LOCOMOÇÃO DE UM ROBÔ MÓVEL COM ESTEIRAS EM ESCADAS

Tiago Pereira Azevedo

Projeto de Graduação apresentado ao Curso de Engenharia de Controle e Automação, da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro de Controle e Automação.

Orientador: Fernando Cesar Lizarralde

Rio de Janeiro
Fevereiro de 2017

LOCOMOÇÃO DE UM ROBÔ MÓVEL COM ESTEIRAS EM ESCADAS

Tiago Pereira Azevedo

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE CONTROLE E AUTOMAÇÃO.

Examinado por:

Prof. Fernando Cesar Lizarralde, D.Sc.

Prof. Lilian Kawakami Carvalho, D.Sc.

Eng. Alex Fernandes Neves, M.Sc.

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2017

Azevedo, Tiago Pereira

Locomoção de um robô móvel com esteiras em escadas/Tiago Pereira Azevedo. – Rio de Janeiro: UFRJ/Escola Politécnica, 2017.

XIV, 68 p.: il.; 29,7cm.

Orientador: Fernando Cesar Lizarralde

Projeto de graduação – UFRJ/Escola Politécnica/Curso de Engenharia de Controle e Automação, 2017.

Referências Bibliográficas: p. 46 – 48.

1. Robô móvel com esteiras. 2. Escada ascendente. 3. Locomoção semiautônoma. I. Lizarralde, Fernando Cesar. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Controle e Automação. III. Título.

A minha família, que me deu a
oportunidade de ser

Agradecimentos

Gostaria de agradecer, primeiramente, a Deus, a minha família, pais, irmãos, avós e tios, que dividem o diploma, ideais e fé comigo. A minha namorada, cúmplice da minha vida. E aos amigos pelo suporte e carinho, principalmente à amiga Lili, que divide seus resumos antes das provas.

Agradeço também aos mestres do conhecimento, que não só contribuíram mas moldaram a cultura que levo comigo, principalmente meu orientador Fernando Lizarralde, que foi suporte teórico e prático para o desenvolvimento de uma tarefa que se mostrou muito mais complicada conforme fazíamos.

Resumo do Projeto de Graduação apresentado à POLI/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Controle e Automação.

LOCOMOÇÃO DE UM ROBÔ MÓVEL COM ESTEIRAS EM ESCADAS

Tiago Pereira Azevedo

Fevereiro/2017

Orientador: Fernando Cesar Lizarralde

Curso: Engenharia de Controle e Automação

A crescente utilização de robôs em aplicações de exploração e resgate torna maior a demanda de desenvolvimento de tecnologias que auxiliem a operação dos mesmos. Esse cenário incentiva o desenvolvimento de técnicas semiautônomas e autônomas para a realização de tarefas. Portanto, é apresentado o desenvolvimento de um programa de controle e uma metodologia de locomoção em escadas, capazes de produzir um controle semiautônomo.

O *software*, principal contribuição deste trabalho, foi desenvolvido baseado em ROS, reunindo as informações necessárias à locomoção e ao gerenciamento de sua operação.

A metodologia de locomoção permite que os dados dimensionais da escada sejam estimados, e o operador atue sobre o movimento, sendo auxiliado por um computador na centralização e orientação sobre a escada. Esses dados são extraídos por meio do sensor RGB-D *kinect*, que auxilia também na teleoperação do robô escolhido. Os algoritmos de controle e detecção foram desenvolvidos em Matlab.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Control and Automation Enginner.

LOCOMOTION OF A TRACKED MOBILE ROBOTS ON STAIRS

Tiago Pereira Azevedo

February/2017

Advisor: Fernando Cesar Lizarralde

Course: Control and Automation Engineering

The increase in utilization of robots in urban search and rescue applications makes the demand for developing new tecnologies that helps operating those robots greater. The ability to overcome obstacles, like wreckages and stairs, is in the center of all the problems in operating robotic systems. This cenario encourages the development of semi-autonomous techniques to acomplish such tasks. Therefore, it's developed of a control software and a methodology for locomotion on stairways, capable of producing semiautonomous control.

The developed software, main contribution in this work, was based on ROS, gathering information from locomotion and managing its operation.

The locomotion methodology allows dimensional data estimation, and the operator acts on the movement, being assisted by a computer on centralization and orientation on stairs. The approach used is based on the RGB-D sensor *kinect*, that also helps teleoperation of the chosen robot. The control and detection algorithms were developed using Matlab.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

C_i	centróides de cada degrau, p. 35
D_{image}	*Depth image* fornecida pelo sensor, p. 31
D_s	profundidade da escada, p. 35
E_{image}	*Edge image* resultante de D_{image}, p. 31
F_{Stair}	Indicativo de detecção de escada, p. 31
H_R	altura do robô, p. 38
H_{col}	Histograma de frequência de linhas nas colunas de E_{image}, p. 32
H_s	altura da escada, p. 35
L_{3D}	Lista dos pontos das bordas dos degraus, separados por detecção e por linha, p. 31
L_3	comprimento da perna, p. 38
L_{Hough}	Matriz de votação resultante da SHT, p. 32
L_i	pontos das bordas da escada ordenados por degrau, p. 35
L_{merge}	Linhas encontradas pela SHT, p. 32
L_{proj}	pontos detectados e projetados sobre Pl_{Model}, p. 35
$Lim_{Sup,Inf,Esq,Dir}$	Limites de detecção da escada em E_{image}, p. 32
N_{linhas}	Número de linhas da escada, p. 32
O_L	Sistema de coordenadas do laser, p. 52
O_b	origem do sistema de coordenadas inercial, p. 36
O_k	origem do sistema de coordenadas do *kinect*, p. 36
P_{In}	*Point cloud* fornecido pelo sensor, p. 31

P_{Out}	*Point cloud* dos pontos das bordas dos degraus, p. 31
Pl_{Model}	modelo do plano da escada, p. 34
Pl_{temp}	Plano ajustado em P_{Out}, p. 32
R	Raio maior do braço do DIANE, p. 49
S_{Model}	modelo matemático da escada, p. 34
W_s	largura da escada, p. 35
α_L	Ângulo da varredura do *laser scan*, p. 52
α_a	ângulo entre a esteira e o plano do solo, p. 38
α_{max}	Máxima angulação de transposição de obstáculo, p. 32
α_{temp}	Ângulo entre Pl_{temp} e o plano horizontal, p. 32
$\dot{\varphi}_d$	Velocidade tangencial da esteira direita, p. 10
$\dot{\varphi}_e$	Velocidade tangencial da esteira esquerda, p. 10
$\omega(t)$	Velocidade angular do robô, p. 10
θ_3	ângulo de rotação da junta da perna frontal, p. 38
$\vec{n}_{Pl}$	Vetor diretor de Pl_{temp}, p. 32
$\vec{n}_{horz}$	Vetor diretor do plano horizontal, p. 32
d_L	distância medida do *laser scan*, p. 52
d_s	profundidade média dos degraus, p. 35
h_s	altura do degrau, p. 38
h_s	altura média dos degraus, p. 35
r	Raio menor do braço do DIANE, p. 49
$v(t)$	Velocidade linear do robô, p. 10

Lista de Abreviaturas

3D	três dimensões, p. 3
C++	linguagem de programação orientada a objeto, p. 7
CAN	do inglês, *Controller Area Network*, p. 11
EFK	do inglês, *Extended Kalman Filter*, ou Filtro de Kalman Estendido, p. 6
EOD	Do inglês, *Explosive Ordnance Disposal*, p. 1
RAM	Do inglês, *Random Access Memory*, p. 13
RANSAC	Do inglês, *Random Sample Consensus*, p. 4
RGB-D	do inglês, Red, Green, Blue and Depth, p. 3
ROS	Do inglês, Robot Operating System, p. 6
RPM	rotações por minuto, p. 18
SHT	Do inglês, *Standard Hough Transform*, p. 33
SSD	do inglês, Solid State Drive, p. 51
TCP/IP	do inglês, *Transmission Control Protocol/ Internet Protocol*, p. 12
USB	do inglês, *Universal Serial Bus*, p. 12

Capítulo 1

Introdução

Exploração está no cerne do ser humano. Seja em terra, mar, céu ou até no espaço, a humanidade anseia a exploração de novos horizontes. Entretanto, nem sempre as condições permitem que os homens cheguem ao local desejado, seja pela presença de temperatura e pressões extremas, radiações, baixa luminosidade ou terrenos acidentados ou de difícil acesso.

É nesse intuito que a comunidade científica desenvolve sistemas robóticos capazes de explorar, observar, reconhecer, estudar e manipular objetos, sem a presença do ser humano. Uma classe desses robôs muito utilizada para exploração urbana é a de robôs com esteiras. Devido a maior superfície de contato, simplicidade de movimento e capacidade de movimentação em vários terrenos, esses robôs são a escolha ideal para exploração de terrenos desconhecidos.

Um grupo mais específico desses robôs, denominado *Bomb Disposal/ Explosive Ordnance Disposal* (tipo *EOD*), usado por militares para desarmar bombas (o que dá nome a classificação), apresenta também, em sua maioria, braços (ou pernas) robóticos articulados, que auxiliam a transposição de obstáculos, e manipuladores robóticos, permitindo aos robôs manipular objetos. Exemplos dessa classe podem ser observados na figura 1.1.

Independente da classificação dada aos robôs, a superação de obstáculos; como escombros, escadas, terrenos acidentados; é um desafio comum aos robôs. Esse cenário incentiva diversos estudos e desenvolvimento de metodologias que prometem mudar esse quadro (Li et al. [12], Colas et al. [4] e Wang et al. [20]).

Pode-se separar os tipos de operação para locomoção desses robôs em basicamente três: teleoperado; semiautônomo; e autônomo.

A teleoperação conta exclusivamente com a intervenção humana, capaz de reagir adequadamente às mais diversas e complexas situações. Entretanto, a limitada visão das câmeras e informações de outros sensores, dificulta a operação, além de exigir grande habilidade do operador, que deve ser bem treinado em sua manipulação, para tentar compensar, inclusive, atrasos na comunicação.

A operação semiautônoma ainda é uma operação que precisa da intervenção humana,

(a) Robô desarmando bomba

Fonte: www.upi.com

(b) Robô subindo escada

Fonte: br.pinterest.com/lmorhac/eod-robots/

Figura 1.1: Exemplo de robôs *EOD*

mas possui um computador auxiliando a locomoção. Isso garante ao sistema robótico segurança, precisão de movimento e reduz significativamente o tempo de treinamento do operador.

Já a operação autônoma necessita de uma elevada inteligência computacional, pois deve controlar todos os aspectos do robô sem qualquer comando humano. Esse tipo de operação traz, além dos benefícios citados anteriormente, velocidade de reação, pois além dos computador realizar tarefas mais rápido que um ser humano, retira os atrasos de comunicação. Contudo, oferece uma limitação quanto aos cenários de operação, pois depende da programação das rotinas para identificar as situações e pode não agir adequadamente, caso fuja da operação normal.

1.1 Motivação

Durante uma operação de resgate em um ambiente urbano, é comum deparar-se com uma escada como obstáculo, principalmente no interior de uma construção civil. Nem sempre têm-se certeza da capacidade de transpor algumas escadas, devido à limitação visual que as câmeras apresentam. Evidencia-se, assim, a importância de um método de locomoção eficiente sobre a mesma.

Os robôs do tipo *EOD* foram, ao longo dos anos, se tornando populares na solução do problema de locomoção em ambientes urbanos. Devido às suas características, este tipo de robô garante estabilidade e tração durante o movimento, além da maioria possuir

braços que permitem a reconfiguração de sua postura a fim de superar o obstáculo, como visto na figura 1.1b.

A complexidade e dificuldade da teleoperação em escadas é tal que uma operação incorreta pode levar ao tombamento do robô, colisão com obstáculos laterais (normalmente ocultos à câmera), escorregamento, o que pode significar em danos ao robô ou até mesmo fracasso da missão. Para que isso não aconteça, o operador deve estar extremamente familiarizado com o equipamento, rigorosamente treinado e disponha de uma habilidade e destreza muito acima da média. Assim, um sistema capaz de auxiliar a transposição de uma escada se torna mais que necessário.

1.2 Objetivo

O objetivo deste trabalho é desenvolver um programa, que possibilite o controle da operação semiautônoma de um robô com esteiras. Para isso, foi proposto uma ideologia de *software* que seja capaz de identificar e modelar escadas, baseado na proposta de de Lima [5], e permitir a teleoperação ao usuário.

O programa de controle desenvolvido foi baseado na plataforma ROS, aliado ao ambiente de desenvolvimento Qt, que dispõem de bibliotecas que auxiliam na comunicação entre componentes e gerenciamento do *software*. O mesmo deve ser escalável, segmentado e reutilizável. Este programa deve funcionar por comandos remotos, enviados por um operador, e um controle, proposto por de Lima [5].

Para detecção e modelagem da escada, a metodologia desenvolvida por de Lima [5] foi baseada na proposta em Delmerico et al. [6]. Utilizando sensores RGB-D , como o *Kinect*, extrai-se da *depth image* as linhas da escada. Com a informação de pontos 3D , pode-se encontrar o modelo aproximado da escada.

O controle de locomoção sobre a escada é baseado no rastreamento de trajetórias. Deve ser proposto um controle que corrija a centralização do robô sobre a escada, bem como sua orientação e permita o operador informar a velocidade do movimento.

Assim, o robô deve, ao final do projeto, ser capaz de subir uma escada de forma semiautônoma, com o auxílio do computador no controle de centralização e orientação da escada, e na modelagem do mesmo, a fim de retirar os parâmetros para o controle e na determinação da possibilidade de transposição do obstáculo.

1.3 Revisão de Literatura

Para fazer qualquer sistema robótico transpor uma escada, antes é necessário detectá-la e modelá-la. Para tal, diversos trabalhos foram analisados e foi escolhida uma linha de atuação baseada nas características do robô disponíveis, descritas no capítulo 2.

Detecção e modelagem da escada

A modelagem da escada é essencial para determinar se o robô será capaz de superar o obstáculo, bem como para fornecer dados geométricos, visando o planejamento de posicionamento e controle do sistema robótico. Só após extrair seus dados, o programa é capaz de preparar o sistema robótico para sua transposição.

Foram observados nos principais métodos de percepção e modelagem de escadas três tipos de sensores: câmeras monoculares (como em Carbonara and Guaragnella [3]), sensores *laser scan* (como em Mihankhah et al. [15]) e sensores 3D (como em Luo et al. [14]), como o *Kinect*.

Em Luo et al. [14], um método é desenvolvido baseando-se em conceitos de análise estatística de dados em *point cloud* (nuvem de pontos) 3D e modelos geométricos ajustados. Utilizando os métodos estatísticos, os planos do ambiente são identificados e seus pontos correspondentes são agrupados. Em seguida, a relação geométrica entre pontos de diferentes grupos com espaçamentos iguais é usada para identificar os degraus da escada. Já utilizando a *depth image* (imagem em profundidade), são extraídos os pontos das bordas de cada degrau. Com isso, o modelo proposto pelo autor é ajustado aos planos e limites de cada degrau, permitindo determinar os parâmetros necessários para análise da escada, como altura e profundidade dos degraus. Entretanto, este método não pode ser aplicado a robôs pequenos, pois necessita de uma altura mínima do solo para identificar os planos dos degraus.

Já em Delmerico et al. [6], é desenvolvido um método de detecção e modelagem de escada, utilizando o *kinect*, que analisa as descontinuidades presentes na *depth image* devido aos degraus. Aplicando um detector de bordas *Canny*, um conjunto de linhas paralelas é evidenciado. Em seguida, o algoritmo utiliza uma transformada de Hough para destacar as retas da imagem. Depois de devidamente agrupadas, faz-se uma correspondência com a matriz *point cloud* da mesma imagem, extraindo as informações dos pontos relativos às bordas de cada degrau. Por fim, um plano é ajustado nos pontos das bordas para determinar a inclinação aproximada da escada. A relação geométrica entre os pontos agrupados de degraus distintos informam os outros parâmetros necessários para modelagem da escada.

Em Beno et al. [1], a aplicação de reconstrução de mapa é explorada. Essa prática é muito bem vinda para armazenamento das características do ambiente, principalmente para modelos robóticos autônomos ou semiautônomos. Os autores exploram o método RANSAC (do inglês, *Random Sample Consensus*) para atualizar as informações do ambiente em tempo real. Este algoritmo é capaz de modelar o espaço em sua volta, bem como localizar o *kinect* no ambiente, sem perder em performance, gerando, segundo resultados do autor, uma taxa de amostragem de 30 fps (*frame per second*).

Também passível de uso em tempo real, em Wagner et al. [18], os autores desenvol-

vem um método de reconhecimento de superfícies somente utilizando um mapa 3D, e reconhecendo os padrões de arranjo dos planos. Eles utilizam *point cloud* como massa de dado, mas o algoritmo deveria aceitar qualquer tipo de dado amostral. A grande vantagem da metodologia desenvolvida é a utilização de um único tipo de dado. Outros métodos utilizam associação de 2 ou mais imagens, como *depth image* e *point cloud*. O algoritmo reconhece superfícies e modela cada uma delas. Com a representação da superfície extraída, e o cálculo do centróide da mesma superfície, é possível determinar o padrão de defasagem entre os vetores diretores dos planos. É ele que dá característica ao plano. Entretanto, para gerar um mapa do ambiente em que o sensor *kinect* se encontra, é preferível uma altura relativa ao solo bem elevada. Uma posição mais baixa impossibilita a visão das superfícies tanto de degraus ascendentes quanto descendentes, devido à zona escura do nível, resultante da posição desprivilegiada do sensor. Essa falta de informação não só prejudica o reconhecimento dos padrões, mas também modela erroneamente.

Tentando também resolver o problema da exposição muito intensa à luz solar, Wang et al. [19] apresenta um método desenvolvido para câmeras RGB-D, baseado não na imagem em profundidade, mas na imagem RGB, fornecida pelo *kinect*. A presença de linhas paralelas na imagem, com mesma frequência luminosa, produz a identificação de um obstáculo e seu consequente estudo. A separação de identificação e modelagem, sem variáveis compartilhadas acelera o processamento de dados, pois o segundo recurso só é utilizado quando necessário. Contudo, outros objetos apresentam o padrão de linhas paralelas da escada, como estantes, o que não garante a presença da escada. Para determinar de fato a existência da mesma, é necessário a análise da detecção em profundidade. Além disso, quando o que se quer achar é justamente a escada para modelagem, o retrabalho de uma nova extração de informações se torna dispendiosa. O fato de a detecção ser feita por uma imagem RGB dificulta mais ainda esse retrabalho, já que o próprio erro de paralaxe inerente ao sensor gera uma inutilização dos dados extraídos da imagem RGB, quando se modela.

Controle de locomoção em escadas

Devido à geometria do robô móvel do tipo *EOD* e às características do contato entre a esteira e a escada, a maioria das abordagens de controle são focadas em manter a orientação do robô constante durante o deslocamento e reduzir os desvios de posição em relação ao centro da escada.

Desvios consideráveis podem acarretar em uma colisão com as laterais da escada, escorregamento ou até tombamento do robô. Por isso, deve-se manter o robô dentro de uma faixa de trabalho segura.

Os principais estudos podem ser divididos em estratégias de controle cinemático e controle dinâmico.

Estratégias de controle dinâmico são apresentadas em Helmick et al. [10] e Mourikis et al. [16] propondo algumas soluções ao problema de locomoção sobre escadas. Ambos os trabalhos realizam a fusão de informações do *laser scan*, ou câmera monocular, e giroscópio, por meio de um *Filtro de Kalman Estendido (EFK)*, para estimar a orientação do robô na escada.

Nesses trabalhos, são propostos dois controles separados, um para minimizar o erro de centralização e outro para erro de orientação. Ao desviar-se do centro da escada, o controle de centralização produz um sinal de referência ao controle de orientação, diferente do sinal desejado original, a fim de trazer o sistema para a centralização. Ao aproximar-se da trajetória desejada, o controle de centrlização retorna gradativamente ao sinal original.

Já as estratégias de controle cinemático, em sua maioria, utilizam o modelo do robô móvel com direção diferencial, a fim de representar a cinemática dos robôs com esteiras do tipo *EOD*.

Como em Kalantari et al. [11], o controle cinemático proposto visa também minimizar os erros de orientação e centralização do robô em relação à escada. O controle é baseado em modelo de referência que produz a trajetória desejada a partir das velocidades linear e angular desejadas. Define-se, depois, o erro de centralização e orientação do robô em relação à escada e obtém-se suas dinâmicas. A partir da linearização das dinâmicas dos erros obtidas, o autor calcula a lei de controle e estabiliza o sistema ao redor da origem.

Este controle produz bons resultados, porém, devido à linearização em torno do ponto de equilíbrio, nada se pode afirmar quanto à estabilidade assintótica global do sistema em malha fechada.

Utilização da plataforma ROS

Aplicações robóticas necessitam de um programa para rodar. Uma plataforma disponível utilizada para auxiliar o desenvolvimento dessas aplicações é o *framework* ROS (do inglês, *Robot Operating System*).

O ROS possui características semelhantes a um sistema operacional, como lidar com a execução de processos, troca de mensagens entre estes e um sistema de arquivos. Este permite a programação de suas rotinas nas linguagens C++ e Python, linguagens orientadas à objeto, possibilitando ao programador segmentar seu programa e integrar as diversas rotinas por meio de mensagens, de fácil determinação de protocolo.

Uma boa definição e utilização desta plataforma é encontrada em Loureiro [13]. Além do definido pelo autor, faz-se uso da ferramenta de inicialização de processos do ROS, o *launch*. Essa ferramenta permite configurar um arquivo XML (*eXtensible Markup Language*) para execução de processos específicos com configurações predefinidas. Mais informações sobre as ferramentas do ROS podem ser encontradas em seu sítio (wiki.ros.org).

1.4 Ferramentas utilizadas

Este trabalho consiste na aplicação dos métodos propostos em um robô específico. Sendo assim, foram adotadas algumas ferramentas difundidas no mercado para possibilitar o êxito da missão.

Primeiramente, o robô escolhido para realizar este projeto se chama DIANE. É um robô do tipo *EOD*, como pode ser observado na figura 1.2.

Figura 1.2: Robô DIANE

Este robô, mostrado na figura 1.2, tem quatro motores atuando em duas esteiras e dois braços. Um motor para o conjunto de braço frontal, outro para o conjunto traseiro, um para a esteira direita e um último para a esteira esquerda. O movimento do robô se dá pela associação do movimento das duas esteiras. Mais informações a respeito das configurações do robô, de suas dimensões, parâmetros de motores, entre outros, podem ser encontrados no apêndice A. Contudo, as características necessárias para este trabalho serão descritas no capítulo 2.

Todas as funções básicas de atuação e sensoriamento foram desenvolvidas em C++. A interpretação e funções básicas do robô foram desenvolvidas nessa linguagem, bem como uma interface gráfica para concentrar e facilitar a operação do robô. Porém, pela facilidade de escrita, robustez, simples integração e já comprovada eficiência em controle, foi escolhido o Matlab para o desenvolvimento da locomoção na escada.

1.5 Estrutura do texto

Este trabalho é dividido em cinco capítulos.

No capítulo 2, o robô utilizado é apresentado e descrito. Suas características fundamentais ao desenvolvimento do *software*, detecção da escada e controle são discutidas para que se tenha base para todo o desenvolvimento do trabalho.

No capítulo 3, é apresentado o programa desenvolvido para controle do robô. Sua ideologia apresentada e as principais funcionalidades descritas.

A locomoção em escadas é discutida no capítulo 4. Nele, a metodologia proposta por [5] é discutida, com algumas modificações destacadas. O método de detecção e modela-

gem da escada, bem como a metodologia para locomoção e o controle sobre a mesma são abordados da forma como se idealizou implementar.

Já no último capítulo, o 5, as conclusões acerca do trabalho, bem como trabalhos em desenvolvimento e futuros são mostrados.

Capítulo 2

Descrição do robô com esteiras DIANE

O DIANE é um robô móvel com esteiras, do tipo EOD, possuindo duas esteiras laterais, utilizadas para seu deslocamento, dois pares de braços (ou pernas) com esteiras que se movimentam na mesma velocidade tangencial das esteiras laterais, usado para transposição de obstáculos, e um manipulador robótico de três graus de liberdade, que serve para manejar objetos, como bombas. A figura 2.1 mostra o projeto deste robô.

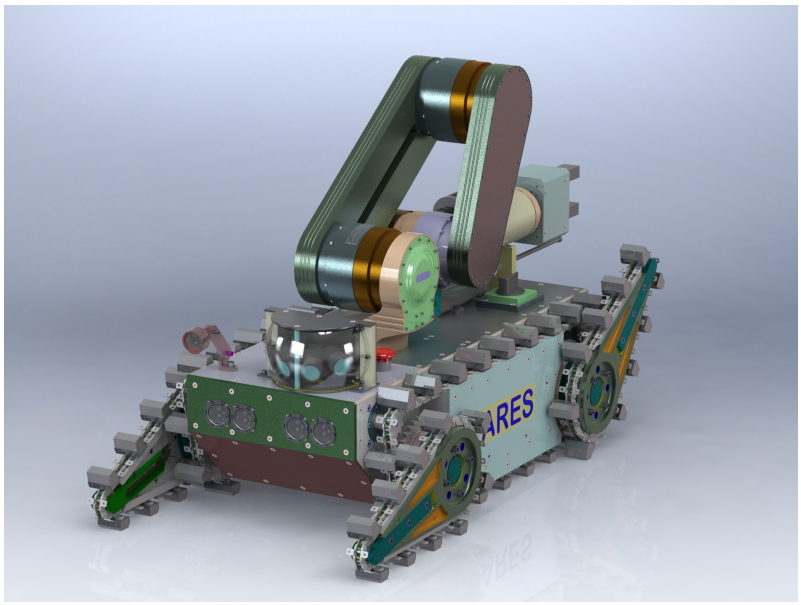


Figura 2.1: Projeto em SolidWorks original do DIANE.

As informações do manipulador não serão descritas neste trabalho, visto que não faz parte da locomoção do robô. Todavia, o programa descrito no capítulo posterior deve possibilitar a inclusão do controle do mesmo.

Para se movimentar no solo, o sistema robótico utiliza esteiras laterais, como mostrado na figura 2.1, que compõem o movimento linear e angular do mesmo. O movimento associado das esteiras permite que o robô se mexa para frente e para trás (esteiras com mesmo módulo e sentido de velocidades), em torno do seu próprio eixo (esteiras com sentidos contrários de movimento) e fazendo curvas, com o movimento associado por diferença

nos módulos das velocidades das esteiras. A equação 2.1, mostra o comportamento de movimentação do robô, sendo $v(t)$, a velocidade linear do robô, $\omega(t)$ sua velocidade angular, $\dot{\varphi}_d$ a velocidade tangencial da esteira direita e $\dot{\varphi}_e$ a velocidade tangencial da esteira esquerda. A figura 2.2 ilustra esse movimento.

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \frac{\dot{\varphi}_d}{2} + \frac{\dot{\varphi}_e}{2} \\ \frac{\dot{\varphi}_d}{2} - \frac{\dot{\varphi}_e}{2} \end{bmatrix} \tag{2.1}$$

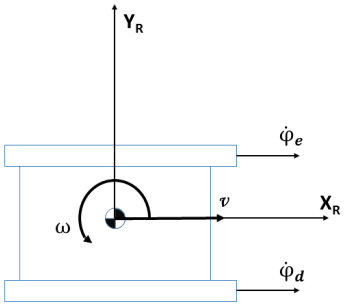


Figura 2.2: Movimento das esteiras

Para a transposição de obstáculos, o robô conta com dois pares de braços, dianteiros e traseiros, que se movimentam de acordo com a necessidade da situação de controle. Os braços se movimentam em pares, sendo o primeiro par frontal e o segundo o traseiro, assegurando estabilidade ao mesmo.

Para movimentar todo esse conjunto, o robô móvel possui quatro motores, dois para as esteiras e dois para os braços. Os motores das esteiras são do modelo *Maxon EC45*, com sensor de efeito Hall, e os motores dos braços são do modelo *Maxon EC32*, também com sensor de efeito Hall. Esses sensores são utilizados para averiguar a velocidade do motor. Estes motores estão acoplados a caixas de redução de 53:1 e 111:1, respectivamente, somado a uma rosca sem fim, no caso dos motores dos braços, que produz uma redução de 60:1. Mais informações sobre os motores são encontradas no apêndice A.

As EPOS, também desenvolvidas pela Maxon Motor, são *drivers* que permitem que o usuário grave parâmetros mecânicos, elétricos e térmicos do motor, por meio do programa do fabricante, e controle seu movimento por três modos diferentes. Isso quer dizer que, uma vez corretamente configurado, o *driver* aceita três tipos de informações diferentes para controle do motor. São elas velocidade, posição e corrente.

O controle por corrente é o mais básico deles. Nesse motor de corrente contínua, a velocidade de rotação é proporcional à corrente que passa pelo seu circuito.

No modo de velocidade, o *driver* calcula a corrente que faz o motor chegar a velocidade desejada por meio de um modelo do mesmo, construído a partir das suas especificações e de ganhos ajustados pelo programador, utilizando o *software* do fabricante. Esse modo é muito útil pois retira do programador a responsabilidade de calcular o modelo do robô e diminui os erros na movimentação. A unidade de velocidade no *driver* é o RPM

(rotações por minuto). Essa velocidade é atingida utilizando um controlador interno PI, com ganhos, limites de corrente, velocidade e aceleração, ajustados pelo programador.

Já no modo por posição, é acoplado um *encoder*. Esse *encoder* também tem suas informações gravadas na EPOS e gera uma realimentação da informação de movimento do motor para determinar a posição do mesmo. Essa informação permite ao *driver* que ele conte passos de movimentação do motor. Esses passos são determinados de acordo com a resolução do *encoder*, da característica do eixo do motor, e da rotação total do motor, definindo um alcance de movimento do mesmo e dividindo pela rotação do motor. Essas informações são descritas na equação 2.2, retirada do manual das EPOS. Para chegar a velocidade, a EPOS também conta com um controlador PID, com ganhos ajustados pelo programador.

$$1\ passo = y\ qc(quadcounts) \quad = 4.x \text{ (pulsos por revolução)} \tag{2.2}$$

As EPOS são comandadas pelo computador por meio de uma rede CAN (do inglês, *Controller Area Network*) . Por cima desta rede existe um protocolo CANOpen, que estabelece a garantia na entrega de pacotes como prioridade. O protocolo contém um cabeçalho que define as prioridades dos pacotes, bem como a identificação dos dispositivos pela determinação de um *ID*, e ainda o tipo de solicitação da mensagem. Esse tipo de solicitação são divididos em mensagens NMT (relativo a cofigurações do próprio proocolo), SYNC (mensagem de sincronização usada para disparar as rotinas periódicas), PDO (objetos de mensagem síncrona - este tipo de mensagem deve ser configurada ao iniciar o dispositivo, pois é responsável por enviar as mensagens periódicas disparadas pela mensagem SYNC) e SDO (objeto de solicitação de informação não periódico). Esta estrutura necessita de um *mestre* que deve administrar essas mensagens e dispará-las quando necessário. A rede foi montada de maneira que todas as EPOS, identificadas por um Id, fossem interligadas ao computador. Os Ids de cada EPOS podem ser encontrados no apêndice B. Uma ilustração de como foi montada fisicamente a rede é encontrada na figura 2.3.

Agora que foi definida a estrutura física e a movimentação do robô, pode-se definir os componentes de percepção do ambiente, os sensores. Depois, define-se o computador, componente responsável por administrar todos os componentes do robô e interpretar os comandos enviados pelo controle.

2.1 Sensores utilizados

O robô DIANE apresenta 2 sensores, além dos *encoders* e sensores de efeito Hall já mencionados: um *laser scan*, para mapeamento do ambiente em torno do robô; e um *kinect*, usado para mapeamento de distâncias, teleoperação e determinação do ângulo de inclinação do robô.

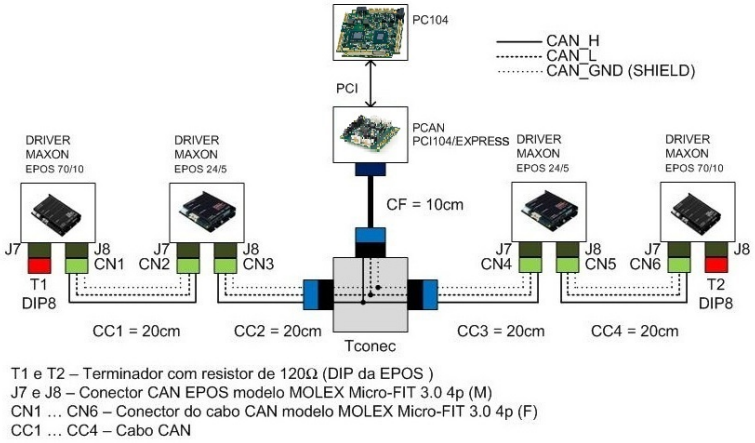


Figura 2.3: Arquitetura da Rede CAN

O *laser scan* é uma ferramenta muito útil de mapeamento de ambientes, identificação de marcos, e monitoramento de objetos muito próximos. O sensor escolhido foi o UST-10LX, cujas características se encontram na tabela A.4, desenvolvido pela Hokuyo, por ser pequeno e leve. Ele devolve uma estrutura compacta de dados que permite um processamento ágil de seus dados. Sua comunicação com o computador utiliza o protocolo TCP/IP (do inglês, *Transmission Control Protocol/ Internet Protocol*), através de um cabo Ethernet ligado à rede do DIANE.

O *kinect* é um sensor RGB-D (do inglês, *Red Green Blue - Depth*) que possui uma câmera comum, um emissor e um receptor infravermelhos, aliado a um acelerômetro capaz de fornecer dados de inclinação do dispositivo em relação ao solo. Este é um sensor acessível que fornece dados de um mapeamento 3D de profundidade, utilizado para geração de mapa, navegação e reconhecimento de padrões, como uma escada. Sua utilização na instrumentação robótica vem crescendo justamente pela sua acessibilidade. Sua comunicação com o computador utiliza uma interface serial USB (do inglês, *Universal Serial Bus*). A câmera RGB é utilizada na teleoperação do DIANE, fazendo dele um sensor completo para movimentação de sistemas robóticos.

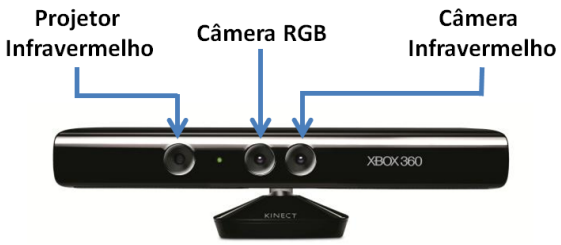


Figura 2.4: Composição dos sensores no *kinect* Fonte: de Lima [5], pp.37

2.2 Computador embarcado

Administrando todos os dispositivos embarcados no DIANE, foi escolhida a placa ADLQM67PC, da *ADL Embedded Solutions*. Sua filosofia de arquitetura permite que dispositivos periféricos sejam acoplados a um barramento PCI-e na vertical. Isso faz com que o espaço utilizado num robô seja muito menor que um computador normal. Aliado a um poderoso processamento, já que possui um processador Intel i7 de segunda geração e 4Gb de memória RAM (do inglês, *Random Access Memory*) , a placa se torna o computador ideal para sistemas de controle moderno que precisam de poder de processamento para realizar tarefas mais complexas.

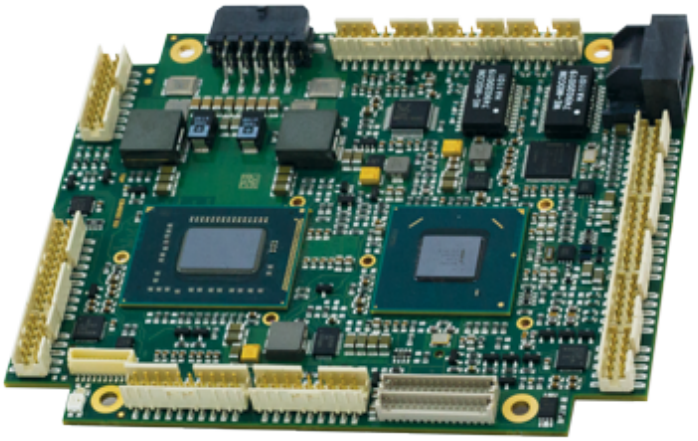


Figura 2.5: Placa ADLQM67PC

O computador possui interfaces USB e ethernet nativas, possibilitando a comunicação com os sensores, e, instalada nele, uma placa de interface com a rede CAN para comunicação com os *drivers* dos motores. Essa placa vai acoplada logo abaixo da mesma, utilizando uma interface *PCI-express*, e possui duas portas para comunicação com a rede CAN. A interface ethernet possibilita a comunicação por meio da rede interna do DIANE. Mais informações a respeito das configurações, basta olhar no apêndice A.

Junto com a placa, encontra-se um SSD (do inglês, *Solid State Drive*) que foi escolhido como dispositivo de armazenamento devido ao seu desempenho. As velocidades de escrita e leitura deste dispositivo são muito superiores a qualquer disco rígido encontrado no mercado. Isso permite que as tarefas realizadas pelo computador sejam completadas num tempo muito menor, o que é fator essencial para o desenvolvimento de controle em tempo real. Além disso, por não possuir partes mecânicas, seu funcionamento não é tão prejudicado por vibrações e choques, situações normais no funcionamento de um robô tipo EOD.

Já em relação ao sistema operacional, foi escolhido o Linux, mais especificamente a distribuição Ubuntu, muito utilizada no mercado. O sistema operacional foi selecionado

devido a sua robustez, suporte prolongado, grande comunidade colaborativa e acessibilidade. Ele possibilita a execução do *framework* robótico ROS, ferramenta escolhida como base para o desenvolvimento do software. Mais definições sobre o ROS podem ser encontradas no site do mesmo (www.ros.org). O Ubuntu é multitarefa e permite a rápida comunicação e resolução de tarefas, característica chave para o controle.

Capítulo 3

Software de controle do DIANE

O programa, ou *software*, de controle tem o papel de unificar os diversos formatos e informações que farão parte da operação do robô. Nele estão as diretrizes de funcionamento real do robô móvel, informações e configurações básicas de cada componente, entre outros. Além disso, no caso semiautônomo, o computador deve ter uma inteligência capaz de auxiliar o operador.

Os benefícios de um computador controlando o sistema robótico são inúmeros, como torná-lo mais resiliente, aumentar o desempenho da atuação a distância, e ser capaz agir mesmo sem comando do operador. Assim, a estrutura descrita na sessão 2.2 se torna justificada.

Como mencionado anteriormente, o programa é baseado na plataforma ROS. Este é um *framework para escrever programas robóticos. É uma coleção de ferramentas, bibliotecas e convenções que focam em simplificar a tarefa de criar comportamentos robóticos complexos e robustos numa grande variedade de plataformas robóticas*(traduzido do sítio www.ros.org). Programado em linguagens de programação orientadas à objeto, **C++** ou **Python**, o *framework* permite o reuso de código, tendo uma grande comunidade que compartilha esses códigos, e simplifica a comunicação entre processos e máquinas. Dessa forma, escolheu-se o **C++**, para programação da plataforma.

Para desenvolver e organizar os códigos, o *Qt Creator* foi escolhido como ambiente de desenvolvimento. Apesar do ROS possuir ferramentas gráficas para o desenvolvimento de robôs, estas não são adequadas ao usuário final. Assim, optou-se por usar a interface gráfica *Robot GUI*, que encontra-se em estágio de desenvolvimento, para operação do robô.

3.1 Organização do Software

O desenvolvimento do *software* de controle do robô móvel DIANE envolveu inúmeras etapas. Neste trabalho, escolheu-se focar na contribuição para o controle de robôs,

principalmente no deslocamento.

Para operar o robô remotamente são necessários, pelo menos, dois computadores: um do robô, que deve ir embarcado; e outro de base, que funciona como interface com o operador. Seguindo esse raciocínio, o programa separa suas funcionalidades por computador que será executado.

No computador embarcado são executadas as rotinas de ROS, ou *nós*, necessárias para a tradução dos dados de seus componentes, sensores e atuadores, e uma rotina que centraliza e distribui corretamente as informações entre as funcionalidades. Por questões de implementação no ROS, foi definido um processo principal, dividido em diversas rotinas paralelas, os *nodelets*. Assim, existem oito processos sendo executados na operação normal do sistema robótico, sendo quatro deles do mesmo tipo:

- ***epos***: *nodelet* de controle das informações relativas aos *drivers* dos motores
- ***freenect***: *nó* de publicação das informações relativas aos sensores ópticos do *kinect*
- ***kinect_aux***: *nó* de publicação das informações relativas aos sensores inerciais do *kinect*
- ***urg_node***: *nó* de publicação das informações do *laser scan*
- ***diane_controller***: *nodelet* de gerenciamento das informações de locomoção do DIANE

Os *nós* executados no computador remoto são referentes aos dispositivos ligados ao computador, processos da interface gráfica, e um único processo enviando comandos de velocidade para o computador embarcado. Os *nós* executados são descritos a seguir. A figura 3.1 ilustra a organização do *software*.

- ***linux_devs***: *nodelet* de gerenciamento de periféricos do *Linux*, como teclado, mouse e *joystick*
- ***keyboard***: *nodelet* de publicação das informações do teclado
- ***gamepad***: *nodelet* de publicação das informações do *joystick*
- ***diane_mapper***: mapeamento dos comandos do teclado e *joystick* em dados de velocidade do robô.

Os processos aqui apresentados serão descritos mais profundamente nas secções posteriores. Além dos *nós* para controle, existem processos inerentes à interface gráfica, para visualização das informações que também serão discutidos à frente.

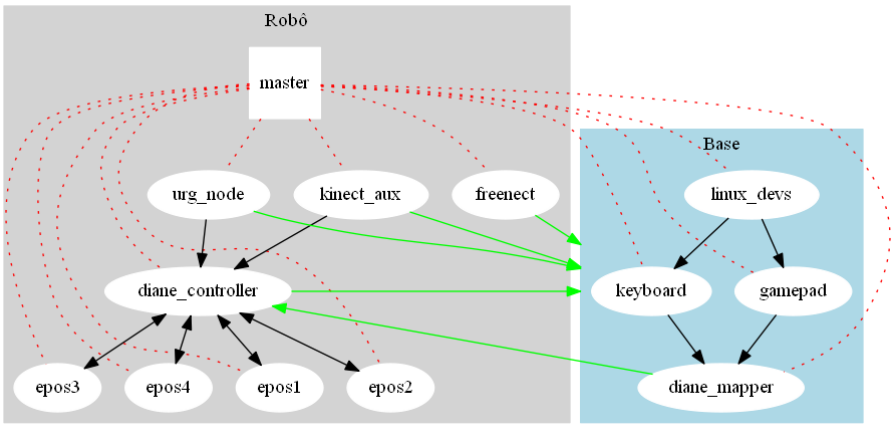


Figura 3.1: Estrutura do software de controle do DIANE

3.2 Processos embarcados

Os processos do robô podem ser separados em: sensoriamento; atuação; e integração.

Os processos de sensoriamento são aqueles ligados aos sensores do robô. Ou seja, cada sensor tem uma rotina responsável por tratar seus dados e publicar para outras rotinas. O *laser scan* e o *kinect*, sensores de profundidade, imagem e inercial, tem rotinas que tomam conta de seus dados.

Os processos de atuação são os processos que cuidam, basicamente, dos motores. Como dito anteriormente, os motores estão ligados às EPOS embarcadas. Existem quatro EPOS, uma para cada motor, como apresentado na figura 3.1. Assim, existe uma rotina para cada EPOS.

Os processos de integração são os responsáveis por interpretar os dados e organizá-los da melhor forma possível para o controle, seja por teleoperação ou por um algoritmo. Nestes é que se encontram as lógicas de controle básicas para locomoção. No caso do DIANE, o processo responsável pelo cálculo das velocidades de cada motor, dado velocidade linear e angular desejados, é o *diane_controller*.

O *nó* do sensor *laser scan*, o *urg_node* foi desenvolvido por terceiros, sendo encontrado no site do ROS. Este fornece os dados da varredura através da mensagem *scan*, do tipo *sensor_msgs::LaserScan*. A única configuração feita para a comunicação com o dispositivo é seu endereço IP, no caso **172.16.0.10**, através da porta **10940**.

O sensor *kinect* possui dois *nós*, também desenvolvido por terceiros, controlando suas funções, *kinect_aux* e *freenect*. A única configuração feita foi no *nó freenect* para só fornecer os dados de profundidade, que pode ser encontrado em sua página do ROS. Os tópicos usados para o controle são:

- *cur_tilt_angle* (std_msgs::Float64): Informação de *pitch* do robô
- *camera/depth/points* (sensor_msgs::PointCloud2): *point cloud* fornecido pelo *kinect*

- *camera/depth/image* (sensor_msgs::Image): imagem de profundidade em escala de cinza

Agora, discute-se um pouco mais profundamente acerca de cada nó desenvolvido para o controle do robô.

3.2.1 Pacote *epos*

O pacote *epos* é responsável por todas as informações a respeito do *driver* de controle dos motores. Esse pacote permite a troca de informações entre computador e *driver* EPOS, como velocidade do motor. Este se encarrega de gerar o sinal de corrente correspondente aquela velocidade.

Este pacote, primeiramente, tem configurado os protocolos de comunicação CANOpen, assim como a lista de objetos e comandos que devem ser enviados às EPOS para seu funcionamento. Estas informações foram escritas em classes que fazem parte integral do pacote *epos*.

Para trabalhar adequadamente, as EPOS devem estar configuradas com as informações dos motores. As informações necessárias são de caráter mecânico, elétrico e térmico, pois a saída de corrente deve ser adequada as características do motor, para que não ocorra dano algum. Essas informações foram retiradas do sítio do fabricante e devidamente configuradas em cada EPOS, seguindo a configuração do manual. As configurações de cada EPOS são encontradas na tabela A.2.

O dispositivo EPOS tem modos de controle que auxiliam o programador, e consequentemente o operador, a desenvolver o movimento desejado. Ademais à entrada de corrente, o dispositivo permite que seja definido uma velocidade ou uma posição almejada. Estes modos de controle vieram retirar a responsabilidade do desenvolvimento de um modelo de cálculo para controle do motor. A partir das informações já configuradas, o dispositivo gera um modelo que calcula as correntes devidas. Essa corrente pode ser, também, limitada, para que o motor não sofra de desgaste térmico ou elétrico.

A unidade de velocidade no *driver* é dado por rotações por minuto (RPM) . O controle por velocidade é feito por malha fechada, na presença de sensor de efeito Hall e/ ou um *encoder*. O programador, por meio de uma ferramenta disponibilizada pelo fabricante das EPOS, ajusta os ganhos do controlador interno PI.

O controle por posição utiliza o *encoder*. A EPOS recebe os pulsos do *encoder* e soma até a posição desejada. Os ganhos de transformação variam para cada configuração *encoder-motor*, assim, os cálculos dos ganhos foram feitos e depois configurados na EPOS. A equação 2.2 mostra como é feito o cálculo de transformação dos pulsos do *encoder* para passo. Acima dessas informações é que podem-se jogar os ganhos de reduções aplicadas e transformações para grandezas inteligíveis, como rad/s (radiano por segundo).

$$
\begin{aligned}
(m/s \implies rpm) &= \frac{(s \implies min) \times (\text{Reduções})}{2.\pi.(\text{Raio da roda})} \\
(m \implies passo) &= \frac{(passo) \times (\text{Reduções})}{2.\pi.(\text{Raio da roda})}
\end{aligned}
\tag{3.1}
$$

Esses ganhos estão todos embarcados na rotina *epos*, sendo alguns deles configuráveis em tempo de execução nos arquivos *launch*, como mostra a tabela 3.1. Transformações de rpm para m/s (metros por segundo), metros para número de passos, entre outros são parâmetros de entrada dos *nodelets* do pacote *epos* e precisam ser configurados antes de iniciar seu funcionamento.

Tabela 3.1: Configurações das EPOS no arquivo *launch*

	epos1	epos2	epos3	epos4	Comentários
can_device	can0	can0	can0	can0	adaptador CAN usado
device_id	1	2	8	4	ID no protocolo CANOpen
inverted	false	true	false	true	movimento em sentido contrário
velocity	0.3	0.3	–	–	velocidade máxima
acceleration	2000	2000	2000	2000	
deceleration	2000	2000	2000	2000	
gain_step_to_meter	421974.5223	421974.5223	7575068.244	7575068.244	
gain_mps_to_rpm	28117.373279568	28117.373279568	227252.0473	227252.0473	
digital_outputs	–	–	–	[1,2,3]	saídas digitais controladas
analog_outputs	–	–	–	–	saídas analógicas controladas

As EPOS disponibilizam portas lógicas e analógicas de entrada e saída. Visto isso, os componentes *laser scan* (configurado na porta lógica 2, terceira porta lógica na EPOS), *kinect* (configurado na porta lógica 1, segunda porta lógica na EPOS) e LEDs (não configurado) para iluminação estão ligados logicamente a *switches* para energizar estes componentes. Isso torna seu acendimento muito mais simples, já que faz parte de uma estrutura conhecida e mapeada, além de economizar a energia das baterias. Basta chamar um serviço de ROS *epos4/d_a_outputs* com valor *true* para ligá-los. Esse serviço deve ser chamado pela interface gráfica, mas ainda não foi implementado.

O *nodelet* da EPOS deve receber muitas informações acerca dos modos de operação do motor. Devido às muitas funções que as EPOS possuem, o *nodelet epos* deve ser capaz de configurar cada uma delas. Para discutir sobre a funcionalidade da rotina, inicia-se a análise introduzindo uma lista dos tópicos e serviços do nó *epos*, com as descrições do que cada tópico faz para o controle do programa.

1. Tópicos:

(a) Publicados:

- epos/state (std_msgs::UInt8): publica o estado da conexão com o dispositivo EPOS.
- epos/info (epos::Info): Publica as informações a respeito de velocidade, posição, corrente, *setpoint* de controle, modo de controle (velocidade, posição ou corrente), e estados das portas digitais e analógicas de entrada.
- epos/emergency (std_msgs::ByteMultiArray): publica mensagens de emergência enviados pelas EPOS.

(b) Subscritos:

- epos/load_vars (epos::Info): tópico que carrega as informações nas variáveis correspondentes.

2. Serviços:

(a) Anunciados:

- epos/control (epos::Control): serviço de envio de dados do controle do robô.
- epos/d_a_output (epos::DAOutputs): serviço que estabelece os valores de saída das portas lógicas e analógicas das EPOS.
- epos/connect (epos::Connect): estabelece a conexão entre o computador e determinada EPOS.

Primeiramente, o *nodelet* estabelece a conexão entre o computador e a EPOS designada. Depois, ele configura o dispositivo de acordo com os ganhos determinados pelo operador, do arquivo *launch*. Somente agora é que o processo espera um comando de velocidade, posição ou corrente para enviar o dado para o dispositivo.

Este modo garante que a EPOS estará sempre em funcionamento normal antes de liberá-la para operação. Mesmo com a configuração dos parâmetros limites, ainda é possível, nesta etapa, estabelecer lógicas que protejam o dispositivo. O funcionamento desta rotina foi testado exaustivamente em todas os modos de controle e apresentou um resultado muito bom. O motor responde com atrasos de milissegundos em relação ao comando enviado pelo operador. Atribui-se esse atraso à comunicação entre computadores.

3.2.2 Pacote *diane_controller*

O pacote *diane_controller* é o que concentra as informações primordiais ao movimento. Por ele passam as atuações e os devidos modos de operação que o usuário poderá escolher e configurar. Estes modos de operação são diferentes dos modos de controle das EPOS, mas são baseados neles.

Este pacote é importante pois é a conexão do mundo exterior com os motores. O controle foi desenvolvido para fornecer velocidades linear e angular do robô. Entretanto, a informação passada ao *epos_nodelet* é a velocidade dos motores. Desse modo, existe uma matriz que relaciona as velocidades linear e angular com as velocidades das esteiras, descrita em 2.1. A implementação no programa são observadas na secção *Esteiras*, do apêndice B.

Este pacote também está responsável por devolver as informações dos motores para o controlador, desenvolvido em Matlab.

Devido as limitações da EPOS, por causa do controle de posição, foi desenvolvido um método capaz de salvar as últimas posições alcançadas pelos motores. Esta necessidade vem da possibilidade de perda de conexão ou algum desligamento, seja esperado ou inesperado. Assim, o *diane_controller*, já que concentra todas as informações de posição de todas as EPOS, é responsável por guardar as últimas posições dos motores, para que, ao religar o robô, o controlador e o operador não tenham que mudar seu referencial ou reiniciar de um referencial fixo.

O *diane_controller* pode receber diversas publicações de velocidade de diversos computadores. Existe um controle, feito por ID que é distribuído pelo próprio nó na forma de serviço, que só aceita, durante um intervalo de tempo de dois segundos, publicações de velocidade daquela origem. Isso garante, também, que, caso haja atraso na comunicação devido ao congestionamento de pacotes, o operador não vai perder o controle da locomoção por conflito de pacotes ou tempo de ausência. Ainda na perda de conexão, o nó foi programado para que seja persistido o último dado de controle por dois segundos, até que o comando padrão de frear imediatamente assuma. Esse comando foi configurado para que, na ocasião de perda de comunicação no meio da subida da escada, o robô não desça de volta. Sua implementação pode ser encontrado no apêndice B como *StandardInput*.

Novamente, a seguir é apresentada uma lista dos tópicos que são publicados, para fins de análise da sua funcionalidade.

1. Tópicos:

   (a) Publicados:

   - epos/load_vars (epos::Info): publica as informações salvas de cada EPOS para serem carregadas pelos nós das mesmas.
   - controller/ActualInput (controller::Control): devolve o dado que foi enviado aos nós das EPOS.
   - controller/Feedback (std_msgs::Float64MultiArray): mensagem de realimentação para o controle baseado em MatLab. Segue o formato {modo de controle, ângulo de inclinação do robô, velocidade linear, velocidade angular, velocidade/ posição do braço dianteiro, velocidade/ posição do braço traseiro}.

(b) Subscritos:

- epos/state (std_msgs::UInt8): estado da conexão do computador com a respectiva EPOS.
- epos/info (epos::Info): Informações de cada EPOS.
- cur_tilt_angle (std_msgs::Float64): ângulo de inclinação do kinect em relação a horizontal, em graus.
- controller/input (controller::Control): dado de modo de controle e valores de velocidades linear e angular e velocidade ou posição que os braços devem se movimentar.

2. Serviços:

(a) Cliente:

- epos/control (epos::Control): Envia os dados de modo de controle e *set-point* para cada um dos *epos_nodelet*.

A rotina de controle calcula, a partir das velocidades recebidas em *controller/input*, as informações que devem ser passadas para cada uma das EPOS. O dado é enviado para os respectivos *nodelets* de acordo por meio do serviço *epos/control*.

O tópico *cur_tilt_angle* devolve a informação de ângulo com a horizontal do *kinect*. Como o *kinect* se movimenta junto com o robô móvel, pode-se inferir sobre seu ângulo de inclinação junto. Essa informação é enviada por meio do tópico *controller/Feedback*.

Vale ressaltar que o meio de comunicação com o Matlab é feita por meio de um processo, chamado *diane_remap*. Este *nodelet* faz um De-Para dos comandos do Matlab, para o *diane_controller*. Como o tópico *controller/input* tem um tipo customizado, este nó recebe um *array* padronizado, mesmo que o *controller/Feedback*, mas chamado *cmd*, e publica traduzido pro formato *controller::Control*, no tópico *controller/input*. Assim, a estrutura completa de rotinas executadas no robô pode ser vista na figura 3.2.

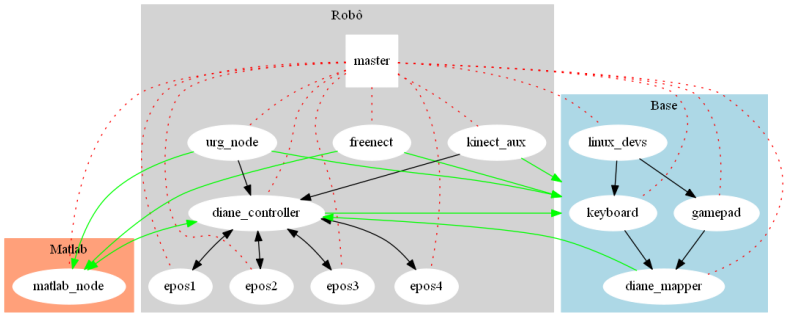


Figura 3.2: Estrutura do programa com Matlab

3.2.3 robot_upstart

Para iniciar os processos do robô quando ligá-lo, foi utilizado o pacote do ROS *robot_upstart*. Este é utilizado para auxiliar na configuração de inicialização de processos do ROS. É criado um serviço de Linux a partir do pacote utilizado e este se encarrega de iniciá-lo, ao entrar o sistema operacional.

Para isso, deve ser configurado um arquivo *launch* que carrega todas as informações necessárias à execução do serviço. O *launch* criado foi o *diane_robot.launch* que carrega todas as configurações descritas nesta secção, e pode ser encontrado no apêndice B. Para utilizar o pacote, basta seguir o tutorial encontrado no sítio do ROS.

3.3 Robot GUI

Os processos da base são aqueles necessários para se atuar no robô a partir do comando do operador, ou seja, todos os nós que devem ser atrelados ao operador, e não ao robô, são agrupados e executados no computador junto ao usuário. Os *nodelets keyboard*, *gamepad* e *linux_devs* seguem a mesma filosofia dos nós dos atuadores e sensores do DIANE. Já o nó *diane_mapper* é o nó de tradução entre os dados enviados pelo *joystick*, ou pelo *keyboard*, em informações de velocidade que o nó *diane_controller* seja capaz de compreender.

Devido a estarem no computador remoto, os processos executados são atrelados à interface gráfica desenvolvida no laboratório. O Robot GUI é o programa criado para controlar os aspectos de qualquer robô, com customizações relativas às especificidades de cada um, utilizando como base os processos do ROS. Uma boa explicação do funcionamento da interface é encontrado em Loureiro [13].

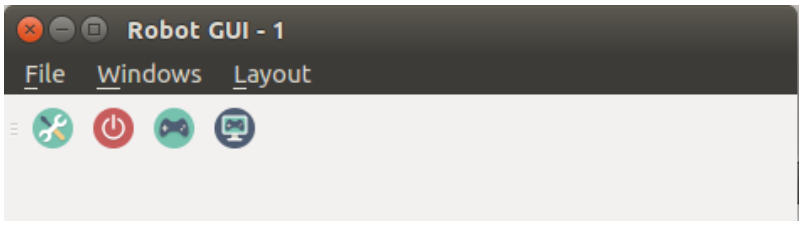


Figura 3.3: Tela inicial da interface gráfica desenvolvida

Este trabalho vai focar na especificidade do robô e dos processos aqui encontrados. Assim, as rotinas necessárias para controlar o movimento do robô são encontrados no pacote *diane_mapper*, que faz o mapeamento dos dados dos eixos e botões do *joystick* e os traduz para velocidades linear e angular desejadas.

Para iniciar esses processos, foi configurado outro arquivo *launch*, chamado *diane_base.launch*. Este arquivo é utilizado numa ferramenta da interface gráfica para inicializar corretamente, se comunicando com o *master* do robô. Esta tela de configu-

ração permite que vários *launches* estejam rodando ao mesmo tempo. A figura 3.4 ilustra essa funcionalidade.

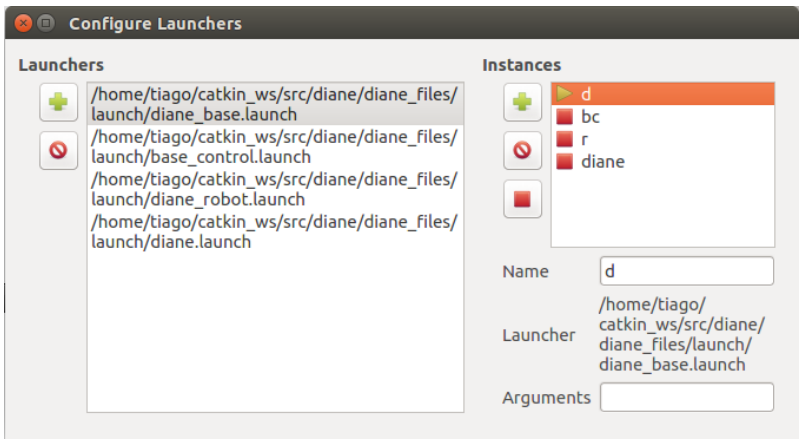


Figura 3.4: Tela de configuração de arquivos *launch*

Assim, o operador não precisa saber nada sobre ROS para controlar o robô, bastando abrir a interface gráfica que o programa já configura tudo o que for necessário para a operação normal do robô móvel.

Os pacotes *linux_devs*, *keyboard* e *gamepad* foram desenvolvidos no laboratório. As configurações necessárias para sua utilização são encontradas no arquivo *diane_base.launch*, mas não é necessário que o código seja customizado.

Utilizando o argumento *sender_name* do pacote *linux_devs* o processo sabe quem é o dispositivo que deve mandar os comandos para o controle. As configurações do pacote *diane_mapper* serão discutidas posteriormente.

Uma vez iniciado o arquivo *launch*, o usuário deve selecionar o *joystick* que quer utilizar. Ele faz essa seleção por meio de uma janela na interface gráfica, chamada *Device Management*. O dispositivo que se deseja deve ser selecionado e dado o botão "*Apply*" pressionado.

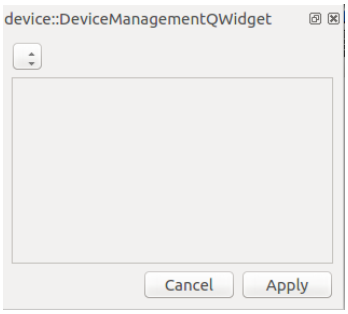


Figura 3.5: Tela de seleção de dispositivos

Index	Nome do botão no controle
0	A
1	B
2	X
3	Y
4	LB
5	RB
6	back
7	start
8	power
9	Button stick left
10	Button stick right
Index	**Nome do eixo no controle**
0	Left/Right Axis stick left
1	Up/Down Axis stick left
2	LT
3	Left/Right Axis stick right
4	Up/Down Axis stick right
5	RT
6	cross key left/right
7	cross key up/down

Tabela 3.2: Informações do joystick Fonte: www.ros.org

3.3.1 Pacote *diane_mapper*

Agora que o dispositivo foi selecionado, a rotina *diane_mapper_nodelet* fica encarregada de receber as informações de botões pressionados e valores dos eixos por um tópico com o nome do dispositivo. Esses botões e eixos devem obedecer a padrões definidos pelo ROS, que são transpostos aqui na tabela 3.2, para facilitar a referência de estudo.

Como pode ser observado na figura 3.7, os dados de modos de controle e valores são selecionados e mapeados de acordo com botões pressionados. Por exemplo, caso o operador pressione o botão "LB" (ver esquema do *joystick* na figura 3.6), o modo de *velocidade 1x* é selecionado e de acordo com a porcentagem do eixo do analógico esquerdo, a velocidade aumenta por um fator multiplicador de 1x. Ou seja, a velocidade limite estabelecida no arquivo *launch* é dividida em 4 vezes e, nesse modo, a velocidade máxima que o robô é capaz de atingir é um quarto (1/4) do valor máximo da velocidade.

Essa informação é calculada toda vez que o botão correspondente estiver fazendo a ação configurada. Botões podem estar em estados: "Pressionado"(P), "Pressionando"(H), "Não pressionado"(NH) e "Solto"(R). De acordo com a configuração estabelecida do par *botão-estado*, o modo de controle muda. Os comandos podem ser encontrados na tabela 3.3, que mostram os modos de controle e as configurações para movimento do robô.

Os modos de controle *Velocity 1x* e *Velocity 2x* movimentam os braços na mesma velocidade, quando o eixo direito é acionado. Já nos modos *Velocity Rear Leg* e *Velo-*

Figura 3.6: Esquema do *joystick xbox* 360

Tabela 3.3: Configurações de mapeamento do *joystick*

	Coast	***Brake***	***Velocity 1x***	***Velocity 2x***	***Velocity Front Leg***	***Velocity Rear Leg***
Acionamento	A/H	B/H	LB/H	RB/H	Button stick left/H	Button stick right/H
Toggle	no	yes	no	no	yes	yes
Velocidade linear	–	0	Up/Down Axis stick left	Up/Down Axis stick left $\times 2$	Up/Down Axis stick left	Up/Down Axis stick left
Velocidade angular	–	0	Left/Right Axis stick left	Left/Right Axis stick left $\times 2$	Left/Right Axis stick left	Left/Right Axis stick left
Braço dianteiro	0	0	Up/Down Axis stick right	Up/Down Axis stick right	Up/Down Axis stick right	0
Braço traseiro	0	0	Up/Down Axis stick right	Up/Down Axis stick right	0	Up/Down Axis stick right

city Front Leg, cada braço é controlado separadamente, com possibilidade do movimento linear e angular ainda ser controlado pelo eixo esquerdo, com velocidade normal. Os modos *Coast* e *Brake* servem somente para parar o robô (*Brake*) ou mantê-lo sem controle (*Coast*).

Essas informações são salvas num arquivo XML chamado *gamepad_mapper*, salvo na pasta de configuração, encontrada no arquivo *diane_base.launch* (argumento *config_location*).

Essa informação de velocidade é publicada no mesmo tópico *controller/input*, mencionado na secção anterior. O pacote *diane_mapper* precisa saber quem está mandando e para quem vai mandar, assim, os argumentos *gamepad_manager* e *controller* devem referenciar os nomes dos *nodelets* responsáveis pelas funções de botões pressionados e gerenciamento dos dados de velocidade do robô. Além disso, o argumento *max_velocity* configura a máxima velocidade atingida pelo robô e utiliza essa informação para calcular os valores dos modos de controle já mencionados.

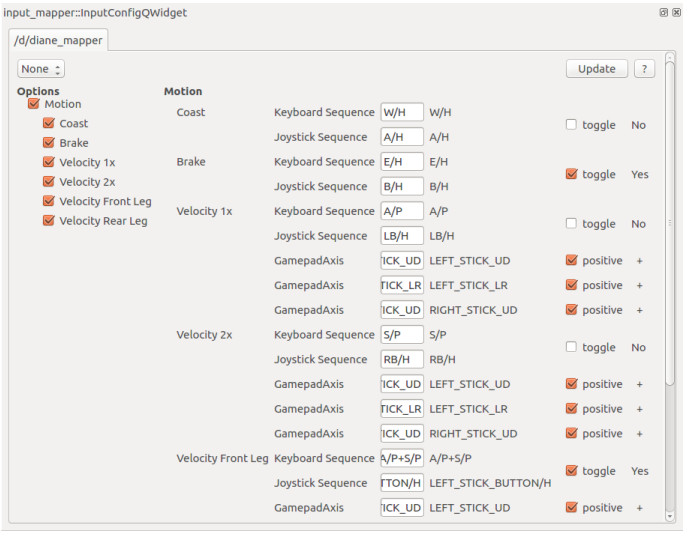


Figura 3.7: Tela de configuração dos botões e eixos do *joystick*

Dessa forma, o software embarcado do DIANE foi desenvolvido para tornar possível sua operação. Este software foi testado exaustivamente para que as situações de erro estejam bem tratadas. A resposta dos motores aos comandos enviados tanto pelo *joystick*, quanto pelo Matlab, ambos em computadores clientes, se dá em milissegundos. A interface possibilita a configuração dos componentes embarcados no robô de modo intuitivo e fácil. Estes resultados são mais que satisfatórios pro desenvolvimento do software, que se mostrou uma excelente ferramenta de controle.

3.4 Utilização do Matlab com ROS

Toda a estrutura para o desenvolvimento do controle foi preparada. Agora, para a escrita dos algoritmos de detecção de escadas ainda é necessário definir mais um passo: a integração com o software Matlab.

O poder do Matlab é conhecido por todos os engenheiros, por sua enorme aplicabilidade e versatilidade, além da disposição de múltiplas ferramentas para diferentes finalidades em uma única plataforma. Por esse motivo foi escolhido para desenvolver a locomoção na escada. Agregando esse time está a integração com o ROS. Desde sua versão r2015b o Matlab oferece integração nativa com o software. Esta integração funciona com uma simples configuração de variáveis de ambiente, as mesmas configuradas para utilizar o ROS em diversas máquinas. São elas **ROS_IP** e **ROS_MASTER_URI**. Estas variáveis designam, respectivamente, o endereço da máquina remota, que deseja se conectar aos processos executados no DIANE, e o endereço do *master* do programa, no caso o endereço do computador embarcado.

Essas variáveis podem ser configuradas no *command line* do MatLab pelos seguintes comandos:

- *setenv('ROS_IP','172.16.11.42')*
- *setenv('ROS_MASTER_URI','http://172.16.11.27:11311')*
- *rosinit*

Vale o destaque para a especificação da porta utilizada para comunicação do ROS, que por padrão é a porta 11311.

O comando *rosinit* tenta estabelecer a conexão do Matlab com o *master* do ROS, que deve estar na mesma rede. O *master* interpreta o Matlab como um nó (com nome aleatório com prefixo *matlab*), e o adiciona nas conexões entre os nós executados. Dessa forma, o Matlab é capaz de receber e enviar as informações, por meio de tópicos e serviços.

O envio de dados é feito pela mesma estrutura encontrada nos programas escritos em ROS, por meio de objetos *publishers* e *subscribers*, ou *clients* e *servers*. Esses objetos são variáveis no Matlab, que devem ser inicializadas conforme seu tipo:

- *subscriber*: comando *rossubscriber('/topicName',@TopicCallback)*
- *publisher*: comando *rospublisher('/topicName','MessageType')*
- *client*: comando *client = rossvcclient('/serviceName')*, seguido de *call(client,message);*
- *server*: comando *rossvcserver('/serviceName', 'ServiceType', @ServiceCallback)*

Assim, o Matlab deve receber as informações do *kinect*, *laser scan* e *diane_controller*, bastando se sobrescrever aos tópicos:

- *camera/depth/points*: informações do *point cloud*, do *kinect*
- *camera/depth/image*: informações da *depth image* (em escala de cinza), do *kinect*
- *scan*: informações de varredura do *laser scan*
- *controller/Feedback*: informações das velocidades do robô, posição dos braços e inclinação em relação ao solo.

Os objetos recebidos por mensagem são interpretados pelo Matlab e possuem métodos de conversão em estruturas do próprio Matlab. Assim, um *point cloud* de ROS, recebido como mensagem no Matlab, pode ser transformado no objeto *point cloud* do próprio *software*.

Para controlar o robô, o Matlab deve publicar as informações de comando no seguinte tópico:

- */cmd*: tópico de recebimento de comando do processo *diane_remap*

Para facilitar a comunicação, foi preparada uma função que publica essas informações pelo Matlab no tópico *cmd*, chamada *SendControlMessage*. Essa função pode ser encontrada na secção B.

Agora que tudo foi definido, pode-se testar a locomoção. A teleoperação foi testada em bancada, com atrasos de comando de menos de 1 s, demonstrando a agilidade do programa. Quanto a robustez, foi simulado a perda de conexão durante o movimento e o robô foi capaz de se manter parado, como programado. Apesar dos excelentes dados de atuação, a imagem do *kinect*, quando transmitida pela rede, começa a ter atrasos. Isso se deve a largura da banda da rede *wireless* do DIANE, que apesar do bom alcance, é pequena para a transferência dos dados. Esse resultado piorou quando os dados do *point cloud* do *kinect* começaram a ser transmitidos. Os resultados da locomoção são discutidos no capítulo 4.

Capítulo 4

Locomoção semiautônoma em escadas

A técnica para locomoção semiautônoma sobre a escada foi baseada em de Lima [5], mas, devido a particularidades da implementação, bem como desempenho computacional, os algoritmos foram modificados.

Para o robô se locomover sobre a escada foi necessário dividir o problema em três: detecção e modelagem da escada; metodologia para entrar e sair da escada, e controle sobre a escada. Cada uma das partes será analisada separadamente. Vale lembrar que toda a metodologia deste capítulo foi desenvolvida em Matlab.

4.1 Detecção e modelagem de escadas

Para esta etapa, foi escolhido o sensor de profundidade *kinect* para detecção e modelagem da escada. Além de fornecer uma imagem 3D, é uma ferramenta barata, acessível, e em sua primeira versão possui um acelerômetro, utilizado para determinar a angulação do robô em relação ao solo. Apesar de ruidoso, o dado devolvido pelo *kinect* é bastante acurado. Uma massa de dados mais densa tornaria a atenuação estatística dos ruídos possível, fornecendo com melhor precisão as informações necessárias para a detecção e modelagem da escada.

4.1.1 Detecção de escadas utilizando imagem 3D densa

Em [6], o autor se aproveita das descontinuidades da imagem ocasionadas pelas bordas das escadas. Devido às dimensões do DIANE, foi implementado este método, proposto por de Lima [5], e modificado neste trabalho. Como o robô móvel utilizado, DIANE, tem uma altura de 0.23 m, a identificação da escada através das descontinuidades foi a única encontrada para tornar possível a identificação. A posição de detecção é ilustrada na figura 4.1.

Outras limitações quanto a modelagem da escada devido às características do DIANE estão listadas na tabela 4.1. Estas foram retiradas das justificativas encontradas em

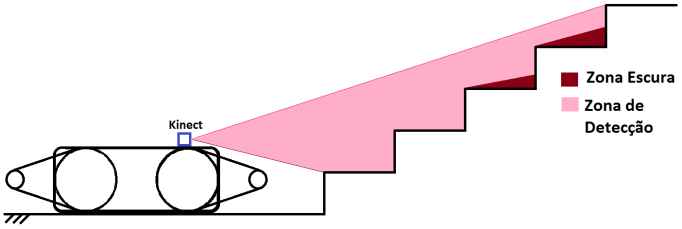


Figura 4.1: Posição de detecção do DIANE em relação à escada

de Lima [5].

Tabela 4.1: Informações referentes a detecção da escada

Altura máxima do degrau ($h_{s_{max}}$)	0.3065 m
Largura mínima da escada (W_s)	0.5 m
Angulação máxima do plano da escada (α_{max})	40°

Características padrão da escada

Segundo a norma NBR 9077, as escadas devem seguir um padrão. Para comparação posterior com os dados obtidos, suas características são apresentadas na tabela 4.2.

Tabela 4.2: Padrão de escadas segundo a norma NBR 9077

Piso (p)	$0.28\ \text{m} \leq \text{p} \leq 0.32\ \text{m}$
Espelho (e)	$0.16\ \text{m} \leq \text{e} \leq 0.18\ \text{m}$
Fator de construção	$0.63\ \text{m} \leq (\text{p} + 2\text{e}) \leq 0.65\ \text{m}$
Largura mínima para escadas fixas	1.20 m

4.1.2 Método proposto

O método pode ser separado em duas etapas: a detecção de bordas da escada da imagem de profundidade; e a modelagem da escada. O pseudocódigo 4.1 mostra o passo a passo do algoritmo de detecção implementado neste trabalho. As variáveis utilizadas são descritas a seguir.

a) D_{image}: é a imagem, *depth image* em escala de cinza, devolvida pelo sensor. *b*) P_{In}: *point cloud*, ou nuvem de pontos, correspondentes à captura do *kinect*, do ambiente. *c*) P_{Out}: *point cloud*, ou nuvem de pontos, correspondentes ás bordas dos degraus da escada. *d*) F_{Stair}: booleano indicativo de presença de escada. *e*) L_{3D}: *array*, ou lista, de pontos das bordas de cada degrau, separado por degrau. Ou seja, cada ponto está agrupado com outros pertencentes ao seu degrau, e a nada mais. *f*) E_{image}: *edge image*, ou imagem de bordas, extraído de D_{image}. *g*) L_{Hough}: matriz de votação gerada pela a Transformada

de Hough (SHT). *h)* L_{merge}: linhas encontradas pela SHT, aglutinadas de acordo com suas características angulares e lineares, bem como proximidade entre os pontos. *i)* H_{col}: histograma de frequência de linhas nas colunas da imagem E_{image}. *j)* $Lim_{Sup,Inf,Esq,Dir}$: Limites superior, inferior, esquerdo e direito da imagem da escada, nas coordenadas da imagem. *k)* N_{linhas}: número de linhas detectadas na imagem E_{image}. *l)* Pl_{temp}: plano ajustado por mínimos quadrados aos pontos de P_{Out}. *m)* α_{temp}: ângulo entre plano Pl_{temp} e o plano horizontal (xy). *n)* $\vec{n}_{Pl}$: vetor diretor do plano Pl_{temp}. *o)* $\vec{n}_{horz}$: vetor diretor do plano horizontal, ou seja $[x,y,z] = [0,0,1]$ *p)* α_{max}: máxima angulação entre o plano Pl_{temp} e o plano horizontal que permita transposição do obstáculo pelo robô.

Pseudocódigo 4.1 Detecção da escada

Entrada: D_{image}, P_{In}
Saída: $F_{Stair}, L_{3D}, P_{Out}$

1: Aplicar *Canny Edge Detection* em D_{image} e gerar E_{image}
2: Aplicar *Standard Hough Transform* em E_{image} e filtrar por θ próximo à horizontal, gerando L_{Hough} $\{-5 \leq \theta \leq 5\}$
3: Juntar linhas próximas, com coeficientes angular e linear próximos, em L_{merge}
4: Montar histograma H_{col} de frequência de linhas em colunas da imagem E_{image}
5: Extrair linhas correspondentes as colunas de maior frequência em H_{col} e obter limites, $Lim_{Sup}, Lim_{Inf}, Lim_{Dir}, Lim_{Esq}$, do quadro que contém a escada na imagem
6: Redimensionar linhas L_{merge} para ficar dentro dos limites anteriores, gerando L_{3D}
7: {O menor número de bordas para ser considerado uma escada é 3}
8: **if** $N_{linhas} \leq 3$ **then**
9: $F_{Stair} = false$, $L_{3D} = \emptyset$, $P_{Out} = \emptyset$
10: **else**
11: Extrair de P_{In} os pontos 3D correspondentes as linhas L_{3D}, gerando P_{out}
12: Ajustar um plano Pl_{temp} aos pontos de P_{out}, por mínimos quadrados
13: Calcular $\alpha_{temp} = \arccos \vec{n}_{Pl} \cdot \vec{n}_{horz}$
14: **if** $0 < \alpha_{temp} < \alpha_{max}$ **then**
15: $F_{Stair} = true$
16: **else**
17: $F_{Stair} = false$, $L_{3D} = \emptyset$, $P_{Out} = \emptyset$
18: **end if**
19: **end if**

Assim, a *depth image*, ou imagem de profundidade, exibe em cada *pixel* um nível de intensidade em escala de cinza. O nível de intensidade é proporcional a distância do sensor ao objeto representado pelo pixel em questão. Devido às diferenças de profundidade de degraus consecutivos, existe uma abrupta varição de intensidade observada na imagem, sendo o princípio de exploração do método. Na figura 4.2a pode-se observar exatamente do que se trata.

Como pode-se observar, a *depth image*, é uma extração ruidosa. Esse ruído é prejudicial a detecção e principalmente a modelagem. Assim, para atenuar os efeitos dos ruídos, serão feitas diversas amostras e detecções nas imagens retiradas do *kinect*.

Com a imagem em mãos, é possível aplicar o método de extração de bordas *Canny*

(Canny [2]), no passo 1, evidenciando as bordas da imagem. A imagem resultado é uma imagem binária contendo somente as bordas da imagem, como pode-se ver na figura 4.2b (E_{image}). Em seguida, no passo 2, o método *Standard Hough Transform* (SHT) (Duda and Hart [7]) é aplicado à E_{image} a fim de extrair as linhas das bordas da imagem. As linhas da borda da escada, independentemente da posição relativa do *kinect*, formam um conjunto de retas paralelas no espaço. Apesar da perspectiva da imagem, certo grau de tolerância ainda é permitido, já que a detecção é feita utilizando as coordenadas da imagem, e não a representação do real. Essa sequência pode ser bem observada na figura 4.2.

(a) Imagem de profundidade da escada pela frente

(b) Imagem depois da detecção de bordas *Canny*

(c) Imagem com as linhas horizontais obtidas pela transformada de *Hough*

Figura 4.2: Passo a passo do método de detecção das linhas

Como somente as linhas horizontais interessam para a extração, já que se trata das bordas dos degraus, somente linhas com angulação entre $-5°$ e $5°$ são interessantes. Depois, as linhas colineares são aglutinadas de forma a representarem melhor a escada (passo 3).

Assim que as linhas forem rigorosamente extraídas, uma análise estatística de frequência de linhas nas colunas da imagem é realizada (passo 4), a fim de extrair somente as linhas verticais referentes à escada. As linhas da escada aparecem uma sobre a outra repetidamente, com um padrão de defasagem. Assim, as colunas que tiverem o maior número de linhas são consideradas as colunas que localizam a escada na figura, como visto no passo 5. Esse passo é ilustrado na figura 4.3.

Logo após, as linhas são redimensionadas para estarem dentro dos limites, como determinado no passo 6. Caso o número de linhas detectadas depois de todos esses filtros seja maior ou igual a 3 (passo 8), pode-se finalmente fazer a correspondência com os dados de referência das coordenadas reais, medida em metros, e não das coordenadas da imagem, representada em *pixels*.

Com essas linhas, pode-se, finalmente, extrair do *point cloud* os pontos referentes as linhas (passo 11). Cada ponto é agrupado de acordo com sua linhas respectiva e compõe um *point cloud* específico para a escada. Em seguida, no passo 12, um plano é ajustado por mínimos quadrados nos pontos extraídos, para medição da angulação do plano em relação ao solo. Só aí que, caso tenha uma angulação dentro dos padrões esperados, tanto para caracterização de uma escada quanto para a transposição da mesma, é que o conjunto

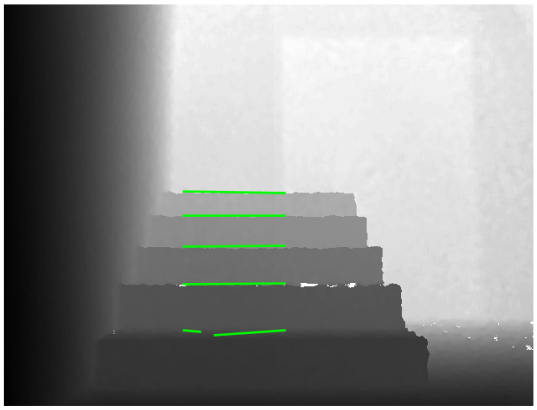
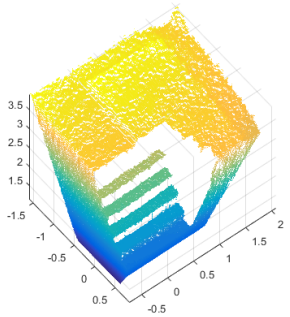
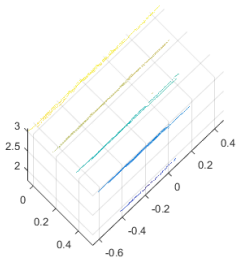


Figura 4.3: Linhas da escada

de pontos pode ser classificado como uma escada (passo 13). No caso positivo, os dados são armazenados para a modelagem (passo 15), em caso negativo, são descartados e uma nova amostragem é analisada (passo 17).



(a) *Point cloud* de entrada do algoritmo P_{In}



(b) *Point cloud* de saída do algoritmo P_{Out}

Figura 4.4: Passo a passo do método de detecção das linhas

Após um considerável número de capturas, obtém-se uma nuvem de pontos densa. Aplica-se um filtro baseado no método *RANSAC* (do inglês, *RANdom SAmple Consensus*) para remoção de ruídos, estando preparada para o próximo, e último, passo do método: a modelagem da escada.

Para a modelagem, utiliza-se um modelo geral que fornece altura e profundidade médias dos degraus, bem como inclinação do plano da escada, largura e número de degraus da escada, e sua orientação.

a) P_{Out}: *point cloud*, ou nuvem de pontos, correspondentes ás bordas dos degraus da escada. Gerado no pseudocódigo anterior. *b*) L_{3D}: *array*, ou lista, de pontos das bordas de cada degrau, separado por degrau. Gerado no pseudocódigo anterior. *c*) S_{Model}: modelo da escada gerado a partir de P_{Out} e L_{3D}. *d*) Pl_{Model}: moledo do plano ajustado por mínimos quadrados, dos pontos de L_{3D}. *e*) L_{proj}: pontos das linhas extraídas, L_{3D}, projetados sobre

o plano ajustado, Pl_{Model}. *f*) L_i: pontos das linhas de L_{proj} agrupados, e ordenados, por degrau. Também contém informações do vetor diretor de cada reta. *g*) C_i: centróides de cada degrau, obtido através de L_i *h*) h_s: altura média dos degraus, obtido usando as diferenças de altura dos centróides consecutivos de C_i. *i*) d_s: profundidade média dos degraus, obtido usando as diferenças de profundidade dos centróides consecutivos de C_i. *j*) W_s: Largura da escada, extraído como o maior comprimento dentre as retas L_i. *k*) H_s: Altura da escada, extraído como a diferença de altura entre os primeiro e último centróides C_i. *l*) D_s: Profundidade da escada, extraído como a diferença de profundidade entre os primeiro e último centróides C_i.

Pseudocódigo 4.2 Modelagem da escada

Entrada: P_{Out}, L_{3D}
Saída: S_{Model}, Pl_{Model}

1: Ajustar um plano Pl_{Model} nos pontos de P_{Out} e calcular seu ângulo com o plano horizontal, α_{Pl}
2: Agrupar os pontos de L_{3D} por degrau e projetar cada ponto sobre o plano Pl_{Model}, gerando L_{proj}
3: Ajustar uma reta para cada grupo de pontos, gerando L_i
4: Projetar os pontos nas retas ajustadas para extrair os limites de cada degrau
5: Achar o centróide, C_i, de cada reta L_i,
6: Ordenar os centróides por altura e calcular as dimensões dos degraus (h_s, d_s), através das médias das diferenças de altura e profundidade dos centróides
7: Obter os limites da escada de acordo com os limites das retas dos degraus. A maior dimensão entre todos gera W_s e as diferenças de altura e profundidade entre o primeiro e último centróides geram H_s e D_s
8: Com as informações obtidas $(L_i, C_i, h_s, d_s, W_s, H_s, D_s)$, gerar o modelo da escada S_{Model}

Ajusta-se um plano inclinado nos pontos extraídos, agora não em cada amostra, mas no conjunto denso de pontos (passo 1). Em seguida, projetam-se os pontos extraídos sobre o plano ajustado (passo 2) e retira-se os limites da caixa delimitadora da escada, B_l (passos 3 e 4). Agora, calcula-se os os centróides de cada linha da borda do degrau, e pela defasagem entre os centróides extrai-se a profundidade e altura de cada degrau (passo 5 e 6). Com uma média, obtém-se a altura e profundidade média dos degraus (passo 7).

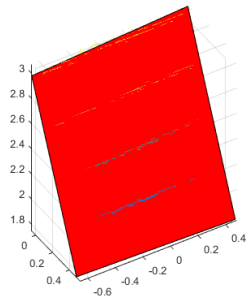


Figura 4.5: Plano ajustado da escada

4.1.3 Experimentos e Resultados

O método descrito anteriormente foi validado com dados reais, utilizando o sensor *kinect*. Alguns dos principais scripts podem ser encontrados no apêndice B, como *DetectStairsKinect* e *AjustaPCloud*.

Para os testes realizados, o sistema de coordenadas inercial O_b foi escolhido que sua origem estivesse posicionada na escada, como ilustra a figura 4.6. A origem do sistema de coordenadas do *kinect* é representado por O_k.

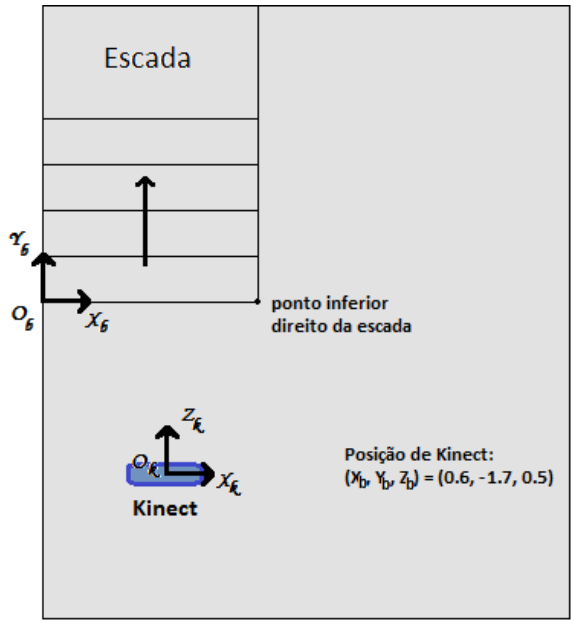


Figura 4.6: Sistemas de coordenadas do experimento

Na tabela 4.3 tem-se um comparativo dos dados obtidos pelo método e o real.

Dado	**Experimental**	**Real**	**Erro**
Altura média dos degraus (m)	0.134	0.17	0.036
Profundidade média dos degraus (m)	0.294	0.30	0.006
Ângulo de inclinação do plano da escada	24.36°	29.53°	5.17°
Altura da escada (m)	0.536	0.68	0.144
Largura da escada (m)	1.105	1.20	0.095
Profundidade da escada (m)	1.178	1.20	0.022

Tabela 4.3: Dados experimentais e reais da extração do *kinect*

Devido ao visível ruído do *kinect*, as medições obtidas, principalmente na altura, sofreram grandes desvios. Observa-se que o erro médio é de cerca de 6 cm. Um erro considerável, visto que está na mesma ordem de grandeza que a altura do degrau.

Entretanto, o erro de inclinação do plano não é tão considerável assim. Principalmente considerando a faixa baixa encontrada na escada. Variações de 5° mais perto da vertical são mais perceptíveis que perto da horizontal.

Os resultados foram satisfatórios para efeito de transposição de obstáculos. Entretanto, o erro agregado ao sensor é considerável, sendo necessário mais capturas para se ter uma massa de dados satisfatória para modelagem. Quando comparado com o esperado de uma escada, segundo 4.2, a altura é a mais comprometida, não podendo nem ser classificada como escada, segundo os resultados.

Contudo, sua aplicação em tempo real é questionável. Como o algoritmo foi desenvolvido para validação do software e ter-se base para o desenvolvimento da modelagem e controle reais, seu desempenho não foi priorizado. Além disso, a quantidade massiva de dados torna a iteração sobre os pontos da imagem muito dispendiosa. Esse cenário só piora quando é necessário que várias capturas sejam feitas para atenuar do ruído da imagem. Somado a isso, existe o atraso da comunicação, principal vilão de qualquer sistema robótico, que limita o tráfego de dados e demora alguns segundos (5.6 segundos, em média), para passar uma única captura do *kinect* no formato *point cloud*. Talvez com informações comprimidas e uma linguagem mais rápida, como o próprio C++, sejam capazes de acelerar o algoritmo a ponto de tornar possível embarcá-lo e aplicá-lo em tempo real.

4.2 Metodologia para locomoção

Agora que o ambiente em que se encontra o robô móvel DIANE foi interpretado, é possível atuar sobre o robô e de fato explorar.

O problema de subida na escada foi separado em 2 partes: transição para entrada e saída da escada; e controle sobre a escada. Esta secção tratará da primeira parte somente, as transições.

São duas transições a serem realizadas: do piso inferior à escada; e da escada ao piso superior. Para cada transição foi desenvolvido um algoritmo que executa a mudança, baseado em de Lima [5].

O método se baseia na definição de arranjos do movimento dos braços, por cinemática direta. Portanto, a partir dos ângulos das juntas, é possível determinar as posições das extremidades dos braços, além de ponto de contato entre os braços e a escada.

4.2.1 Piso inferior à escada

A transição do piso inferior à escada é a primeira etapa do processo de locomoção sobre escadas. O robô deve-se encontrar próximo à escada, com suas orientações alinhadas, a fim de iniciar o movimento de subida.

A configuração dos braços escolhida foi a de ângulação máxima permitida para que o robô suba os obstáculos. Essa altura é discutida em de Lima [5]. A figura 4.7 ilustra o a posição do braço relativa ao primeiro degrau da escada.

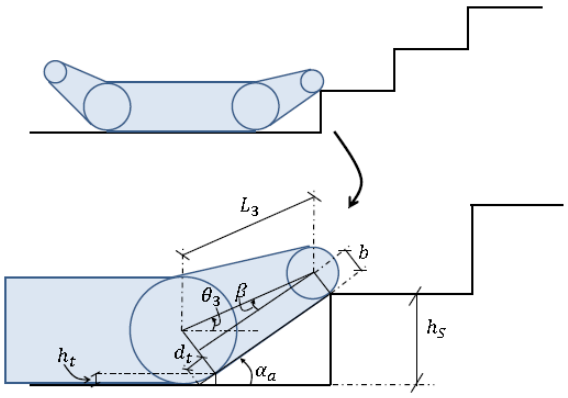


Figura 4.7: Representação do braço (perna) robótico em contato com o primeiro degrau da escada Fonte: de Lima [5] pp. 83

onde:

a) θ_3: ângulo de rotação da junta da perna frontal. *b*) β: ângulo entre o eixo principal da perna e a reta formada pela esteira. *c*) α_a: ângulo entre a esteira e o plano do solo. *d*) h_t, d_t: parâmetros temporários para auxiliar os cálculos. *e*) L_3: comprimento da perna. *f*) H_R: altura do robô. *g*) b: raio da circunferência menor que compõe a perna robótica. *h*) h_s: altura do degrau.

Com as informações do robô em mãos, pode-se inferir sobre a capacidade de superar o obstáculo em questão, pela comparação com a altura de degrau h_s obtido por 4.2.

Sabendo se o robô será capaz de transpor o obstáculo, pode-se iniciar a locomoção sobre a escada. A etapa de entrada na escada, ou transição do piso inferior à escada se dá por meio de um algoritmo de 3 partes.

Essas três partes do algoritmo, ou subtarefas, são separadas para melhor auxiliar o entendimento e estabelecer uma estratégia para resolução do problema de transição. As três etapas são denominadas S1-A, S1-B e S1-C. Estas apresentam um conjunto de ações que juntas compõem o algoritmo de transição.

Fase S1-A

Nesta fase o robô se encontra numa posição de repouso, inicialmente. Para iniciar a transição, os braços devem estar na posição de máxima angulação, $\theta_{3_{max}}$, para que seja garantido o avanço sem problemas. O robô então inicia o avanço em direção à escada, estabelecendo contato com o primeiro degrau.

O contínuo avanço do robô nesta configuração garantirá a elevação do corpor principal, agora em contato com a borda do primeiro degrau. Assim que os sensores inerciais detectarem a excedência do ângulo do robô ao ângulo da escada, o avanço das esteiras é interrompido e inicia-se a fase S1-B.

O algoritmo é encontrado no apêndice B, com nome *Climb_1*.

Fase S1-B

Caso o avanço continue, o aumento de α_R pode fazer com que aconteça uma queda brusca do robô sobre a escada, no momento que o centro de gravidade do robô ultrapassar o ponto de contato com a borda do primeiro degrau.

Para evitar esse cenário, o centro de massa será deslocado o máximo possível para trás, com a movimentação do braço traseiro exclusivamente. A posição do braço traseiro esticado para trás, ou seja, $\theta_2 = 0$, garante que mesmo que o robô acabe tombando, este será o mais suave possível, além de diminuir sua angulação, tornando o avanço angular menos proeminente. Como veremos na secção posterior, 4.2.2, o deslocamento do centro de massa e o tombamento do robô são de suma importância para a manutenção, e preservação, do robô.

O algoritmo é encontrado no apêndice B, com nome *Climb_2*.

Fase S1-C

Depois de reduzir sua angulação, e garantir o movimento mais suave possível, as esteiras retomam movimento, impulsionando o robô. Este movimento linear deve continuar até que a angulação do robô seja igual à ângulação da escada, ou seja, $\alpha_R = \alpha_{Pl}$.

Devido a erros de detecção possíveis do plano da escada, outra condição deve ser adicionada. A aproximação de α_R a α_{Pl} indica que o robô está no fim, mas também monitorando a variação de α_R no tempo, checando se sua derivada é superior à um valor de tolerância experimental, pode indicar que o robô já entrou na escada.

Desse modo, o robô entra na escada de forma segura e controlada. A próxima etapa, o deslocamento sobre o plano da escada, merece um capítulo a parte, devido à sua complexidade e inúmeras discussões acerca da problemática de locomoção em planos inclinados, e possíveis obstáculos no caminho.

O algoritmo é encontrado no apêndice B, com nome *Climb_3*.

4.2.2 Aterrisagem no nível superior

O tombamento observado, quando o centro de massa do robô ultrapassa a linha de contato da borda da escada, pode gerar danos à operação do robô, ao chassi e às pernas do robô, sem mencionar os danos a eletrônica embarcada. Quando o centro de massa ultrapassa a borda da escada, o momento resultante da força gravitacional acelera o robô em direção ao solo, podendo gerar danos irreversíveis. A figura 4.8 ilustra a situação de limite descrita.

Para evitar estes danos, um algoritmo foi idealizado, capaz de minimizar os danos. Este algoritmo foi baseado na movimentação das pernas para uma posição que minimize a velocidade resultante no momento do impacto.

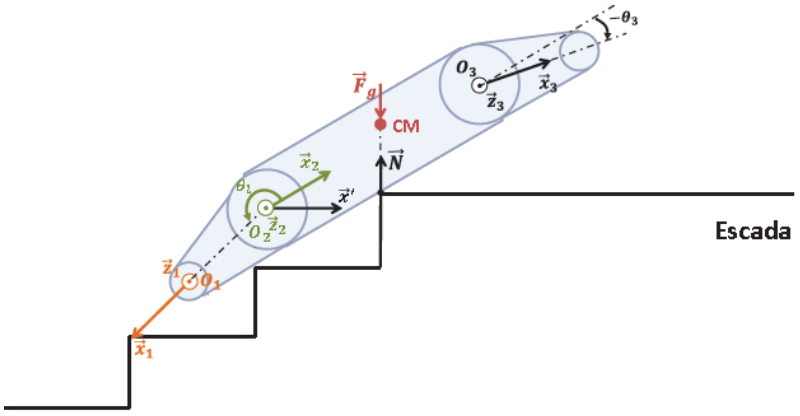


Figura 4.8: Ilustração da iminência do tombamento do DIANE ao chegar ao topo da escada Fonte: de Lima [5] pp. 91

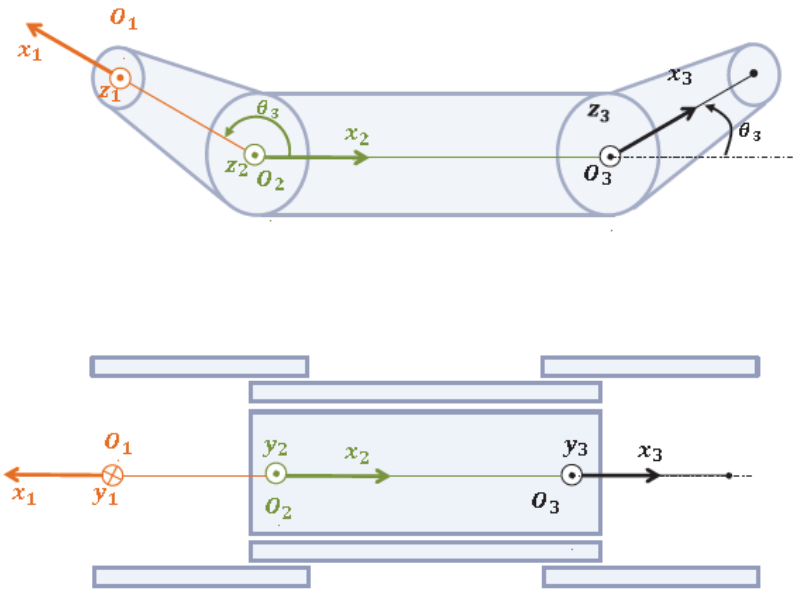


Figura 4.9: Vista superior e lateral dos sistemas de coordenadas dos corpos rígidos que compõem o robô Fonte: de Lima [5] pp.88

O robô é dividido em três corpos rígidos, braços traseiros, chassi, e braços dianteiros, de modo que cada corpo rígido apresenta seu próprio centro de massa. Dessa forma, é fácil determinar os centros de massa de cada corpo em separado. A posição do centro de massa do corpo como um todo fica sendo função exclusivamente do arranjo dos braços. O desenvolvimento da metodologia pode ser encontrado em de Lima [5].

$$(p_{CM})_2 = \begin{bmatrix} (L_1 - l_{c_{1x}})\cos(\theta_2)m_{c_1} + l_{c_{2x}}m_{c_2} + (L_2 + l_{c_{3x}}\cos(\theta_3))m_{c_3} \\ (L_1 - l_{c_{1x}})\sin(\theta_2)m_{c_1} + l_{c_{2y}}m_{c_2} + (L_2 + l_{c_{3x}}\cos(\theta_3))m_{c_3} \\ l_{c_{2z}}m_{c_2} \end{bmatrix} \frac{1}{m_c} \tag{4.1}$$

A partir disso, temos:

$$p_{N_x} = -\frac{H_R}{2}\tan(\alpha_s) + \Big[(L_1 - l_{c_{1x}})\cos(\alpha_s+\theta_2)m_{c_1} + l_{c_{3x}}\cos(\alpha_s+\theta_3)m_{c_3} + L_2\cos(\alpha_s)m_{c_3} \quad -l_{c_{2y}}\sin(\alpha_s)m \tag{4.2}$$

Logo, quando p_{N_x} for menor que a expressão encontrada em 4.2 o momento resultante no corpo do DIANE provoca a rotação do robô em torno do ponto de contato com o último degrau, em direção ao solo.

Basta identificar a chegada da última borda da escada, para iniciar este algoritmo, utilizando um método de localização ou um *laser* na parte debaixo do DIANE. O método proposto em de Lima [5] utiliza um avanço gradual, de acordo com a variação de *pitch* do robô, devido ao contato da perna frontal com o piso superior. Entretanto, o resultado é o mesmo quando se eleva o máximo possível, sem passar do limite de *pitch* do robô, e se reconfigura a posição do braço frontal para $-\theta_{3_{max}}$, para depois avançar. Quando se detecta que não existe mais alteração do *pitch* de acordo com o avanço, o mesmo começa um movimento de descida, pousando sobre o solo de forma suave, fazendo $\theta_3 = 0$, enquanto continua avançando.

Este algoritmo foi descrito e é melhor entendido quando apresentado como está em 4.3.

Pseudocódigo 4.3 Aterrisagem sobre o andar superior

```
1: θ3 = −θ3max
2: while αR ! = 0 do
3:     v ≠ 0
4:     θ3 = 0
5: end while
```

4.2.3 Experimentos e Resultados

A experimentação deste e do próximo passo foram realizadas somente na bancada. Isso quer dizer que o robô não foi colocado sobre a escada para validação do algoritmo. Entretanto, como as variáveis chave da rotina são as posições dos braços dianteiros e traseiros e o ângulo de inclinação do robô, pôde-se fazer testes sobre a bancada e observar a movimentação do robô.

A validação da posição dos braços pode ser feita com o retorno dos dados fornecidos pelas EPOS, aliado à inspeção visual. A posição dos braços na primeira parte do algoritmo, teve um erro de 0.008 (0.5° de incerteza) do ângulo observado. Esse valor foi obtido por instrumentos de inspeção visual, então o erro é devido a incerteza do instrumento.

A variação do ângulo do robô foi conseguida por meio de alteração manual do ângulo do *kinect*. Ao iniciar o movimento de cada etapa, o usuário alterou manualmente o eixo

do *kinect* e simulou a subida. Como este algoritmo não está preocupado com sinais de controle ou respostas no tempo, não se constatou problema nessa simulação.

4.3 Controle para locomoção em escadas durante a subida

O objetivo do controle é realizar o deslocamento do robô durante a subida, mantendo sua posição equidistante em relação às laterais da escada e a orientação no valor desejado. Deste modo, a solução do problema se baseia no rastreamento de trajetórias, utilizando um controle cinemático.

Assim, em de Lima [5], o autor obtém seu modelo a partir de um robô com duas rodas diferencial. Essa aproximação leva à relação da equação 2.1. Como a configuração do robô móvel com esteiras em movimento planar é determinada pelas coordenadas de posição $p = [xy]^\intercal$ e orientação ϕ, pode-se chegar a relaçõ (de Lima [5]):

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 \\ \sin(\phi) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{4.3}$$

Assim, utilizando as variáveis de estado s, l e e_ϕ, e $c(s)$ a trajetória desejada, têm-se a dinâmica do sistema:

$$\begin{aligned} \dot{s} &= v\cos(e_\phi)\left(\frac{1}{1-c(s)l}\right) \\ \dot{l} &= v\sin(e_\phi) \\ \dot{e_\phi} &= \omega - v\cos(e_\phi)\left(\frac{c(s)}{1-c(s)l}\right) \end{aligned} \tag{4.4}$$

Mas no caso proposto, basta uma trajetória retilínea para manter o robô em trajetória equidistante em relação às paredes e com orientação na direção de subida, y_s (ver figura 4.10). Assim, a dinâmica é reduzida à:

$$\begin{aligned} \dot{l} &= v\sin(e_\phi) \\ \dot{e_\phi} &= \omega \end{aligned} \tag{4.5}$$

Durante o deslocamento, o operador pode alterar a velocidade linear $v(t)$ do robô, sendo um sinal externo de entrada do sistema. A figura 4.11 ilustra o controle em malha fechada.

Assim, a lei de controle proposta é definida como (de Lima [5]), onde k e k_2 são constantes maiores que zero:

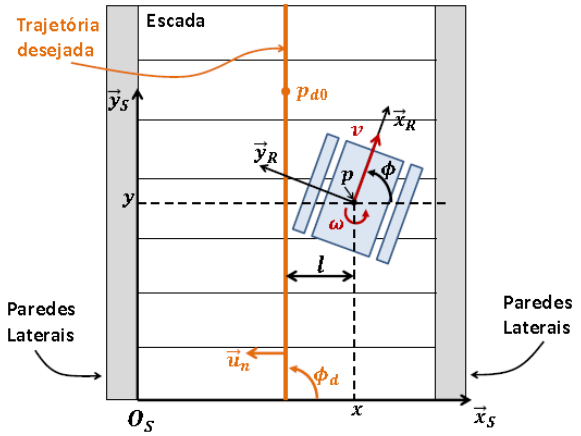


Figura 4.10: Vista superior de um robô tipo EOD realizando rastreamento de trajetória retilínea em escadas Fonte: de Lima [5] pp. 124

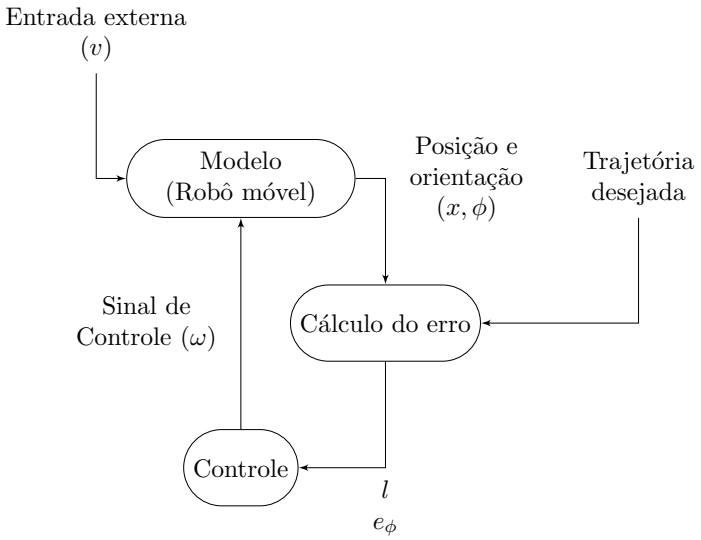


Figura 4.11: Diagrama de blocos do controle aplicado a locomoção de escadas Fonte: de Lima [5] pp.126

$$\omega = -k_2 v l \frac{\sin(e_\phi)}{e_\phi} - k e_\phi \tag{4.6}$$

O controle não foi testado experimentalmente, já que o robô não estava pronto. Entretanto, as simulações e considerações a respeito do controle, em situações diferentes, pode ser encontrado em de Lima [5].

Assim, para a implementação deste controle, é necessário a criação de um novo modo de controle no *nó diane_controller* para aceitar a informação de velocidade linear $v(t)$ vindo do *joystick* e velocidade angular $\omega(t)$ vindo do controle.

Este controle deve enviar a mensagem para o tópico *cmd*, do *nodelet diane_remap* e o *diane_controller* aceitar as duas origens para o comando.

Capítulo 5

Conclusões e trabalhos futuros

Este trabalho apresentou uma filosofia de software aplicada aos robôs móveis. O intuito do projeto é fazer o robô DIANE subir a escada assistido pelo computador. Este trabalho focou numa parte desse projeto que é base para todo o desenvolvimento do trabalho, o software.

Os robôs móveis estão sendo objetos de estudo cada vez mais aprofundados e aplicados. Para aplicações de resgate e exploração, os robôs do tipo EOD, como o DIANE, são muito utilizados. Esta aplicação se deve pela robustez e resiliência do tipo de robô, que devido a forma de contato com o solo, por meio de esteiras, torna a modelagem mais fácil que outros modelos e emprega um movimento mais estável.

O software desenvolvido teve como principal característica a escalabilidade. Isso tornou fácil a integração com diversas formas de controle. A teleoperação, apesar de limitada à visão da câmera, foi possibilitada por meio do sensor *kinect*. A assistência pelo computador não pôde ser testada pois o robô não estava pronto até a data da defesa. Entretanto, o software foi devidamente testado e comprovada sua eficiência na aplicação de comandos na locomoção, com envios de pacotes que responderam em menos de um segundo.

O método de detecção se mostrou eficaz, porém ineficiente. Devido ao tempo de resposta, a modelagem da escada demora cerca de um minuto. Esse tempo, para aplicações em tempo real, impossibilita seu uso embarcado. Os erros agregados a detecção da escada são aceitáveis, mas sucessivas amostragens atenuam seus efeitos. Apesar disso, a abordagem da modelagem da escada por um plano inclinado ajuda muito o controle, que não se preocupa mais com irregularidades, mas com o movimento num plano, muito mais simples de modelar e controlar.

As reconfigurações da postura do DIANE se mostraram satisfatórias, visto que seu comportamento se assemelhou ao esperado. Um algoritmo geral para transições em obstáculos desse tipo torna sua contribuição importante.

Já o controle desenvolvido precisa de mais testes, de preferência com a situação real de subida. Sua falta de testes dificulta a discussão dos resultados, mas, através das simulações realizadas em de Lima [5], o controle se mostra promissor. A modelagem cinemática

do problema auxiliou muito seu entendimento e a atuação sobre o robô.

5.1 Trabalhos futuros

Este trabalho contribuiu, principalmente, para o projeto com o desenvolvimento do software de controle do robô. Como forma de continuidade do trabalho, deve se implementar o controle de centralização e orientação e testá-lo m sitações reais.

Poderia ser implementado, também, a geração de mapas, bem como o reconhecimento dos obstáculos por meio desse arquivo de mapa guardado.

Outro trabalho é a implementação da detecção e modelagem de escadas em outra linguagem, agora priorizando o desempenho. A ideia por trás do algoritmo de detecção pode ser aplicada de diversas formas, gerando conhecimento.

Ainda pode-se contribuir de forma que o controle dinâmico do robô possa corrigir qualquer erro de posicionamento. O controle cinemático foi válido mas é limitado quanto às condições que restringem a reutilização em qualquer outro código. Os robôs devem ter características semelhantes ao robô para que seja aplicado em outras situações.

Outro campo de estudo é o desenvolvimento de algoritmos que reconheçam escadas descendentes. Apesar de citados trabalhos que reconheçam essas escadas, além do próprio trabalho desenvolvido em de Lima [5], não se aplicam à situação do DIANE devido a altura que o sensor RGB-D fica do solo e da falta do manipulador. Assim, se faz necessário para a exploração de ambientes urbanos um método de reconhecimento e modelagens de escadas descendentes.

Outra possibilidade de estudo é o controle dinámico do robô. Apesar de considerações cinemáticas serem feitas, um controle que leve em consideração as forças atuantes sobre o sistema robótico é mais robusto. Para isso, ainda se deve realizar um estudo das forças que atuam sobre o DIANE e como se comportam com o movimento e variação de ângulo em relação ao solo.

Referências Bibliográficas

[1] Beno, P., Duchon, F., Tölgyessy, M., Hubinsky, P. and Kajan, M. [2014], '3d map reconstruction with sensor kinect', *IEEE International Conference on Robotics in Alpe-Adria-Danube Region* pp. 1–6.

[2] Canny, J. [1986], 'A computational approach do edge detection', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-8**(6), 679–698.

[3] Carbonara, S. and Guaragnella, C. [2014], 'Efficient stair detection algorithm assisted navigation for vision impaired people', *IEEE International Symposium on Innovations in Intelligent Systems and Applications* pp. 313–318.

[4] Colas, F., Mahesh, S., Pomerleau, F., Liu, M. and Siegwart, R. [2013], '3d path planning and execution for search and rescue ground robots', *IEEE/RSJ International Conference on Intelligent Robots and Systems* pp. 722–727.

[5] de Lima, L. C. [2016], Locomoção semiautônoma em escadas para robôs móveis com esteiras, M.Sc. dissertation, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

[6] Delmerico, J. A., Baran, D. and David, P. [2013], 'Ascending stairway modeling from dense depth imagery for traversability analysis', *IEEE International Conference on Robotics and Automation* pp. 2283–2290.

[7] Duda, R. O. and Hart, P. E. [1972], 'Use of the hough transformation to detect lines and curves in pictures', *Commun. ACM* **15**(1), 11–15.

[8] Endres, F., Hess, J., Engelhard, N., Sturm, J., Cremers, D. and Burgard, W. [2012], 'An evaluation of the rgb-dslam system', *IEEE International Conference on Robotics and Automation* pp. 1691–1696.

[9] Guerrero, J., Gutiérrez-Gómez, D., Rituerto, A., López-Nicola, G. and Pérez-Yus, A. [2015], Human navigation assistance with a rgb-d sensor, *in* 'VI Congreso Internacional de Diseño, Redes de Investigación y Tecnología para todos DRT4ALL', pp. 286–311.

[10] Helmick, D. M., Roumeliotis, S. I., McHenry, M. C. and Matthies, L. [2002], 'Multi-sensor, high speed autonomous stair climbing', *IEEE/RSJ International Conference on Intelligent Robots and Systems* pp. 733–742.

[11] Kalantari, A., Mihankhah, E. and Moosavian, S. A. A. [2009], 'Safe autonomous stair climbing for a tracked mobile robot using a kinematics based controller', *IEEE/ASME International Conference on Advanced Intelligent Mechatronics* pp. 1891–1896.

[12] Li, N., Ma, S., Li, B., Wang, M. and Wang, Y. [2012], 'An online stair-climbing control method for a transformable tracked robot', *IEEE International Conference on Robotics and Automation* pp. 923–929.

[13] Loureiro, G. S. M. [2017], Desenvolvimento do software para posicionamento dinâmico do rov luma, Graduate dissertation, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil.

[14] Luo, R. C., Hsiao, M. and Liu, C. [2013], 'Multisensor integrated stair recognition and parameters measurement system for dynamic stair climbing robots', *IEEE International Conference on Automation Science and Engineering* pp. 318–323.

[15] Mihankhah, E., Kalantari, A., Aboosaeedan, E., Taghirad, H. D. and Moosavian, S. A. A. [2009], 'Autonomous staircase detection and stair climbing for a tracked mobile robot using fuzzy controller', *IEEE International Conference on Robotics and Biomimetics* pp. 1980–1985.

[16] Mourikis, A. I., Trawny, N., Roumeliotis, S. I., Helmick, D. M. and Matthies, L. [2007], 'Autonomous stair climbing for tracked vehicles', *The International Journal of Robotics Research* **26**(7), 737–758.

[17] Valenti, R. G., Dryanovski, I., Jaramillo, C., Ström, D. P. and Xiao, J. [2014], 'Autonomous quadrator flight using onboard rgb-d visual odometry', *IEEE International Conference on Robotics and Automation* pp. 5233–5238.

[18] Wagner, D., Kalischewski, K., Velten, J. and Kummert, A. [2015], 'Detection of ascending and descending staiways by surface normal vectors', *IEEE 9th International Workshop on Multidimensional Systems* pp. 1–5.

[19] Wang, S., Pan, H., Zhang, C. and Tian, Y. [2014], 'Rgb-d image-based detection of stairs, pedestrian crosswalks and traffic signs', *Journal of Visual Communication and Image Representation* **25**(2), 263–272.

[20] Wang, W., Wu, D., Wang, Q., Deng, Z. and Du, Z. [2012], 'Stability analysis of a tracked mobile robot in climbing stairs process', *IEEE International Conference on Mechatronics and Automation* pp. 1669–1674.

Apêndice A

Configurações do DIANE

O robô do tipo *EOD* DIANE foi desenvolvido pelo Laboratório de Controle da COPPE/UFRJ, encomendado pela empresa ARES. o DIANE é peça fundamental para implementação dessa pesquisa no laboratório. Agora, com o desenvolvimento e estabelecimento do *ROS* no mercado, o *software* do DIANE foi refeito nessa plataforma. Desenvolvido com as bibliotecas do Qt, e na linguagem C++, seu software se tornou mais robusto, reutilizável, encapsulado, eficiente e mais fácil de ser modificado/atualizado.

O projeto do DIANE consiste em braços robóticos dianteiros e traseiros, capazes de auxiliar na transposição de obstáculos e em esteiras laterais e nos braços, fazendo com que se movimento linearmente ou angularmente, devido ao movimento associado de cada esteira.

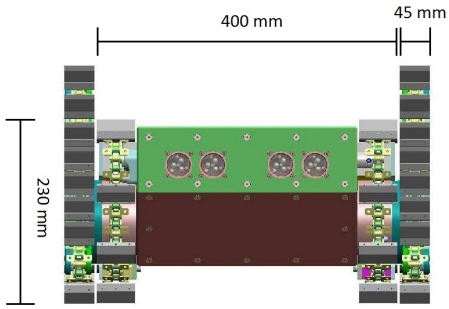
Segue, na figura A.1, as vistas lateral, superior, frontal e perspectiva, a fim de dar as dimensões do robô.

Outras informações que não foram passadas nos desenhos é a dos raios maior e menor dos braços do robô móvel. Sejam eles R e r, respectivamente maior e menor raios dos braços, seu valor é $R = 100mm$ e $r = 20mm$.

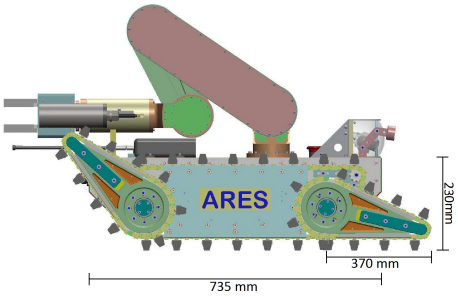
Para movimentar o sistema robótico como um todo, são necessários 4 motores: um para o braço dianteiro; para o braço traseiro; para a esteira direita; e para esteira esquerda. Para que o movimento de cada par fique condizente foram escolhidos 2 pares de motores diferentes, 2 para as esteiras, com as mesmas características mecânicas e elétricas, e outros 2, diferentes dos primeiros, para os braços. A tabela A.1 descreve algumas das informações importantes para configurações posteriores, modelagens ou cálculos de ganhos.

Para controlar esses motores escolhidos, foram selecionadas 2 *drivers* diferentes, da mesma empresa, EPOS 24/5 e EPOS 70/10. Depois de corretamente configurada, através de comandos digitais pelos computador, podemos controlar os motores a vontade, visto que suas características podem ser todas salvas no *driver* e este se encarrega de manter as condições ideais de uso.

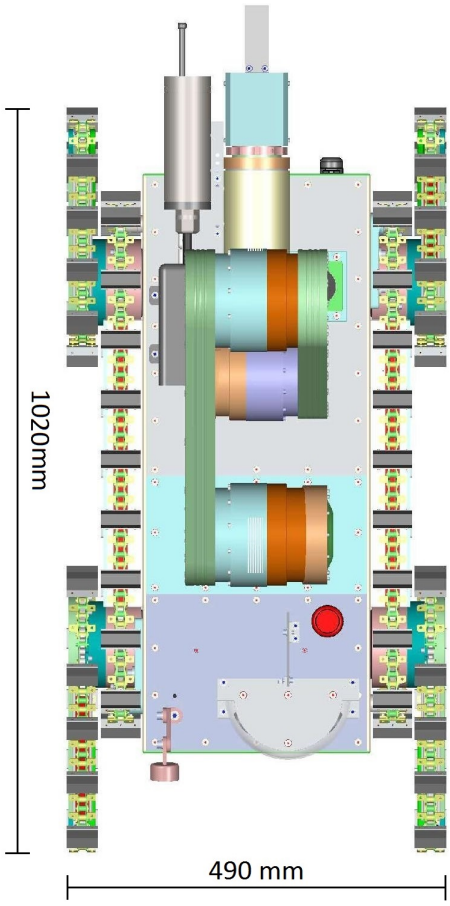
Apesar dessa característica dos *drivers* selecionados, foram tomadas algumas pre-



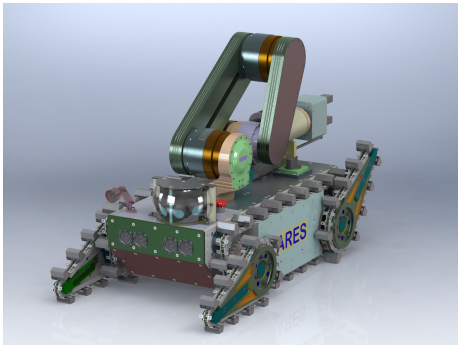
(a) Vista Frontal do DIANE



(b) Vista Lateral Direita do DIANE



(c) Vista Superior do DIANE



(d) Projeto em SolidWorks original do DIANE.

Figura A.1: Vistas do projeto original do DIANE.

caução̧es quanto ao limite dos motores. Na tabela A.2, pode-se checar as configurações essênciais ao funcionamento do motor. Também na mesma tabela, pode-se verificar os objetos em código hexadecimal para confiiguração da mesma. Estes códigos se fazem necessários uma vez que a comunicação do computador com os *drivers* é feito utilizando o barramento *CAN* (*Controller Area Network*). Este tipo de comunicação é excelente para controle em tempo real tendo um rigoroso tratamento de erro e recebimento das mensagens.

Chegando ao computador, primeiro objeto de atualização no robô móvel, o PC PUMA, que utilizava o sistema operacional QNX foi substituído por um ADLQM67PC, com pro-

cessador i7 quad-core de segunda-geração. Junto com 4 Gb e um SSD (*Solid State Drive*) o novo computador embarcado roda um sistema operacional Ubuntu, baseado em Linix. Além da vantagem de ser gratuito, muitas aplicações robóticas atualmente utilizam plataformas baseadas em Unix. Já a escolha do Ubuntu foi feita pela estabilidade, suporte e familiaridade com o ROS.

Rodando nativamente no Ubuntu, o ROS (*Robot Operating Systems*) é um framework voltado para utilização em robótica. Sua grande vantagem é uma interface e uma estrutura facilmente configurável e inteligível, além de tornar a troca de mensagens uma tarefa simples.

A.1 Sensoriamento

Agora que a atuação do DIANE foi descrita, pode-se cobrir a parte de sensoriamento. A escolha dos sensores foi coberta no capítulo 4.2, mas sua configuração, ajustes e características serão cobertas aqui.

A.1.1 Kinect

O *kinect* foi desenvolvido pela empresa *Microsoft* em conjunto com a companhia *Prime Sense*, para ser utilizado como sensor de movimentos em jogos de videogames. Atualmente, câmera RGB-D são muito utilizadas em aplicações robóticas como localização e mapeamento (Endres et al. [8]), rastreamento e detecção (Guerrero et al. [9]) e navegação autônoma (Valenti et al. [17]).

Composto basicamente por um projetor infravermelho, uma câmera infravermelho e uma câmera RGB, o *kinect* é responsável por tornar possível o mapeamento 3D, utilizando os projetor e câmera infravermelhos, como mostrado na figura 2.4.

Outro elemento utilizado no DIANE, que só faz parte do dispositivo na sua primeira versão, é o acelerômetro. Usado na estabilização e compensação das imagens, a informação do acelerômetro é usada para determinar o ângulo de inclinação (*pitch*) do DIANE.

Simplificadamente, seu princípio de funcionamento se baseia na projeção de padrões. O projetor infravermelho realiza a projeção de um conjunto de padrões no ambiente que apresenta objetos com diferentes formas e profundidades em relação ao sensor.

Os padrões projetados na superfície dos objetos sofrem distorções que variam de acordo com a distância entre a superfície do objeto e o sensor. Assim, os padrões são mapeados e comparados com os padrões de referência e, através de processamento de imagens, as distâncias dos padrões projetados no sensor são obtidos baseadas nas distorções correspondentes.

O *kinect* fornece diferentes tipos de imagens e dados, sendo utilizados neste trabalho somente o *point cloud* e *depth image*. O *point cloud*, como a própria tradução remete,

fornece as cordenadas x, y e z de uma nuvem de pontos correspondentes ao quadro da imagem captada no sensor, que representam as superfícies do ambiente. No caso da *depth image*, o conjunto de dados gerado consiste em uma imagem que tem as cores dos pixels alteradas de acordo com a distância obtida em relação ao sensor. Essa alteração é representada em uma escala predefinida de cinza ou de cores.

No caso do DIANE, este dispositivo está sendo ligado por uma porta digital ligada a uma das EPOS. Sua comunicação com o computador embarcado se dá utilizando uma porta USB. Para ligá-lo, basta chamar o serviço de ROS *d_a_outputs* da epos correspondente, ou seja, */epos4/d_a_outputs*, com: (número da porta - 1) e valor *true*, correspondente ao n'ivel lógico 1, ou 5 V. Esta porta liga um relé na placa de monitoramento que alimenta o *kinect* com 12 V. Mais informações sobre as portas são encontradas na tabela A.3.

A.1.2 Laser

Os sensores *laser scan* são instrumentos largamente utilizados na indústria, em aplicações como navegação, posicionamento e dimensionamento.

Seu funcionamento consiste basicamente na emissão de feixes de luz e na medida do tempo em que o feixe retorna ao sensor, depois de refletido no obstáculo. Com a medida do tempo, e conhecendo a velocidade do feixe emitido, é possivel determinar a distância do sensor ao obstáculo.

A obtenção dos dados em duas dimensões consiste na emissão dos feixes de luz em diversos ângulos em um mesmo plano de escaneamento. Assim, a varredura produzida é um conjunto de distância medida, d_L, e ângulo da varredura, α_L, em relação ao eixo x do sistema de coordenadas O_L do *laser*.

Desta maneira, as coordenadas cartesianas (x_L, y_L) dos pontos refletidos em relação ao sistema de coordenadas O_L do *laser* são dadas pela seguinte relação:

$$\begin{bmatrix} x_L \\ y_L \end{bmatrix} = \begin{bmatrix} d_L \cos \alpha_L \\ d_L \sin \alpha_L \end{bmatrix} \tag{A.1}$$

No DIANE, o laser utilizado é o UST-10LX, da Hokuyo, possuindo uma interface ethernet, conectada à rede do DIANE. Também foi utilizada uma porta digital de uma das EPOS para ligar o dispositivo. Informações sobre as portas podem ser encontradas na tabela A.3. Assim, para ligá-lo, basta chamar o serviço de ROS */epos4/d_a_outputs* com a informacção de (porta - 1) e valor *true*, correspondendo ao nível lógico 1, ou 5 V.

A.2 Comunicação

Existem 2 tipos de protocolos de comunicação embarcadas no DIANE: TCP/IP, para comunicação entre o computador embarcado e outros computadores e entre nós do ROS; e CAN, para comunicação do computador embarcado com as EPOS. O objetivo desta secção não é definir ou explicar cada comunicação, mas sim detalhar as configurações feitas para o robô móvel.

Na parte de comunicação TCP/IP, temos as configurações para comunicação entre computadores e as configurações entre nós do ROS. Na parte de configuração entre computadores, a rede do DIANE possui 3 elementos principais: o ADLQM67PC; o roteador Wi-Fi; e o Switch. A tabela A.5 exibe algumas informações úteis acerca da rede.

Já a estrutura da rede CAN foi montada para que o PC/104 funcionasse como *client*, ou *master*, e as EPOS como *server*, ou *slave*. Cada EPOS recebeu um ID, selecionado manualmente pelas chaves presentes na EPOS, e configurada com um *bitrate* de acordo com a velocidade de processamento ideal para cada componente. Esse *bitrate* tem valores tabelados, de acordo com o manual da EPOS. Na tabela A.6 encontram-se algumas das informações e na figura 2.3, uma ilustação da rede física implementada.

Tabela A.1: Informações dos motores Fonte: www.maxonmotor.com

Specifications\Motor	Maxon EC 45	Maxon EC 32
Values at Nominal Voltage		
Nominal Voltage	48 V	12 V
No load speed	6160 rpm	15100 rpm
No load current	244 rpm	662 mA
Nominal speed	5490 rpm	13400 rpm
Nominal torque (max. continuous torque)	347 mNm	44.4 mNm
Nominal current (max. continuous current)	4.86 A	6.51 A
Stall torque	3530 mNm	428 mNm
Stall current	47.7 A	57.2 A
Max. Efficiency	86%	80%
Characteristics		
Terminal resistance	1.01 Ω	0.21 Ω
Terminal inductance	0.448 mH	0.03 mH
Torque constant	73.9 mNm/A	7.48 mNm/A
Speed constant	129 rpm/V	1280 rpm/V
Speed / torque gradient	1.76 rpm/mNm	35.8 rpm/mNm
Mechanical time constant	3.85 ms	7.49 ms
Rotor inertia	209 gcm²	20 gcm²
Thermal Data		
Thermal resistance housing-ambient	1.7 K/W	5.4 K/W
Thermal resistance winding-ambient	1.1 K/W	2.5 K/W
Thermal time constant winding	35.1 s	14.8 s
Thermal time constant motor	1570 s	1180 s
Ambient temperature	-20...+125 °C	-20...+100 °C
Max. winding temperature	+125 °C	+125 °C
Mechanical Data		
Bearing type	ball bearings	ball bearings
Max. speed	12000 rpm	25000 rpm
Axial play	0 – 0.15 mm	0 – 0.14 mm
Max. axial load (dynamic)	20 N	5.6 N
Max. force for press fits (static)	170 N	98 N
(static, shaft supported)	5000 N	1200 N
Max. radial load	180 N, 5 mm from flange	28N, 5 mm from flange
Other Specifications		
Number of pole pairs	1	1
Number of phases	3	3
Number of autoclave cycles	0	0
Protection	IP54	–
Weight	1100g	270 g
Reduction GearBox	GP52C 53:1	GP32C 111:1
Worm Drive	–	60:1
Encoder	–	HEDL-5600 A06
Encoder resolution	–	500 cpr in quadrature

Tabela A.2: Configurações das EPOS

Valores											
		EPOS 70/10				EPOS 24/5					
		Esteira esquerda		Esteira direita		Esteira braço traseiro		Braço dianteiro			
	ID	1		2		4		8			
Objeto		Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal	Decimal	Hexadecimal		
Nome	Código										
Min. Position Limit	0x607D-01	-2147483648	80000000	-2147483648	80000000	-2147483648	80000000	-2147483648	80000000		
Max. Position Limit	0x607D-02	2147483647	7FFFFFFF	2147483647	7FFFFFFF	2147483647	7FFFFFFF	2147483647	7FFFFFFF		
Max. Profile Velocity	0x607F	5250	1482	5250	1482	11000	2AF8	11000	2AF8		
Profile Velocity	0x6081	5000	1388	5000	1388	10000	2710	10000	2710		
Profile Acceleration	0x6083	2000	7D0	2000	7D0	5000	1388	5000	1388		
Profile Deceleration	0x6083	2000	7D0	2000	7D0	5000	1388	5000	1388		
Continuous Current Limit	0x6410-01	4500	1194	4500	1194	3500	DAC	3500	DAC		
Pole Pair Number	0x6410-03	1	1	1	1	1	1	1	1		
Thermal Time Constant Winding	0x6410-04	351	15F	351	15F	148	94	148	94		
CAN Bitrate	0x2001	500k	2	500k	2	500k	2	500k	2		

Tabela A.3: Informações das portas da placa de monitoramento

Board Label	**Signal**	**EPOS Signal**	**EPOS Pin**	**Comment**
1	GND	AGND	14	
5	DO3	DIO2	12	Kinect activation
6	DO2	DIO3	10	Hokuyo Laser activation
7	DO1	DIO4	11	LED Activation
8	AN1	ANI1	16	Medida Corrente
9	AN2	ANI2	15	Medida Voltagem
10	DIN6	DI6	3	
11	DIN5	DI5	4	
15	DIN3			Barra Cindal
17	DIN4			Barra Cindal
19	DIN5			Barra Cindal
21	DIN6			Barra Cindal
31	RL1+			12 V
32	RL1-			Device 1 (LED)
33	RL2+			12 V
34	RL2-			Device 2 (Laser Hokuyo)
35	RL3+			12 V
36	RL3-			Device 3 (Kinect)

Tabela A.4: Informações do laser UST-10LX Informações retiradas do manual

Alimentação	12/24 V (DC)
Ângulo de varredura	270°
Resolução angular	0.25°
Alcance	0.06 m – 30 m
Período de varredura	25 ms

Tabela A.5: Configurações de rede dos dispositivos

Device	**IP**	**User/Password**	**Comments**
Access Point Esteem	172.16.1.1	admin/admin	DHCP server (172.16.x.x – 172.16.x.x, SSID: diane)
Switch	172.16.10.1	admin/admin	—
Laser Hokuyo	172.16.0.10	—	port: 10940
ADLQM67PC	172.16.11.27	diane/labcon	—
Network Netmask	255.255.0.0	—	—
Network Gateway	192.16.1.2	—	—
Network Broadcast	172.16.255.255	—	—

Tabela A.6: Configurações da rede CAN

#DEFINE em Epos.h	ID Programa	ID CANOpen	Motor	Comentários	Baudrate
EPOS_ID_NO	2	2	Esteira Direita	EPOS 70/10	500k
EPOS_ID_NO	1	1	Esteira Esquerda	EPOS 70/10	500k
EPOS_ID_NO	3	8	Braço Dianteiro	EPOS 24/5	500k
EPOS_ID_NO	4	4	Braço Traseiro	EPOS 24/5	500k
EPOS_ID_ENV_HEARTBEAT	–	10	—	Heartbeat	—

Apêndice B

Scripts Utilizados

diane_robot.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
        <arg name="bag_location" default="" />
        <arg name="config_location" default="$(find diane_files)/config/" />
        <arg name="controller_name" default="diane_controller"/>
        <arg name="manager" default="diane_manager"/>
        <arg name="max_velocity" default="5"/>
        <arg name="namespace" default="r"/>

        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="$(arg manager)" args="manager"/>
        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="epos1" args="load epos/EposNodelet $(arg
              manager) _can_device:=can0 _device_id:=1 _inverted:=false _velocity:=0.3 _acceleration:=2000
              _deceleration:=2000 _gain_meter_to_step:=421974.5223 _gain_mps_to_rpm:=28117.373279568">
                <!--Epos esquerda corresponde ao id 1-->
        </node>
        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="epos2" args="load epos/EposNodelet $(arg
              manager) _can_device:=can0 _device_id:=2 _inverted:=true _velocity:=0.3 _acceleration:=2000
              _deceleration:=2000 _gain_meter_to_step:=421974.5223 _gain_mps_to_rpm:=28117.373279568">
                <!--Epos direita corresponde ao id 2-->
        </node>
        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="epos3" args="load epos/EposNodelet $(arg
              manager) _can_device:=can0 _device_id:=8 _inverted:=false _acceleration:=2000 _deceleration:=2000
              _gain_meter_to_step:=7575068.244 _gain_mps_to_rpm:=227252.0473">
                <!--Epos dianteira corresponde ao id 8-->
        </node>
        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="epos4" args="load epos/EposNodelet $(arg
              manager) _can_device:=can0 _device_id:=4 _inverted:=false _acceleration:=2000 _deceleration:=2000
              _gain_meter_to_step:=7575068.244 _gain_mps_to_rpm:=227252.0473">
                <!--Epos traseira corresponde ao id 4-->
                <rosparam param="digital_outputs">[1,2,3]</rosparam>
                <!--Para ligar o laser, chamar o servico para a porta 2 com valor true-->
                <!--Para ligar o kinect, chamar o servico para a porta 1 com valor true-->
        </node>

        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="$(arg controller_name)" args="load
              diane_controller/DianeControllerNodelet $(arg manager) _name:=Motion _max_velocity:=$(arg max_velocity)
              _config:=$(arg config_location)controller" />
        <node ns="$(arg namespace)" pkg="nodelet" type="nodelet" name="diane_remap" args="load diane_controller/
              DianeControllerRemap $(arg manager) _controller_name:=$(arg controller_name)" />
        <node pkg="urg_node" type="urg_node" name="laser_horizontal" args="load urg_node diane_manager _ip_address:
              =172.16.0.10" />-->
        <!--<include file="/opt/ros/indigo/share/freenect_launch/launch/freenect.launch" />-->

</launch>
```

diane_base.launch

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
        <arg name="bag_location" default="" />
```

```
        <arg name="config_location" default="$(find diane_files)/config/" />
        <arg name="manager" default=""/>
        <arg name="controller_name" default="diane_controller"/>
        <arg name="max_velocity" default="0.3"/>
        <arg name="namespace" default=""/>

        <node output="screen" pkg="nodelet" type="nodelet" name="keyboard" args="load keyboard/KeyboardNodelet $(arg
            manager)"/>
        <node output="screen" pkg="nodelet" type="nodelet" name="gamepad" args="load gamepad/GamepadManager $(arg
            manager) _config:=$(arg config_location)gamepad _field_name:=gamepad _autorepeat_rate:=0.5" clear_params
            ="true"/>
        <node output="screen" pkg="nodelet" type="nodelet" name="linux_dev_gamepad" args="load linux_devs_device/
            LinuxDevDevice $(arg manager) _sender_name:=gamepad" respawn="false"/>
        <node output="screen" pkg="nodelet" type="nodelet" name="diane_mapper" args="load diane_mapper/
            DianeMapperNodelet $(arg manager) _config:=$(arg config_location)gamepad_mapper _gamepad_manager:=
            gamepad _controller:=$(arg controller_name) _holding_wait:=100 _max_velocity:=$(arg max_velocity) _name:
            =DianeMapper"/>
</launch>
```

diane_mapper OnInit

```
void DianeMapperNodelet::onInit()
{
    input_mapper::InputMapperNodelet::onInit();

    // Get node handles
    ros::NodeHandle nodehandle = getNodeHandle();
    ros::NodeHandle& privateNH = getPrivateNodeHandle();

    string controllerName;
    privateNH.param("controller", controllerName, (std::string)"diane_controller");
    privateNH.param("max_velocity", maxVelocity, 1.0);

    privateNH.param("config", xmlSequenceFileName, (std::string)"");
    ReadModeSequenceXml();

    msgInputPub = nodehandle.advertise<controller::Control>(controllerName + "/input", 1000, true);
    srvOriginIDcli = nodehandle.serviceClient<controller::RequestID>(controllerName + "/request_id");
}
```

Epos Control Callback

```
bool epos::EposNodelet::ControlCallback(epos::Control::Request & req, epos::Control::Response & res)
{
    mutUpdate.lock();

    res.ok = true;

    // If these SDOs must be sent, the NMT state must be the Pre-Operacional because PDOs can only be configured in
        that state
    // The SendNMT may return false because there is no wait time (time for an answer to be acquired of the NMT state
        change).
    // The code implemented just believe that the NMT will be changed.

    if (res.ok)
    {
        // Even if it's not in the Operation Enable, the desired operation mode and value will be updated
        switch (req.mode)
        {
        case ControlType::Current:
            driveStateSP = EPOS_DRIVE_STATE_OPERATION_ENABLE;
            operationModeSP = EPOS_MODES_OPERATION_CURRENT_MODE;
            currentSP = (short)req.data;
            rPDO[1] = true;
            break;
        case ControlType::Velocity:
            driveStateSP = EPOS_DRIVE_STATE_OPERATION_ENABLE;
            operationModeSP = EPOS_MODES_OPERATION_PROFILE_VELOCITY_MODE;
            velocitySP = req.data;
            rPDO[1] = true;
            break;
        case ControlType::Position:
            driveStateSP = EPOS_DRIVE_STATE_OPERATION_ENABLE;
            operationModeSP = EPOS_MODES_OPERATION_PROFILE_POSITION_MODE;
```

```
            positionSP = req.data-positionOffset;
            positionDegrees = false;
            rPDO[2] = true;
            break;
        case ControlType::PositionDegrees:
            driveStateSP = EPOS_DRIVE_STATE_OPERATION_ENABLE;
            operationModeSP = EPOS_MODES_OPERATION_PROFILE_POSITION_MODE;
            positionSP = req.data-positionOffset;
            positionDegrees = true;
            rPDO[2] = true;
            break;
        case ControlType::NotControlled:
            driveStateSP = EPOS_DRIVE_STATE_READY_TO_SWITCH_ON;
            rPDO[1] = true;
        default:
            res.ok = false;
            break;
        }

        if (driveState != EPOS_DRIVE_STATE_OPERATION_ENABLE)
            res.ok = false;
    }

    mutUpdate.unlock();

    return true;
}
```

CommandRemap

```
void diane_controller::DianeControllerRemap::CommandRemap(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
    sendPos = (msg->data[0] >= 1) ? true : false;
    velLin = msg->data[1];
    velAng = msg->data[2];
    if (sendPos)
    {
        posArmF = msg->data[3];
        posArmB = msg->data[4];
    }
    else
    {
        velArmF = msg->data[3];
        velArmB = msg->data[4];
    }
    timeOfLastInput = boost::posix_time::microsec_clock::local_time();
}
```

Esteiras

```
void diane_controller::DianeController::CollectiveVelocity(vector<double> inputs)
{
//inputs[0] = velocidade linear
//inputs[1] = velocidade angular
//inputs[2] = velocidade do braco 1
//inputs[3] = velocidade do braco 2

    outputs[0] = (inputs[0] - inputs[1])/2;
    outputs[1] = (inputs[0] + inputs[1])/2;
    outputs[2] = inputs[2];
    outputs[3] = inputs[3];

    for (int i = 0; i < noEpos; i++)
        outputModes[i] = epos::ControlType::Velocity;
}

void diane_controller::DianeController::CollectivePosition(vector<double> inputs)
{
    //inputs[0] = velocidade linear
    //inputs[1] = velociddade angular
    //inputs[2] = posicao do braco 1
    //inputs[3] = posicao do braco 2

        outputs[0] = (inputs[0] - inputs[1])/2;
```

```
        outputs[1] = (inputs[0] + inputs[1])/2;
        outputs[2] = inputs[2];
        outputs[3] = inputs[3];

        for (int i = 0; i < noEpos; i++)
        {
            if (i<2)
                outputModes[i] = epos::ControlType::Velocity;
            else
                outputModes[i] = epos::ControlType::PositionDegrees;
        }
}
```

FeedbackPub

```
vector<double> diane_controller::DianeController::FeedbackPub()
{
    vector<double> msg;
    bool pos = (this->eposMode[2] = epos::ControlType::Position) ? true : false;//position mode
    msg.push_back(pos);
    msg.push_back(pitch);//Pitch Angle
    msg.push_back((this->eposVelocities[0]+this->eposVelocities[1])/2);//Linear Speed
    msg.push_back((this->eposVelocities[0]-this->eposVelocities[1])/2);//Angular Speed
    if (pos)
    {
        msg.push_back(this->eposPositions[2]);//Front Arm
        msg.push_back(this->eposPositions[3]);//Rear Arm
    }
    else
    {
        msg.push_back(this->eposVelocities[2]);//Front Arm
        msg.push_back(this->eposVelocities[3]);//Rear Arm
    }
    return msg;
}
```

InputSender

```
void diane_controller::DianeControllerRemap::InputSender()
{
    std::vector <double> inputs;
    inputs.push_back(velLin);
    inputs.push_back(velAng);


    std::vector <unsigned char> inputModes;
    if (sendPos)
    {
        inputModes.push_back(diane_controller::ControlTypes::Position);
        inputs.push_back(posArmF);
        inputs.push_back(posArmB);
    }
    else
    {
        inputModes.push_back(diane_controller::ControlTypes::Velocity);
        inputs.push_back(velArmF);
        inputs.push_back(velArmB);
    }

    controller::Control actualcontrol;
    actualcontrol.originId = controlId;
    actualcontrol.modes = inputModes;
    actualcontrol.data = inputs;
    msgInputPub.publish(actualcontrol);
}
```

StandardInput

```
void diane_controller::DianeController::SetStandardInput()
{
    vector<unsigned char> modes;
```

```
    vector<double> inputs;

    modes.push_back(ControlTypes::Brake);

    newOriginId = 1;
    inputNewModes = modes;
    newInputs = inputs;
}
```

StairModel.m

```
function [ stairModel, planeModel ] = StairModel( pCloud, lines3D )
%%Receives a pointCloud object (pCloud) containing points of a stair, the
%%lines of that stair (lines3D) as a custom struct and returns structs
%%containing the stair model and the plane model of that particular stair.

%Angulacao do plano da escada.
[plane,~,~] = pcfitplane(pCloud,0.02);
planeAlfa = acos(dot(plane.Normal,[0 -1 0]));
if (planeAlfa > (pi/2))
    planeAlfa = abs(pi-planeAlfa);
end

%Agrupando por degrau cada linha
clear steps;
step = struct('xy',lines3D(1).xy,'points',lines3D(1).points,'step',1);
steps(1) = step;
stepNum = 1;
stepsLen = 1;
for i=1:length(lines3D)-1
    stepi = 0;
    for j=1:stepsLen
        if (abs(lines3D(i).xy(1,2)-steps(j).xy(1,2)) < 5)
            stepi = steps(j).step;
            break;
        end
    end
    if (stepi == 0)
        stepi = 1 + stepNum;
        stepNum = stepi;
    end
    step = struct('xy',lines3D(i).xy,'points',lines3D(i).points,'step',stepi);
    steps = vertcat(steps,step);
    stepsLen = stepsLen + 1;
end

%Projetando linhas no plano da escada
point = [0 0 -plane.Parameters(4)/plane.Parameters(3)];
for i=1:stepsLen
    pointsLen = size(steps(i).points,1);
    j = 1;
    while (j<pointsLen)
        if (ismember(steps(i).points(j,:),pCloud.Location,'rows'))%filter points. Only points in filtered point cloud
            steps(i).points(j,:) = steps(i).points(j,:) - dot(steps(i).points(j,:)-point,plane.Normal)*plane.Normal;%
                projetando no plano. q_proj = q - dot(q-p,n)*n
            j = j + 1;
        else
            steps(i).points(j,:) = [];
            pointsLen = pointsLen - 1;
        end
    end
end
clear pointsLen j;

%%Verificar utilidade.
%Rotacionar plano para achar centroides.
% rot_theta = atan2(plane.Normal(2),plane.Normal(1));
% R = rotz(-rot_theta);%Rotacao em Z para deixar alinhado com eixo Y.

%Inicio da montagem do modelo da escada.
degrau = [];
stairTemp = [];
syms t;
for i = 1:stepNum
    %Agrupa todos os pontos correspondentes a um degrau.
    for j = 1:length(steps)
        if (steps(j).step == i)
```

```
            degrau = vertcat(degrau,steps(j).points);
        end
    end

    %Achando equacao da linha do degrau.
    r0=mean(degrau);
    degrau=bsxfun(@minus,degrau,r0);
    [~,~,V]=svd(degrau,0);
    r(t) = r0 + t*V(:,1)';
    line = struct('equation',r,'point',r0,'v',V(:,1)');

    %Achando limites e calculando centroide do degrau.
    limits = LineLimits(line,degrau);
    stairTemp = vertcat(stairTemp,struct('stepNum',i,'equation',r,'limits',limits,'centroide',mean(limits)));
    degrau = [];
end

%Ordenacao dos degraus por profundidade.
A = zeros(stepNum,2);
for n = 1:stepNum
    A(n,:) = [stairTemp(n).stepNum stairTemp(n).centroide(3)];
end
A = sortrows(A,2);%Ordenar por profundidade

stair(stepNum) = stairTemp(A(end,1));
stair(stepNum).stepNum = stepNum;
for n = 1:stepNum - 1
    stair(n) = stairTemp(A(n,1));
    stair(n).stepNum = n;
end

LimEsq = stair(1).limits(1,1); LimDir = stair(1).limits(2,1);
LimInf = stair(1).centroide(2);   LimSup = stair(end).centroide(2);
LimFront = stair(1).centroide(3); LimBack = stair(end).centroide(3);

degAlt = 0;
degProf = 0;
for i=1:stepNum-1
    degAlt = degAlt + abs(stair(i).centroide(2)-stair(i+1).centroide(2));
    degProf = degProf + abs(stair(i).centroide(3)-stair(i+1).centroide(3));

    if (stair(i+1).limits(1,1) < LimEsq)
        LimEsq = stair(i+1).limits(1,1);
    end
    if (stair(i+1).limits(2,1) > LimDir)
        LimDir = stair(i+1).limits(2,1);
    end
end
degAlt = degAlt/(stepNum-1);%altura media dos degraus
degProf = degProf/(stepNum-1);%profundidade media dos degraus

planeModel = plane;
stairModel = struct('LimHorz',[LimEsq LimDir],'LimVert',[LimInf LimSup],'LimDepth',[LimFront LimBack],'StepHeight',
      degAlt,'StepDepth',degProf,'Stair',stair,'Alpha',planeAlfa,'NSteps',stepNum,'Width',abs(LimEsq-LimDir),'Height',
      abs(LimSup-LimInf),'Depth',abs(LimFront-LimBack));

end
```

CanClimbStairs.m

```
function [ canClimbStairs ] = CanClimbStairs( robot , stairs )
%%Returns if a robot can climb stairs.

canClimbStairs = false;

if (robot.Width < stairs.Width)
    if (robot.maxStepHeight > stairs.StepHeight)
        if (robot.maxAlpha > stairs.Alpha)
            canClimbStairs = true;
        end
    end
end
end
```

DetectStairsKinect.m

```
function [ foundStairs, RCloud, lines3D] = DetectStairsKinect( Imd, PCloud, CThreshold, HThreshold, numPeaks,
    minLength )
%DetectStairsKinect Detect if exists any stairs in Imd and return a
%struct containing stairs info.
%   Imd is already an image (not a msg from ROS). PCloud
if ~exist('CThreshold', 'var')
    CThreshold = [0.02 0.039];
end
if ~exist('HThreshold', 'var')
    HThreshold = 15;
end
if ~exist('numPeaks', 'var')
    numPeaks = 1000;
end
if ~exist('minLength', 'var')
    minLength = 50;
end

depthImage = Imd;
numColImg = size(Imd,2);

edgeImage = edge(depthImage,'canny',CThreshold);

%Apply theta range so it extract only Horizontal (or close to horizontal) lines
[H,theta,rho] = hough(edgeImage,'Theta',-90:0.5:-86,'RhoResolution',1);
[H1,theta1,rho1] = hough(edgeImage,'Theta',85:0.5:89,'RhoResolution',1);
H = [H H1];
theta = [theta theta1];
clear H1 theta1 rho1;

HPeaks = houghpeaks(H,numPeaks,'threshold',HThreshold); %Peaks extracted from Hough Transform applied to theta range
    above.
HLines = houghlines(edgeImage,theta,rho,HPeaks,'FillGap',7,'MinLength',7); %Returns Hough Lines that will be used
    afterwards.

%aglutinar linhas quase colineares em uma unica.
tempMergedLines = MergeLines(HLines);
%Filtrar linhas aglutinadas em relacao ao seu angulo de Hough e cmprimento minimo.
k=1;

for i=1:length(tempMergedLines)
    if (tempMergedLines(i).theta <= -86 || tempMergedLines(i).theta >= 85)
        if ((norm(tempMergedLines(i).point1 - tempMergedLines(i).point2)) >= minLength)%length of segment
            mergedLines(k) = tempMergedLines(i);
            k = k+1;
        end
    end
end
clear k;

numLin = length(mergedLines);

%Montagem do histograma de frequencia de linhas em colunas da imagem.
histColumn = zeros(1,numColImg);
linesPoses = zeros(numLin,2);
for i=1:numLin
    xy = [mergedLines(i).point1; mergedLines(i).point2];
    minX = min(xy(:,1));
    maxX = max(xy(:,1));
    temp = zeros(1,numColImg);
    temp(minX:maxX) = ones(1,maxX - minX+1);%xy(:,1) corresponde a valores de X. Valores de Y nao variam, ja que
        estamos fazendo analise por coluna.
    histColumn = histColumn + temp;
    linesPoses(i,:) = [minX, maxX];%Pegar minima e maxima abcissas da linha para depois analisar e interpretar como
        uma unica linha.
end

maxFreq = max(histColumn);
maxFreqIndex = find(histColumn==maxFreq);

k=1;
limInf = 1000; limSup = 0;%Obtendo limites superior e inferior.
for i=1:numLin
   for j=1:length(maxFreqIndex)
       if (maxFreqIndex(j)>=linesPoses(i,1) && maxFreqIndex(j)<=linesPoses(i,2))
           filtLines(k) = mergedLines(i);
           y = [filtLines(k).point1(2);filtLines(k).point2(2)];
```

```
            if (min(y) < limInf)
                limInf = min(y);
            end
            if (max(y) > limSup)
                limSup = max(y);
            end
            k = k+1;
            break;
        end
    end
end
clear k y;
numLin = length(filtLines);
%obtendo limites esquerdo, direito.
limEsq = min(maxFreqIndex);
limDir = max(maxFreqIndex);

Lim = struct('LimEsq',limEsq,'LimDir',limDir,'LimInf',limInf,'LimSup',limSup);

%tracando novas retas de acordo com os limites obtidos.
for i=1:numLin
    xy = [filtLines(i).point1; filtLines(i).point2];
    [minX, indMinX] = min(xy(:,1));
    [maxX, indMaxX] = max(xy(:,1));
    [minY, indMinY] = min(xy(:,2));
    [maxY, indMaxY] = max(xy(:,2));

    if (minX < limEsq)
        xy(indMinX,1) = limEsq;
    end
    if (maxX > limDir)
        xy(indMaxX,1) = limDir;
    end
    if (minY < limInf)
        xy(indMinY,2) = limInf;
    end
    if (maxY > limSup)
        xy(indMaxY,2) = limSup;
    end
    filtLines(i).point1 = xy(1,:);
    filtLines(i).point2 = xy(2,:);
end

% figure, imshow(edgeImage), hold on
%     for k = 1:length(filtLines)
%     xy = [filtLines(k).point1; filtLines(k).point2];
%     plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');
%     end

%retorno da funcao com o devido PointCloud.
if (numLin < 3)
    RCloud = [];
    return
else
    [ foundStairs, RCloud, lines3D] = AjustaPCloud(filtLines,PCloud,Lim);
end

end
```

AjustaPCloud.m

```
function [ foundStairs, rPCloud, lines3D] = AjustaPCloud( lines , pCloud, Lim)
%AjustaPCloud ajusta e devolve as linhas em 3D correspondentes as linhas
%dadas (extraidas da depthImage).

foundStairs = false;
maxAngle = 60;
widthBoxMin = 50;

if (abs(Lim.LimDir-Lim.LimEsq) > widthBoxMin)%se limites estao de acordo com o manimo necessario para o robo subir,
      continue.

    A = [];%Variavel auxiliar.
    lines3D = [];

    for k = 1:length(lines)
        stairsPointCloud = [];
```

```
        linePoints = LinePoints(lines(k));

        for j = 1:size(linePoints,1)
            x = linePoints(j,1);
            y = linePoints(j,2) + 4; %Correcao de bordas. Os Pontos correspondentes as linhas sao os pontos do espelho
                   do degrau, e nao da borda. Somando esse pequeno numero, conseguimos achar as bordas do degrau.

            if (~(isnan(pCloud(y,x,3))))
                stairsPointCloud = vertcat(stairsPointCloud,pCloud(y,x,:));
            else
                kn = 4;%Tolerancia em pixel da coordenada 'y' da depthImage para PointCloud.
                diffVec = [];
                vec = pCloud(y-kn:y+kn,x,:);
                for i=1:size(vec,1)-1
                    diffVec = vertcat(diffVec,(vec(i+1,1,3)-vec(i,1,3)));
                end
                [~,I] = max(diffVec);
                if (~(isnan(pCloud(y-kn+I+1,x,3))))
                    stairsPointCloud = vertcat(stairsPointCloud,pCloud(y-kn+I+1,x,:));
                end
            end
        end
        clear y x;

        if(~isempty(stairsPointCloud))
            points = [stairsPointCloud(:,1,1) stairsPointCloud(:,1,2) stairsPointCloud(:,1,3)];
            s = struct('xy',[lines(k).point1;lines(k).point2],'points',points);
            lines3D = vertcat(lines3D,s);
            A = vertcat(A,stairsPointCloud);
        end
    end

    if (~isempty(A))
        rPCloud = A;

        A = [A(:,1,1) A(:,1,2) A(:,1,3)];

        B = ones(length(A(:,1,1)),1);
        Abar = A'*A;
        Bbar = A'*B;
        v = Abar\Bbar;

        q1 = [1 0 0];
        q2 = [0 0 1];

        Q = [q1' q2'];
        Qbar = Q'*Q;
        p = (Q/Qbar)*Q'*v;
        pn = p/norm(p);
        vn = v/norm(v);
        angle = (180/pi)*acos(dot(pn,vn));
        if(angle > 90)
            angle = 180 - angle;
        end

        if ((90 - angle) < maxAngle)
            foundStairs = true;
        end

    end

else
    rPCloud = NaN;
    lines3D = NaN;
end

end
```

Climb_1.m

```
global rearArm pitch;
while(rearArm < Diane.maxAlpha)
    SendControlMessage(pubCmd,1,0,0,Diane.maxAlpha,Diane.maxAlpha);
    pause(0.2);
end

stairAng = stairs.Alpha*180/pi;
```

```
while (pitch <= stairAng)%Checar se stairs.Alpha esta em radianos. Deve ser em graus.
    SendControlMessage(pubCmd,0,0.1,0,0,0);%Acrescentar controle para nao bater em laterais.
    pause(0.2);
end
```

Climb_2.m

```
global rearArm frontArm;
while (rearArm > 0.2)
    SendControlMessage(pubCmd,1,0,0,frontArm,0);
    pause(0.2);
end
```

Climb_3.m

```
%3a etapa de inicio da subida.
global pitch;
format shortg;
lastPitch = pitch;
lastTime = clock;
while (true)
    SendControlMessage(pubCmd,0,0.1,0,0,0);
    pause(2)
    if (pitch-lastPitch <= 0.5)
        if (etime(clock, lastTime) > seconds(2))
            break;
        end
    else
        lastTime = datetime;
    end
    lastPitch = pitch;
end
```

Control.m

```
reachedEnd = false;
target = CalculateTarget(stairs);
stairEnd = batch(@KinematicModelIntegrator,1,{target(1)});
global pos;
%This is the control part
while (~reachedEnd)
%     lin = Diane.maxVel/2;
%     ang = 0;%Diane.maxAngVel/2;

    [erho, ealpha, ebeta, reachedEnd] = TargetError(target,pos);

    [lin, ang] = ControlLaw(Diane.kinematicGains,[erho, ealpha, ebeta]);

    [l,a] = AvoidCloseObstacle(Diane,scanData);
    lin = lin + l;
    ang = ang + a;

    [l,a] = MaintainCenter(Diane,stairs,scanData);
    lin = lin + l;
    ang = ang + a;


    SendControlMessage(pubCmd,0,lin,ang,0,0);
    pause(0.3);
end
```

Config_ROS.m

```
setenv('ROS_IP','172.16.180.16');
setenv('ROS_MASTER_URI','http://172.16.11.27:11311');

rosinit;
```

SendControlMessage.m

```
function [ ok ] = SendControlMessage(controlPublisher, mode, linSpeed, angSpeed, frontArm, rearArm)
%%Receives ROS message of type std_msgs/Float64MultiArray and decomposes it
%%to each feedback variable.
%fronArm and rearArm may vary depending on the mode selected (mode = 1, position, mode = 0 velocity)

msg = rosmessage(controlPublisher);
msg.Data = [mode, linSpeed, angSpeed, frontArm, rearArm];
send(controlPublisher,msg);

end
```