grande número de implementações para o aprimoramento das imagens digitais através da aplicação de técnicas embasadas em operações matemáticas, alterando os *pixels* e corrigindo os defeitos. Conforme descrito em [Gonzales e Woods 2001], após a imagem ser capturada e digitalizada, pode-se melhorá-la a partir das correções de defeitos oriundos da aquisição, bem como realçar os detalhes de maior interesse, de modo a facilitar sua segmentação.

Entende-se por segmentação de uma imagem, o processo responsável por subdividir uma imagem qualquer em regiões homogêneas, que compartilhem certa similaridade. Esse processo é de grande importância para a visão computacional e para o processamento e a análise de imagens, apresentando grandes desafios e tendo vários estudos na área. [Gonzales e Woods 2001]

Um processo de fundamental importância para o processamento de imagem é o filtro de cores, que em implementação conjunta com a segmentação da imagem, permite localizar os objetos importantes para o controle do robô. Mas o grande desafio de aplicar tal filtro, está na dificuldade ocasionada pela diferença de luminosidade, que ocorre nas diferentes partes do ambiente por onde o robô se locomove, no caso do presente trabalho o campo. Outro problema observado em filtros simples é a dificuldade de separação correta de cores devido à sua proximidade no espectro em que a descreve.

No caso da visão computacional, a mesma também vem tendo grande expansão, pois para implementações como a proposta para este trabalho, tal sistema deve ser rápido e tolerante à variações de luminosidade, e quando possível, ser capaz de tolerar ruídos e variações de intensidade luminosa. Com relação ao reconhecimento de padrões, várias técnicas vem sendo aplicadas em trabalhos mais avançados de futebol de robôs, para auxiliar na localização e orientação dos robôs e na distinção de seus respectivos times, conforme apresentado [Bianchi e Reali-Costa 2000] e [Grittani, Gallinelli e Ramírez 2000], dentre outros.

3.1 Fundamentos da Cor

Dentre as várias teorias sobre a visão das cores, a mais aceita define que o olho humano possui em torno de 6 a 7 milhões de cones em cada olho, esses são mais conhecidos como receptores sendo divididos em três tipos, os quais são sensíveis à luz vermelha, à luz verde e à luz azul. No caso da incidência da luz branca sobre a retina do olho humano, todos os três receptores são estimulados igualmente. E quando da incidência de uma dessas cores primárias, somente os respectivos receptores serão estimulados, como por exemplo com a luz vermelha, que é percebida apenas pelos receptores sensíveis a essa radiação. No caso da incidência da cor amarela, a sensação resulta do fato de que tanto os receptores sensíveis ao verde quanto ao vermelho estão sendo estimulados com com a mesma intensidade [Cattin 2012].

Em 1666, Isaac Newtom descobriu que se um feixe de luz solar atravessasse um prisma de vidro, o feixe de luz emergente não seria mais o branco, mas sim um espectro contínuo de cores indo do violeta até ao vermelho, ou seja, numa sucessão de diferentes matizes semelhantes às observadas em um arco-íris. O fato de o prisma promover a decomposição da luz branca comprova a natureza ondulatória da radiação, uma vez que esta se deve à variação do índice de refração do prisma nos diferentes comprimentos de onda. Sendo assim, cada matiz decomposto estará relacionado a uma determina frequência de radiação, logo a um determinado comprimento de onda [Janeczko 2010].

As cores que conhecemos, conforme apresentado na Figura 3.1, pertencem ao intervalo da luz visível e representam uma pequena parcela do espectro eletromagnético, a qual se estende de 400nm a 700 nm aproximadamente. De acordo com a CIE *(Commisson Internationale de l'Eclairage)* é aceito internacionalmente desde de 1931, que os valores de 435,8 nm, 546,1 nm e 700 nm representam espectralmente as três cores primárias azul, verde e vermelho, respectivamente [Souto 2003].

As cores que são percebidas ao observar um dado objeto, são determinadas pela natureza da luz refletida pelo mesmo, assim se um objeto refletir de forma balanceada todas as componentes do estreito espectro visível ele será branco. Caso esse objeto reflita uma cor preferencial do espectro visível, ele passará a ter uma coloração.

3.1.1 Espaço de Cor

O Espaço de cor também pode ser chamado de modelos de cor ou ainda de sistema de cor e tem por finalidade facilitar a especificação das cores em um dado padrão aceito. Ele é um sistema tridimensional de coordenadas, onde cada eixo refere-se a uma cor primária. E a

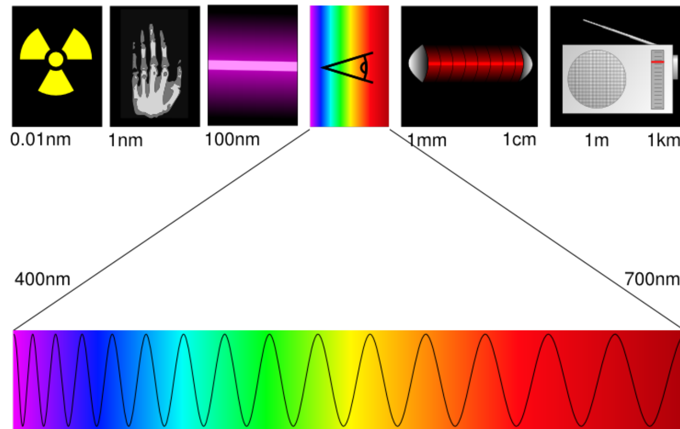


Figura 3.1: Espectro Eletromagnético.
Fonte: [Cattin 2012]

quantidade de cada cor primária necessária para reproduzir uma determinada cor, é atribuída a um valor sobre o eixo correspondente [Foley et al. 1996].

3.1.1.1 Espaço de Cor RGB

Este espaço de cor é baseado nas três cores aditivas primárias, ou seja, no vermelho, no verde e no azul, que do inglês tem-se respectivamente: *red* (R), *green* (G) e *blue* (B). A soma dessas três cores, duas a duas, resulta nas cores subtrativas, também conhecidas como secundárias, ciano (verde e azul), magenta (vermelho e azul) e amarelo (vermelho e verde), que do inglês tem-se respectivamente: *cyan* (C), *magenta* (M) e *yellow* (Y).

Na Figura 3.2, são apresentados três círculos preenchidos com as cores primárias, que interceptam-se entre si dando origem às cores secundárias. Onde os três círculos interceptam-se é formada a cor branco, resultando da soma das três cores primárias.

Figura 3.2: Círculos Representando as Cores Primárias do Sistema Aditivo.
Fonte: [Cattin 2012]

O espaço de cor RGB é tradicionalmente representado através de uma figura geométrica

denominada de “cubo de cores”, a qual está apresentada na Figura 3.3. Os valores sobre os eixos R, G e B, estão normalizados, ou seja, variam de 0 a 1. É possível observar que os vértices do cubo representam as cores aditivas primárias e subtrativas, com exceção de dois vértices, os referentes ao preto (*black*) e ao branco (*white*), onde ainda a diagonal que os une apresentam os tons de cinza (*grey scale*). Sobre as arestas do cubo estão localizadas as cores saturadas [1] neste espaço de cores.

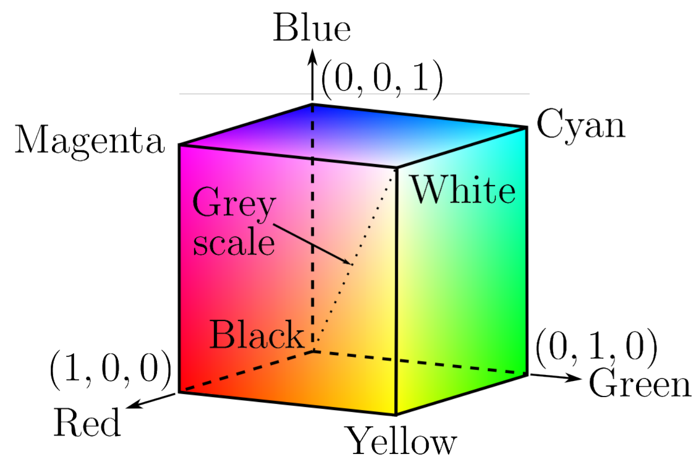


Figura 3.3: Cubo de Cores RGB.
Fonte: [Cattin 2012]

O espaço RGB é muito empregado em todo o tipo de aplicações digitais que envolvam o uso da cor, como no caso de armazenamento de imagens, descrição de cores em páginas web, dentre outras. O uso desse espaço como sendo o padrão (*standard*) para apresentação de cor na internet tem suas raízes no *standard RCA Color – TV* de 1953 e no uso por *Edwin Land* deste formato nas câmaras *Land / Polaroid*.

No entanto, este espaço de cor apresenta dois problemas principais, sendo o primeiro por ele não possuir sua representação de maneira similar à percepção de cor humana e a segunda por suas componentes estarem muito correlacionadas, tornando muito difícil as tarefas de identificação de cores nesse espaço e de segmentação de imagens que tenham sido obtidas com iluminação não uniforme ou com sombra [Júnior, Facon e Neto 1997].

Uma imagem digital no espaço de cor RGB é um *array* de cor de *M x N x 3 pixels*. Cada *pixel* é formado por um triplete de valores, que corresponde aos componentes vermelho, verde e azul, conforme pode ser visualizado na Figura 3.4. O número de bits usados para representar cada um desses *pixels* no espaço de cor RGB é chamado de *pixel depth*. Assim, em uma imagem de 8 bits, cada triplete de valores RGB irá possuir uma profundidade de 24 bits, visto que são três planos de imagem vezes o número de bits por cor de plano [Janeczko 2010].

[1] Cor saturada não significa pura, mas sim a cor que está restrita ao espaço de cor ao qual a mesma está inserida. O espaço de cor RGB não é capaz de reproduzir uma cor pura devido às suas limitações.

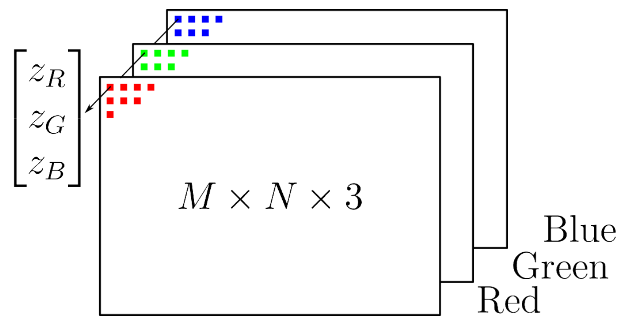


Figura 3.4: Imagem RGB.
Fonte: [Cattin 2012]

3.1.1.2 Espaço de Atributos de Cor

No espaço de cor RGB, os seus componentes, identificados como sendo os eixos do cubo de cor, representam uma quantização da energia, referente aos comprimentos de onda respectivos às cores primárias do espectro da luz visível.

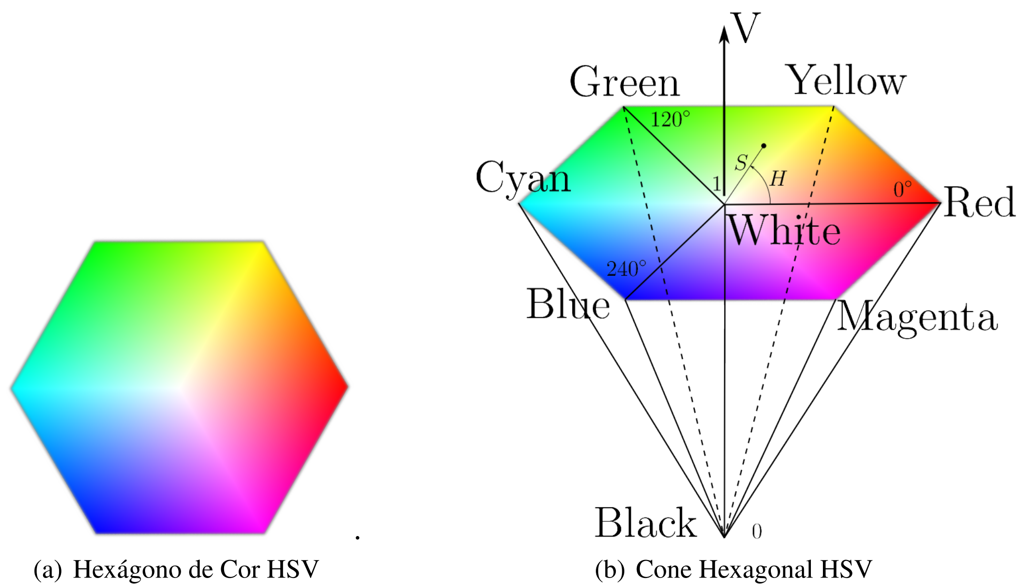
Sabe-se ainda, que qualquer ponto interior a esse espaço de cor irá representar uma cor a partir da combinação dessas três cores primárias. E dependendo dos valores dos componentes selecionados resultará em uma intensidade (brilho) associada, uma quantidade de luz branca, a qual determina a sua saturação e uma cor predominante denominada matiz ou tonalidade. Com essas novas referências, conhecidas como atributos, é possível de forma similar construir uma representação espacial chamada de atributos de cor para a mesma cor obtida pelo espaço de cor.

Esse espaço de atributos pode ser definido como um sistema de coordenadas cilíndricas (r, ϕ, z), onde o valor de saturação corresponde ao raio r, o valor de matiz corresponde ao ângulo ϕ e o valor de intensidade é distribuído ao longo de eixo z. Os valores de intensidade e de saturação variam de 0 a 1, enquanto os valores de matiz vão de 0 a 360 graus, sendo que para valores de ϕ iguais a 0, 120 e 240 graus, estão representadas as cores primárias vermelho, verde e azul, respectivamente [Gonzales e Woods 2001].

O sistema de coordenadas com esses atributos de cor recebe o nome de espaço de atributos HSI, onde H vem da palavra em inglês *Hue*, que é a tradução para matiz ou tonalidade, S vem da palavra em inglês *Saturation*, que é a tradução para saturação e I vem da palavra em inglês *Intensity*, que é a tradução para intensidade.

Existem outras representações espaciais do espaço de atributos, como é o caso do HSV abreviatura de *Hue*, *Saturation* e *Value*, que é a tradução para valor. Esse espaço de atributos também é conhecido como HSB (*Hue*, *Saturation* e *Brightness*, que significa brilho). É importante ressaltar que os atributos V do espaço de atributos HSV e o atributo I do HSI representam a mesma grandeza, ou seja, o brilho.

Esse espaço de atributos não possui uma representação cilíndrica, mas sim hexacônica, conforme apresentado na Figura 3.5(a). Essa diferença é devida a este espaço ser resultado de transformações geométricas do espaço de cor RGB a partir da forma como se observa o cubo de cores RGB. Para visualizar tal transformação, imagine-se observando o cubo ao longo do eixo da escala de cinza, ou seja, como se o cubo de cores RGB estivesse equilibrado sobre o vértice preto e as faces que formam o vértice branco fossem " empurradas" para o plano formado pelos demais vértices, formando o hexágono de cor HSV. Uma representação completa do espaço de atributos HSV pode ser visualizada na Figura 3.5(b), onde observa-se o hexacone ou cone hexagonal HSV [Souto 2003].



(a) Hexágono de Cor HSV (b) Cone Hexagonal HSV

Figura 3.5: Espaço de Atributos HSV.
Fonte: [Cattin 2012]

As transformações geométricas do espaço de cor RGB para o espaço de atributos HSV serão apresentadas a seguir nas equações 3.1, 3.2, 3.3 e 3.4, sendo que tais equacionamentos foram formulados para os valores RGB normalizados, ou seja, variando de zero a um.

$$H = \begin{cases} indefinido, geralmente = 0 & se \quad MAX = MIN \\ 60 \times \left(0 + \frac{G-B}{MAX-MIN}\right) & se \quad R = MAX \\ 60 \times \left(2 + \frac{B-R}{MAX-MIN}\right) & se \quad G = MAX \\ 60 \times \left(4 + \frac{R-G}{MAX-MIN}\right) & se \quad B = MAX \end{cases} \tag{3.1}$$

$$H = H + 360 \qquad se \quad H < 0 \tag{3.2}$$

$$S = \begin{cases} 0 & se \quad MAX = 0 \\ 100 \times \left(\frac{MAX-MIN}{MAX}\right) & se \quad H < 0 \end{cases} \tag{3.3}$$

$$V = 100 \times MAX \tag{3.4}$$

Sendo que: $MAX = max(R,G,B)$ e $MIN = min(R,G,B)$. E os intervalos para os valores obtidos serão: $H = [0, 360°]$, S e $V = [0, 100\%]$

Ao observar as relações de transformação é possível encontrar a primeira desvantagem do modelo de atributos de cor HSV. Há uma singularidade não removível próximo do centro do hexágono de cor (R=G). Fazendo com que pequenas variações nos valores de entrada R, G, B possam ocasionar grande variação nos valores transformados [Cheng et al. 2001] [Chapron 1992]. Tal singularidade pode trazer descontinuidades na representação da cor tornando-a instável, principalmente nos pontos de baixa saturação ou intensidade aproximando-se do branco ou preto [Cheng et al. 2001]. Outra desvantagem é o fato das relações de transformação não serem lineares. Isto resulta em um maior custo computacional para se obter esta representação, uma vez que o hardware utilizado trabalha em RGB. Todavia os processadores dos computadores atuais são capazes de efetuar tal transformação para aplicações em tempo real.

Uma das principais vantagens da utilização do espaço de atributos HSV é o caso do atributo matiz apresentar baixa influência da variação de iluminação na cena captada, isso devido às componentes HSV estarem muito descorrelacionadas, o que torna fácil a exclusão das informações de intensidade na segmentação por cor, o que torna o sistema praticamente insensível à iluminação não uniforme [Júnior, Facon e Neto 1997] [Cheng et al. 2001]. Outra vantagem é o fato desse modelo ser consideravelmente mais próximo, do que o sistema de cor RGB, para a maneira pela qual os seres humanos detectam e descrevem as sensações de cor.

3.2 *OpenCV*

OpenCV é uma biblioteca de visão computacional de código aberto (*open-source*) e de software de aprendizagem de máquina. Tendo a licença *BSD*[2], o que a possibilita de ser utilizada tanto para a construção de *softwares* livres quanto comerciais, igualmente ao ROS. Ela foi idealizada com o objetivo de proporcionar uma infra-estrutura comum para aplicações de visão

[2]Essa licença foi originada do sistema operacionam de mesmo nome (*Berkeley Software Distribution*), desenvolvido na Universidade da California, Berkeley, em 1988. As licenças da "Família BSD" são integrantes do grupo das licensas permissivas, as quais impõem restrições mínimas relativas à redistribuição do software aberto, ou seja, qualquer pessoa ou empresa pode utilizar o código conforme desejar e ainda realizar alterações para a criação de novos produtos fechado, como fazem a *Microsoft* e *Apple*.[Digital 2013] [Initiative 2013]

computacional e acelerar a implementação deste segmento como meio de percepção de máquina em produtos comerciais, tornando dessa forma a visão computacional acessível a usuários e programadores em diversas áreas, dentre as quais se destacam a interação humano-computador em tempo real e a robótica [OpenCV 2013b].

A seguir está apresentado um pequeno histórico, do surgimento até aos dias atuais dessa biblioteca:

- Em 1999, é desenvolvida a primeira versão da biblioteca *OpenCV* pela Intel Corporation [Intel 1999-2001];
- Em 2000 surge a sua versão Beta;
- Em 2006 é lançada a versão 1.0, a qual é uma API – Interface de Programação de Aplicativos baseada em C;
- Em 2008, o grupo de pesquisa Willow Garage assume o desenvolvimento da biblioteca;
- Em 2009 é lançada a versão 2.0, sendo essencialmente uma API em C++;
- Em 2011 é lançada a versão 2.3.1;
- Em 2012 é lançada a versão 2.4, a qual está sendo utilizada no presente trabalho;
- E em 2013 é lançada a versão mais atual 2.4.4, sendo uma API para Java.

O código fonte e os executáveis (binários) dessa biblioteca são otimizados para os processadores da Intel. Sendo que ao ser executado o programa em *OpenCV*, ele invoca automaticamente uma DLL (*Dynamic Linked Library*) que detecta o tipo de processador e a partir daí carrega uma DLL otimizada para tal processador [Intel 1999-2001].

A biblioteca possui interfaces em C++, C, Python e Java, sendo mais de 2500 algoritmos otimizados de visão computacional relacionados às várias áreas como: processamento de imagens, segmentação de imagens, detecção de movimento e rastreamento de objetos, reconhecimento de padrões em imagens, calibração de câmera e reconstrução 3D. Ela é suportada nas plataformas: Windows, Linux, Android e Mac OS [OpenCV 2013b].

OpenCV tem uma estrutura modular, dessa forma seu pacote inclui bibliotecas compartilhadas ou estáticas. Sendo que os seguintes módulos estão disponíveis [OpenCV 2013a]:

- *core* – um módulo compacto definindo as estruturas de dados básicas, incluindo o *array* multi-dimensional **Mat** e funções básicas usadas por todos os demais módulos;

- *imgproc* – um módulo de processamento de imagem, incluindo: filtros lineares e não-lineares de imagem, transformações geométricas de imagem (redimensionamento, deformação de perspectiva e remapeamento genérico baseado em tabela), conversão de espaço de cor, histogramas, dentre outros;
- *video* – um módulo de análise de vídeo, o qual inclui estimação de movimento, subtração de fundo e algoritmos de rastreamento de objetos;
- *calib3d* – algoritmos básicos de geometria de *multi-view*, calibração de câmera monocular e estéreo, estimação de pose de objeto, algoritmos de correspondência estéreo e reconstrução de elementos 3D;
- *features2d* – detecção de características salientes, descritores e descritor de correspondência;
- *objdetect* – detecção de objetos e instâncias de classes pré-definidas, como por exemplo: faces, olhos, carros e outros);
- *highgui* – uma interface de fácil uso para captura de vídeo, imagem e codecs de vídeo;
- *gpu* – algoritmos avançados para GPU (*Graphics Processing Unit*), a partir de diferentes módulos do *OpenCV*.

Para se ter uma ideia da disseminação dessa biblioteca na atualidade, a Equipe de Desenvolvedores do *OpenCV* apresentou em [OpenCV 2013b], um quantitativo onde mais de 47 mil pessoas estão inscritas na comunidade de usuários da biblioteca, sendo que o número estimado de downloads da mesma é superior a 5 milhões. Comprovando quão amplamente é utilizada essa biblioteca, principalmente em empresas, grupos de pesquisas e órgãos governamentais.

No decorrer desse capítulo serão apresentados os conceitos de processamento de imagens e de visão computacional, juntamente com as principais funções da *OpenCV*, utilizadas no presente trabalho.

3.2.1 Processamento de Imagens

A utilização de imagens ou de vídeos pelos processos de visão computacional, em sua grande maioria, requer uma etapa de pré-processamento, etapa essa que engloba o processamento de imagens. Em alguns casos, essas imagens que servem de base para extração das informações desejadas, precisam ser convertidas para um determinado formato ou tamanho, ou

ainda de serem filtradas, para assim, terem seus ruídos provenientes do processo de aquisição removidos [Marengoni e Stringhini 2010].

Os ruídos podem ser originados de várias fontes, como por exemplo: o tipo de sensor da câmera, a iluminação do ambiente, a posição relativa entre o objeto de interesse e a câmera, dentre outros. Tais ruídos não interferem apenas no sinal de captura da imagem, mas também podem prejudicar a interpretação ou o reconhecimento de objetos na imagem em análise. Um exemplo dessa interferência é a variação na tonalidade dos marcadores utilizados na identificação do robô móvel, ao longo do campo de futebol, devido à iluminação do mesmo não ser uniformemente distribuída em todas as suas extremidades.

Para remoção dos ruídos é feito o uso de ferramentas básicas, denominadas de filtros. Todavia, em alguns casos o uso somente do filtro não é capaz de eliminar tais ruídos, não dando assim suporte para a elaboração de um algoritmo robusto para se obter as características desejadas. Nessas situações é necessária a utilização de transformações no espaço da imagem.

Os filtros podem ser aplicados no domínio de espaço (filtros espaciais), em que se atua diretamente na imagem, ou no domínio da frequência, onde a imagem é inicialmente transformada para o domínio da frequência através da *Transformada de Fourier*, onde é filtrada neste mesmo domínio e na sequência é transformada novamente para o domínio de espaço. Apesar das duas possibilidades de filtro, os algoritmos utilizados, neste trabalho, são embasados no domínio do espaço.

O termo domínio espacial refere-se à imagem propriamente dita, sendo assim, os métodos aplicados nesse domínio farão uso da manipulação direta dos *pixels* da imagem. Dessa forma, os processos no domínio espacial são caracterizados pela Equação 3.5:

$$g(x,y) = T(f(x,y)) \tag{3.5}$$

onde:

$f(x,y)$ é a imagem original;
$T(.)$ é uma transformação na imagem;
e $g(x,y)$ é a imagem transformada.

A operação T é definida sobre uma vizinhança de influência do *pixel*, o qual está localizado na posição (x,y), ou seja, considera a contribuição dos *pixels* ao redor do mesmo. Tal vizinhança é definida como sendo uma região quadrada ou retangular e com tamanho ímpar. Na Figura 3.6 é possível visualizar alguns exemplos dessas regiões de tamanhos variados - $(k \times l)$, as quais

também são denominadas de máscaras, ou ainda *kernel* e definem as matrizes nas operações de transformação [Marengoni e Stringhini 2010] [OpenCV 2012].

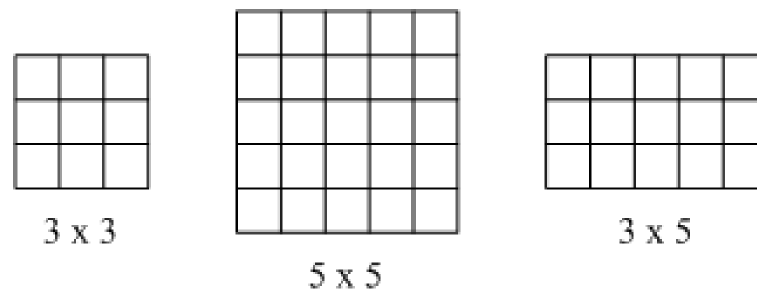


Figura 3.6: Exemplos de Regiões de Vizinhança.
Fonte: baseada em [Marengoni e Stringhini 2010]

Na sequência serão apresentadas, conceitualmente, as duas principais funções da *OpenCV*, que se enquadram na etapa de processamento de imagens.

3.2.1.1 Filtro

As funções e classes usadas para executar as várias operações de filtragem de uma imagem 2D, essa representada em um *array* **Mat**, podem ser do tipo linear ou não-linear. Sendo que para cada *pixel* de localização (x,y) da imagem fonte, normalmente na forma retangular, sua vizinhança é considerada e utilizada para calcular a imagem resposta. No caso do filtro linear, a influência da vizinhança é obtida através de uma soma ponderada dos valores de cada *pixel*. Já no caso de operações morfológicas, essa influência pode ser considerada por exemplo, através dos valores mínimos e máximos desses *pixels*. A resposta calculada a partir dessas ponderações será armazenada na imagem de destino, também um *array* **Mat**, e exatamente na mesma posição (x,y), sendo essa imagem do mesmo tamanho da imagem de origem [OpenCV 2012].

Uma outra característica comum dessas funções e classes é que ao contrário de funções aritméticas simples, essas necessitam extrapolar os limites da imagem fonte, fazendo assim o uso de *pixels* não existentes. Um exemplo dessa característica pode ser observado quando da aplicação de um filtro *Gaussian* 3*x*3, onde ao processar o *pixel* mais à esquerda da imagem em cada linha, será preciso o *pixel* mais à esquerda dele, isto é, fora da imagem. Para contornar esta dificuldade podem ser feitas algumas considerações de extrapolação, como por exemplo considerar que esses *pixels* inexistentes assumam os mesmos valores dos *pixels* limite, tal método de extrapolação recebe o nome de ”borda replicada”. Ou ainda, assumir que todos esses *pixels* inexistentes sejam iguais a zero, sendo esses método de extrapolação o de ”borda constante”[OpenCV 2012] [OpenCV 2013c] [Laganière 2011].

Os filtros muitas das vezes são utilizados com a finalidade de suavização, ou também cha-

mada de desfocagem de imagem, para assim atenuar o ruído e tornar mais uniforme as regiões na mesma [OpenCV 2013d]. Para executar a operação de suavização, necessária nesse trabalho, foi aplicado um filtro do tipo linear na imagem fonte. Dessa forma, o valor de um *pixel* de saída, aqui representado por: $g(x,y)$ será determinado pela soma ponderada dos valores de *pixels* da imagem de entrada, denotado por: $f(x+k,y+l)$, conforme a Equação 3.6 a seguir [OpenCV 2012]:

$$g(x,y) = \sum_{k,l} f(x+k,y+l)h(k,l) \tag{3.6}$$

onde:

x e y são as coordenadas de localização do pixel, respectivamente na horizontal e na vertical;
k e l são as dimensões do *kernel*, respectivamente na horizontal e na vertical;
$h(k,l)$ é a denominada máscara ou *kernel*, com os coeficientes do filtro.

O Filtro *Gaussian* é provavelmente o filtro mais útil, embora não seja o mais rápido. Sua aplicação se dá pela convolução de cada *pixel* do *array* **Mat** de entrada com o *kernel Gaussian*, resultando no *array* de saída. Para facilitar a visualização, considere a imagem como sendo de uma dimensão, dessa forma o *kernel Gaussian* pode ser visto como mostrado na Figura 3.7 [OpenCV 2013c].

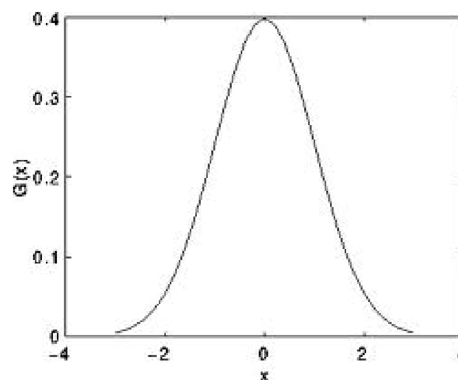


Figura 3.7: Comportamento do Filtro *Gaussian* 1D.
Fonte: [OpenCV 2012]

Ainda considerando a imagem como sendo 1D e a Figura 3.7, pode-se perceber que o *pixel* centrado na máscara terá o maior peso, o mesmo não é observado para os *pixels* vizinhos, onde seus pesos vão diminuindo à medida que se distanciam do elemento central.

Na realidade, um filtro *Gaussian* para uma imagem 2D é regido pela Equação 3.7, mostrada a seguir [OpenCV 2013d]:

$$G_0(x,y) = A \cdot e^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}} \tag{3.7}$$

onde:

A é uma constante;
x e y são as coordenadas de localização do pixel, sendo respectivamente a abscissa e a ordenada;
μ_x e μ_y são as médias dos valores em x e y;
σ_x e σ_y são as variâncias dos valores em x e y.

Na biblioteca *OpenCV* e considerando a linguagem de programação *C++*, esse filtro é realizado através da função *GaussianBlur*, a qual pertence ao módulo *imgproc* seção *Image Filtering* do *OpenCV* e possui a seguinte estrutura:

void **GaussianBlur** (InputArray **src**, OutputArray **dst**, Size **ksize**, double **sigmaX**, double **sigmaY**=0, int **borderType**=BORDER_DEFAULT)

onde os parâmetros significam:

src - imagem de entrada com qualquer número de canais, os quais são processados independente, mas a profundidade deve ser `CV_8U, CV_16U, CV_16S, CV_32F ou CV_64F`;
dst - imagem de saída como as mesmas dimensões da de entrada;
ksize - tamanho da máscara *Gaussian*. `ksize.width` e blueksize.heigth podem ser diferentes, mas ambos devem ser positivos e ímpares. Ou eles podem ser zero, desse forma são calculados a partir de `sigma*`;
sigmaX - Desvio Padrão do *kernel Gaussian* na direção X;
sigmaY - Desvio Padrão do *kernel Gaussian* na direção Y. Se **sigmaY** for zero, implica que será igual à **sigmaX**. Caso ambos sejam iguais a zero, eles serão obtidos a partir de `ksize.width` e `ksize.heigth`, respectivamente;
borderType - especifica o método de extrapolação de *pixel*, como já explicado anteriormente.

3.2.1.2 Conversão entre Espaços de Cor e de Atributos

O espaço de cor RGB se baseia no uso das cores: vermelho, verde e azul, sendo essas, conforme apresentado na Seção 3.1, as cores aditivas primárias. A escolha desse espaço de cor, para implementações visuais, se deve ao fato da vasta gama de cores distintas que podem ser obtidas a partir da combinação entre suas cores base, além de ser normalmente o espaço de cor utilizado para representação de imagens digitais. Outro fato importante a ser considerado é que o sistema visual humano também é baseado na percepção tricromática de cores, visto que a sensibilidade das células dos olhos está localizada ao redor do espectro do vermelho, verde e azul [Laganière 2011].

Entretanto, o cálculo da distância entre as cores, pertencentes ao espaço de cor RGB, não é a melhor maneira de se medir a proximidade ou distinção entre elas, visto que esse não é um espaço de cor perceptualmente uniforme. Por exemplo, duas cores a uma dada distância podem parecer muito semelhantes, enquanto que outras duas cores que apresentam a mesma distância podem ser muito diferentes. Com o intuito de sanar tal problema, podem ser utilizados outros espaços de cor ou de atributos conforme já demonstrado na Seção 3.1, os quais apresentam a propriedade de serem perceptivelmente uniforme.

Dessa forma ao se converter uma imagem base para um desses espaços, a distância euclidiana entre um pixel da imagem e a cor de interesse será uma medida da semelhança visual entre as duas cores. Com base em toda teoria já apresentada, para aplicações de visão computacional, se torna extremamente importante a realização dessa conversão entre espaços, sendo os mais indicados os espaços de atributos HSV e HLS, pois representam a forma mais natural de descrição das cores pelos seres humanos.

Neste trabalho além da conversão entre o espaço de cor RBG para o espaço de atributos HSV, foi necessária a conversão para escala de cinza, para utilização na etapa de segmentação da imagem.

Esses dois tipos de conversões são funções que pertencem ao módulo *imgproc* seção *Miscellaneous Image Transformations* do *OpenCV*, cujas estruturas serão apresentadas e detalhadas a seguir [OpenCV 2013e].

void **cvtColor** (InputArray **src**, OutputArray **dst**, int **code**, int **dstCn**=0)

onde os parâmetros significam:

src - imagem de entrada: `8-bit unsigned, 16-bit unsigned (CV_16UC, ...),`

ou com precisão de ponto flutuante;

dst - imagem de saída como a mesma dimensão e profundidade como a entrada;

code - código de cor da conversão de espaço;

dstCn - número de canais da imagem de destino, se o parâmetro for 0, o número de canais será derivado da **src** e do **code**.

No caso de uma transformação de espaço de cor a partir do RGB, as ordem dos canais devem ser explicitamente especificado, ou seja, RGB ou BGR. Pois o formato de cor padrão no OpenCV é referida constantemente como sendo BGR, o que na verdade é apenas uma inversão na ordem de armazenamento dessas componentes no *array* **cvMat**. Desse jeito, o primeiro *byte*[3] em uma imagem a cor padrão de 24 *bits* (tamanho do *pixel*) será a componente azul, o segundo *byte* será o verde e o terceiro o vermelho.

No caso da transformação do espaço de cor RGB para a escala de cinza, será realizada uma compressão das três componentes da imagem origem, uma vez que de três canais será obtido um único. Essa transformação segue o seguinte raciocínio [OpenCV 2013e]:

RGB to Gray:

$$Y \leftarrow (0.299 \times R) + (0.587 \times G) + (0.114 \times B) \tag{3.8}$$

E a programação ficaria da seguinte forma:

```
cvtColor (src, dts_gray, CV_BGR2GRAY);
```

Já no caso da conversão entre o espaço de cor RBG e o espaço de atributos HSV, serão realizadas as transformações conforme as Equações: 3.1, 3.2, 3.3 e 3.4, apresentadas na Seção 3.1. Tendo a sua programação conforme mostrado a seguir:

```
cvtColor (src, dts_hsv, CV_BGR2HSV);
```

3.2.2 Visão Computacional

Na Subseção 3.2.1 foi abordado o processamento de imagens, onde ficou visível que os processos apresentados possuíam um caráter de baixo nível no tratamento das imagens, pois consistiam de recursos aplicados pontualmente nos *pixels*, com o intuito de reduzir ruídos ou suavizar as imagens, bem como transformações para melhorar a aquisição das características de interesse.

[3]Vale ressaltar que um *byte* é o conjunto de 8 *bits*, sendo esse por sua vez a menor unidade de representação de uma informação em formato digital.

Já nesta Subseção, serão abordados processos em um nível intermediário, sendo esses embasados na segmentação de imagens, a qual consiste na extração de informações da imagem de origem para a formação de uma imagem secundária, onde estarão presentes menos informações, contudo mais relevantes para a visão computacional propriamente dita.

3.2.2.1 Subtração de Fundo

O procedimento de subtração de fundo é um passo fundamental em aplicações no campo da visão computacional, onde deseja-se capturar um movimento e consequentemente um novo objeto em um cena com o fundo estático, como é o caso do robô no campo de futebol. Pois como será apresentado na próxima seção, sobre Transformada de Hough, a busca pelas características de interesse passará a ser feita apenas na imagem com os elementos adicionados ao plano de fundo e não na imagem num todo, trazendo assim uma redução significativa do tempo de busca pelas características desejadas e do processamento do algoritmo.

Esse processo consiste na comparação entre uma imagem recém observada com uma outra de referência que representa uma estimativa do plano de fundo, onde as regiões que apresentarem uma diferença acima de um determinado limiar, indicarão a localização dos objetos de interesse. Geralmente a subtração de fundo inclui as seguintes etapas:

- inicialização do modelo do plano de fundo até um determinado limiar;
- manutenção deste modelo após o limiar e;
- detecção do primeiro plano.

Vários são os métodos de subtração de fundo que foram desenvolvidos ao longo dos anos, conforme alguns listados em [Sobral 2013], porém cada um desses segue a estrutura, a linguagem de programação, a formatação do código fonte, o sistema operacional etc, conforme as familiaridades dos próprios autores. Tais particularidades acabam por dificultar o desenvolvimento e a integração de sistemas que fazem uso dessa tecnologia.

Para sanar essas inconveniências foi lançada, em março de 2012, uma biblioteca de código aberto e livre para uso acadêmico e não-comercial, padronizada e independente de sistema operacional, a BGSLibrary. Essa biblioteca fornece um *framework* em C++ para a realização da subtração de fundo em imagens ou em vídeo. Inicialmente, essa contava com 14 algoritmos para execução desse procedimento, mas na atual versão 1.5.0, já são contabilizados 29 algoritmos, sendo seu uso vinculado à biblioteca *OpenCV*.

O processo de subtração de fundo só é viável em aplicações em tempo real, como é o caso da aplicação efetuada no presente trabalho, quando o mesmo é avaliado quanto ao seu consumo de memória, tempo de execução e percentual de ocupação da CPU (*Central Processing Unit*). Pois essas características poderão ser críticas e decisivas na pesquisa e desenvolvimento de uma solução.

Dessa forma, a escolha dos métodos que serão utilizados nesta etapa foi fundamentada no *benchmark*[4] e na avaliação da precisão dos 29 algoritmos de subtração de fundo apresentada em [Sobral 2013].

3.2.2.2 Transformada de Hough

A Transformada de Hough (TH) foi desenvolvida por Paul Hough em 1962, sendo um método padrão para detecção de formas que são facilmente parametrizadas como: linhas, círculos, elipses, dentre outros em imagens digitalizadas. Seu principal conceito está em definir um mapeamento entre o espaço de imagem e o espaço de parâmetros, também conhecido como: Espaço de Hough [Jain, Kasturi e Schunck 1995]. A parametrização de uma classe de objeto define a forma do objeto, portanto, variações de cor na imagem, ou até mesmo sobre os objetos, não afetam o desempenho e a eficiência da TH. Para detectar os objetos em uma imagem, a TH tenta combinar as bordas localizadas na imagem com o modelo parametrizado do objeto.

A ideia é aplicar uma transformação na imagem de origem, tal que todos os pontos pertencentes a uma mesma curva sejam mapeados em um único ponto no novo espaço de parametrização da curva procurada. Para para melhor explicar o funcionamento dessa técnica, suponha que a forma a ser encontrada seja uma linha (Transformada de Hough para linhas), sendo essa descrita pela Equação 3.9:

$$y = mx + c \tag{3.9}$$

Nessa Equação 3.9, observa-se que x e y são as variáveis, e m e c representam os parâmetros. Caso os valores dos parâmetros sejam conhecidos, a relação entre as coordenadas do ponto é especificada claramente. Reescrevendo a equação, mas desta vez em função dos parâmetros obtém-se a Equação 3.10:

[4] Na computação, um *benchmark* pode ser entendido como o ato de executar um conjunto de avaliações sobre determinadas aplicações, de modo a obter o desempenho relativo de cada uma delas. Tais avaliações são efetuadas utilizando-se um conjunto padrão de testes e devem permitir, a partir da análise dos resultados obtidos, que seja realizada alguma comparação entre as mesmas [Gomes 2009].

$$c = -xm + y \tag{3.10}$$

Verifica-se agora que as variáveis de interesse são m e c, e x e y são constantes. Ainda assim, a Equação 3.10 representa uma reta, porém agora no espaço $m \times c$, sendo a inclinação e a interceptação desta linha com o eixo das ordenadas representada respectivamente por x e y. Outro fato a ser observado é que um ponto (x, y) qualquer no plano $x \times y$ corresponde a uma reta no espaço $m \times c$, conforme pode ser visualizado na Figura 3.8. É importante mencionar que a forma da curva no espaço de parâmetros dependerá da função original usada para representação, sendo na prática utilizada a equação polar da linha, conforme a Equação 3.11. Essa representação polar se faz necessária para evitar problemas com as linhas que por ventura vierem a ficar muito próximas da vertical.

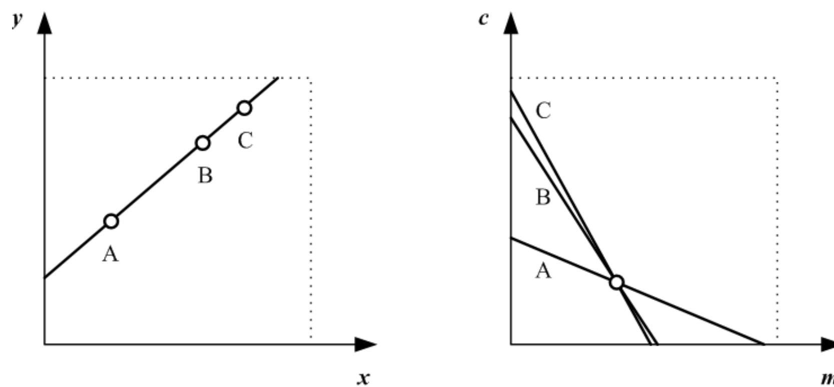


Figura 3.8: Mapeamento de uma linha existente no plano $x \times y$ para o plano $x \times y$.
Fonte: baseada em [Jain, Kasturi e Schunck 1995]

$$\rho = x \cos\theta + y \sin\theta \tag{3.11}$$

Com a realização dessa transformação, vários serão os candidatos, no espaço de parâmetros, a serem a linha desejada no espaço original. Entretanto, todos esses terão suas ocorrências armazenadas em um acumulador, ou seja, seus votos contabilizados, e de posse dos mesmos será viável a seleção dos vencedores quando um valor máximo for alcançado. Os vencedores por sua vez, irão fornecer os parâmetros originais da linha.

No caso da Transformada de Hough para círculos, que é a função de interesse para o presente trabalho, ela é mais fácil de representar no espaço de parâmetros, em comparação com a linha, uma vezes que os parâmetros do círculo podem ser transferidos diretamente para o espaço de parâmetros. Em contrapartida, seu espaço de parâmetros é tridimensional, apresentando dois parâmetros que definem o centro - (x_c, y_c) e o outro, o raio do círculo - r [Pedersen 2007].

A equação implícita para o círculo é:

$$(x-x_c)^2+(y-y_c)^2=r^2 \tag{3.12}$$

Já as equações paramétricas para um círculo em coordenadas polares são:

$$x=x_c+r\cos\theta \tag{3.13}$$

$$y=y_c+r\sin\theta \tag{3.14}$$

Fazendo a manipulação das Equações 3.13 e 3.14, obtém-se as equações em função dos parâmetros do círculo, conforme a seguir:

$$x_c=x-r\cos\theta \tag{3.15}$$

$$y_c=y-r\sin\theta \tag{3.16}$$

Contudo, os círculos a serem identificados na imagem, os quais são utilizados para a identificação do robô quanto ao time e ao jogador, não sofrem alteração de tamanho, ou seja de raio, visto que o robô apenas se desloca em duas dimensões - o plano. Dessa forma, o espaço de parâmetros passa a ser bidimensional o que viabiliza a utilização deste método em uma aplicação em tempo real. Sendo assim, de posse do valor de r e das Equações 3.15 e 3.16 é possível realizar a Transformada de Hough para círculos, conforme demonstrado na Figura 3.9.

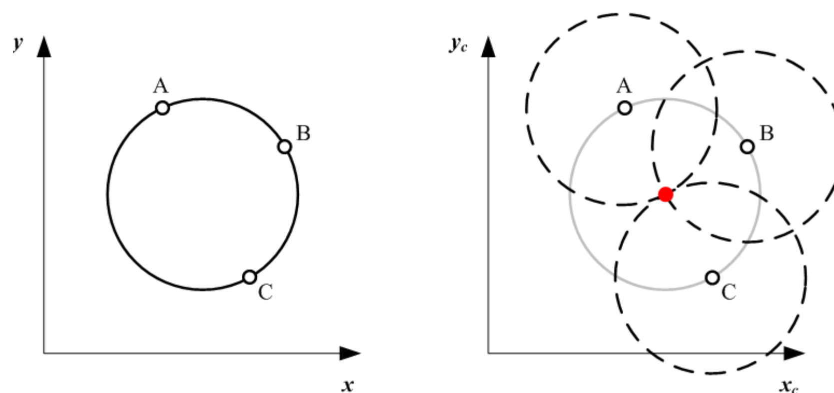


Figura 3.9: Exemplo de geração de pontos no espaço de Hough.
Fonte: baseada em [Martins 2007]

Como pode ser observado, cada ponto da borda da imagem é responsável por gerar um

círculo no espaço de Hough e cada ponto da borda desse círculo, no espaço de parâmetros, recebe um voto. Quanto maior o número de votos recebidos por um ponto, maior a probabilidade desse ponto ser o centro de um círculo verdadeiro. Tais pontos de maior probabilidade são os máximos relativos do espaço de Hough e definem os centros dos objetos na imagem.

A estrutura dessa técnica de segmentação na biblioteca do *OpenCV* está relacionada ao módulo *imgproc* seção *Feature Detection*, a qual está apresentada a seguir [OpenCV 2013f].

> void **HoughCircles** (InputArray **image**, OutputArray **circles**, int **method**, double **dp**, double **minDist**, double **param1**=100, double **param2**=100, int **minRadius**=0, int **maxRadius**=0)

onde os parâmetros significam:

image - imagem de entrada em escala de cinza, sendo de 8 *bits* e canal simples;
circles - vetor de saída dos círculos encontrados. Cada vetor é codificado como um vetor de 3 elementos em ponto flutuante $(x_c, y_c, raio)$;
method - Método de detecção utilizado. Até o momento de implementação desse código, o único método implementado é o `CV_HOUGH_GRADIENTE`.
dp - proporção inversa da resolução do acumulador para a resolução da imagem. Ou seja, se DP = 1, o acumulador terá a mesma resolução da imagem de entrada, caso DP = 2, o acumulador terá a metade das dimensões de largura e altura.
minDist - mínima distância entre os centros dos círculos detectados. Caso esse seja muito pequeno, vários falsos círculos vizinhos podem ser detectados, além de um verdadeiro. No caso de ser muito grande, alguns círculos verdadeiros podem ser desconsiderados.
param1 - primeiro parâmetro do método de detecção especificado. No caso do `CV_HOUGH_GRADIENT`, é o *threshold* superior dos dois valores passados para o detector de bordas **Canny()**[5], sendo que o inferior é metade desse valor.
param2 - Segundo parâmetro do método de detecção especificado. No caso do `CV_HOUGH_GRADIENT`, é *threshold* do acumulador para os centros de círculos na fase de detecção, ou seja, é o número mínimo de votos para o centro ser considerado como sendo de um círculo verdadeiro.
minRadius - raio mínimo do círculo a ser detectado.
maxRadius - raio máximo do círculo a ser detectado.

[5]Método de detecção de bordas em uma imagem usando o algoritmo de Canny.

Como já abordado anteriormente, o método da Transformada de Hough necessita de alguns procedimentos anteriores ao processo de transformação propriamente dito. Dessa forma, na própria função da *OpenCV* são solicitados alguns parâmetros para tais procedimentos.

De posse da imagem de origem, sendo que essa já deve estar em escala de cinza, é preciso aplicar o método de Canny, o qual será responsável por converter a imagem de entrada em uma imagem binária onde serão enfatizadas as bordas de todas as estruturas presentes na imagem.

Apesar de se ter demonstrado o desenvolvimento do método de Hough para detecção de círculos, com a simplificação quanto ao raio ser constantes, na prática é considerado uma pequena variação para mais e para menos em cima do valor do raio do círculo procurado. Isso se faz necessário para tornar o código de busca mais robusto com relação à interferência de ruídos na imagem binária.

3.3 Sistema de Captura de Imagens

A câmera escolhida para a tarefa de captura da imagem é uma câmera do modelo *Basler Scout scA640-120fc*, sendo a mesma digital e com conexão *IEEE 1394*, mais conhecida como *FireWire*, conforme pode ser visto na Figura 3.10.

Figura 3.10: Câmera *Basler Scout scA640-120fc*.
Disponível em: <http://www.baslerweb.com/>

Juntamente a essa câmera foi utilizada uma lente, também conhecida como objetiva, da marca Fujinon modelo DV3.4x3.8SA-1, conforme mostrada na Figura 3.11. Na verdade, trata-se de um sistema óptico que consiste na composição de lentes, as quais são responsáveis pelo direcionamento e dimensionamento dos feixes de luz, provenientes do cenário para o sensor óptico presente na câmera. Tendo por finalidade definir a superfície de foco, bem como a magnificação - ampliação ou redução - da imagem capturada [Stemmer et al. 2005].

As câmeras digitais apresentam suas características básicas semelhantes às analógicas, como por exemplo serem monocromáticas ou coloridas, porém disponibilizam um sinal de vídeo de saída já digitalizado para a transferência das imagens adquiridas, dispensando então o uso de placa de aquisição de vídeo. Além dessa vantagem, observa-se o fato do sinal ser digital,

Figura 3.11: Lente *Fujinon DV3.4x3.8SA-1*.
Fonte: Manual do Fabricante.

o qual é mais robusto, quando comparado ao sinal analógico, por ser praticamente imune aos ruídos eletromagnéticos, permitindo dessa forma, altas taxas de transferência de dados, maiores níveis de contraste de cores ou tons de cinza (de 8 a 16 bits). [Stemmer et al. 2005]

Com relação ao protocolo *IEEE 1394*, ele é baseado na tecnologia *LVDS - Low Voltage Differential Signaling*[6], que dispensa o uso de placas de aquisição de imagens, contando atualmente com duas versões a IEEE1394a e a IEEE1394b, sendo essa última a existente na câmera mencionada. Tal diferenciação se deve ao fato do padrão "a" permitir o alcance de taxas de transferência de até 400Mbits/s, com cabeamento padrão em cobre, e o padrão "b" permitir o alcance de taxas de transferência de até 3200Mbits/s, com cabeamento de fibra ótica. Outra vantagem desse protocolo é a possibilidade de conexão de dispositivos a qualquer momento (*hotplug*). [Stemmer et al. 2005]

[6] Padrão de sinal digital, também conhecido como RS644, que apresenta um sinal diferencial, permitindo maiores taxas de transmissão de dados e a utilização de cabeamento mais longo, sendo um padrão mais recente e superior ao RS422

4 Robotic Operating System - ROS

Desenvolver códigos de programa para controle de robôs é difícil e complexo, pois a área de robótica está em contínuo desenvolvimento, ampliando assim sua escala e seu campo de atuação. Outro fator que contribui é a gama de diferentes robôs, os quais podem apresentar uma ampla variedade de *hardware*, o que torna a reutilização desses códigos, nesses diferentes tipos de robôs, quase inviável. Além de tudo isso, a própria extensão do código pode ser gigantesca, visto que o mesmo deve apresentar uma abordagem iniciando a partir do controle de baixo nível (*drivers*) e continuando através da percepção, raciocínio abstrato e mais além. Essa extensão demanda também um nível de especialização que vai além das capacidades de um único pesquisador e na tentativa de auxiliar o desenvolvimento desses códigos, arquiteturas de programas robóticos vem dando apoio a integração códigos em larga escala [Cousins 2010].

Com o intuito de enfrentar tais desafios, vários pesquisadores criaram uma grande variedade de plataformas (*frameworks*) para gerenciar a complexidade e facilitar a prototipagem de programas para pesquisas, o que resultou em muitos sistemas de programas robóticos que atualmente são usados em Universidades e Indústrias [Quigley et al. 2009]. Cada um desses, foi projetado para atuar em um ponto específico, de modo a atender às fraquezas percebidas de outras plataformas ou enfatizar fatores que antes eram vistos como menos importantes. O ROS é uma dessas plataformas desenvolvidas a partir de compensações e de priorizações realizadas durante o processo de projeto.

4.1 ROS

O ROS é um sistema de código fonte aberto, meta-operacional[1] para robôs, que disponibiliza bibliotecas e ferramentas para ajudar os desenvolvedores de programas robóticos a criarem suas aplicações. Ele fornece os serviços esperados de um sistema operacional, incluindo abstração de *hardware*, controle de baixo nível de dispositivos (*drivers*), implementação de

[1]Sistema operacional no qual vários outros sistemas operacionais são ativados, comumente chamados de supervisor.

funcionalidades de uso comum (bibliotecas), visualizadores, troca de mensagens entre processos e gerenciamento de pacotes. O ROS é similar, em alguns aspectos, à outras plataformas robóticas, tais como *Player*, *YARP*, *Orocos*, *CARMEN*, *Orca*, *MOOS* e *Microsoft Robotics Studio* [Coleman 2013a].

O "grafo" da execução do ROS é uma rede ponto a ponto (*peer-to-peer*) de processos que estão em "baixo acoplamento" (*loosely coupled*) utilizando a infraestrutura de comunicação do ROS. O ROS implementa vários estilos diferentes de comunicação, incluindo a comunicação síncrona, estilo RPC (*Remote Procedure Call*) sobre os serviços, *streaming* de dados assíncronos sobre os tópicos e armazenamento de dados em um servidor de parâmetros [Coleman 2013a].

O ROS não é um *framework* em tempo real, embora seja possível a integração com código em tempo real. Um exemplo de tal integração é o Robô PR2 da *Willow Garage* que faz uso de um sistema *pr2_etherCAT*, o qual transporta mensagens do ROS de dentro para fora de um processo em tempo real [Coleman 2013a].

Para complementar os objetivos principais do ROS de compartilhamento e colaboração, ainda pode-se enumerar as seguintes características:

- Leve:
  O ROS é projetado para ser o mais leve possível, de forma que o código escrito para o ROS possa ser usado com outras plataformas de programas robóticos. Para que isso seja possível é necessário que o desenvolvimento de todo *driver* e algoritmo seja embasado em bibliotecas independentes. Essas por sua vez, devem seguir o modelo de desenvolvimento com interfaces funcionais limpas, sem dependências com o ROS, permitindo mais facilmente a extração do código e a reutilização do mesmo além de sua intenção original. Devido a essa característica o ROS já foi integrado ao *OpenRAVE*, *Orocos* e *Player* [Coleman 2013a].

  Outro fator que contribui para o ROS ser leve é a reutilização dos códigos de inúmeros outros projetos de código fonte aberto, como os *drivers*, o sistema de navegação, e os simuladores do projeto *Player*, algoritmos de visão computacional do *OpenCV*, e algoritmos de planejamento do *OpenRAVE*, entre muitos outros. Em cada um dos casos, o ROS é utilizado apenas para expor as várias opções de configuração e para encaminhar os dados de dentro ou fora dos respectivos programas, com o mínimo de envolvimento ou adaptações possível [Quigley et al. 2009].

- Multi-linguagem:
  A escolha de uma linguagem de programação depende das ponderações pessoais entre

tempo de programação, facilidade de depuração, sintaxe, eficiência no tempo de execução e uma série de outras razões técnicas e culturais. Devido a isso, o ROS foi projetado para ser uma linguagem neutra, sendo atualmente suportado quatro idiomas diferentes: *C++*, *Python*, *Octave* e *LISP*, tendo outras linguagens em vários estágios de avaliação e conclusão. Ao invés do ROS fornecer uma implementação baseada em C com interfaces geradas para todas as principais linguagens, foi escolhido implementar o ROS nativamente em cada linguagem de interesse, para melhor seguir as convenções de cada uma [Quigley et al. 2009].

Para apoiar o desenvolvimento através das linguagens, o ROS utiliza uma linguagem simples e neutra, a *IDL* - linguagem de descrição de interface, para descrever as mensagens enviadas entre os módulos. Os geradores de código para cada linguagem suportada geram implementações nativas, as quais se "sentem" como objetos nativos e são automaticamente serializados e desserializados, ou seja, traduzido de uma linguagem para outra pelo ROS. Isso poupa considerável tempo de programação e evita erros.

O resultado final é uma mensagem de linguagem neutra processando um esquema onde diferentes linguagens podem ser misturadas e combinadas conforme desejado.

- Baseado em ferramentas:
  Com o intuito de melhor gerenciar a complexidade do ROS, definiu-se por um projeto de *microkernel* (micronúcleo), onde um grande número de pequenas ferramentas são usadas para criar e executar os vários componentes do ROS, em oposição a construção de um desenvolvimento monolítico e um ambiente de execução.

  Dentre as várias tarefas que essas ferramentas executam pode-se citar como exemplo: navegar pela a árvore do código fonte, obter e definir parâmetros de configuração, visualizar a topologia da conexão ponto-a-ponto, medir a utilização da largura de banda, plotar graficamente os dados da mensagem, dentre outras. Apesar de se ter buscado a implementação em módulos separados, acreditasse que a perda de eficiência é mais do que compensada pelo ganho de estabilidade e gestão da complexidade [Quigley et al. 2009].

O ROS possui três níveis de conceitos: a nível de Sistema de Arquivos, a nível de Processamento Grafo e a nível de Comunidade. O ROS ainda define dois tipos de nomes: Nomes de Recurso de Pacote e Nomes de Recurso Grafo. Estes níveis e conceitos são apresentados detalhadamente nas seções seguintes.

4.1.1 Sistema de Arquivos do ROS

Os conceitos de nível de sistema de arquivos são recursos do ROS, os quais se encontram salvos no computador, sendo os mesmos divididos em: *pacotes*, *manifestos*, *mensagens*, *serviços* e *pilhas*.

4.1.1.1 Pacotes (*packages*)

O *Software* no ROS está organizado em pacotes, nos quais podem estar contidos executáveis (nós), uma biblioteca ROS independente, um conjunto de dados, arquivos de configuração, parte de *software* de terceiros, ou qualquer outra coisa que, logicamente, constitua um módulo útil. O objetivo desses pacotes é fornecer essas funcionalidades de modo fácil para que o *software* possa ser reutilizado. O princípio que rege os pacotes no ROS é o de *Goldlocks*, ou seja, os mesmos devem ter funcionalidade suficiente para ser útil, mas não muita para que não se torne pesado e difícil de ser usado por outros *software* [Conley 2012a].

Os pacotes no ROS ainda tendem a seguir uma estrutura comum, podendo assim ser compostos por alguns dos arquivos e diretórios listados a seguir.

- **bin/** - contém binários compilados, os quais podem ser executados diretamente em um sistema Unix;
- **include/package_name** - contém cabeçalhos em C++;
- **msg/** - contém arquivos de descrição do tipo *mensagem* (.msg);
- **src/package_name** - contém arquivos fontes (.cpp, .py, etc);
- **srv/** - contém arquivos de descrição do tipo *serviço* (.srv)
- **scripts/** - contém *scripts* executáveis;
- **CMakeLists.txt** - arquivo de construção CMake;
- **launch/** - contém a ferramenta *roslaunch*, a qual lê os arquivos *launch* (.launch/XML);
- **manifest.xml** - arquivo manifesto do pacote;
- **mainpage.dox** - muitos pacotes colocam sua página principal *Doxygen* neste arquivo.

4.1.1.2 Manifestos (*manifests*)

Manifestos são arquivos que estão presentes nos pacotes ROS, os quais recebem o nome `manifest.xml` e são uma especificação mínima sobre o pacote suportando uma ampla variedade de ferramentas ROS, indo desde de compilação a documentação sobre a distribuição. Além disso, eles fornecem uma especificação mínima de metadados[2] sobre seu pacote, outra funcionalidade é declarar as dependências de uma forma neutra tanto a nível de linguagem, quanto à sistema operacional.

Os arquivos de manifesto se assemelham à arquivos do tipo *leia-me*, relatando por exemplo o autor e a licença sob a qual o pacote foi escrito. Nos arquivos mais comuns, são observados *tags* do tipo <depend > e <export>, as quais auxiliam a instalação e a utilização de um pacote ROS [Foote 2013b].

4.1.1.3 Pilhas (*stacks*)

Os pacotes no ROS são organizados em pilhas, para assim, se criar pequenas coleções de códigos com o objetivo de facilitar a sua reutilização, bem como simplificar o processo de compartilhamento dos códigos. As pilhas são o mecanismo primário no ROS para a distribuição de *software*, possuindo uma versão associada e podendo ainda declarar dependências em outras pilhas. Essas dependências por sua vez, também declaram um número de versão, a qual proporciona maior estabilidade no desenvolvimento do ROS [Conley 2012b].

As pilhas colecionam pacotes que fornecem funcionalidade, formando assim uma pilha de navegação ou pilha de manipulação. Ao contrário de uma biblioteca de *software* tradicional, que realiza o *link* apenas no momento da compilação do código, essas pilhas também podem fornecer essa funcionalidade durante a execução via tópicos e serviços.

De forma semelhante a um pacote, as pilhas também possuem um arquivo manifesto (`stack.xml`). Mas é de extrema importância relatar que nas distribuições mais recentes do ROS: ROS Groovy Galapagos e ROS Hydro Medusa, o conceito de pilhas e consequentemente o de manifesto de pilhas está em desuso, pois agora se utiliza o conceito de metapacotes [Conley 2012b].

[2]Metadados podem ser basicamente definidos como ”dados que descrevem dados”, ou seja, são informações úteis para identificar, localizar, compreender e gerenciar os dados [IBGE 2008]

4.1.1.4 Mensagens (*messages*)

O ROS faz uso de uma linguagem de descrição simplificada para definir os valores de dados, sendo esses publicados por um executável no ROS, no caso um nó. A forma pela qual é feita esta descrição, facilita a geração automática do código fonte da mensagem, nas várias linguagens de programação de interesse. Essas descrições por sua vez, são atribuídas em arquivos do tipo `.msg` que se encontram no subdiretório de um pacote ROS. Para a construção das mensagens é necessário adicionar a linha de código `genmsg()` no arquivo `CMakeLists.txt` [Scioli 2013a].

Um arquivo do tipo mensagem é composto pela descrição de campos e constantes, sendo que os campos (variáveis) são os dados enviado para e pela mensagens e as constantes são valores fixos utilizados para a interpretação dos campos.

Cada campo consiste de um tipo e de um nome, sendo os mesmos separados por um espaço, seguido o formato: `fieldtype fieldname`.

Os campos podem ser dos seguintes tipos [Scioli 2013a]:

- Tipos pré-determinados. São tipos de variáveis mais comuns, como: `bol, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float32, float64, string, time` e `duration`, como por exemplo: nome `string` e posicao_x `int32`.
- nome de descrições de mensagens definidas pelo próprio usuário. Caso seja incorporado descrições de outra mensagem, o nome do tipo pode ser uma *mensagem relativa* ou uma *mensagem completa*.
- *arrays* fixos ou de tamanho variável dos anteriores, como: `string[]` ou `Posicao_x[10]`.
- tipo especial `Header`, que mapeia para `std_msgs/Header`. O tipo *Header* é fornecido junto ao ROS e tem a finalidade de promover um mecanismo geral para definir ID's para bibliotecas como `tf` - *Transforms/Frames*. A seguir está apresentada a estrutura dessa mensagem.

```
# Standard metadata for higher-level flow data types
# sequence ID: consecutively increasing ID
  uint32 seq
# Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
```

```
# time-handling sugar is provided by the client library
  time stamp
# Frame this data is associated with
  string frame_id
```

A definição constante é como uma descrição do campo, exceto pelo fato de ser necessário a atribuição de um valor. Tal atribuição de valores é realizada através do sinal de igualdade, seguindo o formato: `constanttype constantname = constantvalue`. As definições constantes podem ser de qualquer um dos tipos primitivos anteriormente mencionados, com exceção dos tipos `time` e `duration`.

4.1.1.5 Serviços (*services*)

O ROS faz uso de uma linguagem de descrição de serviço simplificada para descrever os tipo de serviços do ROS, os quais são construídos diretamente sobre o formato mensagem do ROS, a fim de permitir a solicitação/resposta de comunicação entre os nós. Um arquivo de descrição de serviço, conforme já descrito anteriormente, consiste em um pedido e uma resposta do tipo mensagem, sendo os mesmos separados por três traços (- - -). Na sequência está apresentado um trecho de descrição de serviço, onde está sendo solicitado uma `string` e tem como resposta uma `string` [Lamprianidis 2012a].

```
string str
---
string str
```

As bibliotecas cliente do ROS implementam um gerador de serviços que traduzem arquivos `.srv` em código fonte. Este gerador de serviços deve ser chamado em seu script de construção. Por convenção, todos estes tipos de arquivos são armazenados em um subdiretório do próprio pacote `/srv`, e para serem construídos deve ser adicionada a linha de código `gensrv()` no arquivo `CMakeLists.txt` [Lamprianidis 2012a].

4.1.2 Processamento Grafo do ROS

O Processamento Grafo no ROS é a rede ponto a ponto (*peer-to-peer*) de processos, ou seja, o processamento dos dados em conjunto, tendo como conceitos básicos: *nós*, *tópicos*, *mensagens*, *serviços*, *servidor de parâmetros*, *mestre* e *bolsas*.

Na Figura 4.1 está apresentado esquematicamente como é realizada a ligação entre esses conceitos, os quais serão descritos detalhadamente no itens a seguir.

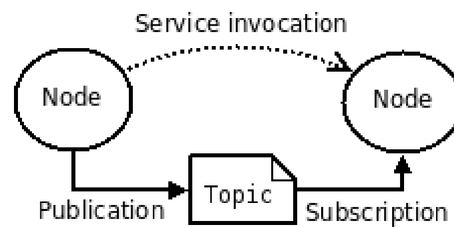


Figura 4.1: Conceitos Básicos do ROS.
Fonte: [Coleman 2013b]

4.1.2.1 Nós (*nodes*)

Os nós são processos que executam instruções e quando combinados em um grafo se comunicam uns com os outros através de *streaming* de tópicos, serviços via RPC (*Remote Procedure Call*) [3] e o servidor de parâmetro. Essa estrutura do ROS, baseada no nós, é projetada para ser modular e em uma escala de pequenas dimensões. Um sistema de controle de robô, como o apresentado nesse trabalho, compreende muitos nós, onde se tem um nó para realizar a captura da imagem, um nó para processamento da imagem, um nó para controle de trajetória, um nó para controle dos motores, um nó que proporciona uma visualização do processamento grafo, e assim por diante. Essa estrutura fundamentada na utilização de nós apresenta várias vantagens para o sistema num todo, dentre elas: maior tolerância a falhas adicionais, redução da complexidade do código, dentre outras.

O nó é construído com a utilização de bibliotecas cliente do ROS, tais como `roscpp` essa baseada em C++ ou `rospy` baseada em Python [Conley 2012c].

Todos os nós que são executados, possuem um nome de recurso grafo que o identifica para o restante do sistema. Os nós possuem ainda um tipo nó, o qual simplifica o processo de referência a um nó executável no sistema de arquivos. Estes tipos nó são nomes de recursos de pacote, os quais são compostos com o nome do pacote em que o nó se encontra mais o nome do nó executável. A fim de resolver um tipo nó, o ROS pesquisa todos os executáveis no pacote com o nome especificado e escolhe o primeiro que encontrar. Dessa forma é necessário ter um cuidado especial para não produzir executáveis, em um mesmo pacote, com o mesmo nome mas com funcionalidade diferente.

[3]Protocolo para execução remota de *procedures* em computadores ligados em rede, podendo ser implementado sobre diferentes protocolos de transportes, como é o caso do TCP e UDP [Thurlow 2009].

4.1.2.2 Tópicos (*topics*)

As mensagens são encaminhadas por meio de um sistema de transporte com uma estrutura de publicador/subscritor. Dessa forma, um nó envia uma mensagem através de sua publicação via tópico, sendo designado a esse um nome, pelo qual será identificado o conteúdo da mensagem. Caso um outro nó queira receber o conteúdo dessa mensagem, o mesmo terá que se subscrever ao tópico apropriado. Pode haver situações onde vários publicadores e subscritores concorrem a um único tópico e um único nó pode publicar e/ou subscrever vários tópicos, visto que em geral os publicadores e subscritores não possuem conhecimento da existência uns dos outros. O intuito principal é dissociar a produção de informação de seu consumo, dessa forma qualquer um pode-se conectar ao barramento para enviar ou receber as mensagens, desde que eles sejam do mesmo tipo [Conley 2012d].

Os tópicos são projetados para um fluxo de comunicação unidirecional. Sendo assim, os nós que necessitam realizar RPC (*Remote Procedure Call*), ou seja, receberem uma resposta a um pedido, devem usar um outro conceito do Processamento Grafo, os serviços. Há também a possibilidade de se usar o servidor de parâmetros para a manutenção de pequenas quantidades de estados.

A estrutura atual do ROS suporta o transporte de mensagens baseado na arquitetura TCP/IP e UDP, sendo a primeira o padrão usado pelo ROS e o único que as bibliotecas clientes são obrigadas a suportarem. O Transporte via TCP/IP é conhecido como TCPROS, onde seu fluxo de dados se dá através de conexões TCP/IP. Já o transporte UDP, conhecido como UDPROS, é atualmente suportado apenas em aplicações com a biblioteca `roscpp`, sendo que o mesmo separa as mensagens em pacote UDP. Este segundo tipo de transporte possui baixa latência, apresentando perdas no transporte, sendo por isso mais indicada para tarefas de teleoperação [Conley 2012d].

A negociação de qual o tipo de transporte será adotado é feito pelos nós e durante a execução dos mesmos. Neste caso se um nó escolher por transportar em UDPROS, mas o outro nó não suportar, ele poderá solicitar que o transporte seja em TCPROS. Tal modelo de negociação permite que novos tipos de transportes possam ser adicionados ao longo do tempo, a medida que forem surgindo aplicações interessantes.

4.1.2.3 Mensagens

A comunicação realizada entre nós é por intermédio do envio de mensagens via tópicos. Tal mensagem é simplesmente uma estrutura de dados, que compreende os campos pré-definidos.

Os tipos primitivos padrão: *integer*, *floating*, *point*, *boolean*, dentre outros, são suportados, além de *arrays* desses tipos. As mensagens podem incluir estruturas aninhadas arbitrariamente e *arrays*, sendo assim muito parecidas com as estruturas na linguagem *C* [Lamprianidis 2012b].

4.1.2.4 Serviços

O modelo de publicador/subscritor de mensagens é um paradigma de comunicação muito flexível, mas seu transporte unidirecional de muitos para muitos não é adequado para interações de solicitação/ resposta via RPC (*Remote Procedure Call*), as quais são muitas vezes necessárias em sistemas distribuídos. A solicitação/resposta é feita através de um serviço, esse definido por um par de mensagens: uma para a solicitação e outra para a resposta. Dessa forma, um nó disponibiliza um serviço com um nome específico e um outro nó cliente utiliza desse, ao enviar uma mensagem de solicitação e aguardar sua resposta. As bibliotecas cliente do ROS, comumente apresentam essa interação ao programador, como se fosse uma chamada de procedimento remoto [Conley 2012e].

4.1.2.5 Servidor de Parâmetro (*parameter server*)

Um servidor de parâmetros é um dicionário multivariado compartilhado, o qual está acessível através de API's de rede. Os nós fazem uso desse servidor para armazenarem e recuperarem parâmetros durante a sua execução. Contudo, ele não é projetado para alta performance, sendo assim melhor empregado para dados estáticos, não-binários, tais como parâmetros de configuração. Ele foi criado para ser visível globalmente, para que as ferramentas possam facilmente inspecionar o estado de configuração do sistema e modificá-lo quando necessário. O servidor de parâmetros é implementado através do protocolo de de chamada de procedimento remoto - XMLRPC, sendo executado internamente ao *Master* do ROS, o que implica em sua API estar acessível via bibliotecas XMLRPC normais [Thomas 2013].

4.1.2.6 Mestre (*master*)

O mestre ROS é responsável pelos serviços de nomeação e de registro para os demais nós no sistema ROS, rastreando os publicadores e os subscritores de tópicos, bem como de serviços. Sua principal função é permitir que os nós possam se localizar mutuamente e quando o fizer, esses irão se comunicar numa conexão *peer-to-peer*. O mestre também fornece o servidor de parâmetros [Conley 2012f].

Os nós se conectam a outros diretamente segundo as informações de pesquisas fornecidas

pelo mestre, de modo semelhante a um servidor DNS. Sendo assim, os nós que subscrevem a um tópico farão a requisição de conexões com nós que publiquem nesse tópico e irão estabelecer essa conexão conforme o protocolo dessa. Tal arquitetura permite a operação de desacoplamento, visto que os nomes são o principal meio pelo qual os sistemas maiores e mais complexos podem ser construídos. Os nomes têm um papel muito importante no ROS, de modo que nós, tópicos, serviços e servidor de parâmetros possuem nomes. Cada biblioteca cliente do ROS suporta o remapeamento de nomes por linha de comando, o que significa que um programa compilado pode ser reconfigurado durante a execução, para que passe a operar em uma topologia diferente de processamento grafo [Coleman 2013b].

O principal meio de executar o mestre é através da linha de comando `roscore`, a qual carrega o mestre ROS, juntamente com outros componentes essenciais ao seu funcionamento, como o servidor de parâmetro e um nó de *logging* rosout [Coleman 2013e].

Para exemplificar o funcionamento do mestre considere o seguinte caso, onde há um nó chamado **Camera** e um outro chamado **Image_viewer**. Uma sequência típica de eventos que iniciam com o nó **Camera**, notificam o mestre que desejam publicar imagens no tópico **images**, conforme apresentado na Figura 4.2.

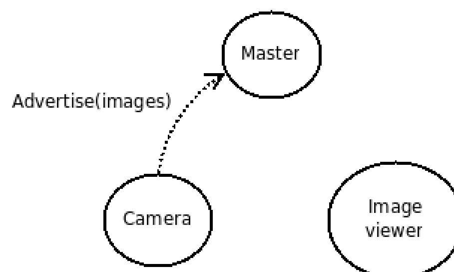


Figura 4.2: Comunicação do Nó Publicador com o Mestre.
Fonte: [Conley 2012f]

Na sequência, o nó **Camera** publica as imagens no tópico **images**, porém não há nenhum nó que esteja subscrevendo a esse tópico ainda, sendo assim nenhum dado é enviado. Então, o nó **Image_viewer** notifica o mestre de seu interesse em subscrever o tópico **images** para ter acesso às imagens publicadas, de acordo com a Figura 4.3.

Como agora o tópico **images** possui tanto um publicador quanto um subscritor, o mestre notifica os nós **Camera** e **Image_viewer** sobre a existência um do outro, para que eles iniciem a transferência de mensagens entre si, ou seja, o envio da imagem, como mostrado na Figura 4.4.

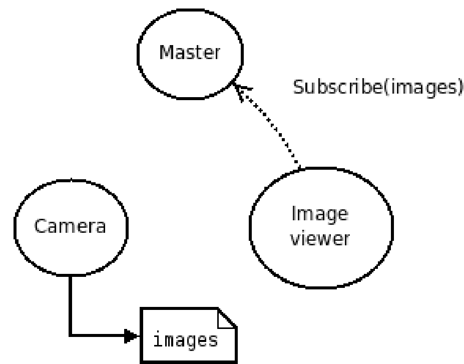


Figura 4.3: Comunicação do Nó Subscritor com o Mestre.
Fonte: [Conley 2012f]

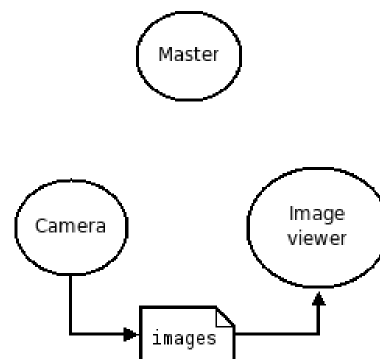


Figura 4.4: Comunicação Direta entre Nós.
Fonte: [Conley 2012f]

4.1.2.7 Bolsas (*bags*)

Bolsa é um formato de arquivo do ROS para armazenamento de dados obtidos a partir de mensagens, o qual recebe esse nome devido a sua terminação *.bag*. Eles têm um papel importante no ROS, pois através deles é possível o armazenamento de dados, por exemplo provenientes de sensores, possibilitando a verificação de algoritmos sem a necessidade de repetidas execuções na prática. Várias ferramentas do ROS foram escritas de modo a permitirem tal armazenamento, para de posse desses dados fosse possível seu processamento, análise e visualização [Scioli 2013b].

4.1.3 Comunidade ROS

Os conceitos a nível comunitário no ROS são os recursos do ROS que possibilitam às comunidades distintas a troca de *software* e de conhecimento [Coleman 2013b]. Dentre esses recursos se destacam:

- Distribuições: as distribuições do ROS são coleções de pilhas agrupadas conforme as

versões, podendo ser instaladas no computador. Estas desempenham um papel semelhante ao das distribuições do Linux, pois dessa maneira se torna mais fácil a instalação do software, além de manter as versões consistentes em um conjunto de *software*.

- Repositórios: o ROS conta com uma rede federada de repositórios de códigos, onde diferentes instituições podem desenvolver e lançar seus próprios componentes de *software* de robôs. Sendo que no Brasil há uma única instituição, o Centro Universitário da FEI, que contribui como o desenvolvimento desses códigos [Coleman 2013c].
- O ROS Wiki: o wiki comunitário do ROS é o principal fórum para documentar informações sobre o ROS. Onde qualquer pessoa pode se inscrever através de uma conta e contribuir com a sua própria documentação ou tutoriais, além de fornecer correções ou atualizações, dentre outras.
- Lista de discussão: a lista de discussão para usuários do ROS é o principal canal de comunicação sobre as novas atualizações do ROS, bem como um fórum para se fazer perguntas sobre o *software* ROS.
- ROS Answers: é um site de perguntas e respostas para questionamentos relacionados ao ROS.
- Blog: O blog *Willow Garage* fornece atualizações regulares, além de fotos e vídeos dos projetos desenvolvidos tendo o ROS como base.

4.1.4 Nomes de Recurso Grafo

Os Nomes de recurso grafo fornecem uma estrutura de nomenclatura hierárquica, a qual é utilizada para todos os recursos no processamento grafo do ROS. Tais nomes são muito importantes no ROS e é a forma central de como sistemas maiores e mais complexos são compostos no ROS, sendo por isso de fundamental relevância entender como funcionam esses nomes e como manipulá-los [Coleman 2013b].

Esses nomes funcionam como um encapsulamento, sendo um mecanismo imprescindível para o funcionamento do ROS, pois evita que partes diferentes do sistema se apropriem de algum recurso erradamente ou algum nome global. Cada recurso é definido dentro de um *namespace*, o qual pode ser compartilhado com muitos outros recursos. De um modo geral, os recursos podem criar outros recursos dentro de seu *namespace* e ainda acessar outros recursos tanto dentro quanto acima de seu próprio *namespace*. As conexões podem ser feitas entre recursos em *namespace* distintos, contudo essas são realizadas por código de integração que esteja

acima de ambos *namespaces*.

Os nomes são resolvidos relativamente, para que assim os recursos não precisem estar cientes a qual *namespace* pertencem. Essa característica simplifica a programação, pois os nós que trabalham juntos podem ser escritos como se todos eles estivessem em um *namespace* de nível superior. Quando esses nós são integrados em um sistema maior, eles podem ser colocados em um *namespace*, o qual será constituído por uma coleção de código. Um exemplo disso, conforme [Coleman 2013b], é a situação em que se pega uma demonstração do repositor *Stanford* e outra do repositor *Willow Garage* e as mesclam em uma nova demonstração com subgrafos `stanford` e `wg`. No caso de ambas demonstrações possuírem, por exemplo, um nó chamado `camera`, não se teria uma situação de conflito. Para implementação de ferramentas ou de parâmetros que precisem ser visíveis para todo o grafo, podem ser criados nós em nível superior.

4.1.4.1 Nomes Válidos

Para que um nome seja válido ele deve seguir as seguintes características:

1. O primeiro caractere deve ser uma letra do alfabeto, um til (∼) ou uma barra (/);

2. Os caracteres seguintes podem ser alfanuméricos, *under line* (_), ou barras (/).

Com exceção dos nomes de base, os quais não podem conter a barra ou o til.

4.1.4.2 Deliberação

Existem quatro tipos de nomes de recurso grafo no ROS: *base*, *relativo*, *global* e *privado*, sendo que esses possuem a seguinte sintaxe:

- base
- relative/name
- /global/name
- ∼ private/name

Por padrão, a deliberação dos nomes é feita relativa ao *namespace* do nó. Por exemplo, o nó `/wg/node1` tem o *namespace* `/wg`, dessa forma no caso de um nó chamado `node2` seu nome será `/wg/node2`.

Os nomes sem nenhum qualificador de *namespace* são todos nomes de base, os quais na verdade são uma subclasse dos nomes relativos e seguem suas regras de deliberação. Esse tipo é o mais usado para iniciar o nome de um nó.

Já os nomes que iniciam com uma barra são globais, sendo considerados completamente deliberados. Os nomes globais devem ser evitados sempre que possível, pois os mesmos limitam a portabilidade do código.

E por último, os nomes que começam com til são os privados, os quais convertem o nome de um nó interno a um *namespace*. Por exemplo, o `node1` pertencente ao *namespace* `/wg` tem o *namespace* privado `/wg/node1`. Esse tipo de nome é útil no envio de parâmetros para um nó específico através do servidor de parâmetros.

4.1.4.3 Remapeamento

Qualquer nome interno a um nó no ROS pode ser remapeado quando o nó é lançado na linha de comando. Para exemplificar tal processo, considere o nó `talker` do pacote `roscpp_tutorials`, um de seus tópicos, o `chatter` será remapeado para um nome privado, com a seguinte linha de comando:

```
$ rosrun roscpp_tutorials talker chatter:=~chatter
```

Com esse remapeamento o nome do tópico passa a ser `/talker/chatter`, e como pode ser observado o nome do nó funciona como um *namespace*.

4.1.5 Nomes de Recurso de Pacote

Os nomes de recurso de pacote são usados no ROS juntamente aos conceitos a nível de sistemas de arquivos com a finalidade de simplificar o processo de referência a arquivos e a tipos de dados no disco do computador [Coleman 2013b]. Eles são muito simples, uma vez que são apenas o nome do pacote em que o recurso se encontra mais o nome do recurso. Por exemplo, o nome `std_msgs/String` refere-se ao tipo de mensagem `String` dentro do pacote `std_msgs`.

Alguns dos arquivos relacionados que podem ser referidos usando nomes de recursos de pacote incluem: o tipo mensagem, o tipo serviço e o tipo nó.

Esses nomes se assemelham aos caminhos de arquivos, exceto pelo fato de serem muito mais curtos. Isso devido à capacidade do ROS em localizar pacotes internos aos sistemas

do computador a fazer suposições adicionais sobre o conteúdo. Um exemplo disso é que as descrições de mensagens são sempre armazenadas no subdiretório `/msg` e com a extensão `.msg`, assim `std_msgs/String` é um atalho para o caminho `path_to_std_msgs/msg/String.msg`.

4.1.5.1 Nomes Válidos

Os nomes de recursos de pacote seguem regras rígidas de nomeação, visto que eles são frequentemente usados em código gerado automaticamente. O que implica em um pacote do ROS não poder ter caracteres especiais diferentes de *under line* e iniciarem com um caractere alfabético. Um nome para ser válido deve seguir as seguintes características:

1. O primeiro caractere deve ser uma letra do alfabeto;
2. Os caracteres seguintes podem ser alfanuméricos, *under line* (_), ou barras (/);
3. Deve haver no máximo uma barra.

4.1.6 Bibliotecas Cliente

A biblioteca cliente do ROS é uma coleção de códigos, a qual facilita o desenvolvimento de aplicações com o ROS. Ela possui ainda muitos conceitos do ROS e torna-os acessíveis através de código. Normalmente, essas bibliotecas permitem escrever nós, publicar e subscrever tópicos, escrever e chamar serviços e ainda fazer uso do servidor de parâmetros. Uma das vantagens dessa biblioteca é que ela pode ser implementada em qualquer linguagem de programação, embora seja dada uma maior ênfase no suporte para linguagens como C++ e Python [Coleman 2013b] [Quigley et al. 2009].

As principais bibliotecas clientes são [Dietrich 2013]:

roscpp
É a biblioteca cliente em C++ do ROS, sendo a mais utilizada e por isso projetada para ser uma biblioteca de alto desempenho para o sistema do ROS.

rospy
É a biblioteca cliente em Python do ROS, sendo projetada para fornecer as vantagens de uma linguagem de *script* orientada a objetos para o sistema do ROS. O projeto dessa biblioteca favorece mais a agilidade no desenvolvimento do que o desempenho de tempo de execução para que os algoritmos possam ser rapidamente prototipados e testados internamente no ROS. Muitas das ferramentas do ROS são escritas em *rospy* para assim,

tirar vantagem das capacidades de introspecção de tipo. O mestre ROS é um exemplo de ferramenta desenvolvida a partir dessa biblioteca, com isso o Python é uma dependência do núcleo do ROS.

roslip

É a biblioteca cliente em Lisp e atualmente está sendo usada para o desenvolvimento de bibliotecas de planejamento. Ela suporta tanto a criação de um nó independente quanto o uso interativo de um sistema ROS em execução.

4.1.7 Variável de Ambiente

Há muitas variáveis de ambiente no sistema do ROS, as quais são utilizadas para definir o seu comportamento. Dentre essas, pode-se destacar algumas mais importantes como: `ROS_MASTER_URI`, `ROS_ROOT`, `ROS_PACKAGE_PATH`, `ROS_IP` e `ROS_HOSTNAME`. A seguir estão apresentadas essas variáveis, bem como uma breve descrição das mesmas no sistema ROS [Pooley 2013].

- `ROS_MASTER_URI`: informa aos nós em que máquina o mestre está sendo executado.
- `ROS_ROOT`: define o local onde os pacotes principais estão instalados.
- `ROS_PACKAGE_PATH`: permite o ROS localizar pacotes e pilhas no sistema de arquivos. Essa variável é opcional, mas de grande ajuda principalmente para localizar pacotes criados pelo usuário.
- `ROS_IP`: é uma variável opcional, usada para especificar um endereço IP.
- `ROS_HOSTNAME`: também é uma variável opcional, usada para definir um *host name*.

A visualização da definição de uma variável de ambiente pode ser feita via linha de comando, seguindo a sintaxe:

```
$ echo $VARIAVEL_AMBIENTE
```

Para realizar a definição de uma variável, basta digitar a linha de comando, conforme a sintaxe:

```
$ export $VARIAVEL_AMBIENTE=INFORMAÇÃO
```

4.1.8 Arquivos *launch*

Os arquivos .launch de uma aplicação, possibilita a execução de vários nós, a configuração de variáveis de ambiente, a atribuição de parâmetros, dentre outras funções [Coleman 2013d]. O formato desse tipo de arquivo é o XML e a ferramenta associada é a `roslaunch`, a qual pode interpretar as *tags* associadas ao arquivo. Esse arquivo, ainda pode ser executado para lançar nós localmente e remotamente via SSH.

A seguir são apresentadas algumas dessas *tags* e suas respectivas funções:

- <launch> - essa *tag* é o elemento raiz, atuando somente como um recipiente para as demais *tags*.
- <node> - especifica o nó que deve ser executado.
- <machine> - define a máquina que executará determinado nó. É utilizada apenas quando for necessário executar nós remotamente.
- <include> - permite a importação de outros arquivos `.launch`.
- <remap> - permite a definição de variáveis de ambiente.
- <param> - define um parâmetro para o servidor de parâmetros.
- <rosparam> - permite o uso de arquivos YAML para o carregamento ou descarregamento de parâmetros a partir do servidor de parâmetros.
- <arg> - permite a entrada de argumentos.

4.1.9 ROS Fuerte Turtle

O ROS Fuerte Turtle é o nome da quinta versão da distribuição do ROS, sendo a versão utilizada para desenvolvimento deste trabalho. Ela apresenta grandes melhorias em relação às versões anteriores, que facilitam a integração com outras plataformas de programas e ferramentas. Nessas melhorias estão incluídas a reestruturação do sistema de compilação, a migração para a plataforma *Qt* e a transição contínua para bibliotecas independentes.

O ROS visa fazer códigos para robôs mais reutilizáveis e esta versão é uma nova base para a próxima geração de grandes bibliotecas robóticas. [Foote 2013a]

O ROS Fuerte está atualmente direcionado para plataformas baseadas em *Unix* e é testado principalmente nos sistemas *Ubuntu* e *Mac OS X*, no caso do *Ubuntu* nas versões *Lucid*, *Oneiric*

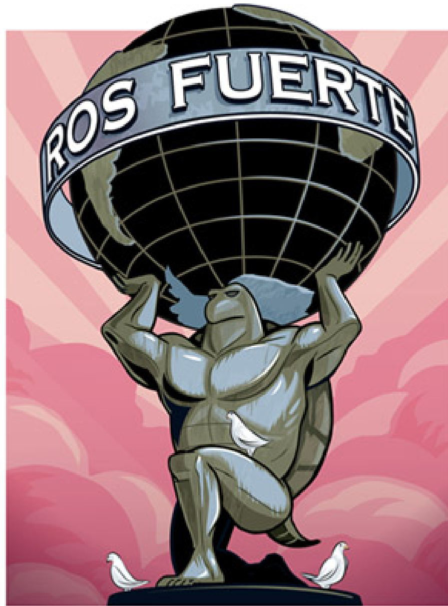


Figura 4.5: ROS Fuerte Turtle.
Fonte:[Foote 2013a]

e *Precise*. Porém, ele também pode ser instalado em outros sistemas *Linux*, como: *Red Hat*, *Debian* e *Gentoo*. Outros sistemas operacionais que podem ser utilizados são: *FreeBSD*, *Android* e *Windows*, embora esse último apresente compatibilidade mais limitada [Foote 2013a] [Coleman 2013a].

O sistema do núcleo do ROS, juntamente com ferramentas e bibliotecas úteis é regularmente lançado em uma nova distribuição do ROS, como é o caso do ROS Fuerte, pois esse procedimento ajuda a garantir a estabilidade da base de código, por bloquear as API's (*Application Programming Interface*) e verificar se a documentação, tutoriais, testes e códigos exemplo estão no lugar [Cousins 2010].

5 *Experimentos e Resultados*

Neste Capítulo serão abordados, passo a passo, todo o processo dos trabalhos desenvolvidos para a confecção do robô móvel, do sistema de visão robótica, a estruturação do sistema num todo conforme o *Framework* do ROS e do controle de trajetória.

É importante ressaltar que o presente trabalho foi desenvolvido por completo sobre a plataforma Linux, sendo utilizada a distribuição *ubuntu 12.04 LTS* - versão 64-bits. Visto que se trata de um Sistema Operacional (SO) livre e com o uso amplamente defendido entre as comunidades de pesquisa, como no caso da Robótica. Sem contar que o próprio ROS, que é a estrutura base desse trabalho, ser voltado para esse SO.

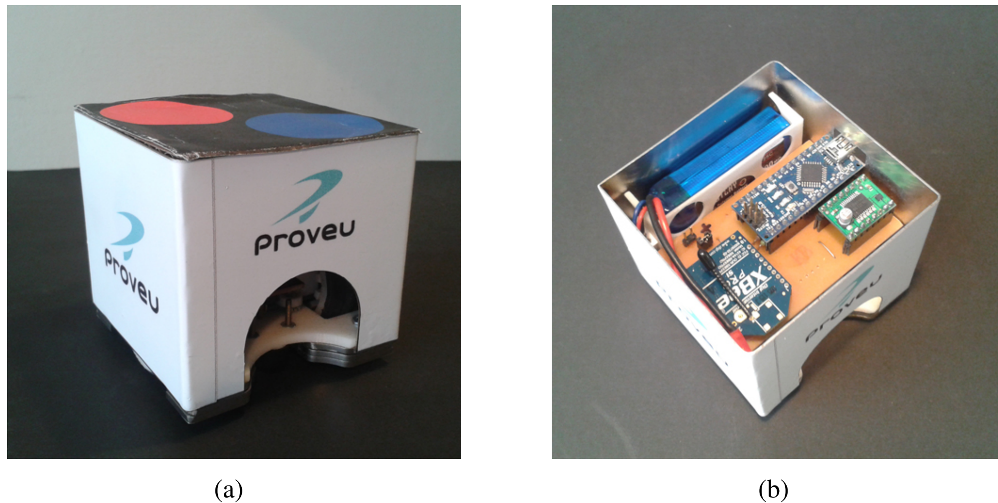
5.1 Robô Diferencial Proposto

O robô diferencial utilizado no presente trabalho foi projetado conforme as regras para a categoria de futebol de robôs *very small size* [IEEE 2008]. Dessa forma, seu tamanho é limitado a 7,5 cm $\times$ 7,5 cm $\times$ 7,5 cm, respectivamente largura, profundidade e altura, não sendo considerada a altura da antena de rádio-frequência como parte integrante do robô, caso haja.

A Figura 5.1(a) mostra o robô construído pela equipe do Grupo de Robótica Inteligente - GRIn, o qual está em concordância com as diretrizes já mencionadas.

Para a identificação deste robô no campo por parte do sistema de visão computacional, são utilizadas etiquetas de cores padronizadas, na parte superior do mesmo, para distinção entre equipes e jogadores do mesmo time. Tais etiquetas não podem conter as cores laranja, sendo essa já utlizada pela bola do jogo, e as cores branco e cinza, as quais são utilizadas na marcação do campo.

As cores utilizadas para a identificação do time obrigatoriamente devem ser azul ou amarelo, sendo que essas cores fazem referência aos times da final da Copa do Mundo de Futebol de 1970, Brasil e Itália [Gomes et al. 2006]. Já as cores utilizadas para identificar o robô do mesmo



(a) (b)

Figura 5.1: Robô Diferencial Proposto.

time, podem ser qualquer outra que não tenha sido definida anteriormente ou que pertença ao outro time.

Conforme as regras em [IEEE 2008][1], estas etiquetas de identificação devem conter, pelo menos, um quadrado com 3,5 cm de lado ou um círculo com 4 cm de diâmetro. Para o desenvolvimento deste trabalho e conforme as regras anteriormente mencionadas foram escolhidas etiquetas no formato circular com 4 cm de diâmetro e com as cores conforme apresentado na Figura 5.2. Além da função de identificação do robô no campo, as etiquetas também são responsáveis por fornecer as características necessárias para a orientação do mesmo.

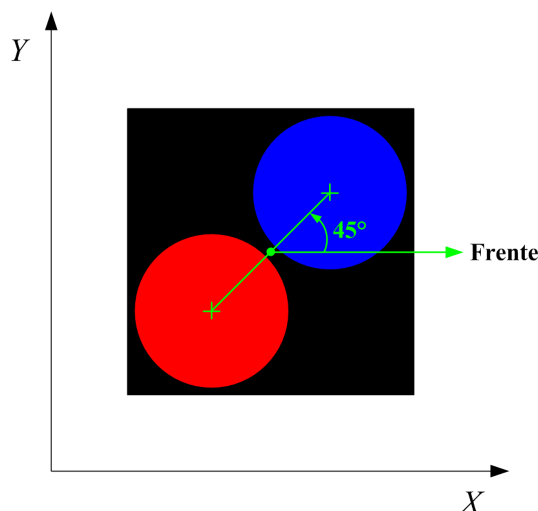


Figura 5.2: Etiqueta de identificação do robô.

Por convenção essas etiquetas devem seguir o padrão apresentado na Figura 5.2, onde o círculo que define o time deve ser disposto à frente e à esquerda do robô, já o círculo referente

[1]Estas regras, apesar de serem de 2008, de acordo com o *site*: Competição Brasileira de Robótica, elas estão vigentes para os eventos atuais.

ao jogador deve ser fixado exatamente ao contrário. Dessa forma, a posição do robô pode ser encontrada a partir da média dos centroides de cada etiqueta da seguinte maneira:

$$\begin{cases} x_{robo} = \frac{(x_{mark1}+x_{mark2})}{2} \\ y_{robo} = \frac{(y_{mark1}+y_{mark2})}{2} \end{cases} \tag{5.1}$$

onde:

x_{mark1} é a coordenada x do círculo referente ao time;
y_{mark_1} é a coordenada y do círculo referente ao time;
x_{mark_2} é a coordenada x do círculo referente ao jogador;
y_{mark_2} é a coordenada y do círculo referente ao jogador.

Já no caso da orientação, ou seja, da posição angular do robô no *Frame* de Referência Global, a mesma pode ser obtida conforme a relação:

$$\theta = \arctan 2\left(\frac{y_{mark1} - y_{mark2}}{x_{mark1} - x_{mark2}}\right) + 45^\circ \tag{5.2}$$

Na Figura 5.1(b) é possível visualizar o interior do robô e por consequência a placa de circuito impresso, na qual estão interligados os módulos de controle de baixo nível e seus periféricos, bem como o módulo de transmissão RF e da bateria utilizada para a alimentação dos mesmos. A seguir serão apresentadas, em detalhes, as partes constituintes do robô móvel desenvolvido.

5.1.1 Arduino Nano

O Arduino Nano é o módulo responsável pelo controle do robô e o gerenciamento do fluxo de dados recebidos, via módulo de transmissão e sensores. O Arduino é uma placa de desenvolvimento programável, em linguagem de alto nível, como as linguagens C e C++, com interfaces analógicas e digitais, além de comunicação serial e I2C, como mostrado na Figura 5.3. A placa que está sendo utilizada é da versão 3.0, o que lhe confere um microprocessador ATmega328, tendo o dobro de memória de armazenamento e de processamento em relação à versão 2.X.

Com a programação de controle de baixo nível embarcada é possível sintetizar os pulsos PWM (Modulação por Largura de Pulso) de controle de velocidade dos motores, conforme as informações recebidas do computador que executa o sistema de visão computacional e dos dados provenientes dos sensores de hodometria, os quais estão presentes em cada uma das rodas.

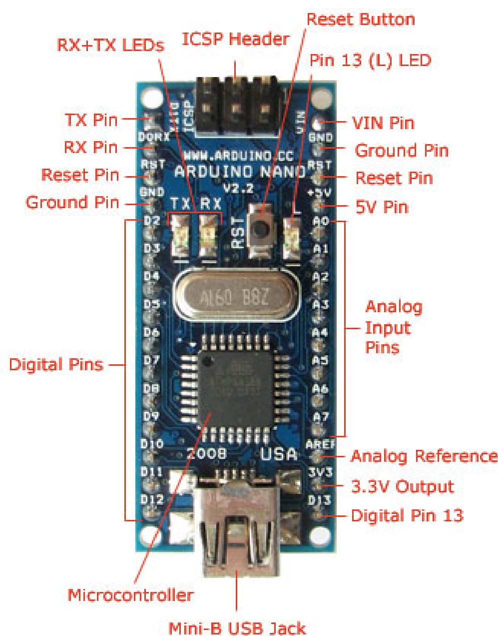


Figura 5.3: Arduino Nano.
Disponível em: <*http://arduino.cc/en/*>

5.1.2 Módulo de Comunicação *XBee*

A comunicação do robô móvel com o computador é feita através da placa de desenvolvimento, por intermédio de uma conexão serial feita por um transmissor dedicado. Esse transmissor, no caso do futebol de robôs, não precisa ser de longo alcance, sendo assim, foi definido o uso do módulo RF (Rádio Frequência) *XBee-PRO S1*, apresentado na Figura 5.4. Este módulo realiza a comunicação através do padrão *ZigBee* IEEE 802.15.4, que é um protocolo padrão para comunicação *wireless*, na frequência de 2.4 GHz.

Figura 5.4: Módulo RF XBee PRO S1.
Fonte: Manual do Fabricante.

O modelo utilizado apresenta, segundo o fabricante, um alcance em ambientes internos ou áreas urbanas de 100 m, o que não é necessário no caso em estudo, mas como já era um módulo

existente. O modelo *OEM XBee* apresenta um menor alcance, em torno de 30 m, o qual serviria para desempenhar o papel necessário.

Para que seja feita a conexão ponto-a-ponto, entre o robô e o computador, é necessário a utilização de dois módulos, sendo um na configuração de transmissor e outro na de receptor. No caso do módulo RF na configuração de receptor, é necessário a utilização de um adaptador/-conversor USB para a conexão ao computador, esta placa CON-USBBEE, está apresentada na Figura 5.5.

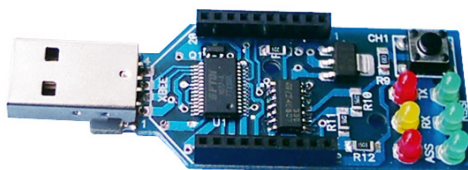


Figura 5.5: Adaptador/Conversor USB - CON-USBBEE.
Disponível em: <*http://multilogica-shop.com/adaptador-con-usbbee*>

5.1.3 Módulo Ponte H

O controle quanto ao sentido de rotação dos motores é realizado com a utilização de um circuito eletrônico, denominado de Ponte H, com a capacidade de realizar a inversão dos terminais positivo e negativo da alimentação nos terminais dos motores CC. Para executar essa tarefa foi escolhido o módulo TB6612FNG, que é um *driver* capaz de acionar no máximo dois motores CC, tendo os mesmos as seguintes características: tensão de alimentação entre 4,5 V e 13,5 V e corrente contínua de 1 A por canal. Na Figura 5.6 está apresentado este módulo.

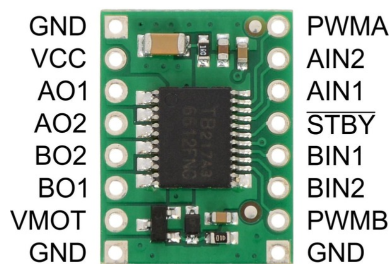


Figura 5.6: Módulo Ponte H.
Disponível em: <*http://www.pololu.com/catalog/product/713*>

Os pares de terminais AO1 e AO2, e BO1 e BO2 são ligados respectivamente nos terminais negativo e positivo dos motores A e B. Já os pares de terminais AIN1 e ANI2, e BIN1 e BIN2 são ligados à portas digitais do Arduino de forma a controlar o sentido de rotação dos motores, ao realizar a inversão da polaridade dos terminais do motor. Os terminais PWMA e PWMB

recebem o sinal PWM de controle enviado pelo Arduino para cada um dos motores e o circuito da ponte H funciona como um *driver* para o controle do nível de tensão de alimentação. O terminal VMOT é a tensão de alimentação destinadas aos motores, onde seu valor excursionará do mínimo ao máximo conforme a largura de pulso injetada nos terminais de PWM. A Tabela 5.1 traz as combinações possíveis de controle para os motores, conforme os níveis de tensão atribuídos aos terminais de entrada.

Entrada				*Saída*		
IN_1	IN_2	PWM	$Stby$	OUT_1	OUT_2	$Modo$
alto	alto	alto/baixo	alto	baixo	baixo	parada rápida
baixo	alto	alto	alto	baixo	alto	anti-horário
		baixo	alto	baixo	baixo	parada rápida
alto	baixo	alto	alto	alto	baixo	horário
		baixo	alto	baixo	baixo	parada rápida
baixo	baixo	alto	alto	alta impedância		parado
alto/baixo	alto/baixo	alto/baixo	baixo	alta impedância		*Standby*

Tabela 5.1: Descrição do controle da Ponte H.
Fonte: *DataSheet* do Fabricante

O termo PWM já foi relacionado algumas vezes, no decorrer da descrição do robô móvel, e esse é a abreviação para as palavras em inglês: Modulação por Largura de Pulso. Sendo essa, Uma técnica para a obtenção de grandezas analógicas a partir de sinais digitais. Esse controle digital é usado para criar uma onda quadrada, a qual varia entre um valor representando o estado ligado, do inglês *on*, e o estado desligado, do inglês *off*.

Conforme o comportamento deste padrão *on-off* é possível simular tensões que variem do valor máximo ao valor mínimo da tensão injetada no terminal VMOT da Ponte H. Tal variação é alcançada através da alteração do tempo em que o sinal *on* permanece ativo em relação ao tempo do sinal em *off*, durante o período de tempo - T. Esse trecho em que o sinal permanece *on* é denominado de largura de pulso, e a relação entre ele e o trecho em *off* é denominado de razão cíclica.

Dessa forma, para se obter diferentes valores analógicos, basta alterar, ou melhor modular a largura de pulso, ou seja, quanto mais o sinal *on* permanecer ativo em relação ao sinal *off* maior será o sinal analógico, em contrapartida, quanto menos o sinal *on* permanecer ativo em relação ao sinal *off* menor será o sinal analógico. E como se sabe, os motores CC apresentam sua característica de controle de rotação proporcional ao nível de tensão injetada em seus terminais.

Na Figura 5.7 é possível visualizar alguns exemplos dessa modulação, conforme a variação da largura de pulso, determinada pelos comandos na programação do Arduino. As linhas em

verde representam o período de tempo - T, o qual é obtido pela Equação 5.3.

$$T = \frac{1}{f} \tag{5.3}$$

onde:

f é a frequência do PWM.

O comando no Arduino responsável por determinar a permanência do sinal em *on* é o `analogWrite (x)`, sendo que o valor de `x` é definido de 0-255. Assim, caso seja enviando o comando `analogWrite (255)`, a razão cíclica (*Duty Cycle)* será de 100%, ou seja, o sinal *on* continuará por todo o período de tempo T. Se o comando for `analogWrite (127)` a razão cíclica será de 50%, fazendo com que o sinal *on* continue ativo metade do tempo e o sinal *off* na metade restante. O comportamento deste padrão *on-off* para outro comandos também estão apresentados.

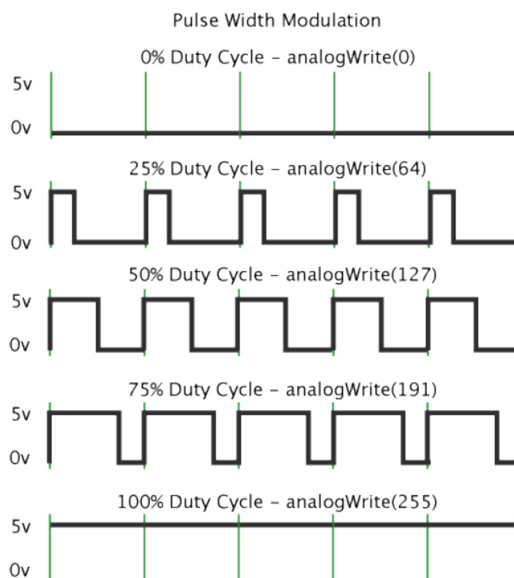


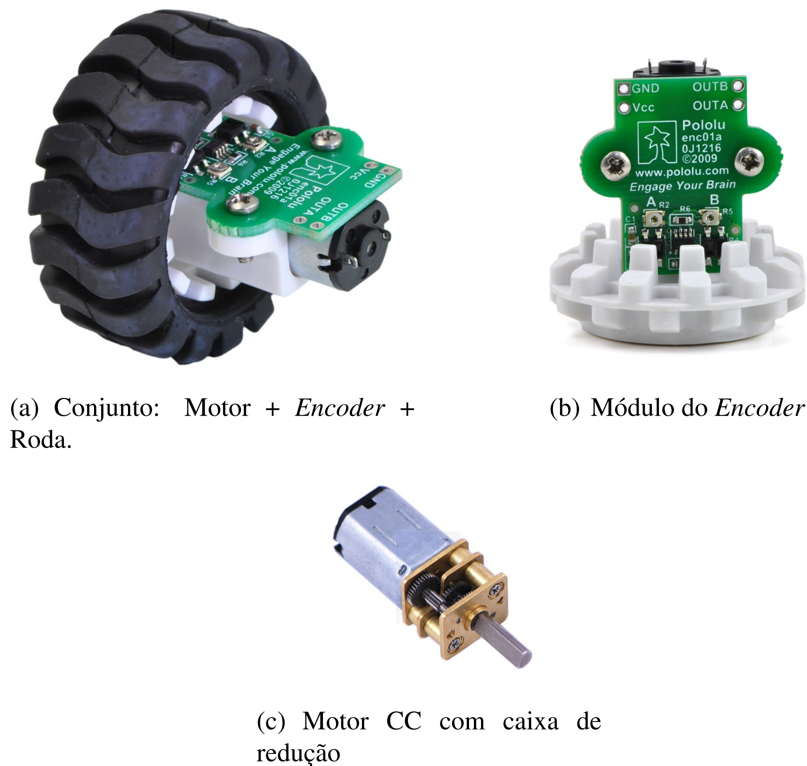
Figura 5.7: Modulação por Largura de Pulso - PWM.
Disponível em: <*http://arduino.cc/en/Tutorial/PWM*>

5.1.4 Módulo de tração

O módulo de tração do robô diferencial é composto por dois conjuntos, conforme apresentado na Figura 5.8(a). Este conjunto é formado pelo motor CC, mostrado na Figura 5.8(c), o qual dispoe de uma caixa de engrenagens acoplada ao seu eixo, para obter a redução da velocidade

de rotação e consequentemente o aumento do torque por parte do motor. Além de um sensor de hodometria, denominado de *encoder* de quadratura, o qual é imprescindível para realizar o controle de baixo nível, ao corrigir as diferenças construtivas de cada um desses conjuntos.

Na Figura 5.8(b), é possível notar a presença de um dispositivo circular com dentes alternados, sendo esse responsável por passar a informação de posicionamento da roda para os sensores de refletância infravermelhos do circuito do *encoder*. Este tipo de *encoder*, por possuir dois sensores, fornece duas formas de ondas defasadas de 90°, o que possibilita descobrir a direção de rotação das rodas.



(a) Conjunto: Motor + *Encoder* + Roda.

(b) Módulo do *Encoder*

(c) Motor CC com caixa de redução

Figura 5.8: Sistema de Tração.
Disponível em: <*http://www.pololu.com/catalog/product/1218*>

5.2 Sistema de Visão Robótica

A primeira etapa a ser realizada para obter as imagens desejadas é a calibração do sistema de visão computacional, uma vez que essa é de fundamental importância, pois por seu intermédio é possível encontrar uma correlação entre as medidas dos objetos e/ou as distâncias entre os mesmos no mundo real e suas respectivas dimensões nas imagens adquiridas. Outro fato relevante é que a busca pelo robô é baseada na identificação de círculos na etiqueta em seu topo, a qual é favorecida uma vez que a imagem não terá suas extremidades distorcidas e o

padrão circular será mantido em toda a extensão do campo.

A calibração da câmera monocular, a partir da captura das imagens puras, é realizada através do nó `cameracalibrator.py` do pacote `camera_calibration` do ROS. Sendo esse pacote fundamentado no módulo *calib3d* do *OpenCV*.

Esse processo, em sistemas de imagens em duas dimensões, é realizado com o auxílio de imagens de gabaritos, no caso um tabuleiro de xadrez, com dimensões conhecidas de 8x6 quadrados, tendo esses 10,8 cm de largura.

Antes de iniciar a calibração propriamente dita é necessário que, além da câmera estar conectada à interface IEEE 1394 do computador, que um nó do ROS esteja executando o *driver* da câmera de modo a publicar as imagens provenientes da mesma. E para que um nó possa realizar tal tarefa é preciso obter todas as dependências para a câmera 1394, sendo essas feitas através do comando:

```
$ rosdep install camera1394
```

Outra etapa fundamental é a atribuição de alguns parâmetros de identificação da câmera utilizada, sendo tais parâmetros obtidos por intermédio de uma ferramenta chamada *Coriander*. Essa ferramenta ainda possibilita a verificação do funcionamento do dispositivo.

Na Figura 5.9, está apresentada uma das janelas de execução do Coriander com algumas das informações da câmera, sendo que a mais importante são os dezesseis dígitos hexadecimais da identificação GUID da câmera, no caso o valor 003053000142e251.

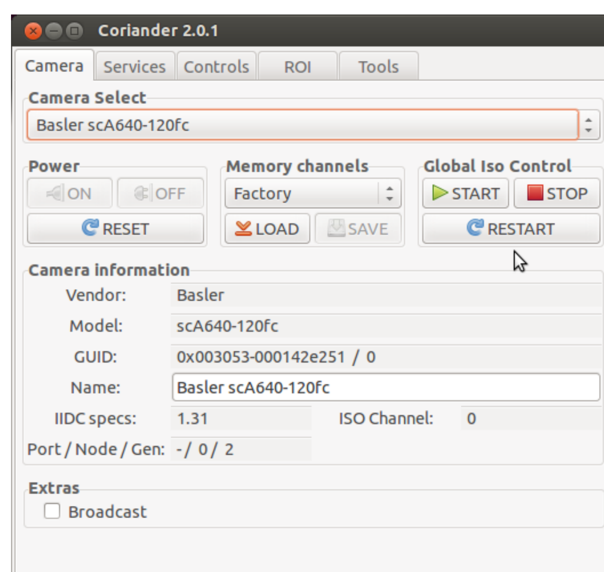


Figura 5.9: Janela de execução do Coriander.

Para que o nó possa ser executado e assim inicie a publicação das imagens é necessário passar outros parâmetros essenciais, como: *video_mode*, *iso_speed* e *frame_id*, sendo esses respectivamente o formato do vídeo, a taxa de transferência de dados e a taxa de amostragem dos frames. Todos esses parâmetros descobertos pelo `Coriander` e outros são passados via arquivo `Camera.launch` do tipo XML, o qual está parcialmente apresentado a seguir.

```
<launch>
  <!-- group ns="Camera0"-->
    <node pkg="camera1394" type="camera1394_node" name="camera1394_node" >
      <param name="guid" value="003053000142e251" />
      <param name="video_mode" value="640x480_yuv422" />
      <param name="iso_speed" value="400" />
      <param name="frame_rate" value="30" />
      <param name="frame_id" value="basler" />
    </node>
  <!-- /group -->
  .
  .
  .
</launch>
```

A publicação e subscrição de qualquer nó em um determinado tópico está vinculada a prévia execução de uma estrutura básica do ROS. Essa estrutura, que é composta pelo mestre ROS, o servidor de parâmetro e um nó de *logging* rosout, é carregada através da linha de comando `roscore`, conforme já explicado na Subseção 4.1.2.6. No entanto, ao realizar o lançamento do arquivo `Camera.launch`, todos esses componentes são inicializados automaticamente, dispensando o comando anterior.

Para a execução do nó de calibração do ROS também se faz necessário que sejam instaladas suas dependências e na sequência que seja feita a compilação desse nó. Esses dois processos são realizados com as seguintes linhas de comando:

```
$ rosdep install camera_calibration
$ rosmake camera_calibration
```

A fim de confirmar se realmente a câmera está publicando a imagem capturada, basta executar a linha de comando `rostopic list`, a qual retornará a listagem de todos os tópicos que estão ativos. Feito isso, necessariamente os tópicos `/camera/camera_info` e `/camera/image_raw` deverão ser listados.

O nó de calibração será inicializado a partir da linha de comando abaixo, onde são passadas as informações da imagem de gabarito, como: o número de interseção entre os quadrados e o tamanho da lateral dos quadrados, e ainda a imagem de origem e a imagem de destino.

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square
  0.108 image:=/camera/image_raw camera:=/camera
```

Na Figura 5.10 é possível visualizar a identificação das interseções na imagem gabarito. Como pode ser visualizado, acima do campo e abaixo do tabuleiro foi adicionado um papel pardo para realçar a imagem do tabuleiro, visto que num primeiro momento, sem o papel pardo o processo de calibração foi muito demorado e não foi possível obter uma calibração satisfatória.

Figura 5.10: Janela inicial do nó de calibração.

Já tendo sido feito todo esse processo preparatório, deu-se início ao próximo passo, o qual consistiu na aquisição de uma sequência de imagens em diferentes posições, varrendo todas as extremidades do *frame* da câmera, em diferentes inclinações do gabarito de calibração, para assim ser formado um vetor de imagens. Alguns dos *frames* utilizados nesta etapa estão apresentados na Figura 5.11.

Retornando a Figura 5.10 é possível notar a presença de quatro nomes, na lateral direita da janela, esses: *X*, *Y*, *Size* e *Skew*, além de três botões em marca d'água, nomeados: *CALIBRATE*, *SAVE* e *UPLOAD*. Sendo que ao efetuar todos os movimentos anteriores essa janela passará para o estado mostrado na Figura 5.12.

Nessa Figura é possível notar que agora estão presentes barras, estas na cor verde, as quais indicam que as características necessárias para o processo de calibração já possuem dados coletados suficiente. Ao passo que o botão *CALIBRATE* agora encontra-se em condições de ser acionado. Ao clicar nesse botão, após alguns instantes, a imagem exibida não apresentará mais as distorções provenientes das lentes, conforme pode ser observado na Figura 5.13

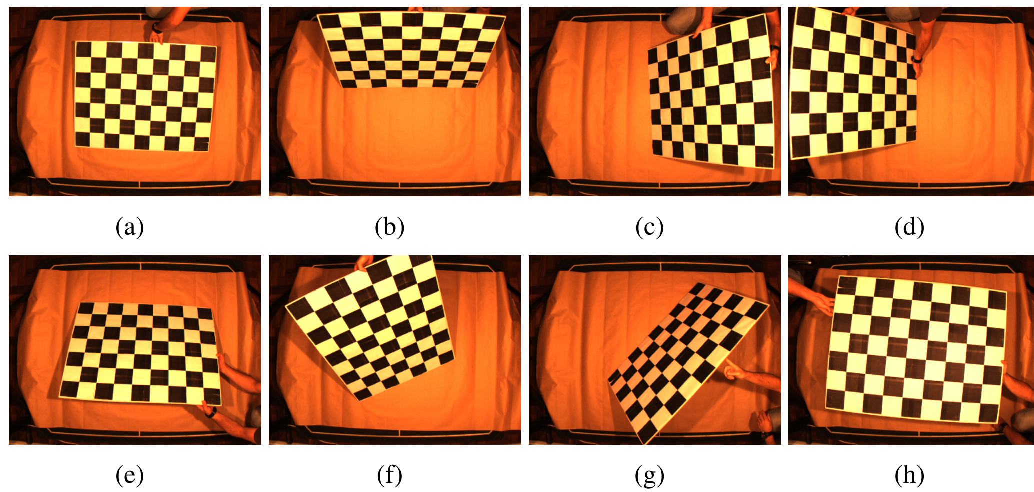


Figura 5.11: Alguns *frames* utilizados na Calibração da Câmera.

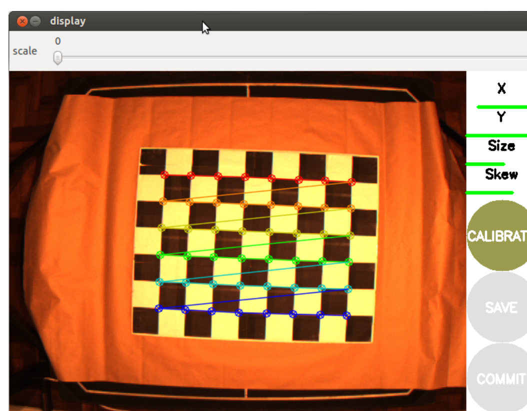


Figura 5.12: Janela do nó de calibração com índices satisfatórios.

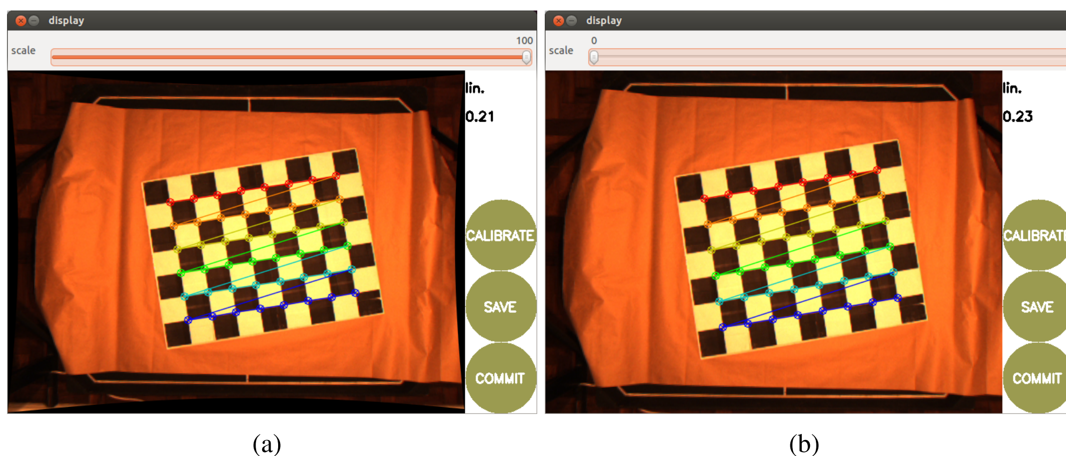


Figura 5.13: Janela do nó de calibração sem distorções.

Pode-se observar ainda na Figura 5.13(a), que suas laterais, apresentam uma curvatura para dentro, sendo que essa faixa em preto representa a falta de *pixels*, demonstrando o quanto de distorção havia na imagem original. Já na Figura 5.13(b) essa porção lateral com a deformação foi retirada, juntamente com alguns *pixels* da imagem original. Porém essa subtração de alguns

pixels não influencia em nada o processo da Visão Robótica.

Terminado o processo de calibração os outros dois botões ficaram ativos, sendo que através do botão *SAVE* é possível salvar o arquivo `.yaml` com as configurações obtidas do processo, e com o botão *COMMIT* é possível salvar essas configurações diretamente na memória interna da câmera, o que não é possível com a câmera escolhida para o trabalho.

Tendo salvo o arquivo `.yaml`, o mesmo deve ser colocado na pasta oculta `/.ros/camera_info` para que todas as vezes que seja necessária a obtenção de uma imagem retificada, todo esse processo não precise ser repetido para obtenção dos parâmetros de configuração. Esse arquivo recebe como nome o número *GUID* da câmera e possui a seguinte estrutura:


```
image_width: 640
image_height: 480
camera_name: 003053000142e251
camera_matrix:
  rows: 3
  cols: 3
  data: [726.643909622187, 0, 369.403453547574, 0, 725.522388447364,
  223.19940911117, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
  cols: 5
  data: [-0.369374647836413, 0.124054800176944, 0.000527054472258484,
  -0.0034730534326618, 0]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
  data: [665.460021972656, 0, 375.616943185196, 0, 0, 692.150817871094,
  221.33470415579, 0, 0, 0, 1, 0]
```


Para ter acesso a imagem retificada é necessário além de um nó publicando a imagem pura, a execução de um segundo, esse com a capacidade de, a partir do arquivo de configuração, gerar

a imagem retificada. Esse outro nó do ROS é chamado `image_proc` e o mesmo é executado em sequência com o anterior, através do arquivo `Camera.launch`, já parcialmente apresentado anteriormente, tendo sua continuação apresentada abaixo.

```
<launch>
  .
  .
  .
  <!-- group ns="camera"-->
    <node pkg="image_proc" type="image_proc" name="image_proc" >
    </node>
  <!-- /group -->
</launch>
```

Com a execução desse nó, obrigatoriamente o tópico `/camera/image_rect_color` deve estar ativo para que seja possível a publicação da imagem colorida retificada. Sendo necessário para a conferência do referido nó, a execução da linha de comando `rostopic list`.

Na Figura 5.14, é possível visualizar a representação de aproximadamente 2/5 de todo o Processamento Grafo do ROS utilizado neste trabalho. Onde os nós, em forma de elipse, estão conectados entre eles através dos tópicos, esses identificados pelas setas.

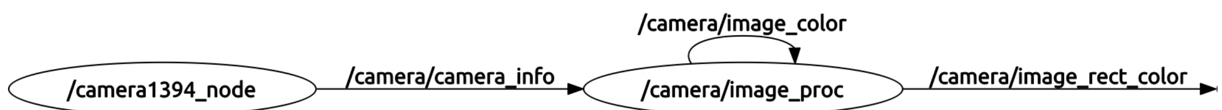


Figura 5.14: Representação do Processamento Grafo do ROS para a Visão Computacional.

5.2.1 Nó de Visão Robótica

Para que um nó possa publicar ou subscrever uma informação em um dado tópico no ROS é necessário que o mesmo apresente uma estrutura padrão, tornando-se uma tarefa simples de ser executada. Dessa forma, o código do nó do ROS para a visão robótica, o `Find_Robot`, deve possuir as seguintes instruções:

- `ros::init(argc, argv, "Find_Robot"`: essa instrução inicializa o ROS. Onde o primeiro campo recebe os argumentos de entrada através do servidor de parâmetros do ROS, já o segundo armazena os argumentos passados e o terceiro define o nome padrão do nó, caso não seja passado um outro pelo servidor de parâmetros. Essa instrução permite ao

ROS fazer o remapeamento de nomes via linha de comando, os quais devem ser únicos no sistema em execução.

- `ros::NodeHandle nh`: cria um manipulador para o nó em processo, onde o primeiro `NodeHandle` criado irá realmente fazer a inicialização do nó, e o último será destruído limpando todos os recursos do nó que estava em uso.

- `ros::Publisher chatter_pub = nh.advertise<visualization_msgs::InteractiveMarkerPose>("msg_Camera", 1000)`: é uma classe que possibilita a criação de um publicador de um tópico, passando ao mestre que será publicada uma mensagem do tipo `visualization_msgs/InteractiveMarkerPose` no tópico `msg_Camera`. Isso permite que o mestre diga a todos os nós em escuta ao `msg_Camera`, que estão sendo publicados dados neste tópico. O segundo argumento é o tamanho da fila de publicação, ou seja, serão armazenadas no máximo 1000 mensagens.

- `image_transport::Subscriber image_sub_ = it.subscribe("camera/image_rect_color", 1000, imageCallback)`: é uma classe que possibilita a criação de um subscritor a um tópico, subscrevendo para o tópico `camera/image_rect_color` com o mestre. Dessa forma, o ROS irá chamar a função `imageCallback` sempre que uma nova mensagem chegar. O segundo argumento também se refere ao número máximo de mensagens armazenadas, sendo que o se esse valor for alcançado as mais antigas darão lugar às mais novas. Como pode ser observado, a grande diferença entre o publicador e o subscritor é que, o subscritor necessita de uma chamada a uma função, essa responsável por subscrever um tópico com o tipo de dado correto.

- `ros::Rate loop_rate(30)`: especifica a taxa de publicação no tópico.

- `ros::spin()`: é um método que permite ao programa entrar em um *loop* infinito, sendo interrompido somente com a finalização do nó (via comando *crtl + c*). Porém, é sensível a interrupção quando houver mensagens publicadas, caso contrário retorna ao *loop*.

- `ros::spinOnce()`: necessário quando da adição de uma subscrição na aplicação, caso não seja utilizado não serão realizados retornos às chamadas do tópico.

É necessário também que todos os nós criados, para serem executados no ROS, tenham duas principais linhas de cabeçalho, essas responsáveis por adicionar bibliotecas fundamentais para a compilação do nó. A primeira é `#include "ros/ros.h"`, onde são incluídos todos os cabeçalhos necessários para usar as partes públicas mais comuns do sistema ROS. A segunda é

#include "std_msgs/String.h", que inclui as mensagens std_msgs/string, essas pertencentes ao pacote std_msgs.

Tendo feito essa etapa, o nó *Find_Robot* está pronto para receber as imagens provenientes do tópico */camera/Image_rect_color*.

Um outro ponto importante é o fato da integração entre o ROS e a biblioteca *OpenCV*, pois como já apresentado anteriormente, foram utilizados dois nós, onde um foi responsável por capturar as imagens puras e outro por realizar a calibração dessas imagens. Sendo que na sequência serão executados os demais procedimentos de processamento de imagem e visão computacional, esses a partir da utilização do *OpenCV*.

A questão se concentra no fato do ROS transmitir suas imagens no formato de mensagem - sensor_msgs/Image, o qual é diferente do padrão do *OpenCV*. Dessa forma, para que seja possível a manipulação das imagens do ROS pelo *OpenCV*, se faz necessário a utilização de uma biblioteca, a CvBridge, essa responsável por proporcionar uma interface entre os dois, conforme apresentado na Figura 5.15. Essa biblioteca está disponível no pacote cv_brigde na pilha vision_opencv.

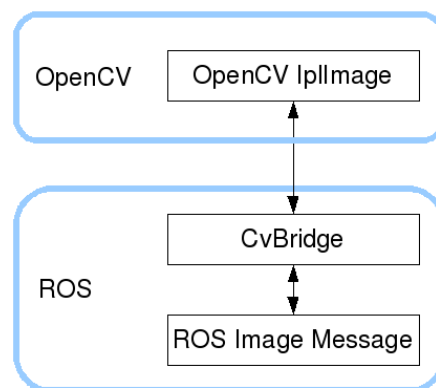


Figura 5.15: Representação da interface ROS *OpenCV* para imagens.

O CvBridge define um tipo CvImage, o qual contém uma imagem *OpenCV*, sua codificação e um cabeçalho do ROS. As informações da mensagem sensor_msgs/Image são armazenadas na estrutura CvImage, de modo que a partir desses três parâmetros seja possível a conversão de um tipo para o outro.

Para que essa conversão possa ser efetuada é necessário que sejam acrescentadas as seguintes dependências no arquivo manifest.xml:

- sensor_msgs;
- opencv2;

- cv_bridge;
- roscpp;
- std_msgs;
- image_transport.

Também se faz necessário acrescentar algumas linhas de cabeçalho no código do nó. Sendo essas:

- #include <image_transport/image_transport.h>: usado para publicar e subscrever imagens no ROS, permitindo subscrever para um fluxo de imagem comprimida.
- #include <cv_bridge/cv_bridge.h> e #include <sensor_msgs/image_encodings.h>: inclui o cabeçalho para CvBridge bem como algumas constantes úteis e funções relacionadas com a codificação de imagem.
- #include <opencv2/imgproc/imgproc.hpp> e #include <opencv2/highgui/highgui.hpp>: inclui os cabeçalhos para os módulos de processamento de imagem e GUI do *OpenCV*, respectivamente.

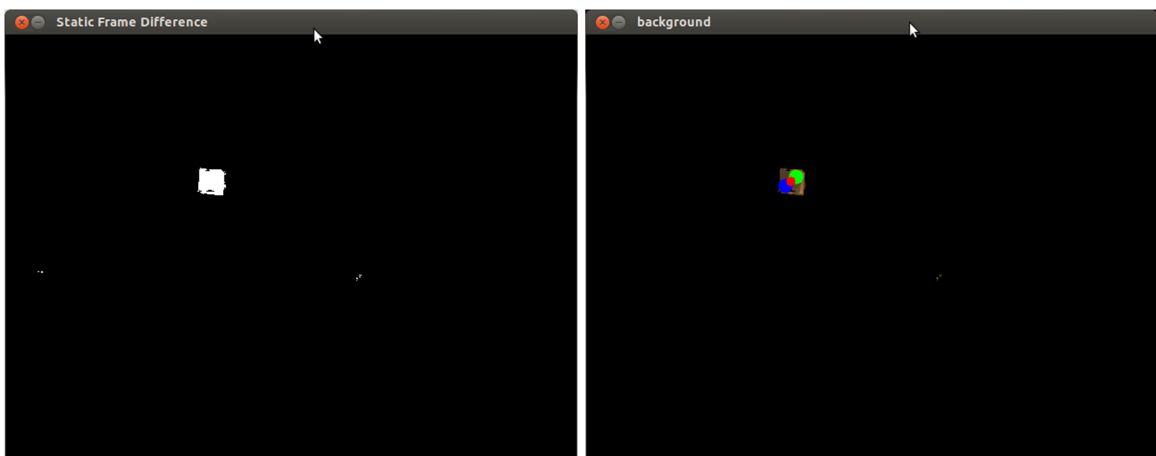
Feito todo esse processo de adequação na estrutura para a compilação do nó, essa conversão entre formatos pode ser obtida a partir do seguinte trecho do código do nó:

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
```

De posse da imagem cv_ptr, já no formato do *OpenCV* (cv::Mat) e capturando apenas o campo, é executado o código para a subtração de fundo. Onde esse retorna uma imagem em preto e branco, sendo que enquanto não são adicionados objetos no campo, é possível visualizar

somente um frame em preto, e apenas quando da inserção de objetos é que verificamos a silhueta dos mesmos em branco. Na Figura 5.16(b) é possível observar que o robô foi identificado como um novo objeto no campo.

(a) Imagem original.



(b) Imagem obtida com a subtração de fundo.

(c) Imagem resultante da subtração de fundo com o robô identificado.

Figura 5.16: Processo de Visão Robótica.

Essa nova imagem obtida, será utilizada como uma máscara para a imagem original, de modo que a partir de uma operação possa resultar em uma outra imagem, essa agora colorida (RGB) e apenas com a região do robô visível. Com isso houve uma redução significativa da região da imagem, pela qual será feita a busca pelos círculos.

Como apresentado na Subseção 3.1.1, a busca pelas cores das etiquetas do robô terá maior êxito com a conversão da imagem no espaço de cor RGB, conforme o padrão de captura da câmera, para o espaço de atributos de cor HSV. Juntamente a essa conversão é executada outra, mas dessa vez do espaço de cor RGB para escala de cinza, a qual será utilizada para a detecção de círculos. Sendo tais conversões obtidas pelas seguintes linhas de comando:

```
cv::cvtColor(result,resultHSV,CV_BGR2HSV);
cv::cvtColor(result,resultG,CV_BGR2GRAY);
```

A imagem em escala de cinza, ainda deve passar pelo filtro gaussiano para retirar os ruídos, de modo a deixar a imagem mais homogênea e por consequência borrada. Entretanto, de forma ideal para a localização dos círculos pela transformada de *Hough* para círculos, segundo a Subseção 3.2.2. A seguinte linha de comando deve ser implementada:

```
cv::HoughCircles(resultFilt,circles,CV_HOUGH_GRADIENT,2,11,100,20,6,10);
```

onde se optou por atribuir:

- A imagem de entrada como sendo a processada pelo filtro gaussiano - `resultFilt`;
- O vetor de saída com os círculos detectados e suas repectivas coordenadas - `circles`;
- O método de detecção utilizado - `CV_HOUGH_GRADIENT`;
- A resolução do acumulador como 2, ou seja, a imagem analizada terá metade da largura e da altura da original;
- A distância mínima entre círculos foi estabelecida em 11, evitando que sejam encontrados círculos em interseção com outros;
- O valor de *threshold* superior de 100 para o detector de bordas *Canny*;
- O número mínimo de votos de 20, para que os círculos detectados pudessem ser considerados verdadeiros;
- Os valores de menor e maior dimensão para o raio, sendo esses definidos em 6 e 10, respectivamente.

Detectados os círculos na imagem em escala de cinza é necessário agora identificar suas cores. Mas antes mesmo dessa etapa de processamento de imagem se iniciar, foi preciso a execução de uma fase de calibração das cores. Na qual, foram coletas informações das cores das etiquetas do robô ao longo do campo e em várias posições. Isso é importante visto que a iluminação sobre o campo não é uniforme, ocasionando diferentes valores, ou seja, uma faixa de valores para representar uma única cor. Apesar da existência de uma faixa de valores, as

cores são completamente classificável para quando no espaço de atributo de cor HSV, o mesmo não ocorrendo para o espaço de cor RGB.

Com as faixas de valores das cores das etiquetas do robô e as coordenadas dos círculos encontrados é necessária a conferência de qual a cor de cada círculo para definir, conforme as Equações 5.1 e 5.2, as coordenadas do robô e sua orientação. Na Figura 5.16(c), está representada a imagem com as etiquetas identificada, sendo que os círculos foram preenchidos nas cores azul e verde, e em vermelho está identificado o centro do robô, o ponto P.

Identificado os parâmetros do robô, esses são publicados na mensagem do tipo `visualization_msgs/InteractiveMarkerPose`, conforme já mostrado no início dessa Subseção.

Na Figura 5.17 está demonstado todo o processo da Visão Robótica, através de um fluxograma, de modo a fornecer uma perspectiva sequencial de todas as etapas executadas.

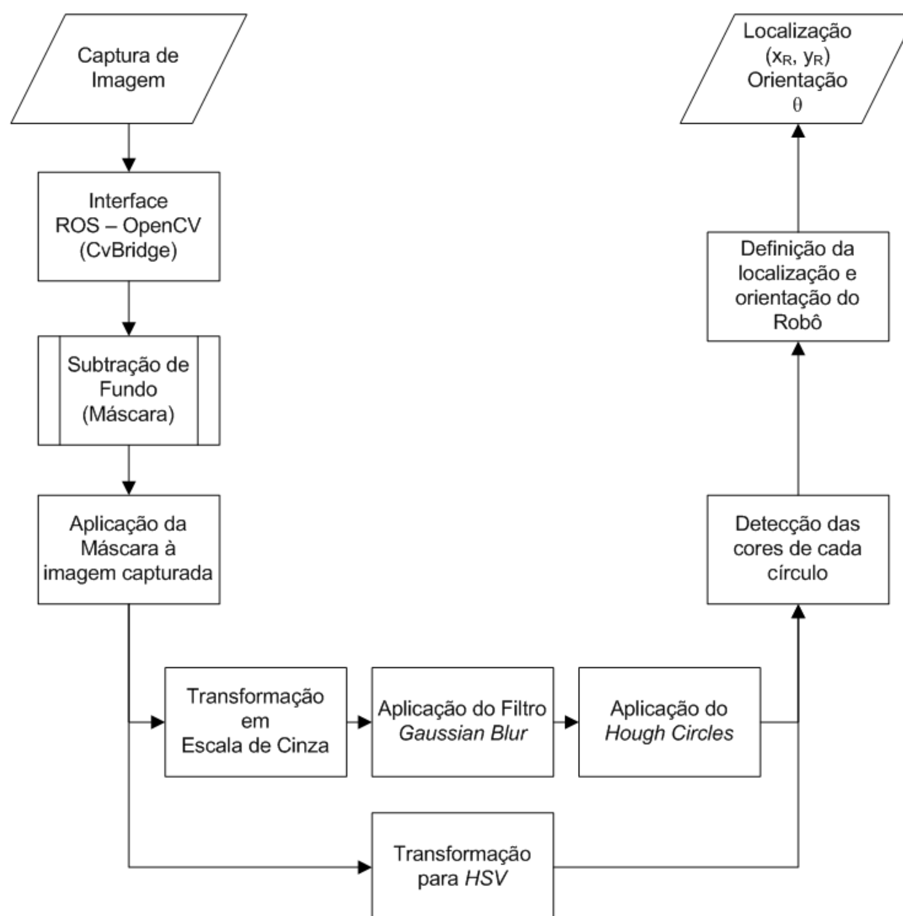


Figura 5.17: Fluxograma da Programação de Visão Robótica.

5.3 Controle de Alto Nível do Robô

Como já apresentado na Subseção 2.5, o controle de alto nível deverá ser capaz de calcular a trajetória que o robô deverá trilhar para que a partir de sua posição de origem possa alcançar a posição de destino. Esse controle trabalha em conjunto com o de baixo nível, uma vez que são necessárias comparações entre os parâmetros das ordens delegadas e a real execução das mesmas pelo robô.

Diante de todo esse arcabouço matemático, foi possível obter o seguinte modelo do controlador de alto nível, conforme apresentado pela Figura 5.18.

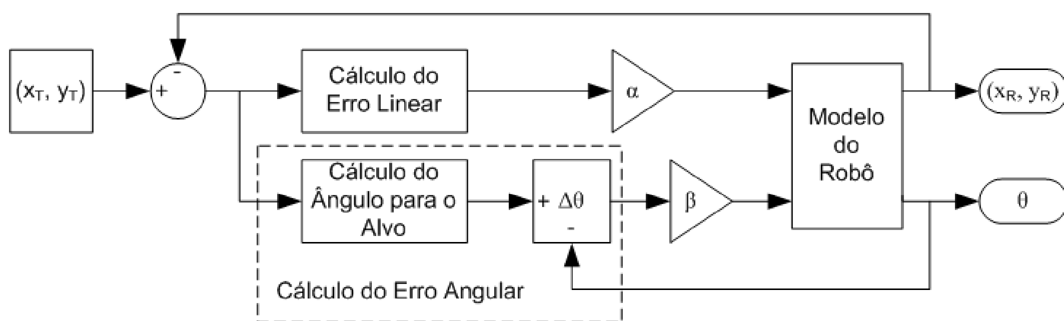


Figura 5.18: Modelo do Controlador de Alto Nível.

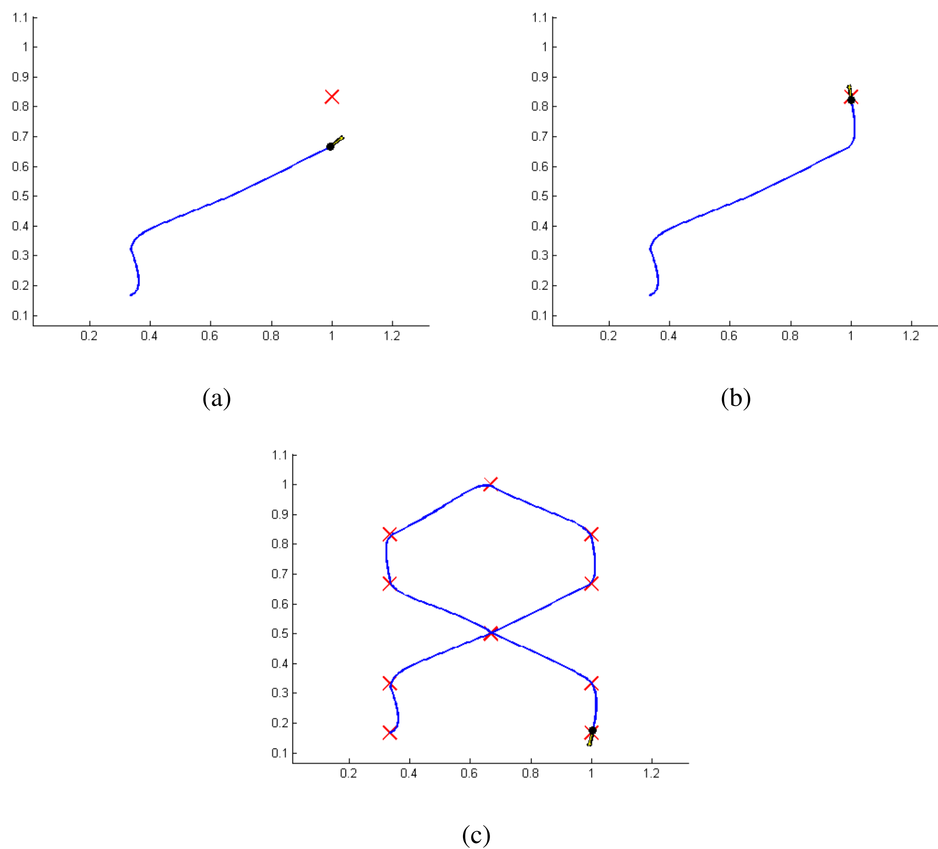
E para verificar a eficácia do modelo de controle proposto, foram realizadas simulações através do *Matlab*, sendo seus resultados mostrados na Figura 5.19.

Como pode ser observado, na Figura 5.19(a), o robô que aqui está representado como uma seta, está em uma posição de origem definida (*way-point*). A partir dela é atribuída uma próxima posição aleatória, sendo essa alcançada como demonstrado na Figura 5.19(b). Na Figura 5.19(c), todo o trajeto foi percorrido pelo robô, quase que em sua totalidade, em linha reta entre os pontos.

5.3.1 Nó de Controle de Alto Nível

Esse nó possui uma estrutura bem parecida com a do nó de visão robótica, pois ele também publica e subscreve mensagens. Como se trata do controle de alto nível e o mesmo é baseado em visão, logo é preciso que ele subscreva ao tópico proveniente do nó de posse das coordenadas do robô. Alem de subscrever e publicar em tópicos com ligação ao controle de baixo nível. Essas publicações e subscrições são feitas, no nó `Robot`, através das seguintes linhas do código:

- `ros::Subscriber sub = n.subscribe("/msg_Camera", 1000, chatter Callback)`: possibilita a criação de um subscritor a um tópico, que no caso é o `/msg_`



(a) (b) (c)

Figura 5.19: Simulação da Técnica de Controle de alto nível.

`Camera`. O ROS ainda irá efetuar uma chamada a função `chatterCallback` todas as vezes que uma mensagem chegar.

- `ros::Subscriber sub_Encoder = n.subscribe("/msg_EncoderRobo1", 1000, chatterCallback_Encoder)`: idem ao anterior, mas o tópico de interesse é o `/msg_EncoderRobo1` e a função é a `chatterCallback_Encoder`.

- `ros::Publisher chatter_pub = n.advertise<geometry_msgs::Twist>("/msg Robo1", 1000);`: possibilita a criação de um publicador de um tópico, no caso o `/msgRobo1`, cuja mensagem é do tipo `geometry_msgs/Twist`

As duas funções chamadas com a subscrição possuem as seguintes estruturas, respectivamente:

```
void chatterCallback(const visualization_msgs::InteractiveMarkerPose msg_Camera)
{
    leitura[0] = msg_Camera.pose.position.x;
    leitura[1] = msg_Camera.pose.position.y;
    leitura[2] = msg_Camera.pose.orientation.z;
    leitura[3] = msg_Camera.pose.orientation.w;
```

```
    leitura[4] = (float)msg_Camera.header.stamp.nsec;
    leitura[5] = (int)msg_Camera.pose.position.z;
}

void chatterCallback_Encoder(const std_msgs::Float32MultiArray::ConstPtr& msg2)
{
    encoder[0] = msg2->data[0];
    encoder[1] = msg2->data[1];
    encoder[2] = msg2->data[2];
    encoder[3] = msg2->data[3];
}
```

De posse das informações de localização do robô, do estado do *encoder* e das posições de destino, esse nó de controle de alto nível é capaz de calcular os erros linear e angular do robô, para que assim, possa atuar no controle do robô.

5.4 Controle de Baixo Nível do Robô

O objetivo do controle de baixo nível é controlar as velocidades angulares dos dois motores do robô diferencial, de modo que o mesmo consiga percorrer as trajetórias com o mínimo de erro. Para tal, é necessário identificar o modelo do sistema. O sistema é composto basicamente de quatro estruturas: controlador, ponte H (ver Subseção 5.1.3), motores e *encoder* (ver Subseção 5.1.4).

O controle de baixo nível é realizado a fim de garantir que cada roda atinja a velocidade necessária para que o robô se movimente com as velocidades linear e angular determinadas pela camada do controle de alto nível, sendo que a velocidade de cada roda pode ser obtida utilizando as Equações (2.5) e (2.6).

O controlador gera um sinal de sentido de rotação e um sinal PWM (*Pulse-Width Modulation* - Modulação por Largura de Pulso), sendo esses entradas para a ponte H, a qual é capaz de amplificar o sinal PWM de entrada e assim alimentar o motor com tensão eficaz variável proporcional à largura do pulso. A velocidade do motor pode ser medida com o uso do *encoder*, que fornece um valor de pulsos em um determinado período, referente à velocidade do motor, sinal esse utilizado como realimentação para possibilitar um controle em malha fechada.

Conforme apresentado em [Medeiros 2003], os motores elétricos CC a imã permanente, igualmente aos controlados pela corrente de armadura, podem ser aproximadamente descritos

em termos de uma função de transferência de primeira ordem, sendo essa:

$$F(s) = \frac{K}{Ts+1} \tag{5.4}$$

onde:

K é uma constante;

T é o tempo de resposta do sistema.

Foram realizados ensaios com o sistema operando em malha aberta para estimar as constantes mostradas na Equação (5.4). Foi possível constatar que o tempo de resposta para os motores escolhidos é desprezível ao se considerar os períodos de amostragem utilizados tanto para o controle de baixo nível quanto para o controle de alto nível, uma vez que o sinal de velocidade obtido através dos *encoders* como resposta à um degrau unitário, atinge o estado estacionário em menos de um período de amostragem (no caso do baixo nível, escolhido como 10ms).

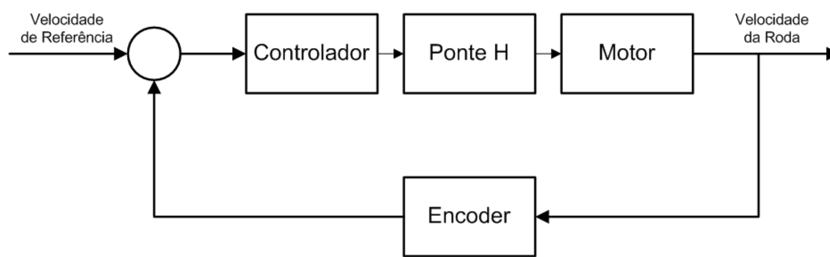
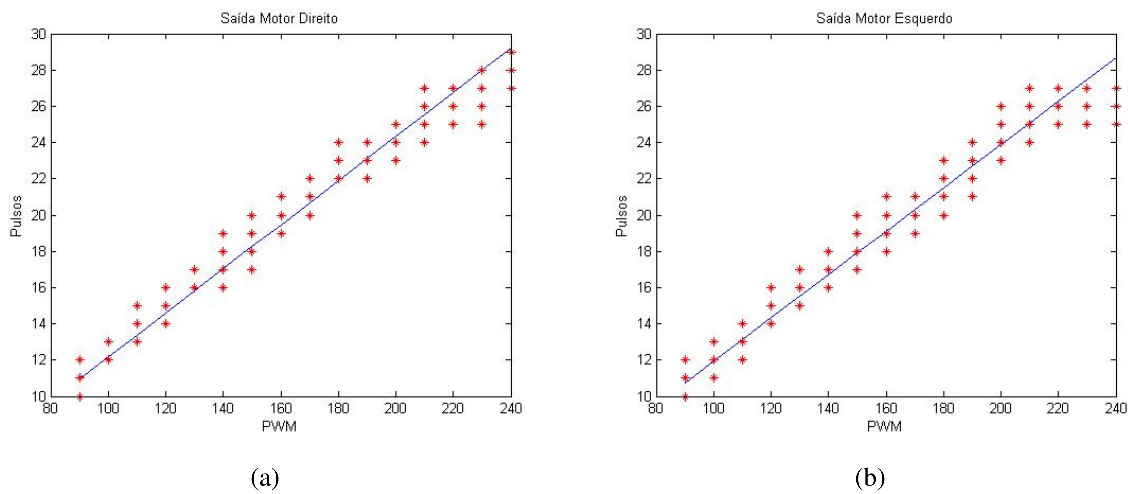


Figura 5.20: Modelo do Controlador PID em malha fechada.

Um outro ensaio realizado excita o sistema com uma sequência de degraus variando o sinal PWM enviado ao motor. No caso da plataforma utilizada, a representação da largura de pulso do sinal PWM pode assumir valores inteiros entre 0 e 255 (8 *bits* sem sinal), e desta forma, 0 corresponde a um valor de tensão eficaz sobre o motor igual à 0% da tensão de alimentação, enquanto 255 corresponde à 100% da tensão, sendo os valores intermediários uma interpolação linear. Apesar dos motores serem do mesmo modelo, verifica-se pequenas variações durante o funcionamento dos mesmos, devido a diferenças construtivas. Por isso, julgou-se necessário a aplicação do sinal para identificação do sistema em ambos os motores.

As Figuras 5.21(a) e 5.21(b) ilustram o segundo ensaio realizado para os motores direito e esquerdo, respectivamente. A curva apresenta um comportamento visivelmente linear entre a saída do *encoder* (número de pulsos em um período de amostragem) e a entrada do sistema ($0-255$ referente à largura de pulso), e desta forma para a determinação da constante angular da reta, e por consequência a constante de ganho K do motor, foi utilizado o método *Least Square* - mínimos quadrados.



(a) (b)

Figura 5.21: Gráficos da saída dos motores direito e esquerdo, respectivamente.

Os valores obtidos para as constantes K de cada um dos motores são: $K_{direito} = 0{,}1217$ e $K_{esquerdo} = 0{,}1194$.

Como a referência utilizada para controle é um valor para velocidade angular de cada roda, a leitura do *encoder* pode ser transformada de forma a realimentar o controlador diretamente com um sinal de velocidade, ao invés do valor obtido diretamente: número de pulsos por período. A transformação pode ser realizada através da Equação 5.5:

$$V = n_p \frac{p_r}{n_{pv}} \frac{1}{\Delta t} \tag{5.5}$$

onde:

n_p é o número de pulsos contabilizados em um instante Δt;
p_r é o perímetro da roda do robô;
n_{pv} é o número de pulsos por volta;
Δt é o tempo de amostragem.

Assim, a equação considera a dimensão da roda - $p_r = 0{,}1225m$, o período de amostragem - $\Delta t = 0{,}01s$, e o número de pulsos do *encoder* que caracteriza uma volta completa da roda - $n_{pv} = 48$.

Aplicando a Equação 5.5, as constantes do sistema tornam-se: $K_{direito} = 0{,}0311$ e $K_{esquerdo} = 0{,}0305$

Após o sistema ter sido identificado, o controlador pode ser encontrado utilizando o bloco PID do *Simulink*. O sistema de controle é apresentado na Figura 5.22.

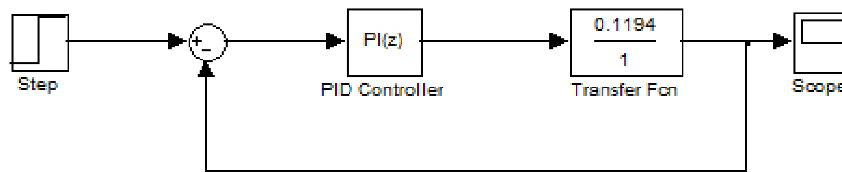


Figura 5.22: Sistema de Controle obtido com o Bloco *PID Controller* do Simulink.

As Figuras 5.23 e 5.24 mostram na ordem, o controlador PI obtido e sua resposta.

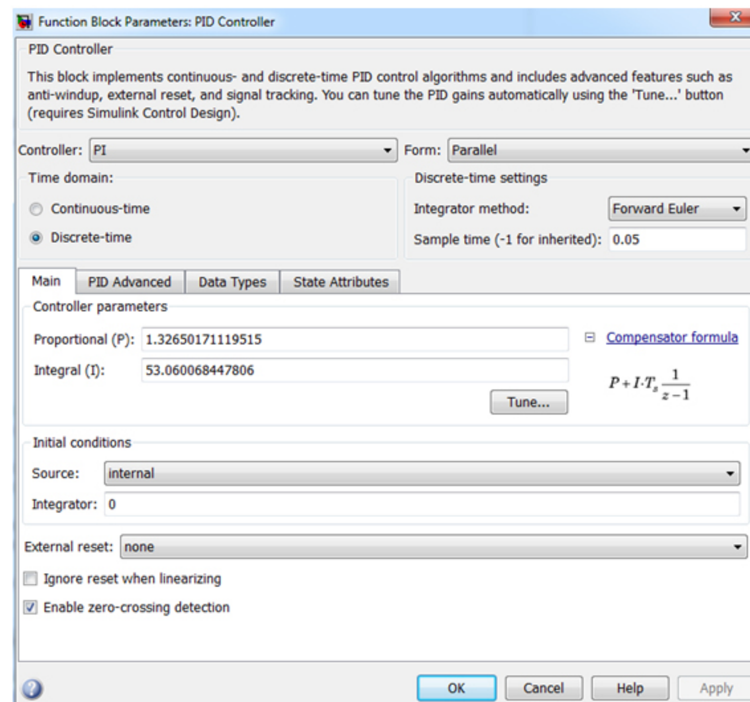


Figura 5.23: Parâmetros do controlador PI.

5.4.1 Nó de Controle de Baixo Nível

Esse nó, aqui chamado de `serial_node`, é responsável por executar todas as atividades de controle, relatadas na Seção anterior, tendo atuação direta sobre os parâmetros dos motores. Como comentado no nó de controle de alto nível, esse nó deve ser capaz de subscrever a um tópico, as instruções das variáveis de atuação linear e angular, sob o formato da mensagem `geometry_msgs/Twist`. Além de fornecer as leituras do *encoder*, via publicação no tópico `/msg_EncoderRobo1`, para uma confirmação do comportamento dos motores conforme os comandos dados, ou para um futuro controle de erro de trajetória via hodometria.

Sendo assim, esse nó deve possuir as seguintes instruções:

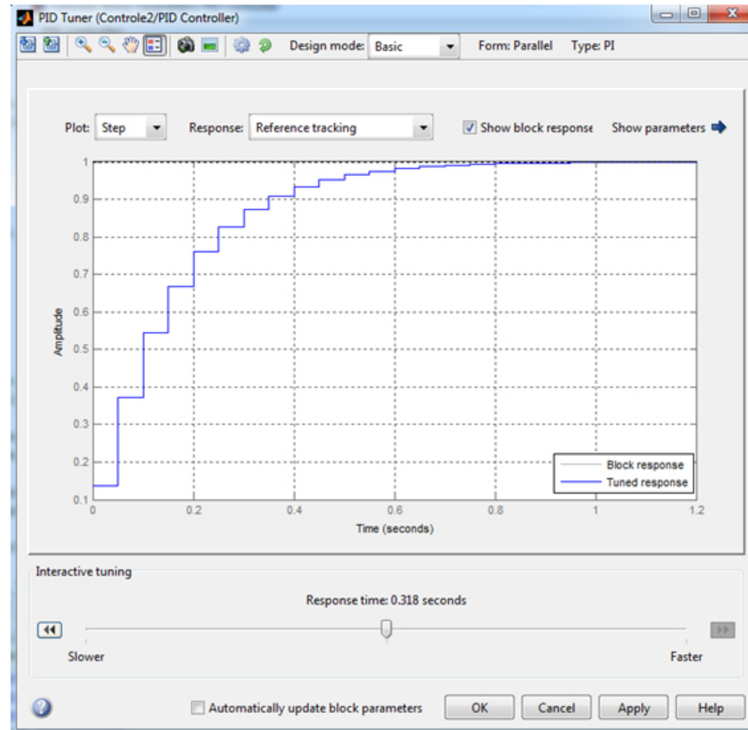


Figura 5.24: Resposta do controlador PI ao degrau unitário.

- `ros::Publisher chatter("msg_EncoderRobo1", &msg_EncoderRobo1)`: possibilita a criação de um publicador de um tópico, que no caso é o `msg_EncoderRobo1`. O ROS ainda irá efetuar uma chamada a função `&msg_EncoderRobo1` todas as vezes que uma nova mensagem chegar.

- `ros::Subscriber<geometry_msgs::Twist> sub("msgRobo1", &chatterRobo)`: possibilita a criação de um subscritor a um tópico de interesse, sendo esse o `/msg_EncoderRobo1` e a mensagem é do tipo `geometry_msgs/Twist`.

Tendo desenvolvido todos esses nós e os interligando a partir dos tópicos listados, referentes à visão robótica e aos controles de alto e baixo nível do robô, foi possível obter os 3/5 restantes do processamento grafo do ROS, conforme demonstrado na Figura 5.25

Figura 5.25: Representação do Processamento Grafo do ROS para a Visão Computacional e para os Controles de Baixo e Alto Nível.

Na Figura 5.26, está ilustrado todo o Processamento Grafo do ROS realizado para a concretização deste trabalho. É importante notar que o mesmo teve que ser seccionado devido à sua

extensão, sendo repetido o tópico /camera/image_rect_color para representar sua continuidade.

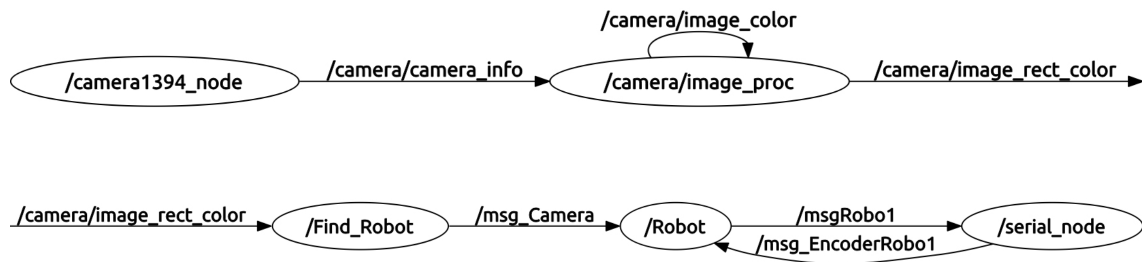


Figura 5.26: Representação do Grafo do ROS.

5.5 Configurações Adicionais

Além de todos os pontos já abordados, outros três fatos importantes devem ser abordados, sendo eles:

- a questão do meio pelo qual é realizada a comunicação entre o nó de controle de alto nível, esse localizado no computador em uso, e o nó de baixo nível, esse localizado no *Arduino Nano* no robô móvel.
- as adaptações necessárias ao código do *Arduino Nano*, para que o mesmo pudesse ser compilado de forma a possibilitar a publicação e a subscrição por parte do mesmo.
- e por último, as alterações que devem ser efetuadas no arquivo CMakeLists.txt, para que todo esse sistema desenvolvido pudesse ser compilado segundo as diretrizes do ROS e do *OpenCV*.

Como já abordado na Seção 5.1, a comunicação entre o robô móvel e o computador é efetuada por um módulo RF, no caso o *XBee*, o qual estabelece uma conexão serial entre ambas as partes. Para que seja concretizada a comunicação é necessário que cada uma das partes possua um módulo *Xbee*, sendo que no caso em estudo, os mesmos deverão ser configurados para uma rede na topologia estrela[2]. De modo que o dispositivo ligado ao computador seja o *Coordenador*, esse responsável por administrar a rede *ZigBee*, fazendo sua inicialização, manutenção e reconhecimento dos demais módulos conectados à mesma. E o dispositivo instalado no robô seja um *Dispositivo Final*, esse com influência direta nos atuadores do robô.

[2]Onde um dispositivo *Coordenador* é responsável por efetuar a comunicação com cada um dos demais, sendo estes dispositivos *Roteadores* ou *Finais*. E esses outros dois tipos são capazes de comunicar exclusivamente com o dispositivo *Coordenador*.

Com o auxílio do adaptador/conversor USB, também mostrado na Seção 5.1, é possível a conexão do módulo *XBee* coordenador ao computador, mas para que o mesmo possa ter o direito de leitura e escrita na respectiva porta é preciso a execução das seguintes linhas de comando, junto ao terminal do Linux, sendo a primeira executada para conferir as permissões até então definidas e a segunda para liberar as mesmas, caso ainda não estejam.

```
$ ls -l /dev/ttyUSB0
$ sudo chmod a+rw /dev/ttyUSB0
```

Para que a rede *ZigBee* funcione da forma descrita, é necessário a prévia configuração de cada um dos módulos *Xbee-PRO*. Os processos de configuração dos dispositivos e de ativação da rede são realizados através do pacote do ROS denominado `rosserial_xbee`. Para a configuração é utilizado o nó `setup_xbee.py`, sendo exeutado a seguinte linha de comando para o módulo *Coordenador*:

```
$ rosrun rosserial_xbee setup_xbee.py -C /dev/ttyUSB0 0
```

Onde: `-C` - é uma opção e atribui que este módulo será o Coordenador; `/dev/ttyUSB0` - atribui qual a porta serial em que está conectado o módulo + adaptador ao computador para a realização da configuração; `0` - é um endereço de 16 bits do módulo na rede, devendo ser único para cada um dos dispositivos e 0 para o Coordenador.

Para a configuração do *Dispositivo Final* é preciso executar a seguinte linha de comando:

```
$ rosrun rosserial_xbee setup_xbee.py /dev/ttyUSB0 1
```

Onde: `/dev/ttyUSB0` - atribui qual a porta serial em que está conectado o módulo + adaptador ao computador para a realização da configuração; `1` - é um endereço de 16 bits do módulo na rede, devendo ser único para cada um dos dispositivos.

A taxa de transferência é definida de fábrica em 9.600 bps, mas ao executar essa configuração, a partir do pacote `rosserial_xbee`, a mesma passa para 57.600 bps. Caso seja necessário alterá-la basta acrescentar, logo após ao nome do nó, o seguinte comando `-c CHANNEL`, onde `CHANNEL` deverá ser substituído por uma frequência de até no máximo 250.000 bps.

A ativação da rede *ZigBee* e inicialização da comunicação é feita através do nó `xbee_network.py`, sendo necessário executar a seguinte linha de comando:

```
$ rosrun rosserial_xbee xbee_network.py /dev/ttyUSB0 1
```

Realizado todo esse procedimento, o *hardware* do sistema de comunicação está apto para a execução. No entanto é necessário que adaptações sejam feitas no IDE (*Integrated Development Environment*) do *Arduino* e no próprio código do *Arduino* para que o robô seja capaz de se comportar como um nó do sistema ROS e consiga comunicar serialmente através do módulo *XBee*.

Essa adaptação é concretizada com o uso do pacote `rosserial_arduino`, onde esse fornece um protocolo de comunicação ROS, que funciona através da UART (*Universal Asynchronous Receiver/ Transmitter*) do *Arduino*, possibilitando o uso do ROS diretamente com o IDE do *Arduino*. Com isso, a placa do *Arduino* Nano passa a ser um nó completo do ROS, podendo publicar e subscrever mensagens ROS diretamente em tópicos.

As vinculações do ROS são implementas como uma biblioteca do *Arduino*, e como toda biblioteca o `ros_lib` funciona adicionado todas as implementações de biblioteca no diretório de bibliotecas do *Arduino*. Tendo realizada a instalação do `ros_lib` no IDE do *Arduino*, quando da execução deste, no menu *File/Examples* estará disponível a opção *ros_lib*, confirmando a perfeita integração entre sistemas.

Na Figura 5.27 é demonstrada uma estrutura do ROS, sendo possível observar suas divisões em gerenciamento, infraestrutura e *drives*, além das interligações entre esses e os demais dispositivos, como o *Arduino* e o *XBee*. É importante comentar, que na configuração apresentada no presente trabalho, o *XBee* está sendo empregado como o elo entre o Rosserial e o *Arduino*.

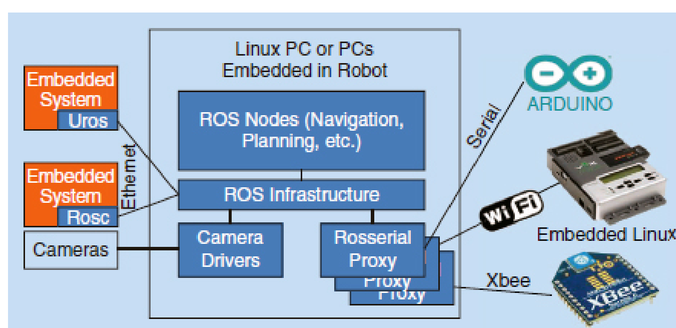


Figura 5.27: Estrutura e interligações para o envio de mensagens ROS.
Disponível em: [Bouchier 2013]

As adequações no código do *Ardino* são feitas através da utilização das bibliotecas `rosserial` no próprio código. Sendo preciso para isso, acrescentar a seguinte linha de código:

```
#include <ros.h>
```

Essa necessariamente antes de quaisquer outros arquivos de cabeçalho para que o IDE do *Arduino* possa ser capaz de localizar suas atribuições.

Os pacotes nos ROS são compilados a partir da ferramenta `rosbuild`, a qual contém *scripts* capazes de gerenciar o sistema de construção baseado na API *CMake*. Dessa forma, cada pacote requer um arquivo, denominado `CMakeLists.txt`, que chama a API *CMake* e conforme as diretrizes contidas no mesmo, possa ser construído o pacote do ROS. Na sequência está apresentado o conteúdo do arquivo utilizado para compilação do pacote criado, o `robot_soccer`.

```
cmake_minimum_required(VERSION 2.4.6)

find_package(OpenCV)

include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)

//#include_directories(${OpenCV_INCLUDE_DIRS})
include_directories(/home/elias/opencv-2.4.5/include/)
include_directories(/home/elias/fuerte_workspace/sandbox/robot_soccer/include/)
include_directories(/home/elias/fuerte_workspace/sandbox/robot_soccer/include/
    package_bgs/)
include_directories(/home/elias/fuerte_workspace/sandbox/robot_soccer/include/
    package_bgs/dp)

//# Set the build type. Options are:
//# Coverage : w/ debug symbols, w/o optimization, w/ code-coverage
//# Debug : w/ debug symbols, w/o optimization
//# Release : w/o debug symbols, w/ optimization
//# RelWithDebInfo : w/ debug symbols, w/ optimization
//# MinSizeRel : w/o debug symbols, w/ optimization, stripped binaries
//# set(ROS_BUILD_TYPE RelWithDebInfo)

rosbuild_init()

//#set the default path for built executables to the "bin" directory
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
//#set the default path for built libraries to the "lib" directory
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

//#uncomment if you have defined messages
rosbuild_genmsg()

//#uncomment if you have defined services
//#rosbuild_gensrv()

//#common commands for building c++ executables and libraries
//#rosbuild_add_library(${PROJECT_NAME} src/example.cpp)
```

```
rosbuild_add_library(ControlePosicao src/ControlePosicao.cpp)
rosbuild_add_library(ControladorPID src/ControladorPID.cpp)
rosbuild_add_library(Trajetoria src/Trajetoria.cpp)
rosbuild_add_library(ManipuladorTrajetoria src/ManipuladorTrajetoria.cpp)

rosbuild_add_library(FrameDifferenceBGS include/package_bgs/FrameDifferenceBGS.cpp)
rosbuild_add_library(StaticFrameDifferenceBGS include/package_bgs/
    StaticFrameDifferenceBGS.cpp)
rosbuild_add_library(DPEigenbackgroundBGS include/package_bgs/dp/DPEigenbackgroundBGS.
    cpp)
rosbuild_add_library(Eigenbackground include/package_bgs/dp/Eigenbackground.cpp)
rosbuild_add_library(DPWrenGABGS include/package_bgs/dp/DPWrenGABGS.cpp)
rosbuild_add_library(WrenGA include/package_bgs/dp/WrenGA.cpp)

//#target_link_libraries(${PROJECT_NAME} another_library)
//#rosbuild_add_boost_directories()
//#rosbuild_link_boost(${PROJECT_NAME} thread)

//#rosbuild_add_executable(example examples/example.cpp)
rosbuild_add_executable(Robot src/Robot.cpp)
rosbuild_add_executable(Find_Robot src/Find_Robot.cpp)

//#target_link_libraries(example ${PROJECT_NAME})
target_link_libraries(Robot ControlePosicao ControladorPID Trajetoria
    ManipuladorTrajetoria)
target_link_libraries(Find_Robot FrameDifferenceBGS StaticFrameDifferenceBGS
    DPEigenbackgroundBGS Eigenbackground DPWrenGABGS WrenGA)
target_link_libraries(Find_Robot /home/elias/opencv-2.4.5/build/lib)
```

./code/CMakeLists.txt

Como pode ser notado, num primeiro momento é feita a inclusão de todos os diretórios onde estão dispostos os arquivos de cabeçalho `.h`. Na sequência são configurados os caminhos de diretórios como: `$PROJECT_SOURCE_DIR/bin` e `$PROJECT_SOURCE_DIR/lib`, esses, respectivamente direcionados ao armazenamento dos executáveis e das bibliotecas. Ainda é necessário definir se no pacote serão geradas mensagens, via comando: `rosbuild_genmsg()`. Além da adição pontual de cada biblioteca, executável e *links* individuais dos nós.

5.6 Execução Prática

Com todo o sistema de visão robótica, controle de alto nível e controle de baixo nível montados em nós do ROS, foram realizados alguns testes para verificar o funcionamento de cada um e o respectivo comportamento de todo o embasamento teórico na aplicação prática.

Durante a execução foram coletados dados referentes a localização do robô, no decorrer do campo, sendo assim possível plotar a real trajetória percorrida pelo robô, conforme mostrado na Figura 5.28. É importante notar que os eixos estão com suas unidades em *pixels*, logo com dimensões de 640x480 que é o formato da imagem.

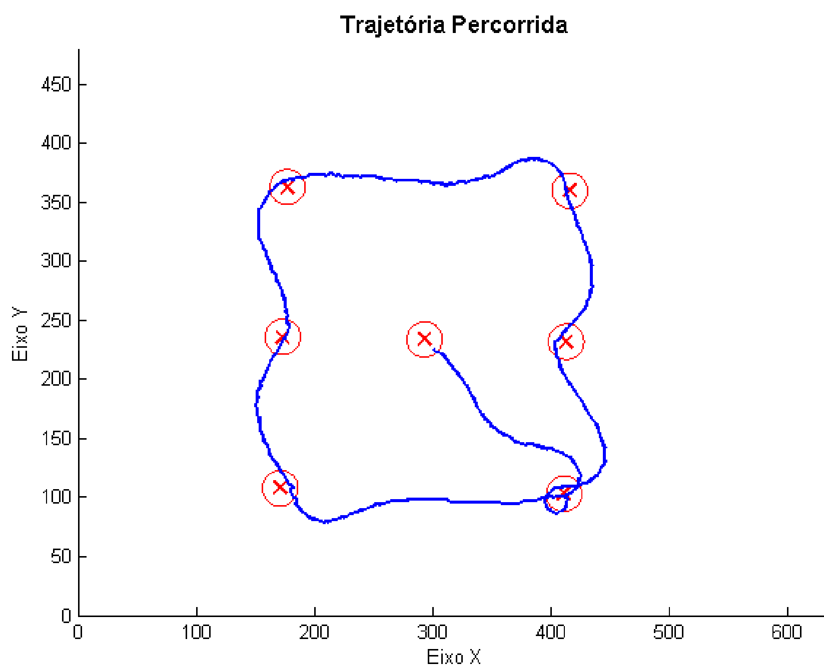


Figura 5.28: Trajetória percorrida pelo robô móvel proposto.

Nessa Figura estão identificados alguns *way-points* atribuídos como sendo os possíveis locais em que se encontraria a bola. Também estão apresentadas circunferências, de diâmetro de 30 *pixels*, ao redor dos referidos *way-points* como sendo a região de erro aceitável para localização do ponto, por parte do controle de alto nível. Sabe-se que o robô possui dimensões de 7,5 cm x 7,5 cm e apesar de não aparecer na referida Figura, sua representação em *pixels* é de 25 x 25.

Com a realização desse teste foi possível colher dados de modo a produzir três outros gráficos, os quais apresentam o deslocamento do robô ao longo do Eixo X, um segundo com o deslocamento ao longo do Eixo Y e um terceiro com a variação de *theta*. É importante relatar que tais gráficos foram plotados a partir de 535 amostras.

Na Figura 5.29 é possível visualizar que o gráfico apresenta duas curvas, sendo uma em azul, a qual representa a variação da coordenada X do *way-point*, e a outra vermelha, a qual apresenta o real comportamento da variação do robô em X. A variação da coordenada do *way-point* é sempre realizada quando o robô alcança o *way-point* de destino ou a região de erro aceitável. De certa forma, buscou-se simular o deslocamento da bola ao longo do campo, mesmo que de forma lenta.

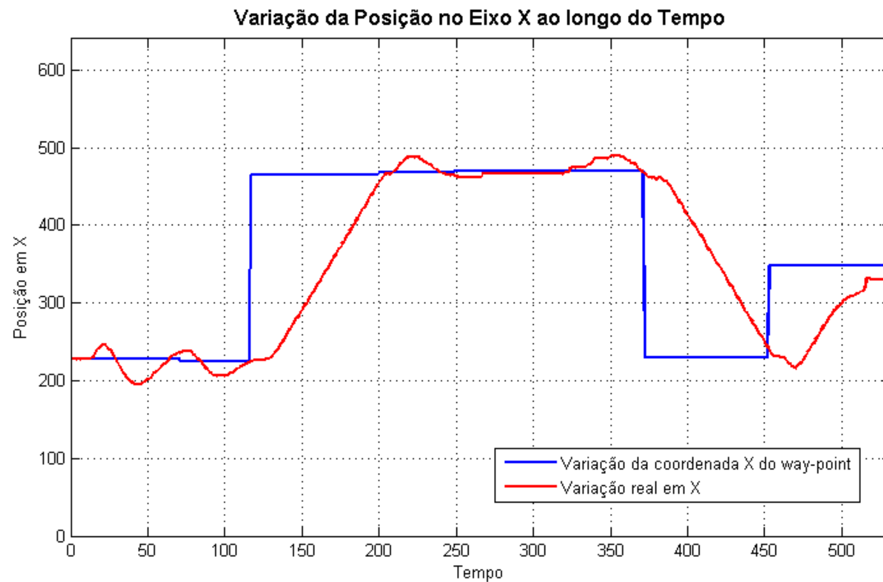


Figura 5.29: Variação da Posição no Eixo X ao longo do Tempo.

Na Figura 5.30 visualiza-se a variação ao longo do tempo, mas agora das coordenadas Y, tanto do robô quanto do *way-point*.

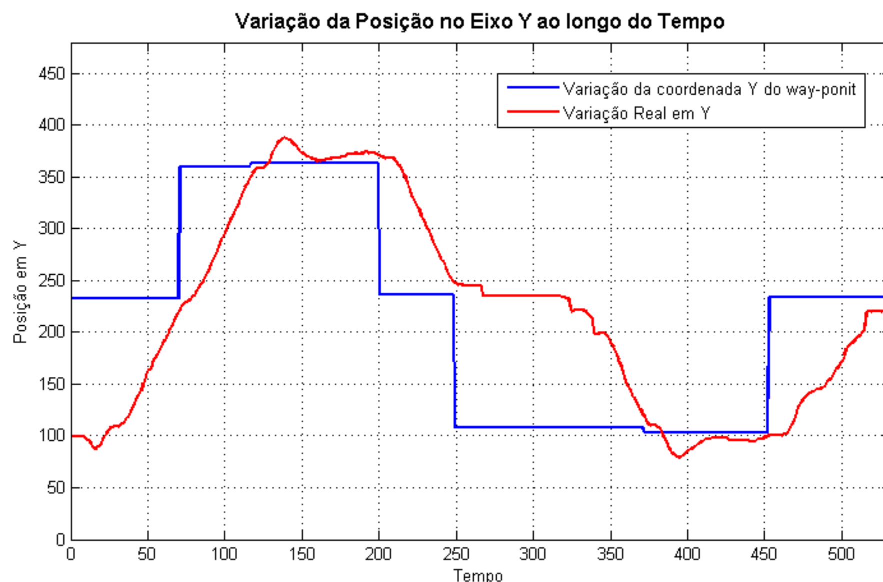


Figura 5.30: Variação da Posição no Eixo Y ao longo do Tempo.

E por último, na Figura 5.31, observa-se a variação do ângulo θ - o ângulo global do robô - ao longo do tempo.

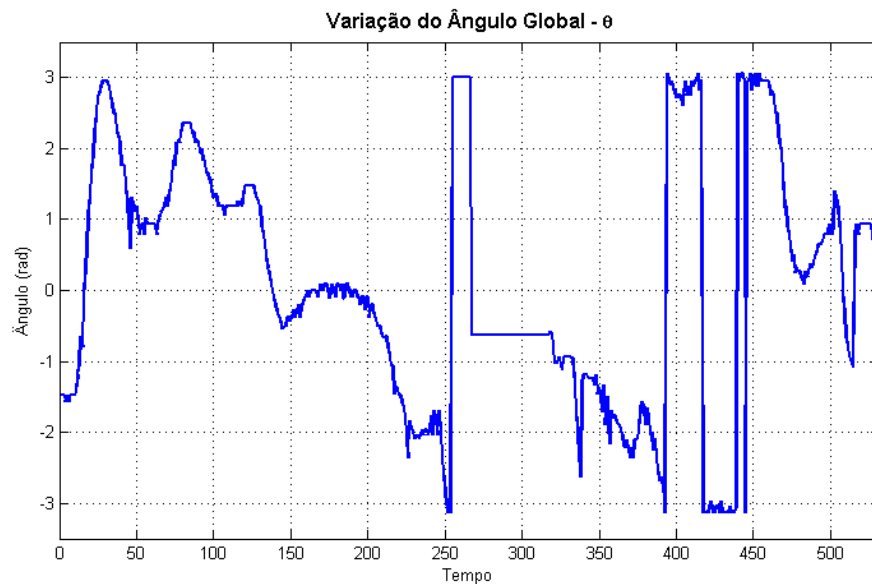


Figura 5.31: Variação do Ângulo Global - θ.

Dois pontos importantes devem ser levantados, sendo um primeiro visível nas Figuras 5.29, 5.30 e 5.31, e um segundo apenas na terceira e última. Num primeiro momento, no intervalo de amostragem de 267 a 318, é possível verificar nos três gráficos que essas váriáveis não se alteram, isso se deve a um pequeno período em que a visão computacional perdeu a localização do robô, mais precisamente no momento em que o algoritmo (*HoughCircles*) não conseguiu localizar uma das etiquetas no topo do robô, esse problema muito provavelmente provocado pela maior incidência de luz nesse ponto.

O segundo ponto, o qual pode ser observado no intervalo de amostragem de 393 a 446, do terceiro gráfico, ou seja, o da variação do ângulo, demonstra que há em alguns instantes pontos de descontinuidades, esses provocados pela passagem de $-\pi$ rad para π rad. Apesar de tal descontinuidade o código de controle de alto nível o contornou.

6 Conclusão

Este trabalho demonstrou a implementação de toda a estrutura necessária para o funcionamento de um sistema baseado no ROS, bem como o desenvolvimento de um algoritmo de visão robótica. Foram apresentados, um simples, porém eficiente, controle de trajetória e ainda os controles de alto e baixo níveis com resultados satisfatórios.

Foram apresentadas considerações gerais e um breve histórico sobre Robótica Móvel, demonstrando o crescimento no desenvolvimento de novos experimentos e de novas teorias e técnicas no decorrer dos anos. Foram enumeradas também, definições e classificações pertinentes ao ramo da Robótica Móvel, sendo essas de extrema necessidade para traçar os rumos a serem seguidos no desenrolar do trabalho e na escolha dos sensores mais adequados para transduzir as grandezas de interesse. Neste contexto foi apresentado a Cinemática para o robô móvel proposto, bem como conceitos importantes para a definição do método de planejamento de caminho e dos controladores de baixo e de alto nível.

Foi também realizada uma discussão sobre alguns conceitos relacionados aos fundamentos da cor e as definições de espaço de cor e de espaço de atributos de cor, os quais possibilitaram ao nó de visão robótica demonstrar maior robustez quanto às interferências de ruídos presentes na imagem capturada e de variações nos índices de iluminância em toda a extensão do campo. Abordou-se ainda a biblioteca *OpenCV*, a qual possibilitou dar maior dinâmica e agilidade no emprego de técnicas de processamento de vídeo e de visão computacional. Num primeiro momento, com uma das técnicas de visão computacional, a de calibração da câmera monocular, foi possível atribuir ao sistema de visão maior robustez ainda, uma vez que o processo de localização de círculos, por parte do algoritmo da Transformada de Hough, tornou-se mais eficiente. Visto que a imagem não apresentava mais distorções nas regiões próximas às extremidades do campo, distorções essas que faziam com que os círculos fossem representados como elipses e consequentemente descartados ou identificados em locais diferentes dos reais.

Ainda referente à visão computacional, com o emprego das técnicas de subtração de fundo, foi viável a redução significativa na detecção de falsos círculos e consequentemente de falsas localizações do robô, visto que, ao ser subtraído o plano de fundo da imagem capturada, a

imagem resultante compreendia um pequena região de interesse em volta do robô móvel.

Outro ponto abordado foi a completa descrição da estrutura do ROS, sendo explicado o sistema de arquivos, o processamento grafo, dentre outras definições e características que rendem ao ROS o seu carácter de fácil aplicação, compartilhamento de códigos e abstração de *hardware*. Foi possível definir a forma como seria estruturado o sistema proposto pelo presente trabalho, os tópicos e suas respectivas mensagens com os dados de interesse, além dos parâmetros de configuração que deveriam ser atribuídos a cada nó.

Tendo sido feita toda essa abordagem, o sistema proposto foi montado em cinco nós, os quais desempenhavam papéis distintos e se comunicavam através das mensagens publicadas e subscritadas em tópicos. A estrutura desse sistema demonstrou uma rede em forma praticamente linear, começando pelo nó de captura de vídeo - `/camera1394_node`, cuja imagem era processada pelo nó `/camera/image_proc`, fornecendo uma imagem com as características de interesse realçadas para o nó de visão robótica - `Find_Robot`. Com a identificação do robô móvel, as coordenadas de localização eram passadas para o nó - `Robot`, nó esse responsável por enviar, através de um módulo RF, uma mensagem com os comandos de atuação para o nó de controle de baixo nível - `serial_node`.

Na sequência foi apresentado o robô diferencial desenvolvido e seus respectivos componentes, bem como todo o desenvolvimento necessário para a elaboração dos códigos dos nós de visão robótica, os controles de alto e baixo nível e resultados de simulações e práticos do adequado funcionamento do conjunto de atividades propostas.

6.1 Trabalhos Futuros

Com intuito de continuar o desenvolvimento deste trabalho e a resolução de diversos temas que foram abordados de forma superficial no seu desenrolar, se propõe as seguintes atividades para trabalhos futuros.

- Um levantamento mais criterioso sobre a modelagem cinemática do robô diferencial, juntamente com a modelagem dinâmica, essa última não abordada no trabalho.
- Uma análise com simulações e práticas para outros métodos de controle de alto nível, de modo a garantir o seguimento da trajetória por parte do robô.
- Desenvolver outros métodos de planejamento de trajetória, como por exemplo o de Campos Potenciais.

- Aprimoramento do código e dos processos da visão robótica, de modo a torná-los mais robustos e capazes de localizar os demais robôs que estiverem em campo, juntamente com os obstáculos, que no caso, seriam os adversários. Tal preocupação se deve ao fato do aumento no tempo de execução desses processos, por ser mais de uma identificação.

- Realizar testes de desempenho do mecanismo de troca de mensagens do ROS - *publisher/-subscriber* - considerando o acréscimo de outros robôs e consequentemente o aumento de tópicos específicos para cada robô.

- Desenvolver o estudo de situações conflitantes na localização dos robôs, quando estes estiverem próximos ao ponto de terem suas etiquetas identificadas como sendo de outro robô.

- Realizar o aprimoramento dos códigos através da Engenharia de *Software*, para assim reduzir o tempo gasto e o custo computacional para a execução dos mesmos.

Referências Bibliográficas

[Bianchi e Reali-Costa 2000]BIANCHI, R. A. C.; REALI-COSTA, A. H. O Sistema de Visão Computacional do Time FutePoli de Futebol de Robôs. *XIII Congresso Brasileiro de Automática*, Florianópolis - SC, p. 2156, 2000.

[Bouchier 2013]BOUCHIER, P. Embedded ROS [ROS Topics]. *Robotics & Automation Magazine, IEEE*, v. 20, n. 2, p. 17–19, 2013.

[Brooks 1986]BROOKS, R. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, v. 2, n. 1, p. 14–23, 1986.

[Cattin 2012]CATTIN, P. *Digital Image Fundamentals*: Introduction to signal and image processing. 2012. Disponível em: <http://miac.unibas.ch/SIP/02-Fundamentals.html>. Acesso em: 22 jun. 2013.

[Chapron 1992]CHAPRON, M. A new chromatic edge detector used for color image segmentation. In: IEEE. *Pattern Recognition Conference C: Image, Speech and Signal Analysis, Proceedings., 11th IAPR International Conference on*. [S.l.], 1992. III, p. 311–314.

[Cheng et al. 2001]CHENG, H.-D. et al. Color Image Segmentation: Advances and Prospects. *Pattern Recognition*, Elsevier, v. 34, n. 12, p. 2259–2281, 2001.

[Coleman 2013a]COLEMAN, D. T. *Getting Started*: Introduction. 2013a. Disponível em: <http://www.ros.org/wiki/ROS/Introduction>. Acesso em: 14 mai. 2013.

[Coleman 2013b]COLEMAN, D. T. *ROS/Concepts*: Introduction. 2013b. Disponível em: <http://wiki.ros.org/ROS/Concepts>. Acesso em: 17 mai. 2013.

[Coleman 2013c]COLEMAN, D. T. *Repositories*: Introduction. 2013c. Disponível em: <http://wiki.ros.org/Repositories>. Acesso em: 5 jun. 2013.

[Coleman 2013d]COLEMAN, D. T. *roslaunch/XML*. 2013d. Disponível em: <http://wiki.ros.org/roslaunch/XML>. Acesso em: 15 jun. 2013.

[Coleman 2013e]COLEMAN, D. T. *roscore*. 2013e. Disponível em: <http://wiki.ros.org/roscore>. Acesso em: 15 jun. 2013.

[Conley 2012a]CONLEY, K. *Packages*. 2012a. Disponível em: <http://wiki.ros.org/Packages>. Acesso em: 12 abr. 2013.

[Conley 2012b]CONLEY, K. *Stacks*. 2012b. Disponível em: <http://wiki.ros.org/Stacks>. Acesso em: 12 abr. 2013.

[Conley 2012c]CONLEY, K. *Nodes*. 2012c. Disponível em: <http://wiki.ros.org/Nodes>. Acesso em: 12 abr. 2013.

[Conley 2012d]CONLEY, K. *Stacks*. 2012d. Disponível em: <http://wiki.ros.org/Topics>. Acesso em: 12 abr. 2013.

[Conley 2012e]CONLEY, K. *Services*. 2012e. Disponível em: <http://wiki.ros.org/Services>. Acesso em: 12 abr. 2013.

[Conley 2012f]CONLEY, K. *Master*. 2012f. Disponível em: <http://wiki.ros.org/Master>. Acesso em: 12 abr. 2013.

[Cousins 2010]COUSINS, S. Welcome to ROS Topics [ROS Topics]. *Robotics & Automation Magazine, IEEE*, v. 17, n. 1, p. 13–14, março 2010.

[Dietrich 2013]DIETRICH, A. *Client Libraries*. 2013. Disponível em: <http://wiki.ros.org/Client%20Libraries>. Acesso em: 10 agosto. 2013.

[Digital 2013]DIGITAL, R. S. *BSD (FreeBSD, OpenBSD e NetBSD)*. 2013. Disponível em: <http://www.segurancadigital.info/dicas/169-bsd-freebsd-openbsd-e-netbsd>. Acesso em: 1 out. 2013.

[Dudek e Jenkin 2010]DUDEK, G.; JENKIN, M. *Computational Principles of Mobile Robotics*. 2. ed. [S.l.]: Cambridge University Press, 2010.

[Flores 2011]FLORES, C. S. *Futebol robótico inspira cadeira de rodas inteligente*. 2011. Disponível em: <http://www.cienciahoje.pt/index.php?oid=47114&op=all>. Acesso em: 04 set. 2013.

[Foley et al. 1996]FOLEY, J. D. et al. *Computer Graphics*: Principles and Practice. [S.l.]: Addison-Wesley Professional, 1996. (Addison-Wesley systems programming series).

[Foote 2013a]FOOTE, T. *ROS Fuerte Turtle*. 2013a. Disponível em: <http://ros.org/wiki/fuerte>. Acesso em: 11 abr. 2013.

[Foote 2013b]FOOTE, T. *Manifest*. 2013b. Disponível em: <http://wiki.ros.org/Manifest>. Acesso em: 11 abr. 2013.

[Garage 2011a]GARAGE, W. *ROS*. 2011a. Disponível em: <http://www.willowgarage.com/pages/software/ros-platform>. Acesso em: 04 set. 2013.

[Garage 2011b]GARAGE, W. *OpenCV*. 2011b. Disponível em: <http://www.willowgarage.com/pages/software/opencv>. Acesso em: 04 set. 2013.

[Gomes et al. 2006]GOMES, M. M. et al. A Visão Computacional - Construção de Robôs Jogadores de Futebol - Parte 4. *Mecatrônica Facil*, Editora Saber, n. 31, 2006.

[Gomes 2009]GOMES, P. R. *Um Estudo sobre Avaliação da Execução do BLAST em Ambientes Distribuídos*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro - RJ, 2009.

[Gonzales e Woods 2001]GONZALES, R. C.; WOODS, R. E. *Digital Image Processing*. 2. ed. [S.l.]: Prentice Hall, 2001.

[Grittani, Gallinelli e Ramírez 2000]GRITTANI, G.; GALLINELLI, G.; RAMÍREZ, J. M. FutBot: A Vision System for Robotic Soccer. In: *Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI: Advances in Artificial Intelligence*. London, UK: Springer-Verlag, 2000. p. 350–358.

[IBGE 2008]IBGE. *Metadados*. 2008. Disponível em: <http://www.metadados.ibge.gov.br/>. Acesso em: 1 set. 2013.

[IEEE 2008]IEEE. *7th Latin American IEEE Student Robotics Competition*: Rules for the IEEE Very Small Competition. [S.l.], 2008.

[Initiative 2013]INITIATIVE, O. S. *BSD license*. 2013. Disponível em: <http://opensource.org/licenses/bsd-license.php>. Acesso em: 1 out. 2013.

[Intel 1999-2001]INTEL, C. *Open Source Computer Vision Library*: Reference Manual. U.S.A., 1999–2001.

[Jain, Kasturi e Schunck 1995]JAIN, R.; KASTURI, R.; SCHUNCK, B. G. *Machine Vision*. [S.l.]: McGraw-Hill, 1995.

[Janeczko 2010]JANECZKO, C. *Processamento Digital de Imagens*: Imagem Colorida. 2010. Disponível em: <http://www.pessoal.utfpr.edu.br/janeczko/>. Acesso em: 22 jun. 2013.

[Júnior, Facon e Neto 1997]JÚNIOR, A. S. B.; FACON, J.; NETO, A. C. Uso da Análise de Componentes Principais na Segmentação de Imagens Coloridas. *XXIX Congresso da Sociedade Brasileira de Pesquisa Operacional*, Salvador - BA, 1997.

[Keramas 1998]KERAMAS, J. G. *Robot Technology Fundamentals*. [S.l.]: Delmar Learning, 1998.

[Khatib 1986]KHATIB, O. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, v. 5, n. 1, p. 90–98, 1986.

[Kitano 1998]KITANO, H. *RoboCup-97: robot soccer world cup I*. [S.l.]: Springer, 1998.

[Kumpel e Sarradilla 1990]KUMPEL, D.; SARRADILLA, F. Robot navigation strategies in a partially known environment using a space-time learning graph. In: *Proceedings of the Sixth CIM-Europe Annual Conference*. [S.l.: s.n.], 1990. p. 65–75.

[Laganière 2011]LAGANIÈRE, R. *OpenCV 2 computer vision application programming cookbook*. [S.l.]: Packt Publishing, 2011.

[Lamprianidis 2012a]LAMPRIANIDIS, N. *srv*. 2012a. Disponível em: <http://wiki.ros.org/srv>. Acesso em: 12 abr. 2013.

[Lamprianidis 2012b]LAMPRIANIDIS, N. *Messages*. 2012b. Disponível em: <http://wiki.ros.org/Messages>. Acesso em: 12 abr. 2013.

[Langer 2007]LANGER, R. A. *Estudo e Implementação de Métodos para Planejamento de Trajetórias e controle de Robôs Móveis não Holonômicos.* Dissertação (Mestrado) — Pontifícia Universidade Católica do Paraná - PUCPR, Curitiba - PR, 2007.

[Lozano-Pérez 1983]LOZANO-PÉREZ, T. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, v. 100, n. 2, p. 108–120, 1983.

[Ma, Kosecka e Sastry 1999]MA, Y.; KOSECKA, J.; SASTRY, S. S. Vision guided navigation for a nonholonomic mobile robot. *IEEE Transactions on Robotics and Automation*, v. 15, n. 3, p. 521–536, 1999.

[Marengoni e Stringhini 2010]MARENGONI, M.; STRINGHINI, D. Tutorial: Introdução à Visão Computacional usando OpenCV. *Revista de Informática Teórica e Aplicada*, v. 16, n. 1, p. 125–160, 2010.

[Martins 2007]MARTINS, M. F. *Aprendizado por Reforço Acelerado por Heurísticas Aplicadas ao Domínio do Futebol de Robôs*. Dissertação (Mestrado) — Centro Universitário da FEI, São Bernardo do Campo - SP, 2007.

[Mataric 1992]MATARIC, M. J. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, v. 8, n. 3, p. 304–312, 1992.

[Medeiros 2003]MEDEIROS, A. A. D. de. *Modelagem e Análise de Sistemas Dinâmicos*. Natal-RN, março 2003.

[Moravec 1999]MORAVEC, H. *Robot: Mere Machine to Transcendent Mind*. [S.l.]: Oxford University Press, 1999. (Science/Computers).

[Nilsson 1969]NILSSON, N. J. A mobile automaton: An application of artificial intelligence techniques. In: *Proceedings of the 1st. International Joint Conference on Artificial Intelligence*. [S.l.: s.n.], 1969. p. 509–520.

[Novak e Seyr 2004]NOVAK, G.; SEYR, M. Simple path planning algorithm for two-wheeled differentially driven (2wdd) soccer robots. *WISES*, v. 4, p. 91–102, 2004.

[OpenCV 2012]OPENCV, D. T. *The OpenCV*: Reference Manual. [S.l.], 2012.

[OpenCV 2013a]OPENCV, D. T. *Introduction*. 2013a. Disponível em: <http://docs.opencv.org/modules/core/doc/intro.html>. Acesso em: 04 jun. 2013.

[OpenCV 2013b]OPENCV, D. T. *About*. 2013b. Disponível em: <http://opencv.org/about.html>. Acesso em: 04 jun. 2013.

[OpenCV 2013c]OPENCV, D. T. *imgproc. Image Processing*. 2013c. Disponível em: <http://docs.opencv.org/modules/imgproc/doc/imgproc.html>. Acesso em: 02 jul. 2013.

[OpenCV 2013d]OPENCV, D. T. *Smoothing Images*. 2013d. Disponível em: <http://docs.opencv.org/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html#smoothing>. Acesso em: 02 jul. 2013.

[OpenCV 2013e]OPENCV, D. T. *Miscellaneous Image Transformations*. 2013e. Disponível em: <http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html>. Acesso em: 02 jul. 2013.

[OpenCV 2013f]OPENCV, D. T. *Feature Detection*. 2013f. Disponível em: <http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html>. Acesso em: 03 ago. 2013.

[Pedersen 2007]PEDERSEN, S. J. K. Circular hough transform. *Aalborg University, Vision, Graphics, and Interactive Systems*, 2007.

[Pereira e Pimenta 2011]PEREIRA, G. A. S.; PIMENTA, L. C. d. A. *Planejamento de Movimento e Estratégias de Controle de Robôs*. Belo Horizonte - MG, 2011.

[Pieri 2002]PIERI, E. R. de. *Curso de Robótica Móvel*. Florianópolis - SC, março 2002.

[Pooley 2013]POOLEY, A. *ROS/EnvironmentVariables*. 2013. Disponível em: <http://wiki.ros.org/ROS/EnvironmentVariables>. Acesso em: 10 jun. 2013.

[Quigley et al. 2009]QUIGLEY, M. et al. ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software*. [S.l.: s.n.], 2009. v. 3, n. 3.2.

[Ribeiro, Costa e Romero 2001]RIBEIRO, C.; COSTA, A.; ROMERO, R. Robos móveis inteligentes: Princípios e técnicas. In: *Anais do XXI Congresso da Sociedade Brasileira de Computação - SBC*. [S.l.: s.n.], 2001. v. 3, p. 258–306.

[Scioli 2013a]SCIOLI, L. D. *msg*. 2013a. Disponível em: <http://wiki.ros.org/msg>. Acesso em: 06 jun. 2013.

[Scioli 2013b]SCIOLI, L. D. *Bags*. 2013b. Disponível em: <http://wiki.ros.org/Bags>. Acesso em: 6 jun. 2013.

[Siegwart e Nourbakhsh 2004]SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. 1. ed. [S.l.]: The MIT press, 2004.

[Sobral 2013]SOBRAL, A. BGSLibrary: An OpenCV C++ Background Subtraction Library. In: *IX Workshop de Visão Computacional (WVC'2013)*. Rio de Janeiro, Brazil: [s.n.], 2013.

[Souto 2003]SOUTO, R. P. *Segmentação de Imagem Multiespectral utilizando-se o Atributo Matiz*. Dissertação (Mestrado) — INPE, São José dos Campos - SP, 2003.

[Stemmer et al. 2005]STEMMER, M. R. et al. *Apostila de Sistemas de Visão*. Florianópolis - SC, 2005.

[Thomas 2013]THOMAS, D. *Parameter Server*. 2013. Disponível em: <http://wiki.ros.org/Parameter%20Server>. Acesso em: 17 ago. 2013.

[Thurlow 2009]THURLOW, R. Rpc: Remote procedure call protocol specification version 2. 2009.

[Zelinsky 1992]ZELINSKY, A. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, v. 8, n. 6, p. 707–717, 1992.

ANEXO A – Ferramentas de Linha de Comando do ROS

Este anexo tem por finalidade apresentar as ferramentas que se aplicam ao conceitos abordados no Capítulo 4. Onde serão mostradas as ferramentas e seus respectivos exemplos de uso, de acordo com os conceitos já abordados. Para obter maiores informações sobre o comando, basta digitar a seguinte linha de comando: `nome_ferramenta --help`.

rospack- esta ferramenta permite inspecionar um pacote, podendo obter a localização, as dependências diretas ou ainda indiretas.

Exemplos:

```
$ rospack find [package]
$ rospack depends [package]
```

roscreate-pkg- esta ferramenta gera um pacote ROS, especificando suas dependências.

Exemplo:

```
$ roscreate-pkg <package-name> [dependencies]
```

rosmake- esta ferramenta constroe um pacote ROS e suas dependências. Sendo necessário a inserção de linhas de código no arquivo *CMakeLists.txt*, como por exemplo: `gensrv()`, responsável por construir os serviços; `genmsg()`, responsável por construir mensagens; `rosbuild_add_executable(example examples/example.cpp)`, responsável por construir nós executáveis.

Exemplo:

```
$ rosmake [package]
```

rosdep- esta ferramenta instala as dependências de um pacote ROS.

Exemplo:

```
$ rosdep install [package]
```

rxdeps- esta ferramenta possibilita a visualização das dependências de um pacote ROS na forma de um grafo.

Exemplo:

`$ rxdeps [options[1]]`

rosstack- esta ferramenta é análogo à `rospack`, a nível de pilha.

Exemplo:

`$ rosstack[2] [options] <command >[stack]`

roscreate-stack- esta ferramenta cria ou atualiza um pilha no diretório corrente.

Exemplo:

`$ roscreate-stack .`

rosmsg/rossrv- esta ferramenta exibe as definições de dados do tipo mensagem/serviço - `msg/srv`. Esta ferramenta possibilita por exemplo imprimir o conteúdo de uma mensagem ou serviço, mostrar e listar mensagens ou serviços que pertencem a um pacote.

Exemplos:

`$ rosmsg/rossrv show` - apresenta os campos de uma mensagem/serviço

`$ rosmsg/rossrv list` - lista todas as mensagens/serviços

`$ rosmsg/rossrv md5` - exibe o *md5sum* de uma mensagem/serviço

`$ rosmsg/rossrv package` - lista todas as mensagens/serviços de um pacote

`$ rosmsg/rossrv packages` - lista todos os pacotes com mensagens/serviços

rosrun- esta ferramenta permite executar um nó de qualquer ponto do sistema, passando como argumento o nome de pacote e o nome do nó a ser executado.

Exemplo:

`$ rosrun package executable`

rostopic- esta ferramenta exibe informações sobre tópicos, incluindo publicadores, subscritores, taxa de publicação e mensagens.

Exemplos:

`$ rostopic bw` - mostra a largura de banda usada pelo tópico.

`$ rostopic echo` - imprime as mensagens na tela.

[1]Para visualizar as opções, basta acessar: http://wiki.ros.org/rxdeps

[2]Para maiores informações, basta acessar: http://wiki.ros.org/Stacks

$ `rostopic hz` - mostra a taxa de publicação usada pelo tópico.

$ `rostopic list` - imprime as informações sobre tópicos ativos.

$ `rostopic pub` - publica dados para um tópico.

$ `rostopic type` - imprime o tipo de tópico.

$ `rostopic find` - localiza tópicos por tipo.

rosservice- esta ferramenta lista e consulta serviços no ROS.

Exemplos:

$ `rosservice list` - mostra a informação sobre um serviço ativo.

$ `rosservice node` - imprime o nome do nó que presta o serviço.

$ `rosservice call` - chama um serviço dado o argumento.

$ `rosservice args` - lista os argumentos de um serviço.

$ `rosservice type` - imprime o tipo de serviço.

$ `rosservice uri` - imprime o serviço ROSRPC uri.

$ `rosservice find` - localiza serviços.

rosparam- esta ferramenta usa e configura parâmetros no servidor de parâmetros através de arquivos YAML.

Exemplo:

$ `rosparam [options]`[3]

rosnode- esta ferramenta exibe informações sobre nós, incluindo publicações, subscrições e conexões.

Exemplo:

$ `rosnode [options]`[4]

roscore- esta ferramenta é uma coleção de nós e programas que são pré-requisitos para execução de um sistema base do ROS. É necessário que um `roscore` esteja rodando para que os nós possam se comunicar. Normalmente, o `roscore` ao ser executado contempla o mestre, o servidor de parâmetros e `rosout`, sendo este último um nó de saída padrão do ROS, o qual reporta erros e alertas.

Exemplo:

$ `roscore`

[3]Para visualizar as opções, acessar: http://wiki.ros.org/rosparam

[4]Para visualizar as opções, acessar: http://wiki.ros.org/Nodes

roslaunch- esta ferramenta inicia nós localmente e/ou remotamente via SSH, além de configurar parâmetros no servidor de parâmetros.

Exemplo:

`$ roslaunch package filename.launch`

rosbag- esta ferramenta grava e reproduz os dados de tópicos.

Exemplos:

`$ rosbag record -a` - grava todos os tópicos.

`$ rosbag record topic1 topic2` - grava tópicos específicos.

`$ rosbag play -i demo log.bag` - reproduz as mensagens sem esperar.

`$ rosbag play demo1.bag demo2.bag` - reproduz arquivos de uma só vez.

rxgraph / rqt_graph- estas ferramentas exibem um grafo dos nós em execução, bem como os tópicos conectados a eles. A diferença entre as duas é que a segunda é uma ferramenta mais atual.

Exemplos:

`$ rxgraph`

ou

`$ rosrun rqt_graph rqt_graph`

rxplot- esta ferramenta plota os dados de um ou mais tópicos.

Exemplos:

`$ rxplot /topic1/field1 /topic2/field2` - representa graficamente os dados em ambientes diferentes.

`$ rxplot /topic1/field1,/topic2/field2` - representa graficamente os dados no mesmo ambientes.

`$ rxplot /topic1/field1:field2:field3` - representa graficamente múltiplos campos de uma mensagem.

rxbag- esta ferramenta permite visualizar, inspecionar e reexecutar históricos de mensagens, salvos em arquivos `.bag`.

Exemplo:

`$ rxbag bag file.bag`

rxconsole- esta ferramenta permite visualizar e filtra mensagens publicadas em `rosout`.

Exemplo:

```
$ rxconsole
```

roswtf- esta ferramenta exibe erros e avisos de um sistema ROS em execução.

Exemplo:

```
$ roswt ou roswtf [file]
```

roscd- esta ferramenta auxilia na alternância entre diretórios, através de uma busca personalizada, de modo que somente o nome do pacote é requerido, sem a necessidade de um caminho completo.

Exemplo:

```
$ roscd [package[/subdir]]
```

rosls- esta ferramenta lista as informações de um pacote ou pilha a partir do argumento passado, sem a necessidade de um caminho completo.

Exemplo:

```
$ rosls [package[/subdir]]
```

ANEXO B – Programas dos Nós do ROS

B.1 Código do Nó - Find_Robot

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "robot/robot.h"
#include <sys/time.h>
#include "calibrated_robots/calibrated_robots.h"
#include "tracking/tracking.h"
#include "visualization_msgs/InteractiveMarkerPose.h"
#include <iostream>
#include <sstream>
#include <StaticFrameDifferenceBGS.h>
#include "geometry_msgs/Twist.h"
#include <time.h>

std::string s_l;
std::stringstream out_l;

namespace enc = sensor_msgs::image_encodings;

static const char WINDOW[] = "Image window";

image_transport::Publisher image_pub_;

//---------------
cv::Mat img_mask = cv::imread("background.jpg");
cv::Mat img_background = cv::imread("background.jpg");
IBGS *bgs;
//--------------
```

```
int dif2;
int xr1 = 0, yr1 = 0;
int xr2 = 0, yr2 = 0;
int xRecebido;
int yRecebido;

FILE * pfile;

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg, enc::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }

    bgs->process(cv_ptr->image, img_mask, img_background);

    cv::Mat result;

    cv_ptr->image.copyTo(result,img_mask);

    IplImage copy = result;
    IplImage* newCopy = &copy;

    int minT[] = {50, 15, 80};
    int maxT[] = {140,120,150};
    int minR[] = {240,190,5};
    int maxR[] = {255, 240,20};
    // --------------------------------

        cv::Mat resultG, resultFilt, resultHSV;

        cv::cvtColor(result,resultG,CV_BGR2GRAY);
        cv::cvtColor(result,resultHSV,CV_BGR2HSV);

        cv::GaussianBlur(resultG,resultFilt,cv::Size(3,3),4);
```

```
std::vector<cv::Vec3f> circles;

cv::HoughCircles(resultFilt,circles,CV_HOUGH_GRADIENT,2,11,100,20,6,10);

std::vector<cv::Vec3f>::const_iterator itc = circles.begin();

int achouT = 0;
int achouR = 0;

while (itc!=circles.end()) {

cv::Scalar Media = cv::mean(resultHSV(cv::Rect((int)((*itc)[0]-3),(int)((*itc
    )[1]-3),6, 6)));

if((Media[0] > minT[2]) && (Media[0] < maxT[2])){
     if ((Media[1] > minT[1]) && (Media[1] < maxT[1])){
          if ((Media[2] > minT[0]) && (Media[2] < maxT[0])){
               xr1 = (*itc)[0];
               yr1 = (*itc)[1];
               achouT++;
          }
     }
}

if((Media[0] > minR[2]) && (Media[0] < maxR[2])){
     if ((Media[1] > minR[1]) && (Media[1] < maxR[1])){
          if ((Media[2] > minR[0]) && (Media[2] < maxR[0])){
               xr2 = (*itc)[0];
               yr2 = (*itc)[1];
               achouR++;
          }
     }
}
++itc;

}

robot[0].pos[0] = (xr1 + xr2)/2;
robot[0].pos[1] = (yr1 + yr2)/2;
robot[0].pos[2] = -(180*(atan2f(-(yr1-yr2),xr1-xr2))/3.14 - 45);

if(robot[0].pos[2] < -180)
```

```
        {
            robot[0].pos[2] = robot[0].pos[2]+360;
        }
        else if(robot[0].pos[2] > 180)
        {
            robot[0].pos[2] = robot[0].pos[2]-360;
        }

        if ((achouT == 1) && (achouR == 1)){
             robot[0].foundFlag=1;
        }else{
             robot[0].foundFlag=0;
        }
        printf("Angulo: %f",robot[0].pos[2]);

        cv::circle(result, cv::Point(xr1, yr1), 8, cv::Scalar(255,0, 0), -1);
        cv::circle(result, cv::Point(xr2, yr2), 8, cv::Scalar(0, 255, 0), -1);
        cv::circle(result, cv::Point((xr1 + xr2)/2.0, (yr1 + yr2)/2.0), 5 , cv::
            Scalar(0, 0, 255), -1);
    // --------------------------------

    cv::imshow(WINDOW, cv_ptr->image);
    cv::imshow("background", result);
    cv::waitKey(3);
}

int main(int argc, char** argv)
{
    pfile = fopen ("/home/elias/Dropbox/Projeto Controle Robo/dados/20junho/
        TesteSincronismoImage.m", "w");
    fprintf(pfile,"Image = [ ");

    //---------
    bgs = new StaticFrameDifferenceBGS;
    //---------

    struct timeval tvBegin, tvEndDelay, tAtual;
    gettimeofday(&tvBegin, NULL);

    ros::init(argc, argv, "Find_Robot");
    setbot();
    ros::NodeHandle nh;
    image_transport::ImageTransport it(nh);
```

```
cv::namedWindow(WINDOW, CV_WINDOW_AUTOSIZE);
image_transport::Subscriber image_sub_ = it.subscribe("camera/image_rect_color",
    1000, imageCallback);
cv::destroyWindow(WINDOW);
image_pub_ = it.advertise("camera/Find_Robot", 1);
ros::Publisher chatter_pub = nh.advertise<visualization_msgs::
    InteractiveMarkerPose>("msg_Camera", 1000);
ros::Rate loop_rate(30); //taxa de publicacao
visualization_msgs::InteractiveMarkerPose msg_Camera;

out_l << 0;
s_l = out_l.str();

int cont = 0;
while(ros::ok())
{
     gettimeofday(&tvBegin, NULL); // Para saber o tempo
     msg_Camera.header.frame_id = s_l ;
     msg_Camera.header.stamp.nsec = dif2 * 1000;
     printf("Tempo: %d\n",msg_Camera.header.stamp.nsec);
     msg_Camera.pose.position.x = robot[0].pos[0];//*realConstant;
     msg_Camera.pose.position.y = robot[0].pos[1];//*realConstant;
     msg_Camera.pose.orientation.z = robot[0].pos[2];
     msg_Camera.pose.orientation.w = robot[0].foundFlag;
     msg_Camera.pose.position.z = cont;
     gettimeofday(&tvEndDelay, NULL); // Para saber o tempo
     dif2 = (tvEndDelay.tv_usec + 1000000 * tvEndDelay.tv_sec) - (tvBegin.tv_usec
         + 1000000 * tvBegin.tv_sec);
     chatter_pub.publish(msg_Camera);
     gettimeofday(&tAtual, NULL);
     fprintf(pfile,"%d %d %d %d\n",(int)robot[0].pos[0],(int)robot[0].pos[1],cont
         ,(tAtual.tv_usec + 1000000 * tAtual.tv_sec));
     ros::spinOnce();

     loop_rate.sleep();
     cont++;
}

fprintf(pfile,"];");

fclose (pfile);
ros::spin();
return 0;
```

```
}
```

./code/Find_Robot.cpp

B.2 Código do Nó - Robot

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_msgs/Float32MultiArray.h"
#include "sensor_msgs/Joy.h"
#include "sensor_msgs/LaserScan.h"
#include "geometry_msgs/Twist.h"
#include "stdio.h"
#include "visualization_msgs/InteractiveMarkerPose.h"
#include <iostream>
#include <string>
#include <time.h>
#include <math.h>
#include <sstream>
#include <stdint.h>

using namespace std;
stringstream ss;
string s;

#include <ControlePosicao/ControlePosicao.h>
#include <ManipuladorTrajetoria/ManipuladorTrajetoria.h>

#include <iostream>
using std::cout;
using std::endl;
using std::ios;

#include <iomanip>
using std::setw;

#include <fstream>
using std::ofstream;
using std::ifstream;

#include <cmath>
using std::sin;
using std::cos;
```

```
#include <algorithm>
using std::min;

#define PI 3.141592

// ====================================
// Valores
// ====================================

double dt = 0.03;  // periodo de amostragem da posicao
int nWaypoints = 0;  // numero de waypoints para interpolacao
double tempoParaComutacao = 30 * dt; // tempo para comutacao entre waypoints
double posicaoRobo[3];

double kp[] = {0.005, 0.25};  // Ganho proporcional para o controlador linear e
    angular
double ki[] = {0.0, 0.1};  // Ganho integral para o controlador linear e angular
double kd[] = {0.0, 0.0};  // Ganho derivativo para o controlador linear e angular
double satIntegral[] = {0.2, 0.2}; // saturador do integrador
double limiteVelocidade = 0.25;  // Limite da velocidade linear para o robo
double limiteAngular = 10;
double limiteErroLinear = 15;
double waypoint[2];  // waypoint considerado em determinado instante
double acaoLinear;  // valor de velocidade linear enviado para o robo
double acaoAngular;  // valor de velocidade angular enviado para o robo
double tempoLoop = 0;
double xtraj = 0;
double ytraj = 0;
// ================================================================================
// ================================================================================

float leitura[6];
float encoder[4];

void chatterCallback(const visualization_msgs::InteractiveMarkerPose msg_Camera)
{
    leitura[0] = msg_Camera.pose.position.x;
    leitura[1] = msg_Camera.pose.position.y;
    leitura[2] = msg_Camera.pose.orientation.z;
    leitura[3] = msg_Camera.pose.orientation.w;
    leitura[4] = (float)msg_Camera.header.stamp.nsec;
    leitura[5] = (int)msg_Camera.pose.position.z;
}
```

```
void chatterCallback_Encoder(const std_msgs::Float32MultiArray::ConstPtr& msg2)
{
     encoder[0] = msg2->data[0];
     encoder[1] = msg2->data[1];
     encoder[2] = msg2->data[2];
     encoder[3] = msg2->data[3];
}
// =====================================

#include <time.h>
#include <string>

#define LOGNAME_FORMAT "%Y%m%d_%H%M%S"
#define LOGNAME_SIZE 20

std::string get_date(void)
{
    static char name[LOGNAME_SIZE];
    time_t now = time(0);
    strftime(name, sizeof(name), LOGNAME_FORMAT, localtime(&now));
    return name;
}

int main(int argc, char **argv)
{
     struct timeval tAtual;

     //Localizacao e nome do arquivo de dados
     //--------------------
     string nome("/home/elias/Dropbox/Projeto Controle Robo/dados/20junho/T");
     string nome2(".m");
     float teste = time(0) - (1.5783 * pow(10.0,6));
     ss << teste;
     s = ss.str();
     nome += get_date() + nome2;
     FILE * pfile;
     pfile = fopen (nome.c_str(), "w");
     fprintf(pfile,"M = [ ");
     //------------------

     FILE * fileSinc;
```

```
    fileSinc = fopen ("/home/elias/Dropbox/Projeto Controle Robo/dados/20junho/
        TesteSincRobot.m", "w");
    fprintf(fileSinc,"Robot = [ ");

// ================================================================================
// ================================================================================

    ros::init(argc, argv, "Robot");

    ros::NodeHandle n;

    ros::Subscriber sub = n.subscribe("/msg_Camera", 1000, chatterCallback);

    ros::Subscriber sub_Encoder = n.subscribe("/msg_EncoderRobo1", 1000,
        chatterCallback_Encoder);

    ros::Publisher chatter_pub = n.advertise<geometry_msgs::Twist>("/msgRobo1", 1000);

    ros::Rate loop_rate(90);

    while ((leitura[3] == 0)&&(ros::ok()))
    {
        ros::spinOnce();

        loop_rate.sleep();
        printf("leitura[3] = %f\n",leitura[3]);
    }
    posicaoRobo[0] = leitura[0];
    posicaoRobo[1] = leitura[1];
    posicaoRobo[2] = (leitura[2] * PI)/180.0;

// ================================================================================
// ================================================================================
    // Instancia manipulador da trajetoria
    // e atribui valores pertinentes
    // ----------------------------------------------
    ManipuladorTrajetoria manip;
    manip.setLimErroLinear(limiteErroLinear);
    manip.setLimTempoTroca(tempoParaComutacao);
    manip.trajeto.setNumeroPontosEntreRef(nWaypoints);
    // ----------------------------------------------

    // Necessita de ter as referencias da trajetoria passadas
```

```
    manip.trajeto.addReferencia(posicaoRobo[0],posicaoRobo[1]);
    ifstream pontosTraj("/home/elias/Dropbox/Projeto Controle Robo/dados/trajetoria/
        path.traj",ios::in);

    while (pontosTraj >> xtraj >> ytraj)
    {
        manip.trajeto.addReferencia(xtraj,ytraj);
        printf("x: %f, y: %f\n",xtraj,ytraj);
    }
    printf ("saiu do criando trajetoria \n");
    manip.trajeto.interpola();

    // Instancia controlador de posicao para o robo 1, e inicializa valores
        pertinentes
    // ---------------------------------------------
    printf ("1 \n");
    ControlePosicao Robo1(1,dt);
    printf ("2 \n");
    Robo1.setGanhosLinear(kp[0],ki[0],kd[0]);
    printf ("3 \n");
    Robo1.setGanhosAngular(kp[1],ki[1],kd[1]);
    printf ("4 \n");
    Robo1.atualizaPosicaoRobo(posicaoRobo);
    printf("x: %f, y: %f\n",posicaoRobo[0],posicaoRobo[1]);
    Robo1.setLimiteVelocidade(limiteVelocidade);
    // ----------------------------------------------
    printf ("6 \n");
    manip.trajeto.getWaypoint(waypoint,0);
    printf ("7 \n");
    manip.setTempoPassado(0);
    printf ("8 \n");
// ================================================================================
// ================================================================================
    int cont = 0;

    while ((ros::ok())&&(!(manip.fimTrajetoria())))
    {
        printf("dado novo: %d\n", (int)leitura[5]);
        if (leitura[5] != 0){
        // ==========================================

        posicaoRobo[0] = leitura[0];
```

```
posicaoRobo[1] = leitura[1];
posicaoRobo[2] = (leitura[2] * PI)/180.0;

tempoLoop = static_cast<double>(((int)(leitura[4]))/1000000.0);

// ==========================================

gettimeofday(&tAtual, NULL);

fprintf(fileSinc,"%d %d %d %d\n",(int)leitura[0],(int)leitura[1],(int)leitura
    [5],(tAtual.tv_usec + 1000000 * tAtual.tv_sec));

if (leitura[3] == 1)
{
    Robo1.atualizaPosicaoRobo(posicaoRobo); // Fornece para o controlador a
        posicao atual do robo

    manip.setTempoPassado((manip.getTempoPassado() + 0));
    manip.setErroLinear(Robo1.getErroLinear());
    manip.trocaWaypointTempo(waypoint); // verifica se deve ocorrer troca de
        waypoint
    printf("xW = %f , yW = %f \n",waypoint[0],waypoint[1]);
    printf("xR = %f , yR = %f \n",posicaoRobo[0],posicaoRobo[1]);

    Robo1.carregaNovoWaypoint(waypoint); // waypoint e passado para
        referencia no controlador de posicao
    manip.setErroLinear(Robo1.getErroLinear()); // manipulador recebe o erro
        de posicao linear atual
    Robo1.acaoControle(&acaoLinear,&acaoAngular); // calcula acao de controle
         para velocidade linear e angular
    acaoAngular = (acaoAngular > 0)? min(acaoAngular,limiteAngular) : -min(-
        acaoAngular,limiteAngular);
    printf("erro linear = %f\n",Robo1.getErroLinear());

    if (manip.fimTrajetoria()){
        acaoLinear = 0.0;
        acaoAngular = 0.0;
        printf("\n\n\nfim trajetoria\n\n\n");
        //break;
    }
    cont = 0;
} else
{
```

```
            if (cont > 5)
            {
                acaoLinear = 0.0;
                acaoAngular = 0.0;
            }
            cont++;
        }
// ==========================================================================
// ==========================================================================

        geometry_msgs::Twist msgRobot;

        msgRobot.linear.x = acaoLinear;
        msgRobot.linear.y = 0;
        msgRobot.linear.z = 0;
        msgRobot.angular.x = 0;
        msgRobot.angular.y = 0;
        msgRobot.angular.z = -acaoAngular;

        printf("acao Linear: %f\n", acaoLinear);
        printf("acao Angular: %f\n", acaoAngular);

        chatter_pub.publish(msgRobot);

        for (int i=0;i<3;i++)
        {
            fprintf(pfile,"%6.3f ", posicaoRobo[i]);
        }
        fprintf(pfile,"%6.3f ", acaoLinear);
        fprintf(pfile,"%6.3f ", acaoAngular);
        for (int i=0;i<4;i++)
        {
            fprintf(pfile,"%10.4f ", encoder[i]);
        }
        fprintf(pfile,"\n");

        leitura[5] = 0;

        }
        ros::spinOnce();

        loop_rate.sleep();
    }
```

```
    printf("saiu do while");
    geometry_msgs::Twist msgRobot;

    msgRobot.linear.x = 0;
    msgRobot.linear.y = 0;
    msgRobot.linear.z = 0;
    msgRobot.angular.x = 0;
    msgRobot.angular.y = 0;
    msgRobot.angular.z = 0;
    chatter_pub.publish(msgRobot);

    fprintf(pfile,"];");

    fclose (pfile);

    fprintf(fileSinc,"];");

    fclose (fileSinc);

    ros::spin();

    return 0;
}
```

./code/Robot.cpp

B.3 Código do Nó - serial_node

```
// Projeto desenvolvido por:
// Ana Sophia Cavalcanti Alves Vilas Boas
// Elias Ramos Vilas Boas
// Leandro Luiz Rezende de Oliveira
// Lucas Correa Netto Machado
// Trabalho de Controle 16-05-2013

//Programa de acionamento e controle de baixo nivel do Robo
#include <ros.h> //Biblioteca Ros, permite que o programa comunique com Ros
#include <std_msgs/Float32MultiArray.h> //Biblioteca da mensagem de output
#include <geometry_msgs/Twist.h> //Biblioteca da mensagem de input
#include <PololuWheelEncoders.h> //Biblioteca do encoder pololu

//declaracao das variaveis
const int PWMA = 5;  //Pino de PWM da roda A
```

```
const int AIN1 = 6;  //Pino de Sentido da roda A
const int AIN2 = 7;  //Pino de Sentido da roda A esquerda
const int STBY = 8;  //Pino que habilita ponte H
const int BIN1 = 9;  //Pino de Sentido da roda B
const int BIN2 = 10;  //Pino de Sentido da roda B
const int PWMB = 11;  //Pino de pwm da roda B direita
const float pi = 3.141592; // Constante numerica
const float raio = 0.0195; // Valor do raio da roda do robo (metros)
float velLinear = 0; // Velocidade linear do robo
float velAngular = 0; // Velocidade angular do robo
float distRodas = 0.06; //distancia entre as rodas do robo (metros)
float velReferencia[] = {0, 0}; // Velocidade linear de referencia {roda direita, roda
     esquerda};
int MT[] = {0, 0}; // Comando enviado para motor (entre 0 e 255) {roda direita, roda
    esquerda};
int maxMotor = 255; // Limite maximo do comando para o motor (255)
int minMotor = 30; // Limite minimo do comando para o motor (255)

float constAngular = 2 * pi; // Constante Angular - corrige o valor do joystick (
    joystick varia entre 0 e 1.0)
//-----------------------------
// controlador de velocidade
// qualquer vetor = {roda direita, roda esquerda};

float controle[] = {0, 0}; //Acao de controle {roda direita, roda esquerda};
int leituraEncoder[] = {0, 0}; //Leitura dos Encoders {roda direita, roda esquerda};
float velRodas[] = {0, 0}; //Velocidade Linear das rodas {roda direita, roda esquerda
    };
float erro[] = {0, 0};  //Sinal de erro {roda direita, roda esquerda};
float integral[] = {0, 0}; //Parcela do bloco integrador {roda direita, roda esquerda
    };
float proporcional[] = {0, 0}; //Parcela do bloco proporcional {roda direita, roda
    esquerda};
float derivativo[] = {0, 0};  //Parcela do bloco derivativo {roda direita, roda
    esquerda};
float kp[] = {18.2, 18.2}; //Constante do bloco integrador {roda direita, roda
    esquerda};
float ki[] = {3639.4, 3639.4}; //Constante do bloco proporcional {roda direita, roda
    esquerda};
float kd[] = {0, 0}; //Constante do bloco derivativo {roda direita, roda esquerda};
float lastErro[] = {0, 0}; //Ultimo sinal de erro {roda direita, roda esquerda};
float dt = 0.01;   //Delta tempo - periodo de amostragem;
long time = 0;    // Tempo de comparacao;
```

```
long time2 = 0;
float satVel = 0.7;  // valor de saturacao da velocidade;
float satVel2 = 0.7;  // valor limite da velocidade; que posteriormente deve ser
    ajustado para 1.0
float minVel = 0.07;  //Valor minimo de Velocidade;
//-----------------------------
PololuWheelEncoders encoders;

ros::NodeHandle nh;   //Instanciando o manipulador do no do sistema ros
std_msgs::Float32MultiArray msg_EncoderRobo1; //Instanciando a variavel de mensagem de
     dados do encoder

// Funcao que recebe a mensagem de velocidade angular e linear
void chatterRobo(const geometry_msgs::Twist& msgRobo1)
{
    velLinear = msgRobo1.linear.x; // velocidade linear
    velAngular = msgRobo1.angular.z; // velocidade angular
}// fim da funcao chatterRobo

// Define o tipo de mensagem a ser recebido pela funcao chatterRobo
ros::Subscriber<geometry_msgs::Twist> sub("msgRobo1", &chatterRobo);
// Define a mensagem a ser publicada para o no externo
ros::Publisher chatter("msg_EncoderRobo1", &msg_EncoderRobo1);

//configuracoes iniciais
void setup()
{
    pinMode(PWMA,OUTPUT);// Habilitando pino como saida
    pinMode(AIN1,OUTPUT);// Habilitando pino como saida
    pinMode(AIN2,OUTPUT);// Habilitando pino como saida
    pinMode(STBY,OUTPUT);// Habilitando pino como saida
    pinMode(BIN1,OUTPUT);// Habilitando pino como saida
    pinMode(BIN2,OUTPUT);// Habilitando pino como saida
    pinMode(PWMB,OUTPUT);// Habilitando pino como saida
    encoders.init(16,17,18,19); // inicializa o encoder
    nh.initNode(); // inicializa o manipulador

    //---------------------------------------------------------
    //configurando a variavel mensagem de saida
    msg_EncoderRobo1.layout.dim = (std_msgs::MultiArrayDimension *)malloc(sizeof(
        std_msgs::MultiArrayDimension));
    msg_EncoderRobo1.layout.dim[0].label = "encoder";
    msg_EncoderRobo1.layout.dim[0].size = 4;
```

```
    msg_EncoderRobo1.layout.dim[0].stride = 1*4;
    msg_EncoderRobo1.layout.data_offset = 0;
    msg_EncoderRobo1.data = (float *)malloc(sizeof(float)*4);
    msg_EncoderRobo1.data_length = 4;
    //-------------------------------------------------------
    //configurando a transmissao de mensagem (receber e enviar)
    nh.subscribe(sub);
    nh.advertise(chatter);
    //--------------------------------------------------------
}//fim da funcao setup

void loop()
{
    float aux = 0; //variavel auxiliar
    // transforma velocidade angular e linear do robo em velocidade linear das rodas
        direita e esquerda
    velReferencia[0] = velLinear * satVel2 + (velAngular * satVel2 * constAngular ) *
        distRodas / 2;
    velReferencia[1] = velLinear * satVel2 - (velAngular * satVel2 * constAngular ) *
        distRodas / 2;
    //---------------------------------------------------
    // bloco que garante que a velocidade de referencia seja menor que 0.7 e maior que
         -0.7
    if ((velReferencia[0] - satVel) * (velReferencia[1] - satVel) < 0)
    {// inicio do if que garante velocidade menor que 1.0
        aux = max((velReferencia[0] - satVel),(velReferencia[1] - satVel));
        velReferencia[0] = velReferencia[0] - aux;
        velReferencia[1] = velReferencia[1] - aux;
    }// fim do if que garante velocidade menor que 1.0
    else if ((velReferencia[0] + satVel) * (velReferencia[1] + satVel) < 0)
    {// inicio do if que garante velocidade maior que -1.0
        aux = min((velReferencia[0] + satVel),(velReferencia[1] + satVel));
        velReferencia[0] = velReferencia[0] - aux;
        velReferencia[1] = velReferencia[1] - aux;
    }// fim do if que garante velocidade maior que -1.0

    //--------------------------------------------------
    //Bloco da acao de controle que se repete a cada dt segundos
    if (micros() - time >= dt*1000000)
    {// inicio do if do bloco da acao de controle
        time = micros();
        // Leitura do encoder
        leituraEncoder[0] = - encoders.getCountsAndResetM2();//Roda direita
```

```
        //A leitura do encoder esta no sentido contrario,
        leituraEncoder[1] = - encoders.getCountsAndResetM1();//Roda esquerda
        //devido a isto existe o sinal de menos (-) na frente da funcao;
        // --
        // Acao de controle para cada roda
        for (int i = 0; i < 2; i++)
        {//inicio do for para cada roda
            //----
            velRodas[i] = (float)leituraEncoder[i] * 2 * pi * raio / (48 * dt); //
                transforma a leitura do encoder em velocidade linear
            //----
            erro[i] = velReferencia[i] - velRodas[i];//sinal de erro para a entrada
                do controlador
            //----
            proporcional[i] = erro[i] * kp[i];//acao proporcional
            //----
            integral[i] += ki[i] * erro[i] * dt;//acao integral
            if (integral[i] > maxMotor) integral[i] = maxMotor;
            else if (integral[i] < -maxMotor) integral[i] = -maxMotor;
                controle[i] = proporcional[i] + integral[i] + derivativo[i];//saida
                    do controlador
            //----
            msg_EncoderRobo1.data[2 * i] = velRodas[i]; //montando a mensagem a ser
                publicada
            msg_EncoderRobo1.data[2 * i + 1] = velReferencia[i];//montando a mensagem
                 a ser publicada
        }//fim do for para cada roda
    }// fim do if do bloco da acao de controle
    //---------------------------------------------------

    if ((micros() - time2 >= dt*5000000))
    {
        time2 = micros();
        chatter.publish(&msg_EncoderRobo1); //publicando a mensagem
    }
    //sinal de saida para motores das rodas

    //------
    //sinal oriundo do controlador
    if(abs(velReferencia[0]) < minVel)
    {
        controle[0] = 0;
        integral[0] = 0;
```

```
    }
    if(abs(velReferencia[1]) < minVel)
    {
        controle[1] = 0;
        integral[1] = 0;
    }
    MT[0] = (int)(abs(controle[0]) > maxMotor ? maxMotor : abs(controle[0]));
    MT[1] = (int)(abs(controle[1]) > maxMotor ? maxMotor : abs(controle[1]));
    //------
    //sinal oriundo sem acao do controlador / passagem direta do valor PWM
    analogWrite(PWMB,MT[0]); //funcao que aciona os motores
    analogWrite(PWMA,MT[1]); //funcao que aciona os motores
    //------
    digitalWrite(STBY,HIGH); //pino STBY recebe valor alto
    //------
    //------------------------------------------------
    //Bloco que defini o sentido de rotacao das rodas
    if(controle[0] > 0 )// && abs(velReferencia[0]) > minVel)
    {
        digitalWrite(BIN1,HIGH);
        digitalWrite(BIN2,LOW);
    }
    else if (controle[0] < 0 )// && abs(velReferencia[0]) > minVel)
    {
        digitalWrite(BIN1,LOW);
        digitalWrite(BIN2,HIGH);
    }

    if(controle[1] > 0 )// && abs(velReferencia[1]) > minVel)
    {
        digitalWrite(AIN1,LOW);
        digitalWrite(AIN2,HIGH);
    }
    else if (controle[1] < 0 )// && abs(velReferencia[1]) > minVel)
    {
        digitalWrite(AIN1,HIGH);
        digitalWrite(AIN2,LOW);
    }
    //------------------------------------------------
    nh.spinOnce(); //funcao necessaria para publicar e subscrever
}//fim da funcao loop
```

./code/serial–node.cpp