Luciano Ricardo Corrêa Laranjeira

Prototipagem de Algoritmos para Controle de Cadeiras de Rodas Motorizadas Utilizando Arquitetura Multiagente, Simulador GAZEBO e MATLAB

São Paulo - Brasil

2016

Luciano Ricardo Corrêa Laranjeira

Prototipagem de Algoritmos para Controle de Cadeiras de Rodas Motorizadas Utilizando Arquitetura Multiagente, Simulador GAZEBO e MATLAB

Dissertação apresentada ao programa de Pós-Graduação em Automação e Controle de Processos do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - IFSP, como requisito para obtenção do título de mestre.

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - IFSP

Campus São Paulo

Mestrado em Automação e Controle de Processos

Orientador: Prof. Dr. Alexandre Simião Caporali

Coorientador: Prof. Dr. Paulo Marcos de Aguiar

São Paulo - Brasil

2016

L332p Laranjeira, Luciano Ricardo Corrêa.
Prototipagem de algoritmos para controle de cadeiras de rodas motorizadas utilizando arquitetura multiagente, simulador GAZEBO e MATLAB / Luciano Ricardo Corrêa Laranjeira. São Paulo: [s.n.], 2016.
77 f.: il.

Orientador: Prof. Dr. Alexandre Simião Caporali.
Coorientador: Prof. Dr. Paulo Marcos de Aguiar.

Dissertação (Mestrado Profissional em Automação e Controle de Processos) - Instituto Federal de Educação, Ciência e Tecnologia de São Paulo, IFSP, 2016.

1. Cadeira de rodas inteligente 2. Algoritmos de controle 3. Arquitetura multiagente 4. GAZEBO 5. MATLAB Simulink I. Instituto Federal de Educação, Ciência e Tecnologia de São Paulo. II. Título

CDU 681.0

INSTITUTO FEDERAL
SÃO PAULO
Campus São Paulo

MINISTÉRIO DA EDUCAÇÃO
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO
CAMPUS SÃO PAULO
DIRETORIA GERAL DO *CAMPUS* SÃO PAULO
Coordenadoria de Registros Escolares de Pós-Graduação

ATA DE EXAME DE DEFESA DE DISSERTAÇÃO

Nome do Programa: **Mestrado Profissional em Automação e Controle de Processos**

Nome do(a) Aluno(a): Luciano Ricardo Corrêa Laranjeira

Nome do(a) Orientador(a): Prof. Dr. Alexandre Simião Caporali

Nome do(a) Coorientador(a): Prof. Dr. Paulo Marcos de Aguiar

Título do Trabalho: "Prototipagem de algoritmos para controle de cadeiras de rodas motorizadas utilizando arquitetura multi-agente, simulador GAZEBO e MATLAB"

Abaixo o resultado de cada participante da Banca Examinadora

Nome completo dos Participantes Titulares da Banca	**Sigla da Instituição**	**Aprovado / Não Aprovado**
Prof. Dr. Paulo Marcos de Aguiar – Coorientador	IFSP – SPO	Aprovado
Prof. Dr. Alexandre Brincalepe Campo – Membro Interno	IFSP – SPO	Aprovado
Prof. Dr. André Leon Sampaio Gradvohl - Membro Externo	Unicamp	Aprovado
Nome completo dos Participantes Suplentes da Banca	**Sigla da Instituição**	**Aprovado / Não Aprovado**
Profª Drª Jussara Pimenta Matos – Membro Externo	IFSP – SPO	
Prof. Dr. Geraldo Dantas Silvestre Filho – Membro Externo	UFPB	

Considerando-o: [X] APROVADO
[ ] NÃO APROVADO

Assinaturas

São Paulo, 08 de abril de 2016

Presidente da Banca

Membro Interno

Membro Externo

Observações:

Agradecimentos

Agradeço à minha esposa Viviane e à minha filha Júlia por me apoiarem e entenderem minha ausência durante os momentos em que precisei me dedicar a este trabalho.

Agradeço, também, aos meus pais que sempre me incentivaram a continuar estudando.

Ao meu orientador, Prof. Dr. Alexandre Simião Caporali, que, além de compartilhar seu conhecimento e experiência, sempre dedicou atenção a mim e ao meu trabalho.

Ao meu coorientador, Prof. Dr. Paulo Marcos de Aguiar, que me incentivou a trabalhar com cadeiras de rodas e que, também, sempre me ajudou prontamente.

Aos professores Dr. Alexandre Brincalepe Campo e Dr. André Leon Sampaio Gradvohl que, durante a fase de qualificação, contribuíram com sugestões importantes para que este trabalho pudesse ser concluído.

A todos os professores que, ao longo da minha formação, contribuíram com seus ensinamentos.

Aos meus colegas de mestrado do IFSP pela cooperação e amizade.

Obrigado a todos!

Resumo

Há uma tendência mundial no envelhecimento da população e a crescente demanda por cadeiras de rodas. O uso de cadeiras motorizadas requer uma série de cuidados (treinamento e indicação médica) para evitar acidentes. A complexidade envolvida em sua operação restringe sua ampla utilização por pessoas com alguma limitação de locomoção. Mesmo aqueles que possuem indicação de uso, passam por um período de treinamento e adaptação. Sistemas de controle que auxiliam ou automatizam a dirigibilidade de cadeiras de roda foram amplamente estudados neste trabalho. Eles abordam problemas variados como, por exemplo, colisão, tração, interpretação de linguagem natural e interação com o navegador (*Joystick*), utilizando-se de diferentes técnicas de controle como lógica Fuzzy, computação visual, análise de atividade cerebral (*Brain Computer Interface*), movimento da língua e vários tipos de sensores e estratégias. Entretanto muitos trabalhos são independentes e de difícil reprodução (necessita de *hardware* específico ou faltam informações). O surgimento de plataformas de desenvolvimento que promovem a reutilização de código, como o ROS, criou um ambiente favorável à troca de conhecimento entre pesquisadores e agilidade no desenvolvimento de controles complexos. Outra vantagem é a possibilidade de testar a solução em ambientes simulados. Este trabalho propõe o uso do ROS, simulador 3D GAZEBO e MATLAB para prototipagem rápida de algoritmos para controle de cadeira de rodas motorizadas com objetivo de explorar características que favoreçam o surgimento de equipamentos a preços mais acessíveis em países em desenvolvimento. Os resultados obtidos mostram que essa técnica é viável. Para se chegar a essa conclusão, testes foram realizados com uma cadeira de rodas virtual. Antes disso, porém, abordou-se o processo de instalação das ferramentas, a criação da cadeira de rodas motorizada virtual e o desenvolvimento de um algoritmo anticolisão utilizando MATLAB Simulink. Integrado a um sistema de controle proporcional, o algoritmo anticolisão processa dados de sensores sonar e, então, atua sobre os motores do veículo para impedir colisões.

Palavras-chaves: Cadeira de Rodas Inteligente, Algoritmos de Controle, Arquitetura Multiagente, ROS, GAZEBO, MATLAB Simulink, Simulação.

Abstract

There is a worldwide population aging trend and, thus, an increasing demand for powered wheelchairs. However, powered wheelchairs driving is not trivial and demands reasonable care (training and medical advice) to prevent accidents. The driving task is not so simple, thus this limits its usage. Even patients that have medical approval to make use of this tool may go through a training and evaluation period. Auxiliary driving control systems have been widely studied. The most common problems solved are collision avoidance, natural language translation, Joystick interaction and motion control. They use different approaches to solve the problems like Computer Vision, Fuzzy, Brain Computer Interface, Tongue Movement and many other different sensors and methods. However studies on this area are usually standlone and difficult to reproduce (there is a lack of information or it demands specific hardware). A new scenario has been created with popularization of systems, like ROS, that promotes code reuse. Among other things, it speeds up complex robotic software development and allows knowledge exchange between researchers and developers. Its use in conjuction with realistic 3D simulators is another great advantage. This work proposes the use of ROS, 3D GAZEBO simulator and MATLAB for rapid motorized wheelchair algorithms prototyping and, then, explore those tools characteristics which may allow creating low cost equipments at developing countries. The result shows that this proposal is feasible and, to demonstrate that, it has being performed several tests with a virtual wheelchair. Before the tests, however, this work went through the tools and development environment setup process, the virtual motorized wheelchair creation and a collision avoidance system development with MATLAB Simulink. Integrated to a proportional controller, the collision-avoidance algorithm processes sonar sensor data and, then, commands the motors to prevent collision.

Key-words:Smart Wheelchair Control, Control Systems Algorithms, Multi-agent Architecture, ROS, GAZEBO, MATLAB Simulink, Simulation.

Lista de ilustrações

Lista de abreviaturas e siglas

2D	Modelo Bidimensional
3D	Modelo Tridimensional
AAL	Ambient Assisted Living
API	Application Programming Interfaces
BSD	Berkeley Software Distribution
CAD	Computer Aided Design
CPL	Component Library
CRM	Cadeira de Rodas Motorizada
DHCP	Dynamic Host Configuration Protocol
GPL	General Public License
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
iGPS	Indoor Global Positioning System
IMU	Inertial Measurement Unit
IP	Protocolo de Internet
PID	Proporcional-Integral-Derivativo
RAM	Random Access Memory
ROS	Robot Operating System
RPC	Remote procedure Call
RTES	Real-Time Embedded Systems
SDF	Simulation Description Format
UML	Unified Modeling Language
XML	Extensible Markup Language

Lista de símbolos

K_p	Ganho Proporcional
K_i	Ganho Integral
K_d	Ganho Derivativo

Sumário

1 Introdução

Segundo Ceres *et al.* (2005), pesquisas tecnológicas em acessibilidade têm focado no desenvolvimento de soluções que permitam a reintegração social de pessoas com necessidades especiais. Cada vez mais, a sociedade tem considerado as dificuldades enfrentadas por esses indivíduos em suas atividades diárias. Cadeiras de Rodas Motorizadas (CRM) contribuem, ainda mais, com a independência de pessoas com limitações de mobilidade, pois podem ser customizadas com diferentes tipos de interface de controle (sopro, movimento de cabeça e outros) para atender, também, pessoas com movimento dos membros superiores limitados e que não poderiam utilizar uma cadeira de rodas comum.

Quando acrescentado de um sistema de controle, as cadeiras de rodas motorizadas são comumente chamadas de "Cadeira de Rodas Inteligente" ou simplesmente "Cadeiras Inteligentes". O artigo de Chipaila *et al.* (2012) traz a seguinte definição: Uma Cadeira de Rodas Inteligente é qualquer plataforma motorizada com uma cadeira projetada para ajudar uma pessoa com limitações físicas, onde um sistema de controle artificial auxilia ou substitui o controle do usuário. É normalmente controlada por um computador que aplica técnicas de robótica móvel e que se utiliza de dados de um conjunto de sensores.

Há uma tendência mundial de crescimento da população de idosos e deficientes físicos e isso tem motivado estudos com cadeiras inteligentes. Grigorescu *et al.* (2012) estimam que, em países desenvolvidos como Estados Unidos, Europa, Japão e Canada, exista uma população de aproximadamente 75 milhões de pessoas com deficiência física e 130 milhões de idosos. Já o artigo de Chipaila *et al.* (2012) estima que existam cerca de 200 milhões de usuários de cadeiras de rodas no mundo.

No Brasil, estimativas do Instituto Brasileiro de Geografia e Estatística - IBGE (2010), publicadas em 2013, apontam que o número de pessoas acima de 65 anos deve quadruplicar até 2060 no Brasil. A população com essa faixa etária deve passar dos atuais 14,9 milhões (7,4% do total) para 58,4 milhões (26,7% do total). Além disso, haverá também um aumento no número de pessoas com deficiência física. Atualmente existem aproximadamente treze milhões de indivíduos que possuem dificuldade de locomoção no Brasil. Esses números indicam que a demanda por cadeiras de rodas continuará aumentando e que pesquisas em equipamentos mais seguros, baratos e confortáveis devem ser de grande importância.

Cadeiras de rodas motorizadas, entretanto, possuem uso restrito devido à complexidade envolvida em sua operação. Primeiro é preciso haver uma avaliação médica que medirá a capacidade motora do paciente para controlá-la. Aqueles que obtiverem indicação ao uso do equipamento ainda serão submetidos a uma segunda etapa na qual

passarão por um período de treinamento e adaptação. Os sistemas de controle para CRM podem facilitar esse trabalho e ampliar sua utilização.

Mesmo usuários mais experientes de cadeiras motorizadas estão sujeitos a colidir com mobílias, pessoas e animais domésticos. Além disso, há problemas de dirigibilidade em locais com piso muito irregular, subidas, descidas ou que possuem corredores estreitos com curvas acentuadas. O acesso a sistemas de transporte público também pode ser desafiador. Um estudo conduzido por Simpson, LoPresti e Cooper (2008) estima que entre 61% e 91% dos usuários de cadeiras de rodas se beneficiariam das funcionalidades oferecidas por cadeiras inteligentes.

Petry *et al.* (2013) afirmam que desde a década de 1980 vários projetos de cadeiras de rodas inteligentes foram propostos para ajudar pessoas com dificuldades de mobilidade. É possível encontrar artigos científicos que propõem soluções de controle que empregam os mais variados métodos e ferramentas. Por exemplo, Ono, Uchiyama e Potter (2004) se utilizam de Lógica Fuzzy e Miro, Poon e Huang (2012) recorrem ao uso de Computação Visual, mas, em geral, as soluções são independentes, ou seja, não há preocupação em se utilizar uma arquitetura de *hardware* ou *software* padronizada que permita a reutilização dos trabalhos desenvolvidos.

A reprodução desses trabalhos também pode ser um limitador. O projeto de sistemas embarcados em tempo real (*Real-Time Embedded Systems - RTES*) é uma tarefa complexa porque estes sistemas incluem *hardware* e componentes de *software* que deve ser altamente otimizados para as necessidades da aplicação (Wehrmeister, Pereira e Becker (2005)).

A reutilização de código, entretanto, tem ganhado força e o ROS (*Robot Operating System*) é uma plataforma que tem se destacado tanto em trabalhos acadêmicos quanto em projetos comerciais por facilitar essa reutilização. O reuso de código visa economizar tempo, recursos e reduzir a redundância, tirando proveito de bibliotecas de *software* que já foram criadas e compartilhadas por outros indivíduos. A ideia-chave na reutilização é que partes de um sistema de controle complexo podem ser facilmente integradas a um novo projeto, deixando o pesquisador/desenvolvedor focado no problema que deseja resolver. Outra vantagem muito importante é que códigos compartilhados acabam sendo submetidos a um grande número de usuários que contribuem para sua maturação. Em geral, módulos ou bibliotecas compartilhados têm baixa incidência de erros.

Nota-se que, a partir de 2013, surgiram vários artigos científicos que citam ou utilizam o ROS como plataforma de desenvolvimento para cadeiras de rodas inteligentes. A todo momento são criados projetos complexos que envolveriam um vasto conhecimento em diversas disciplinas como computação visual, processamento acústico, reconhecimento de linguagem natural, arquitetura de *software* e programação de dispositivos (para citar algumas).

Este trabalho propõe o uso do ROS, simulador 3D Gazebo e MATLAB para prototipagem rápida de algoritmos de controle para cadeiras de rodas motorizadas. O objetivo é validar essa solução como alternativa para desenvolvimento de baixo custo e, ainda, explorar a característica de colaboração e compartilhamento de código intrínseca ao sistema ROS. Isso pode favorecer o surgimento de cadeiras inteligentes com preços acessíveis (fator importante para poder beneficiar o maior número de pessoas possível de países em desenvolvimento). Para se chegar aos resultados, abordou-se o processo de instalação das ferramentas, a criação da cadeira de rodas motorizada virtual e o desenvolvimento de um algoritmo anticolisão utilizando MATLAB Simulink. Finalmente, testes foram realizados com a cadeira de rodas virtual.

As demais seções estão divididas da seguinte maneira:

- Capítulo 2: Traz uma revisão de artigos científicos sobre cadeiras de rodas motorizadas com sistemas de controle utilizando arquitetura própria ou ROS, sendo este último o de maior interesse;
- Capítulo 3: Apresenta a arquitetura computacional e conceitos dos principais sistemas envolvidos na solução;
- Capítulo 4: Descreve os materiais e métodos empregados neste trabalho para se realizar o estudo proposto;
- Capítulo 5: Apresenta os resultados das simulações realizadas com a cadeira de rodas motorizada virtual;
- Capítulo 6: Contém a conclusão do trabalho.

2 Revisão da Literatura

Esta seção apresenta, através de uma revisão analítica, diversos trabalhos envolvendo sistemas de controle para cadeiras de rodas motorizadas. Foram selecionados aqueles artigos que se utilizam da plataforma de desenvolvimento ROS ou similares e, também, os que propõem uma arquitetura de *software* e *hardware* própria.

Ono, Uchiyama e Potter (2004) apresentam um projeto de construção de um veículo autônomo que servirá como prova de conceito para construção de uma cadeira de rodas inteligente. Para isso, eles propuseram uma arquitetura de desenvolvimento de *software* para robótica que eles consideraram ser um *Framework* capaz de facilitar tanto a expansão funcional das aplicações quanto sua portabilidade. A arquitetura do sistema proposto constitui-se de duas camadas HAL (*Hardware Abstraction Layer*) e CPL (*Component Layer*). A primeira é um conjunto de bibliotecas desenvolvidas em Visual C++ .NET para acesso ao *hardware* e que faz parte do *kit* de desenvolvimento utilizado, o ER1 da *Evolution Robotics* cuja imagem está ilustrada pela Figura 1. A segunda, codificada em linguagem de programação JAVA, é modularizada de maneira que possa ser compartilhada por camadas mais altas de *software*, denominadas "agentes" pelos autores. São definidos quatro agentes: o primeiro para tratamento de sensores (*Sensor Handler*), o segundo especializado em identificar corredores utilizando técnicas de computação visual (*Corridor Recognizer*), o terceiro para evitar colisões utilizando-se de lógica Fuzzy (*Collision Detector*) e o quarto responsável pela locomoção (*Drive Controler*). Os resultados, segundo os autores, demonstraram que essa arquitetura distribuída em agentes é viável. Entretanto observaram-se alguns comportamentos indesejáveis como, por exemplo, o sistema de prevenção a colisões não impedir uma batida. Além disso, falou-se em frequente instabilidade que resultou na perda de controle. O autor sugere a adoção de algum mecanismo de segurança contra falhas que tenha confiabilidade.

Chamado de HARSP (*Human-Assistance Robotic System Project*), o trabalho de Jia *et al.* (2006) foca a melhoria na qualidade de vida da população com idade mais avançada ou com limitações físicas a partir do uso da Robótica. Para isso, foi desenvolvido uma cadeira de rodas motorizada e um robô assistente capaz de realizar tarefas preestabelecidas como levar o jornal ou uma bebida para o usuário da cadeira onde quer que ele esteja dentro de uma região mapeada. Os diferenciais desse trabalho foram as tecnologias de controle e comunicação empregadas como o iGPS para identificar o posicionamento da cadeira e o *Robot Technology Middleware (RTM)* para desenvolvimento de robôs baseados em sistemas distribuídos em rede. Essa última é a característica de maior interesse, pois se utilizou uma plataforma padrão de *software* que permite ser estendida a partir da criação de componentes (*RT components*). Essa é uma característica também presente no

Figura 1: Robô ER1 da *Evolution Robotics* customizado. (ONO; UCHIYAMA; POTTER, 2004, p. 2)

sistema ROS, adotado neste trabalho, devido a sua arquitetura multiagente. A Seção 2 do artigo traz uma visão geral sobre a arquitetura do RTM (componentes padrão, bibliotecas e gerenciadores), porém destaca-se ao uso da tecnologia CORBA (*Common Object Request Broker Architecture*) que proporciona computação distribuída, independente de sistema operacional e de linguagem de programação. O artigo cita que, devido ao uso da plataforma RTM, o resultado é uma solução altamente escalável, de fácil integração e que possibilita redução no custo do desenvolvimento a partir da reutilização de componentes. Foram realizadas 22 demonstrações na *Word Exposition* com um índice 86% de sucesso. Segundo os autores, as falhas foram ocasionadas por problemas de calibração em uma câmera do robô utilizada para calcular seu posicionamento. Eles concluem que o sistema pode auxiliar um cadeirante nas tarefas diárias.

Trefler *et al.* (2004 *apud* MIRO; POON; HUANG, 2012) argumentam que, dada sua aceitação social e ubiquidade, cadeiras de rodas motorizadas convencionais têm proporcionado com sucesso os meios pelos quais um grande número de pessoas puderam manter a mobilidade. Além disso, estudos mostraram como os sistemas de cadeira de rodas em instalações hospitalares ou de cuidados interviram de forma a permitir interações sociais e atividades física e mental por mais tempo, proporcionando a capacidade de viver de forma mais independente e melhorando a qualidade de vida das pessoas. Apesar das cadeiras de rodas, manuais ou elétricas, permitirem independência na execução de uma grande quantidade de atividades do dia-a-dia, não há, no entanto, uma visão clara dos limites da tecnologia em apoiar a independência da população idosa ou com deficiência física (CONNELL; YOUNG, 2008). E isso tem motivado pesquisadores a desenvolver recursos tecnológicos mais avançados capazes de dar suporte às pessoas que sofrem de dificuldades de mobilidade (WASSON *et al.*, 2008). Miro, Poon e Huang (2012) apresentam

o desenvolvimento de um sistema de cadeira de rodas inteligente de baixo custo baseado em computação visual para deslocamento autônomo. Criou-se um veículo seguidor de um alvo (uma folha de papel verde de dimensões conhecidas) acoplado em outro veículo à frente. O trabalho foi feito a partir da cadeira de rodas motorizada Invacare M1 equipada com um computador embarcado, codificadores nas rodas e um gerenciador de bateria. A solução proposta é baseada na utilização de uma câmera *USB Logitec QuickCam Webcam* monocular calibrada capaz de gerar imagens a uma taxa de 10 quadros por segundo na resolução de 1280 x 960 pontos. Conforme pode ser observado na Figura 2, a câmera foi fixada em uma haste acoplada na parte de trás do veículo para proporcionar um campo de visão mais amplo. Os autores argumentam que outros sensores, tais como scanners a laser ou radares, poderiam também ter sido utilizados, mas eles são muitas vezes dispendiosos, pesados e com um consumo de energia significativo. Todo o processamento ocorre sobre o sistema ROS instalado em um computador de baixo custo com processador Intel Atom 1.8GHz, 2 GB de memória RAM e algumas portas USB. A biblioteca de computação visual utilizada foi OpenCV. O artigo descreve o algoritmo de rastreamento

Figura 2: Cadeira de Rodas Invacare M1 com câmera USB acoplada. (MIRO; POON; HUANG, 2012, p. 2)

do alvo à frente. Simplificadamente pode-se dizer que a câmera captura e transmite as imagens quadro a quadro para serem analisadas. Características de tom e saturação são processadas para identificar o quadro verde. Aplica-se uma técnica de processamento de imagens chamada de *back-projecton* na qual tenta-se identificar a distribuição de pontos de uma dada imagem num modelo de histograma. Os resultados são convertidos em uma série de contornos, sendo o de maior área tomado como as linhas do alvo. Aplica-se um filtro de Kalman (método de filtragem linear e predição a partir de medições ruidosas) para reduzir os erros nas coordenadas do retângulo alvo. Em seguida, identifica a posição

do alvo em relação a cadeira de rodas (posição inicial conhecida). Finalmente determina a posição à qual o veículo deve se deslocar. Foram conduzidos dois experimentos para validar a abordagem proposta. Em ambos experimentos, o alvo foi transportado por uma pessoa para que a cadeira pudesse seguí-lo de forma autônoma. Os autores afirmam que os resultados obtidos mostraram que um sistema exclusivamente baseado em computação visual foi capaz de seguir o alvo e realizar a trajetória com boa precisão (erros de posicionamento menores do que 0,5m).

Lane *et al.* (2012) argumenta que, nos últimos anos, houve uma grande valorização do reuso de componentes para robótica e que o surgimento do ROS é um bom exemplo desse fato. Apesar disso, os autores afirmam que pouco se fez para componentes voltados à robôs interativos baseados em gestos e reconhecimento por voz que demandam conhecimento em disciplinas variadas como computação visual, processamento acústico, reconhecimento de linguagem natural e muitas outras. Diante disso, o texto propõe a criação de um *framework* para ROS que facilite o desenvolvimento de componentes para interação com os usuários. Chamado de HRItk (*Human-Robot-Interaction toolkit*), o *framework* consiste em protocolos de mensagens, alguns componentes de *software* e ferramentas de desenvolvimento que facilitam e permitem um rápido desenvolvimento de sistemas interativos por comandos de voz. A Seção 2 do artigo traz uma visão geral da arquitetura do HRItk. São apresentadas duas classes de componentes: *Understanding nodes* (nós de tradução) e *Tracking Services* (serviços de rastreio). O primeiro é responsável por reconhecer os comandos dos usuários. Para isso, ele recebe dados de sensores e de nós intermediários capazes de gerar hipóteses. O segundo atua com as intenções dos usuários numa linha de tempo mais longa. Além disso, cinco componentes desenvolvidos para ROS foram detalhados:

- Detecção e Reconhecimento de Voz (baseado na implementação de *Julius Speech Recognition Engine* (LEE; KAWAHARA, 2009);
- Reconhecimento de Linguagem Natural criado utilizando-se de *Conditional Random Fields* (LAFFERTY; MCCALLUM; PEREIRA, 2001);
- Reconhecimento de Gestos, criado utilizando o sensor *Kinect* e o trabalho de Fujimura e Xu (2007);
- Monitoramento do ponto focal dos olhos, utilizando-se de técnicas do tipo *Average of Synthetic Exact Filters (ASEF)* e projeções geométricas;
- Monitoramento de Diálogos (*Dialog State Tracking*) baseado na biblioteca *Hound Belief Tracking* desenvolvida pelo Instituto de Pesquisa Honda (USA) que mapeia as intenções dos usuários na forma de gráficos de dependência probabilística (rede Bayesiana).

Apesar da ausência de dados experimentais no texto do artigo, ele afirma que foi criado um procedimento de geração automática da solução e que foi desenvolvido um exemplo de implementação.

No artigo Li, Oskoei e Hu (2013), os autores argumentam que o envelhecimento da população e o aumento no número de pessoas com limitações físicas ou cognitivas são realidades globais. Como resultado, o custo com cuidados a esses indivíduos cresce rapidamente. Segundo os autores, Tecnologias de Informação e Comunicação (TIC) podem contribuir significativamente na resolução desse problema. São apresentados conceitos de sistemas auxiliares para ambientes domésticos (*Ambient Assisted Living - AAL*) que envolvem diversos dispositivos com recursos e funções variadas (sensoriamento, análise, interação e comunicação). O artigo afirma que já existe uma grande variedade de produtos desenvolvidos para esse fim, porém há um outro desafio: Coordenar e Integrar todas essas soluções para que, de forma colaborativa, elas consigam auxiliar pessoas com limitações em ambientes domésticos. A Seção 3 do artigo descreve um ambiente doméstico em que as mobílias possuem etiquetas RFID, sensores de temperatura, câmeras, um robô humanóide e uma cadeira de rodas robotizada. Todos estes elementos juntos, trocando informações através de rede sem fio, são apresentados como possível solução para AAL que, além de servir e monitorar a saúde do idoso ou deficiente, pode também facilitar o trabalho de acompanhantes e familiares. O trabalho concentra-se, entretanto, em parte dessa solução. Os autores desenvolveram uma interface de comunicação (*communication bridge*) utilizando a infraestrutura do (*Robot Operating System (ROS)*). As motivações pela escolha do ROS são destacadas em diversas partes do texto. Disponível gratuitamente e mantido por uma comunidade de pesquisadores, ROS permite compartilhar conhecimento. Ele possui muitos módulos e bibliotecas de *software* para controle de robôs que estão prontos para uso como, por exemplo, "Navegação Autônoma". Dessa forma, torna-se ideal para desenvolvimento rápido de sistemas complexos com custos reduzidos. A camada de *software* desenvolvida é do tipo cliente/servidor e permite que diferentes tipos de dispositivos sirvam como interface ao usuário. Um acompanhante pode, por exemplo, utilizar seu próprio celular para enviar uma mensagem de texto com comandos em linguagem natural para deslocar uma cadeira de rodas de um determinado ponto a outro da casa. Para demonstrar os resultados utilizou-se uma cadeira de rodas motorizada comercial equipada com computador integrado para controle de sensores e motores e um outro computador portátil para controle em mais alto nível. Este último possui a aplicação desenvolvida em ROS que permite navegação autônoma e uma interface de usuário. A cadeira de rodas foi capaz de chegar a um destino definido pelo usuário.

Li *et al.* (2013) afirmam em seu artigo que as cadeiras de rodas inteligentes ainda são inviáveis economicamente à grande maioria da população. Há uma grande distância entre o ambiente acadêmico e a realidade econômica da população. Para reduzir o custo das cadeiras inteligentes, segundo os autores, seria necessário desenvolver um pa-

drão universal de *hardware* e *software* que pudesse ser oferecido a todos. Diante disso, o artigo apresenta o ROS destacando uma de suas principais características: Sistema de código aberto que possui uma coleção completa de *software* para controle robótico, dentre os quais são citados módulos de navegação, mapeamento, controle de movimento, visão computacional e reconhecimento de voz. Além disso, são citados também módulos de controle mais completos como o robô de resgates e o projeto de robô humanóide. Todo esse conteúdo integrado no ROS permite a troca de conhecimento entre a comunidade de pesquisa e de desenvolvedores de todo o mundo, fazendo do ROS um sistema capaz de reduzir custos de desenvolvimento. O projeto descrito no artigo utiliza uma cadeira de rodas motorizada comercial equipada com computadores dedicados, placas de controle, rede sem fio, dois *scanners* laser Hokuyo, 12 sensores sonar, um sensor de movimento Mongoose com 9 graus de liberdade, uma câmera, um microfone e GPS. O computador embarcado está ligado aos sensores e, também, roda um programa com arquitetura cliente/servidor. A aplicação cliente coleta os dados e os transmite ao servidor, que, em seguida retorna instruções para atuar nos motores. A segunda seção do artigo trás detalhes da arquitetura geral da solução, com um pouco de destaque para o controlador Proporcional - Integral - Derivativo (PID) para controle de tração. A solução final permite navegação autônoma e aceita comandos do usuário. Os resultados são apresentados na Seção 4 do artigo e os testes foram realizados no próprio ambiente de pesquisa. Os parâmetros do controlador PID (Kp, Kd e Ki) foram sendo ajustados experimentalmente até que o movimento da cadeira fosse assertivo. Em seguida, gerou-se um mapa do ambiente (mostrado pela Figura 3) a partir dos dados capturados com os sensores laser durante um movimento exploratório. O passo seguinte foi, então, testar a solução integrada. O usuário pode definir um ponto de destino no mapa e a cadeira foi capaz de atingí-lo.

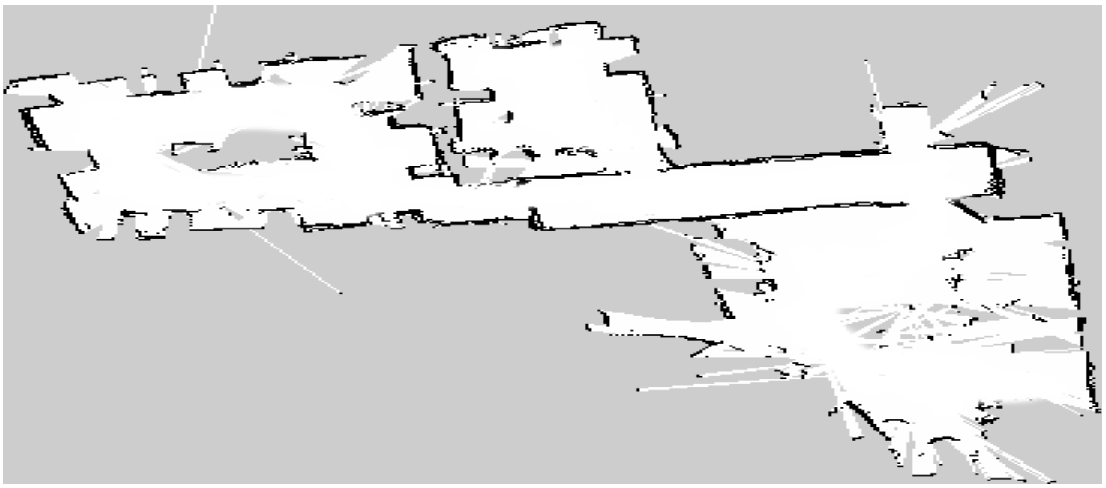

Figura 3: Mapa do ambiente gerado a partir dos dados coletados pelos sensores laser. (LI *et al.*, 2013, p. 4)

Bayar *et al.* (2014) apresenta um trabalho cujo diferencial é a utilização do ROS.

Eles selecionaram um problema clássico de controle de cadeiras de rodas (deslocamento autônomo até uma determinada posição e desvio de obstáculos). Foi escolhido, também, utilizar lógica fuzzy que, segundo os próprios autores, já foi explorado por muitos outros autores. O diferencial deste trabalho seria, então, a aplicação dessas tecnologias em conjunto com a plataforma ROS. Na introdução do artigo são citados vários trabalhos tomados como referência, mas que, em sua maioria, são modelados em Matlab e depois desenvolvidos em linguagem C/C++ ou em Labview. Os autores afirmam não ter encontrado um estudo que empregue lógica fuzzy em conjunto com ROS, apesar deste último ter se tornado bastante popular para desenvolvimento de aplicações em Robótica. O trabalho foi desenvolvido com base na cadeira de rodas inteligente ATEKS, sendo esta palavra um acrônimo da frase em Turco *Adet Akıllı Tekerlekli Sandalye* que significa "Cadeira de rodas inteligente total" ou "Cadeira de rodas totalmente autônoma". A Figura 4 extraída do artigo possui setas indicativas dos principais componentes utilizados para controle da cadeira. A Seção 2 do artigo trás detalhes técnicos desse equipamento cujo grande diferencial é possuir sensores de baixo custo e *software* de código aberto. Além disso é possível contar com um simulador GAZEBO antes de utilizar o equipamento real.

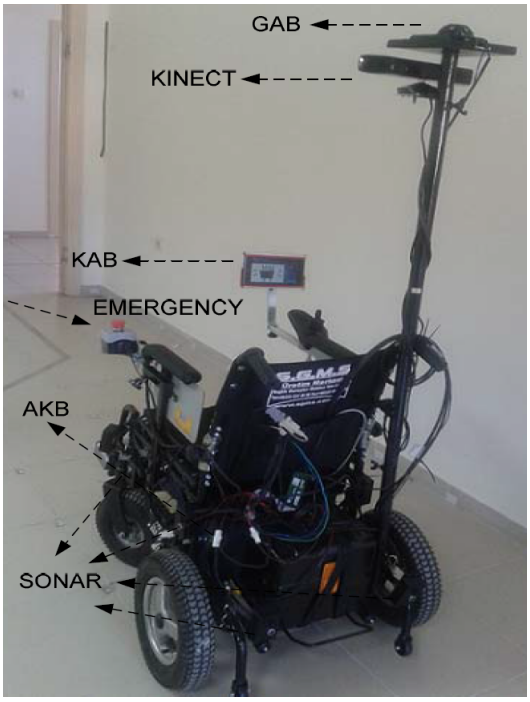

Figura 4: Cadeira de Rodas Inteligente ATEKS. (BAYAR *et al.*, 2014, p. 2)

Ainda na Seção 2 é detalhada a implementação fuzzy. São apresentadas tabelas com as principais regras fuzzy para cada parte do controle. As inferências fuzzy foram criadas com a ajuda de uma aplicação chamada qtfuzzylite e a implementação fuzzy foi feita com uso da biblioteca fuzzylite (uma biblioteca de uso gratuito, com código fonte aberto, programada em C++ e com suporte a múltiplas plataformas que permite criar controladores baseados em lógica fuzzy a partir de programação orientada a objetos). Foi

criada uma camada de adaptação (*bridge*) para ROS que integra os resultados inferidos através da lógica fuzzy com o simulador GAZEBO. Vários testes foram realizados com alteração dos cenários com objetivo de avaliar o comportamento de partes específicas do controle (somente deslocamento autônomo ou deslocamento autônomo com obstáculos). A análise das velocidades angular e linear indicam o predomínio do controle de desvio de obstáculos. Os autores concluem que sua implementação com o ATEKS foi capaz de realizar as tarefas.

O artigo Ozcelikors *et al.* (2014) cita a definição de Simpson, LoPresti e Cooper (2008) para cadeiras de rodas inteligentes: É geralmente uma cadeira de rodas motorizada composta por um computador e uma coleção de sensores ou um robô móvel ao qual foi adicionado um assento. Segundo os autores, esse tipo de equipamento é muito importante para pessoas idosas ou com limitações físicas, pois lhes proporciona mobilidade de maneira segura. A demanda tende a aumentar e isso tem chamado a atenção da indústria. Entretanto ainda há uma distinção entre cadeiras de rodas comerciais ou para pesquisa e as acadêmicas. O artigo faz uma breve revisão técnica das principais cadeiras de rodas comerciais/pesquisa (JiaLong, iWheelchair, Tao Aicle, NavChair e SENARIO) e acadêmicas criadas para aplicações de inteligência artificial e para aprimorar técnicas de controle (MiiChair, Sharioto, SmartWheeler, Wheelesley, VAHM, OMNI-wheelchair e Mister Ed). O trabalho desenvolvido neste artigo, entretanto, utiliza a cadeira de rodas inteligente ATEKS equipada com dois controladores, sensores ultra-sônicos e sensor *Kinect*. A proposta do trabalho é criar uma máquina de estados finitos *(Finite State Machine)* utilizando *software* de código aberto (ROS, GAZEBO, ANDROID). O primeiro é uma plataforma de desenvolvimento de controle robótico que fornece os serviços esperados de um sistema operacional, incluindo abstração de *hardware*, *drivers* de dispositivos, implementações de controle comumente usados, passagem de mensagens entre processos, e gerenciamento de pacotes de *software*. Ele também fornece ferramentas e bibliotecas para obtenção, construção, escrita e execução de código. O Segundo é um simulador de ambientes em 3D que permite a modelagem de robôs com cinemática real e parâmetros dinâmicos como momento de inércia e atrito de forças. O terceiro é um é um sistema operacional móvel baseado no núcleo Linux. O Android foi utilizado para executar o algoritmo de controle do percurso. Dessa forma o processo de planejamento do ATEKS é externo. A Seção 2 traz detalhes das funcionalidades da plataforma ATEKS, com destaque para o sensor *Kinect* e do método de máquina de estados. Este último foi implementado utilizando-se a linguagem visual UML *(Unified Model Language)* e o *software Visual Paradigm* capaz de criar trechos de código a partir dos diagramas. Os autores decidiram criar o seguinte conjunto de estados: "Iniciar Missão" *(Start Mission)*, "Decidir" *(Decide)*, "Virar" *(Turn)*, "Deslocar-se no Ambiente" *(Move in Room)*, "Passar por Porta" *(Doorway Passing)*, "Navegar por Corredores" *(Follow Corridor)*, e "Missão Cumprida" *(Mission Accomplished)*. A Seção 3 traz detalhes da implementação do algoritmo de controle que combina a técnica

de campos potenciais com vetores do histograma. Dessa forma, a navegação é realizada por forças de repulsão e atração (construídas de acordo com os dados dos sensores) que sofrem transformações conforme as circunstancias encontradas durante o trajeto, resultando em velocidades lineares e angulares. O sensor *Kinect* gera dados 3D que precisam ser convertidos em um plano xy. Os dados do sensor sonar são também considerados no mapa 2D durante a execução de movimentos que precisem de detecção mais ampla e precisa, tais como passar por uma porta. Os resultados apresentados foram testados utilizando-se o simulador GAZEBO. Foram apresentados dois casos de teste muito parecidos. A Figura 5 apresenta uma visão superior em perspectiva do ambiente de testes ao qual o robô foi submetido. Nele, o robô teve que se deslocar entre salas passando por portas e corredores. Nos dois casos a missão foi concluída com sucesso. Em uma próxima etapa, os pesquisadores pretendem testar a solução com o equipamento real.

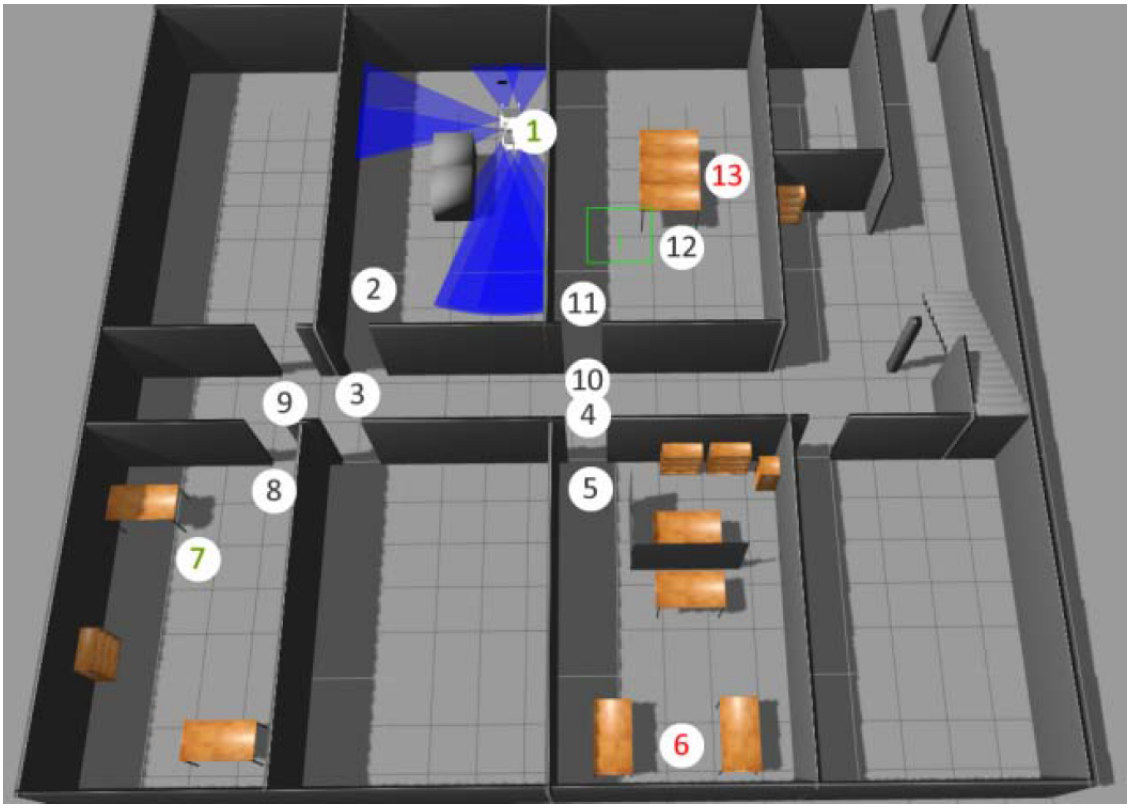

Figura 5: Cenário virtual de testes utilizando simulador GAZEBO. (OZCELIKORS *et al.*, 2014, p. 6)

Du *et al.* (2014) afirmam que a Realidade Virtual (RV) possui um futuro promissor no campo da reabilitação devido aos benefícios trazidos à fisioterapia dos pacientes. Segundo os autores, este trabalho traz um novo método para desenvolvimento de terapias com cenários virtuais e um robô exoesqueleto (com 5 graus de liberdade que permite mover ombro, cotovelo, pulso e antebraço) para reabilitação de pacientes que sofreram acidente vascular cerebral e que, por isso, perderam a capacidade de movimento de um dos lados

do corpo (Hemiplegia). Os exercícios consistem em movimentos com o braço e ombro, pois esta é uma das ações mais essenciais em nossas atividades diárias. A fisioterapia é, geralmente, realizada por um profissional e as técnicas utilizadas são bastante variadas. Entretanto o uso da robótica através de exoesqueletos apresenta algumas vantagens como o fornecimento de exercícios intensivos, consistentes e reproduzíveis, liberando os fisioterapeutas do intenso trabalho manual. O emprego da realidade virtual traz uma vantagem adicional ao tornar o tratamento mais divertido. Os pacientes ficam mais motivados em completar as tarefas. Os autores citam outros trabalhos que também exploram a RV para tratamento como o sistema MIMICS de Ziherl *et al.* (2010), o exoesqueleto L-EXOS de Frisoli *et al.* (2009) e o braço robótico de reabilitação HUST de Jun, Jian e Yongji (2010). No entanto, o ambiente virtual desses projetos é pré-definido e de difícil alteração após seu desenvolvimento. Além disso, perspectivas não podem ser alteradas de forma a melhor atender aos pacientes. O trabalho dos autores, então, supera essas limitações a partir da flexibilidade oferecida pelo uso das tecnologias ROS e Gazebo. A Figura 6, extraída da página 2 do artigo, mostra o equipamento utilizado pelos autores. A Seção 2 do artigo

Figura 6: Exoesqueleto com 5 graus de liberdade para reabilitação de pacientes com Hemiplegia. (DU *et al.*, 2014, p. 2)

descreve a metodologia do trabalho que foi dividida da seguinte maneira:

- *Hardware*: Utilizou-se um computador para controle e projeção do ambiente virtual conectado a um robô exoesqueleto fisioterápico com 5 graus de liberdade;
- Arquitetura de *Software*: Apresentou-se os sistemas Linux Ubuntu, ROS e Gazebo nos quais a solução foi desenvolvida;
- Modelo Humano: Criou-se uma representação humana utilizando o *software Sketchup 3D*. Assim como o robô, esse modelo possui 5 graus de liberdade;

- Controle do Modelo Humano: Desenvolveu-se seis nós ROS para controle do sistema. Um para controle da máquina de estados e outros cinco do tipo PID para controle do posicionamento de cada movimento do braço virtual. O humanóide virtual reposiciona o braço conforme o paciente movimenta o exoesqueleto. Os seis nós ROS possuem o papel de fornecedores de dados (*publishers*). Segundo os autores, para obter uma simulação realista foi necessário criar uma extensão de *software* para o simulador Gazebo (*plugin*) para que o modelo pudesse ter suas propriedades atualizadas dinamicamente. Essas extensões registram-se junto o tópico dos nós de controle e recebem os dados para atualização do modelo virtual;

- Comunicação: Criou-se um nó ROS adicional para a plataforma *Microsoft Windows*. Os autores tiveram que superar a limitação de possuir *driver* de controle do robô de reabilitação apenas para *Windows*. A arquitetura distribuída em rede do sistema ROS, entretanto, facilitou o trabalho de integração.

Com o objetivo de simular uma situação cotidiana, escolheu-se reproduzir uma cozinha como cenário virtual (Figura 7). Nela é possível realizar muitas tarefas como limpar mesas, abrir portas de armário, pegar vegetais e outros objetos das prateleiras. Os pacientes recebem as tarefas e podem observar os resultados dos movimentos em tempo real. Além disso, eles podem escolher diferentes perspectivas da cozinha para realizar as atividades. Como trabalho futuro, os pesquisadores pretendem desenvolver novos cenários virtuais e também criar sensações em resposta à interação com os objetos (*force feedback*).

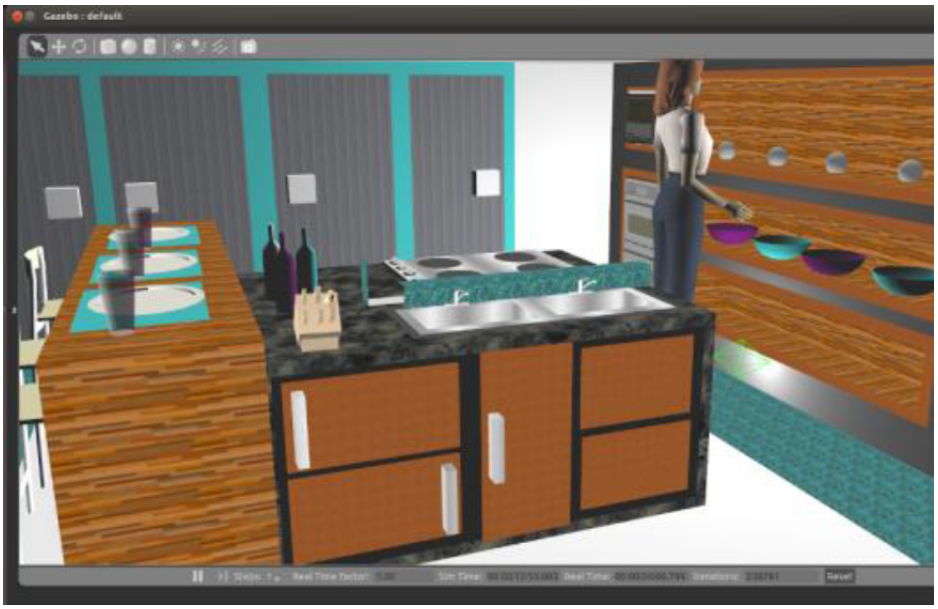

Figura 7: Cozinha Virtual criada no simulador GAZEBO e utilizada no tratamento de pacientes com Hemiplegia. (DU *et al.*, 2014, p. 5)

Pasteau, Krupa e Babel (2014) argumentam que a habilidade de locomoção é uma das necessidades básicas mais importantes dos seres humanos. Sua limitação afeta diretamente a dignidade e, até mesmo, a saúde mental das pessoas. Os autores afirmam que cadeiras motorizadas autônomas ou semi-autônomas já foram alvo de muitos estudos, mas sua importância social a torna item de grande interesse. O trabalho desenvolvido auxilia usuários de cadeiras motorizadas a passar por corredores estreitos, evitando colisões com as paredes. O texto destaca a característica auxiliar de controle em que o usuário continua pilotando a cadeira através do *joystick*, porém com auxílio do algoritmo que previne colisões com as paredes em um esquema que foi denominado *man-in-the-loop*. Outra característica da solução é sua capacidade de adaptação. Não é necessário conhecimento prévio sobre o corredor, pois o controle é capaz de atuar em ambientes desconhecidos. Segundo o artigo, a inovação deste trabalho foi unir o controle manual e o automático utilizando-se de computação visual e materiais de baixo custo (uma câmera monocular). As Seções dois a cinco trazem a teoria e os detalhes sobre o modelo da cadeira de rodas. São identificadas duas variáveis relacionadas ao movimento da cadeira: A velocidade de translação (deslocamento para frente e para trás) e a velocidade angular (mudança de direção). A orientação do equipamento em relação ao corredor é mapeada através de matrizes. Também foi definida uma função que retorna valores entre 0 e 1 que representam uma escala de segurança. Isso permite integrar os dois controles (manual e automático). O controle automático atua gradativamente conforme o índice de segurança deixa o intervalo considerado "seguro". A Seção 6 apresenta os resultados experimentais. Eles foram realizados em uma cadeira de rodas motorizada comercial da marca *Penny and Giles* adaptada para pesquisas em robótica utilizando o *middleware* ROS e uma câmera para *Raspberry PI*. O trabalho não menciona como o algoritmo de controle foi implementado em ROS. Os resultados apresentados mostram que ao se aproximar de uma parede, o risco de colisão aumenta conforme o usuário tenta se direcionar a ela. O sistema de controle, então, entra em ação e evita a colisão. Isso pode ser visto através de três gráficos, apresentados no artigo, que plotam o índice do risco de colisão, a velocidade translacional e a angular segundo comandos do operador e do controle automático. Dessa forma foi possível observar que, em momentos de maior risco de colisão, o controle automático teve prioridade sobre o controle manual. Os autores concluem que os resultados mostram que a solução proposta é eficiente para evitar colisões com as paredes de um corredor. São planejados mais experimentos com voluntários para se chegar a um resultado qualitativo. Eles afirmam, também, estar atuando em uma funcionalidade complementar, baseada nos mesmos princípios, que permitirá passar por vão de portas de maneira segura.

3 Desenvolvimento Teórico

3.1 ROS

Robot Operating System (ROS) é um meta sistema operacional amplamente utilizado na área da robótica. Sua filosofia é fazer com que trechos de código de uma aplicação possam, com pequenas mudanças, trabalhar em diferentes robôs. O objetivo é permitir a criação de funcionalidades que possam ser compartilhadas e utilizadas em outros robôs sem muito esforço, de modo a evitar redundância de trabalho (MARTINEZ, 2013, p. 8). ROS foi originalmente desenvolvido em 2007 pela *Stanford Artificial Intelligence Laboratory (SAIL)* com o apoio do projeto *Stanford AI Robot*. A partir de 2008, seu desenvolvimento continuou sendo feito pela Willow Garage (um instituto de pesquisa em robótica com mais de 20 instituições colaborativas). Distribuído sob os termos da licença de *software* BSD (*Berkeley Software Distribution*), ele tem o código fonte aberto e seu uso é livre tanto para pesquisa quanto para fins comerciais. Entretanto outros pacotes de *software* distribuídos em conjunto podem ter diferentes tipos de licença de *software* para código aberto.

Existem alguns livros sobre ROS, mas a maior fonte de informação sobre o ROS é o seu site oficial que possui uma área dedicada à documentação (ROS.ORG, 2015).

A arquitetura de sistema do ROS está divida conceitualmente em três partes:

- Sistema de Arquivos (*Filesystem level*);
- Arquitetura Computacional (*Computation Graph level*);
- Colaboração (*Community level*).

3.1.1 Sistema de Arquivos

Sistema de arquivos é um conceito aplicado a todos os recursos que podem ser encontrados no disco (memória secundária) do sistema. Assim como em um sistema operacional, um programa ROS está distribuído em pastas cujo conteúdo inclui arquivos que descrevem sua funcionalidade. A Figura 8 mostra como esses recursos estão organizados.

- Pacotes (*Packages*): São os principais elementos na organização de *software*. Eles podem conter processos ROS de tempo de execução (*Nodes*), bibliotecas específicas, coleções de dados (*dataset*), arquivos de configuração e qualquer outro artefato cuja organização seja significativa ao sistema. São os elementos de menor granularidade que pode ser criado e distribuído;

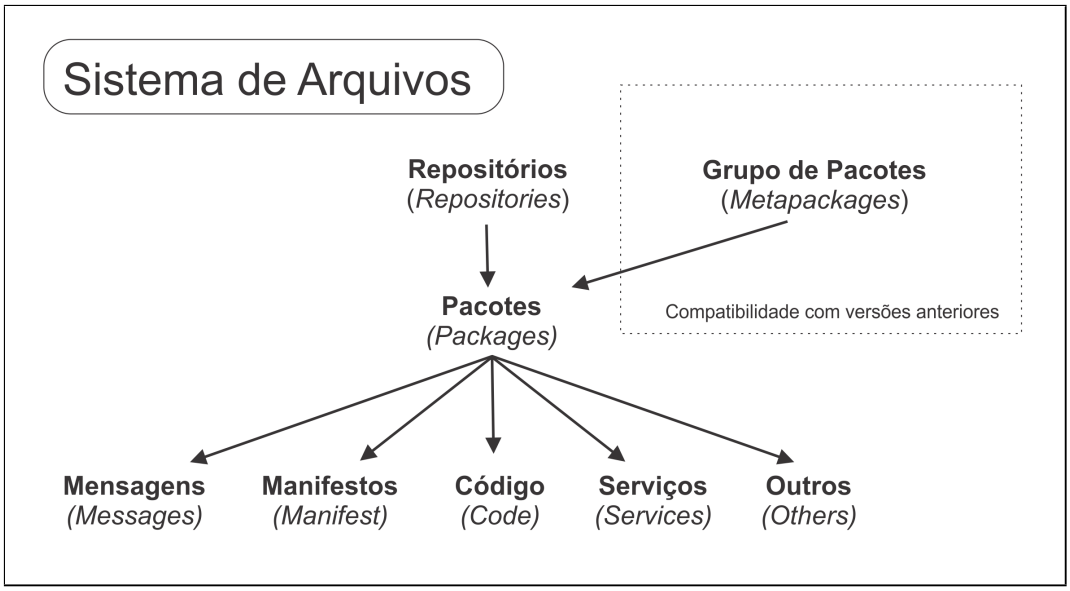

Figura 8: Sistema de Arquivos ROS. Figura criada a partir da imagem encontrada em Martinez (2013, p. 26)

- Grupo de Pacotes (*Metapackages*): São pacotes especializados em representar outros grupos de pacotes relacionados;
- Manifestos (*Package Manifests*): São arquivos XML (package.xml) que possuem informações sobre um pacote como nome, versão, descrição, licença de uso, dependências e dados sobre outros pacotes exportados;
- Repositórios (*Repositories*): É uma coleção de pacotes que compartilham um mesmo sistema de controle de versão de código fonte. Pacotes com a mesma versão podem ser liberados em conjunto. Repositórios também podem conter apenas um pacote;
- Tipo de dados "Mensagem" (msg): Mensagem é a informação que um processo envia a outro. ROS define várias mensagens padrão, cuja estrutura de dados fica armazenada em "my_package/msg/MyMessageType.msg";
- Tipo de dados "Serviço" (srv): Define a estrutura de dados para envio/resposta de serviços. Essa descrição fica armazenada em "my_package/srv/MyServiceType.srv".

3.1.2 Arquitetura Computacional

ROS possui arquitetura de rede ponto-a-ponto onde cada nó da rede funciona tanto como cliente quanto servidor, eliminando a necessidade de um processo centralizador. Os principais elementos da arquitetura computacional do ROS são:

- Nós (*Nodes*): São processos que realizam alguma atividade computacional específica. ROS é um sistema modular e o controle de um robô geralmente compreende muitos

nós. Por exemplo, um nó controla o sensor laser, outro a tração das rodas, outro determina a trajetória e assim por diante. Para se desenvolver um nó ROS é preciso utilizar a biblioteca roscpp ou rospy;

- Mestre (*Master*): O ROS Mestre permite a um nó registrar-se e realizar buscas por outros nós. Sem ele os nós não seriam capazes de localizar uns aos outros para trocar mensagens ou chamar serviços;

- Servidor de Parâmetros (*Parameter Server*): Permite armazenar dados indexados em um repositório central. Ele faz parte do Mestre;

- Mensagens (*Messages*): São estruturas de dados utilizadas pelos nós para se comunicar. Os tipos de dados compreendidos são primitivos (Inteiro, ponto flutuante, booleano e outros);

- Tópicos (*Topics*): As mensagens são trocadas através de um sistema de transporte com semântica publicar (*publish*)/inscrever-se (*subscribe*). Os nós enviam mensagens através da sua publicação a um determinado tópico. O tópico é um nome usado para identificar o conteúdo da mensagem. Um nó que está interessado em um determinado tipo de dados vai se inscrever no tópico adequado. Podem haver vários concorrentes para um único tópico e um único nó pode publicar ou inscrever-se em vários tópicos. Em geral, fornecedores de conteúdo e clientes não estão cientes da existência uns dos outros. Isso desacopla a produção de informações de seu consumo. Pode-se pensar em um tópico como um barramento de mensagens. Cada barramento tem um nome e qualquer nó pode conectar-se a ele para enviar ou receber mensagens;

- Serviços (*Services*): O modelo publicar/inscrever-se é um modelo de comunicação muito flexível, mas esse tipo de transporte muitos-para-muitos e com mensagens em uma única direção não atende quando se deseja trocar informações no modelo pedido (*request*)/resposta (*response*) muito comum em sistemas distribuídos. O pedido/resposta é feito por meio de serviços que são definidos por um par de estruturas de mensagens: uma para o pedido e outro para resposta. Um nó que gera conteúdo (publicando) oferece um serviço com um determinado nome e um cliente utiliza-se desse serviço enviando a mensagem de pedido e aguarda a resposta. Bibliotecas cliente do ROS geralmente apresentam essa interação para o programador como se fosse uma chamada de procedimento remoto (*RPC - Remote procedure Call*);

- Sacolas (*Bags*): As sacolas são um formato para guardar e reproduzir mensagens de dados do ROS. Elas são um importante mecanismo para armazenamento de dados, tais como dados de sensores, que podem ser difíceis de recolher, porém necessários para desenvolver e testar algoritmos.

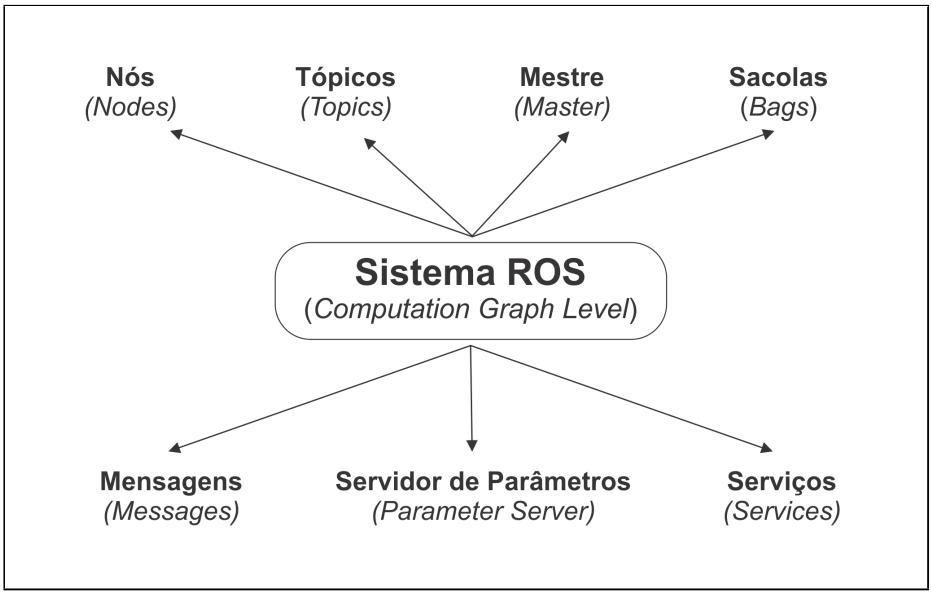

Figura 9: Sistema de Arquivos ROS. Reprodução da imagem encontrada em Martinez (2013, p. 32)

3.1.3 Colaboração

Colaboração é um conceito muito importante para ROS. Ele permite que diferentes grupos de pesquisadores e desenvolvedores troquem *software* e conhecimento. A seguir estão os principais recursos colaborativos:

- Distribuições (*Distributions*): Distribuições ROS são um conjunto de Pilhas (*Stacks*) que podem ser instaladas. São muito parecidas com uma distribuição Linux: Facilitam a instalação de uma coleção de aplicações de *software* e também garantem compatibilidade entre as partes instaladas;
- Repositórios (*Repositories*): ROS conta com uma rede federada de repositórios de código, onde as diferentes instituições podem desenvolver e distribuir seus próprios componentes de *software* para robôs;
- ROS Wiki: Trata-se de um site na Internet que é a principal fonte de informação sobre o ROS;
- Lista de Discussão (*Mailing lists*): A lista de discussão de usuários ROS é o principal canal de comunicação sobre novas atualizações para ROS, bem como um fórum para fazer perguntas.

3.1.4 Criação de Nós (*ROS Nodes*)

Conforme visto nas subseções anteriores, ROS possui vários elementos que compõem sua arquitetura. Para utilizá-lo é preciso ter alguns conhecimentos como saber

criar um ambiente de trabalho (*workspace*), conhecer as várias ferramentas de linha de comando, criar pacotes de código e compilá-los, utilizar o servidor de parâmetros e interagir com Tópicos e Serviços. A criação de Nós, entretanto, é o principal ponto de interesse quando se pretende desenvolver *software* para ROS, pois são eles que possuem algoritmos especializados e oferecem funcionalidades aos robôs.

3.1.4.1 Código Fonte Exemplo

Serão apresentados dois trechos de código fonte escritos em linguagem de programação C++ ("Exemplo 1" e "Exemplo 2") para mostrar como criar Nós ROS. O primeiro exemplo irá publicar dados e o segundo irá consumí-los. Esta é a maneira usual de comunicação entre Nós. Além disso, comentar-se-á os trechos mais importantes de cada código.

A Figura 10 apresenta o código fonte do "Exemplo 1".

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example1_a");
  ros::NodeHandle n;
  ros::Publisher pub = n.advertise<std_msgs::String>("message", 1000);
  ros::Rate loop_rate(10);
  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << " I am the example1_a node ";
    msg.data = ss.str();
    //ROS_INFO("%s", msg.data.c_str());
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Figura 10: Exemplo 1. Extraído do livro (MARTINEZ, 2013, p. 53)

A primeira parte de um programa em C/C++ é, geralmente, a inclusão de outros códigos (dependências). O trecho de código seguinte destaca a inclusão dos cabeçalhos comumente usados para se criar um Nó ROS. "ros/ros.h" inclui todos os cabeçalhos de função e classes mais comuns do sistema ROS e "std_msgs/String.h" inclui definições de mensagens que serão utilizadas pelo nó.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

É necessário instanciar o nó e definir um nome a ele. Esse nome deve ser único. É importante, também, que a função "ros::init()" seja chamada antes de qualquer outra do sistema ROS.

```
ros::init(argc, argv, "example1_a");
```

O trecho seguinte declara uma variável do tipo "ros::NodeHandle". Ela é a principal referência para manipular o processo criado. O primeiro *NodeHandle* construído irá inicializar totalmente este nó, e o último destruído irá encerrar o nó.

```
ros::NodeHandle n;
```

A função "advertise()" permite dizer ao ROS que se deseja publicar em um determinado tópico. Isso resulta em uma chamada ao nó mestre que mantém um registro dos processos que publicam e dos que se inscrevem. Após o anúncio, o Mestre irá notificar qualquer processo que esteja tentando se inscrever a este tópico. Isso fará com que os processos interessados nesse tópico iniciem uma negociação para estabelecer uma conexão ponto-a-ponto com o nó divulgado. Além disso, a função "advertise()" retorna um objeto do tipo *Publisher* que permite publicar mensagens do tópico criado através da função "publish()". O tópico criado será divulgado enquanto o objeto *Publisher* existir. O primeiro parâmetro (*message*) passado à função "Advertise()" no trecho de código a seguir representa o nome do tópico e o segundo parâmetro "1000" é o tamanho da área de armazenamento.

```
ros::Publisher pub = n.advertise<std_msgs::String>("message", 1000);
```

O código seguinte determina a frequência com que este nó irá gerar dados. O parâmetro "10" significa "10Hz".

```
ros::Rate loop_rate(10);
```

A execução cíclica do programa está condicionada ao resultado da chamada à função "ros::ok()". Ela continuará funcionando enquanto não houver o sinal de término gerado pelas teclas "Ctrl + C" ou o ROS comandar a parada de todos os nós.

```
while (ros::ok())
{
```

A chamada à função "publish()" permite publicar a mensagem criada a todos os nós inscritos.

```
    pub.publish(msg);
```

A chamada à função "sleep()" irá suspender o processo o tempo que for necessário para que as mensagens sejam publicadas em uma frequência de 10Hz.

```
    loop_rate.sleep();
```

A Figura 11 apresenta o código fonte do "Exemplo 2". Este código cria um nó que irá se registrar para receber mensagens do nó "Exemplo 1". Além disso, ele irá imprimir na tela as mensagens recebidas.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void messageCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example1_b");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("message", 1000, messageCallback);
  ros::spin();
  return 0;
}
```

Figura 11: Exemplo 2. Extraído do livro (MARTINEZ, 2013, p. 54)

Assim como no "Exemplo 1", o código do "Exemplo 2" inclui os cabeçalhos padrão para se criar um nó ROS.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

Além da função *main*, este programa implementou a função *messageCallback*. Ela será registrada junto ao ROS para ser chamada sempre que houver mensagem destinada a este nó. As mensagens recebidas serão impressas na tela pela função ROS_INFO que possui uma sintaxe muito parecida com o *printf* da biblioteca padrão da linguagem C. Entretanto existem algumas diferenças dentre as quais podemos citar a inclusão automática de data e hora da impressão e cópia das mensagens para o registro (*log*) do sistema (muito útil para depuração de código).

```
void messageCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

O método *subscribe* do objeto "NodeHandle n" recebe três parâmetros: o nome do tópico (*message*), o tamanho da área de armazenamento ("1000") e o nome da função deste nó que será responsável por tratar as mensagens recebidas ("messageCallback"). O resultado dessa chamada será o registro desse nó como cliente de um determinado tópico. Além disso, *subscribe* retornará uma instância do objeto *Subscriber*.

```
ros::Subscriber sub = n.subscribe("message", 1000, messageCallback);
```

"spin()" é uma função cuja chamada é bloqueante. Dessa maneira o processo desse nó continuará existindo para receber mensagens enquanto não houver um comando de término através das teclas "Ctrl + C".

```
ros::spin();
```

Em seguida é necessário compilar os dois códigos fonte e executar os binários. Será necessário abrir uma caixa de comando (*command shell*) para cada um deles. Na caixa de comandos onde foi executado o "Exemplo 2" veremos impressões como as mostradas pela Figura 12:

```
...
[ INFO] [1355228542.283236850]: I heard: [ I am the example1\_a node ]
[ INFO] [1355228542.383221843]: I heard: [ I am the example1\_a node ]
[ INFO] [1355228542.483249861]: I heard: [ I am the example1\_a node ]
...
```

Figura 12: Texto impresso na tela pelo código do "Exemplo 2". Extraído do livro (MARTINEZ, 2013, p. 56)

Isso indica que "Exemplo 1" foi publicar um tópico e que o "Exemplo 2" foi capaz de inscrever-se nele. Além disso, o nó "Exemplo 2" foi capaz de receber as mensagens publicadas pelo nó "Exemplo 1".

3.2 GAZEBO

O'Kane (2013, p. 143) afirma que a arquitetura modular do ROS é uma das grandes vantagens desse sistema. Ela permite reduzir o tempo de desenvolvimento e testes, pois pode-se facilmente trocar componentes do sistema. Outra grande vantagem é a possibilidade de utilizar o Gazebo: Um simulador de robô com alta fidelidade. Com Gazebo é possível definir as características tanto do robô (ou robôs) quanto do ambiente (*World*) e, além disso, interagir com ele via ROS da mesma forma como faríamos com um robô real.

A maior fonte de informações sobre o simulador Gazebo é o seu *website* oficial (GAZEBO, 2015), onde é possível encontrar tutoriais básicos e avançados para utilização e desenvolvimento do simulador. Esse *website* traz um texto em sua página inicial que justifica a utilização de simuladores e, em especial, do Gazebo: Um simulador bem projetado torna possível rapidamente testar algoritmos e projetos de robôs. Além disso é possível realizar testes em condições realistas. O Gazebo permite simular populações de robôs em complexos ambientes internos ou externos. Ele possui simulações físicas realistas, elementos gráficos de alta qualidade e permite interação através de interfaces

programáticas (*API - Application Programming Interfaces*) ou visuais (*GUI- Graphical User Interface*).

O desenvolvimento do Gazebo começou em 2002, na *University of Southern California*. Os criadores originais eram Dr. Andrew Howard e seu aluno Nate Koenig. O conceito de um simulador de alta fidelidade partiu da necessidade de simular robôs em ambientes ao ar livre sob várias condições. Como um simulador complementar ao simulador *2D Stage* para ambientes internos, o nome Gazebo foi escolhido por ser o mais próximo de uma estrutura ao ar livre para um palco (referenciando-se ao *Stage*). Em 2009, John Hsu, engenheiro de pesquisas da *Willow Garage*, integrou o ROS ao Gazebo que passou a ser uma das principais ferramentas utilizadas pelos usuários do ROS. Em 2011 a *Willow Garage* começou a financiar o projeto Gazebo. Em 2012 a *Open Source Robotics Foundation (OSRF)* substituiu-a e passou a coordenar o projeto.

3.2.1 Elementos Principais

As subseções seguintes apresentam uma descrição simplificada dos principais elementos que compõem o simulador Gazebo. Essas informações foram extraídas do Tutorial Gazebo Components (2015).

3.2.1.1 Arquivo Descritivo de Ambientes (*World Files*)

O arquivo de descrição de ambientes (*World Files*) contém todos os elementos que serão simulados como robôs, sensores, iluminação e objetos estáticos. Ele possui extensão ".world" e formatação do tipo SDF (*Simulation Description Format*), cuja sintaxe está especificada em um documento eletrônico (SDF Format Specification (2015)). O servidor Gazebo (gzserver) lê este arquivo para criar o cenário virtual. O instalador do simulador Gazebo inclui vários arquivos exemplo que podem ser encontrados no subdiretório "worlds" cujo caminho de padrão é "<local_instalação>/share/gazebo-<versão>/worlds".

3.2.1.2 Arquivos Modelo (*Model Files*)

Arquivos Modelo definem elementos tridimensionais que podem ser utilizados em ambientes virtuais. Ele forma um conjunto de definições de ligações (*links*), junções (*joints*), colisões (*collision*), aparência (*visual*), dinâmica (*inertial*) e extensões (*plugins*). Um modelo pode definir elementos simples (formas básicas) e complexos (Robôs). Assim como os arquivos descritivos de ambiente, os arquivos modelo possuem formatação SDF. Entretanto eles contém um único modelo delimitado pelas marcações "<model> ... </model>". A Figura 13 apresenta um modelo exemplo que foi extraído do tutorial Make a Model (2015). Ele define uma caixa que não pode ser movida (estática). Nota-se que

o atributo "<static>true</static>" determina sua característica dinâmica como sendo estática. Seria necessário atribuir valor *false* para permitir que a caixa se deslocasse.

```
<?xml version='1.0'?>
<sdf version="1.4">
<model name="my_model">
  <pose>0 0 0.5 0 0 0</pose>
  <static>true</static>
    <link name="link">
      <inertial>
        <mass>1.0</mass>
        <inertia> <!-- interias are tricky to compute -->
          <!-- http://answers.gazebosim.org/question/4372/the-inertia-matrix-explained/ -->
          <ixx>0.083</ixx>       <!-- for a box: ixx = 0.083 * mass * (y*y + z*z) -->
          <ixy>0.0</ixy>         <!-- for a box: ixy = 0 -->
          <ixz>0.0</ixz>         <!-- for a box: ixz = 0 -->
          <iyy>0.083</iyy>       <!-- for a box: iyy = 0.083 * mass * (x*x + z*z) -->
          <iyz>0.0</iyz>         <!-- for a box: iyz = 0 -->
          <izz>0.083</izz>       <!-- for a box: izz = 0.083 * mass * (x*x + y*y) -->
        </inertia>
      </inertial>
      <collision name="collision">
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

Figura 13: Modelo de uma caixa simples em formato SDF (MAKE A MODEL, 2015)

Para incluir modelos no arquivo *world* é necessário utilizar a sintaxe do arquivo SDF conforme mostrado abaixo:

```
<include>
  <uri>model://nome_do_arquivo</uri>
</include>
```

3.2.1.3 Variáveis de Ambiente (*Environment Variables*)

O Gazebo utiliza-se de variáveis de ambiente para localizar arquivos e estabelecer comunicação entre seu processo servidor e os clientes. Não é necessário definir essas variáveis para utilizar os valores mais comuns, pois eles já fazem parte da compilação padrão dos binários Gazebo. Para modificar esses valores é necessário alterar o arquivo "<local_instalação>/share/gazebo/setup.sh" e, em seguida, aplicar as alterações na sessão de comandos atual utilizando-se do comando *Unix "source"*.

3.2.1.4 Servidor Gazebo (*Gazebo Server*)

O servidor Gazebo é o principal processo do sistema. Ele carrega as informações contidas no arquivo ".world" e, em seguida, simula o ambiente virtual com características realistas proporcionadas pelas bibliotecas de simulação física. Para iniciar o gzserver a partir da interface de linha de comando é necessário indicar o arquivo SDF com a descrição do cenário:

gzserver <nome_do_arquivo.world>

A instalação padrão do sistema traz alguns arquivos ".world" que podem ser encontrados em "<local_instalação>/share/gazebo-<versão>/worlds".

3.2.1.5 Interface Gráfica Cliente (*Graphical Client*)

A interface gráfica cliente (gzclient) estabelece uma conexão com o servidor (gzserver) e permite visualizar os elementos renderizados. Com ela é possível, também, realizar algumas alterações no ambiente simulado. Outra interface gráfica disponível é o gzweb que permite visualização a partir de um navegador para Internet.

3.3 MATLAB

Os aplicativos MATLAB e Simulink são amplamente utilizados para prototipagem de algoritmos e modelos. A partir da versão 2014, o MATLAB passou a contar com suporte a desenvolvimento de aplicações para o sistema ROS. Entretanto, há diferenças entre as versões 2014 e 2015. O primeiro requer a instalação de um pacote complementar chamado "ROS I/O". O segundo incorpora essas funcionalidades através do *Robotics System Toolbox* que, adicionalmente, oferece uma interface entre o MATLAB Simulink e o ROS. Além disso, a versão 2015 amplia os recursos disponíveis, como suporte ao servidor de parâmetros, análise de dados gravados (*bags*) e capacidade de gerar código em C++ a partir de modelos Simulink que podem produzir aplicações para rodar em ambientes Linux.

3.3.1 Pacote ROS I/O

A instalação do pacote ROS I/O estende as funcionalidades do MATLAB, versão 2014, permitindo interação direta com o sistema ROS. Dessa forma, é possível testar os algoritmos criados no ambiente MATLAB em conjunto com os outros aplicativos ROS. Todos os nós ROS estarão disponíveis permitindo que a aplicação em teste possa se registrar ou publicar Tópicos e, assim, receber/transmitir dados. Segundo o Guia de Introdução ao pacote de extensão ROS I/O (The MathWorks Inc., 2014), ele estende

a *API* rosjava e disponibiliza novas funções que permitem criar nós ROS a partir das aplicações criadas em MATLAB. As principais funcionalidades oferecidas são:

- Criação de novos nós que irão publicar dados (*publisher*) ou se inscrever para recebê-los (*subscriber*);
- Divulgação de Tópicos;
- Criação de novas mensagens ROS;
- Publicação de dados gerados através do MATLAB a outros nós inscritos num determinado tópico;
- Instanciação do nó mestre na máquina local quando não há outro disponível para conexão remota.

3.3.1.1 Criação de Nós ROS com ROS I/O

A Seção 3.1.4 apresenta um exemplo de como criar nós ROS utilizando-se linguagem de programação C++. Nesta seção será mostrado como criá-los no MATLAB a partir dos recursos disponibilizados pelo pacote ROS I/O. Além disso, serão comentados os principais trechos de código. A Figura 14 apresenta um *script* exemplo para MATLAB que cria um nó ROS que faz os dois papéis simultaneamente: publica e também se inscreve em um determinado tópico. Em computadores que possuam o MATLAB e o pacote ROS I/O instalados, este exemplo poderá sem encontrado no seguinte endereço: "matlabroot\toolbox\psp\rosmatlab\examples\rosmatlab_basic.m"

Todo nó ROS deve se conectar a um nó mestre. Neste exemplo, assumiu-se que não há tal mestre disponível, então utilizou-se o seguinte comando para iniciar um mestre localmente que irá atender requisições na porta de rede TCP/IP número 11311.

```
roscore = rosmatlab.roscore(11311);
```

O comando seguinte cria um novo nó. Ele recebe dois parâmetros: O primeiro é o nome do nó (*NODE*) e o segundo é o endereço de rede do nó mestre cujo valor é retornado pela variável roscore.*RosMasterUri*. A variável "node" recebe o objeto nó instanciado.

```
node = rosmatlab.node('NODE',roscore.RosMasterUri);
```

A função "rosmatlab.publisher" registra um nó junto ao mestre como um nó gerador de informação (*publisher node*). Ela recebe três parâmetros: O primeiro é o nome do Tópico que, no exemplo a seguir, é "TOPIC". O segundo é o tipo da mensagem (std_msgs/String) e o terceiro é o objeto nó (variável "node").

```
publisher = rosmatlab.publisher('TOPIC','std_msgs/String',node);
```

```
%% Creating a Publisher and a Subscriber

% Launch a ROS master on port 11311 on localhost.
roscore = rosmatlab.roscore(11311);

% Create a new node named /NODE and connect it to the master.
node = rosmatlab.node('NODE',roscore.RosMasterUri);

% Add a publisher of a topic named /TOPIC to the node to send message of
% type std_msgs/String.
publisher = rosmatlab.publisher('TOPIC','std_msgs/String',node);

% Add a subscriber to a topic named /TOPIC to the node to receive message
% of type std_msgs/String.
subscriber = rosmatlab.subscriber('TOPIC','std_msgs/String',1,node);

% Set function1 and function2 to execute when a valid message is published
% to /TOPIC.
subscriber.setOnNewMessageListeners({@function1,@function2});

% Create a new message of type std_msgs/String.
msg = rosmatlab.message('std_msgs/String',node);

% Set the data field of the message and then publish the message.
msg.setData(sprintf('Message created: %s',datestr(now)));
publisher.publish(msg);
pause(1);
%% Reassigning Message Listener Tasks of a Subscriber

% Replace function1 and function2 in the standard message listener with
% function3.
subscriber.setOnNewMessageListeners({@function3});

% Update the data field of the message and then publish the message
% iteratively.
for i = 1:10
    msg.setData(sprintf('Iteration %d:\n  Message created: %s',i,datestr(now)));
    publisher.publish(msg);
    pause(1)
end

% Remove function3 from the standard message listener.
subscriber.setOnNewMessageListeners([]);

% Update the data field of the message and then publish the message.
msg.setData(sprintf('Message created: %s',datestr(now)));
publisher.publish(msg);
pause(1);

% Remove the subscriber from the node.
node.removeSubscriber(subscriber);

% Delete the master.
clear('roscore');


%% Clean up workspace.
clear;
```

Figura 14: *Script* em MATLAB para criação de Nós ROS. Distribuído com o pacote MATLAB ROS I/O

Neste exemplo, o nó que publica mensagens registrar-se-á também para recebê-las. A função “rosmatlab.subscriber” registra um nó junto ao mestre para receber mensagens de um determinado tópico. Ela recebe quatro parâmetros: O primeiro é o nome do Tópico ao qual deseja se registrar (“TOPIC”). O segundo é o tipo de dados das mensagens (std_msgs/String). O terceiro é o tamanho da área de memória para armazenar as mensagens (*buffer*) e o quarto é o objeto nó. O terceiro parâmetro foi definido com tamanho “1”. Isso significa que a fila de mensagens poderá conter apenas uma mensagem por vez. Caso a taxa de recebimento seja maior do que a de processamento, a mensagem pendente poderá ser substituída por uma nova.

```
subscriber = rosmatlab.subscriber('TOPIC','std_msgs/String',1,node);
```

É necessário definir a função que será chamada para tratar os dados que chegarem. Para isso, utiliza-se a função "subscriber.setOnNewMessageListeners". Neste exemplo, as funções " function1" e "function2" serão chamadas sempre que houver uma mensagem para o tópico "TOPIC".

```
subscriber.setOnNewMessageListeners({@function1,@function2});
```

Entretanto as funções passadas como parâmetro acima não foram definidas pelo código exemplo mostrado na Figura 14. Elas foram criadas em arquivos separados ("function1.m" e "function2.m") que podem ser encontrados no mesmo diretório onde está o arquivo "rosmatlab_basic.m". Abaixo é possível ver o conteúdo de ambos arquivos. Os dois recebem "message" como parâmetro. A função "function1" lê a mensagem recebida utilizando-se do método"getDate". A função "function" apenas imprime na tela a date e a hora em que foi chamada, não fazendo uso da mensagem recebida.

```
function function1(message)
disp(char(message.getData()));
end

function function2(message)
disp(sprintf('Message received: %s',datestr(now)));
end
```

O próximo comando do exemplo "rosmatlab_basic.m" irá instanciar um objeto "msg" do tipo "std_msgs/String".

```
msg = rosmatlab.message('std_msgs/String',node);
```

A sequência seguinte irá definir o conteúdo da mensagem e depois publicá-la a partir do método "publish" do objeto "publisher".

```
msg.setData(sprintf('Message created: %s',datestr(now)));
publisher.publish(msg);
```

A execução do programa até este ponto resultará em saídas como as mostradas na Figura 15. Os campos "dd-mmm-yyyy" e "HH:MM:SS" deverão apresentar a data e a hora atuais. A primeira saída é impressa pela função "Function1" e a segunda pela "Function2".

```
Message created: dd-mmm-yyyy HH:MM:SS
Message received: dd-mmm-yyyy HH:MM:SS
```

Figura 15: Saída das funções "Function1" e "Function2" quando o *script* "rosmatlab_basic.m" é executado - Figura do documento The MathWorks Inc. (2014, p. 14)

Após a instrução "pause", o programa exemplifica como alterar o nome da função registrada para tratar novas mensagens. Para isso utiliza-se o método "subscri-

ber.setOnNewMessageListeners" novamente passando o nome da nova função como parâmetro. Ao fazer isso, as funções "function1" e "function2" deixarão de ser chamadas e "function3" será a única a tratar novas mensagens.

```
subscriber.setOnNewMessageListeners({@function3});
```

O código da função "function3" também foi criado em arquivo separado ("function3.m") e está disponível junto ao arquivo "rosmatlab_basic.m". O código dessa nova função é:

```
function function3(message)
disp([char(message.getData()),sprintf('\n  Message received: %s',datestr(now))])
    ;
end
```

Para testar esse novo relacionamento com a função "function3", o código abaixo prepara e envia dez mensagens para o tópico "TOPIC". Cada uma das mensagens terá a data e hora da criação e, também, um número de interação que será incrementado de 1 a 10. A função "function3" irá imprimí-las na tela.

```
for i = 1:10
    msg.setData(sprintf('Iteration %d:\n  Message created: %s',i,datestr(now)));
    publisher.publish(msg);
    pause(1)
end
```

Para remover todas as funções registradas para receber mensagens do tópico "TOPIC" sem ter que destruir o nó, basta executar o método "setOnNewMessageListeners" passando um conjunto vazio como parâmetro. Ao fazer isso, qualquer nova mensagem publicada não terá efeito, pois não há funções registradas para recebê-las.

```
subscriber.setOnNewMessageListeners([]);
```

Os métodos "removeSubscriber" e "removePublisher" permitem remover o registro de nós clientes e geradores de conteúdo respectivamente junto ao nó mestre. Além disso, esses métodos também excluem as variáveis que guardam referência aos objetos "subscriber" e "publisher".

```
node.removeSubscriber(subscriber);
```

Para finalizar, a função "clear" irá excluir a variável "roscore" do ambiente de trabalho (*workspace*). Isso resultará em uma chamada ao destrutor do objeto "rosmatlab.roscore" que irá desativar o nó mestre e, depois, remover todos os objetos que estejam ligados a ele.

```
clear('roscore');
```

3.3.2 *Robotics System Toolbox*

Disponível a partir do MATLAB 2015, o *Robotics System Toolbox* oferece mais recursos do que os oferecidos pelo pacote ROS I/O. As principais novidades são:

- Integração com modelos Simulink;
- Suporte ao servidor de parâmetros ROS;
- Capacidade de importar e analisar dados gravados (*ROS Bags*);
- Capacidade de gerar código em C++ a partir de modelos Simulink que podem produzir aplicações para rodar em ambientes Linux;
- Disponibilidade de projetos exemplo.

Outra diferença importante entre o ROS I/O e o *Robotics System Toolbox* é a forma de acesso aos objetos. No primeiro, eles eram acessados a partir do pacote "rosmatlab". No segundo, passaram a ser acessados a partir do pacote "robotics.ros". Entretanto, as funcionalidades continuam as mesmas.

3.3.2.1 Criação de Nó ROS a partir de modelo Simulink

O Simulink permite programar a partir de blocos (programação visual). Ele é amplamente utilizado para simulação de sistemas de controle. Sua capacidade de integração com ROS possibilita integrar blocos pré-existentes aos novos modelos. Ao simular o modelo, o Simulink se conecta a uma rede ROS para enviar e receber mensagens de um ou mais Tópicos. Essa conexão permanece estabelecida até que a simulação seja interrompida.

As informações seguintes foram extraídas do tutorial Getting Started with ROS in Simulink (2015), disponibilizado no site do fabricante, e mostra como desenvolver um projeto com Simulink para criar um nó ROS que publica um Tópico com coordenadas (x,y) de um possível robô. Além disso, esse projeto irá se registrar para receber essas coordenadas e, então, imprimir um gráfico a partir dos dados recebidos.

Primeiro cria-se um novo projeto Simulink. Abre-se, então, a biblioteca de funções em forma de blocos (*Simulink Library Browser*) no qual existirá uma lista, à esquerda, com opções de ferramentas. A Figura 16 é uma captura de tela na qual é possível observar a opção *Robotics System Toolbox* selecionada. À direita, são listados todos os blocos disponíveis para esse conjunto. Para o ROS existem cinco blocos:

- *Blank Message*;
- *Get Parameter*;

- *Publish*;
- *Set Parameter*;
- *Subscribe*.

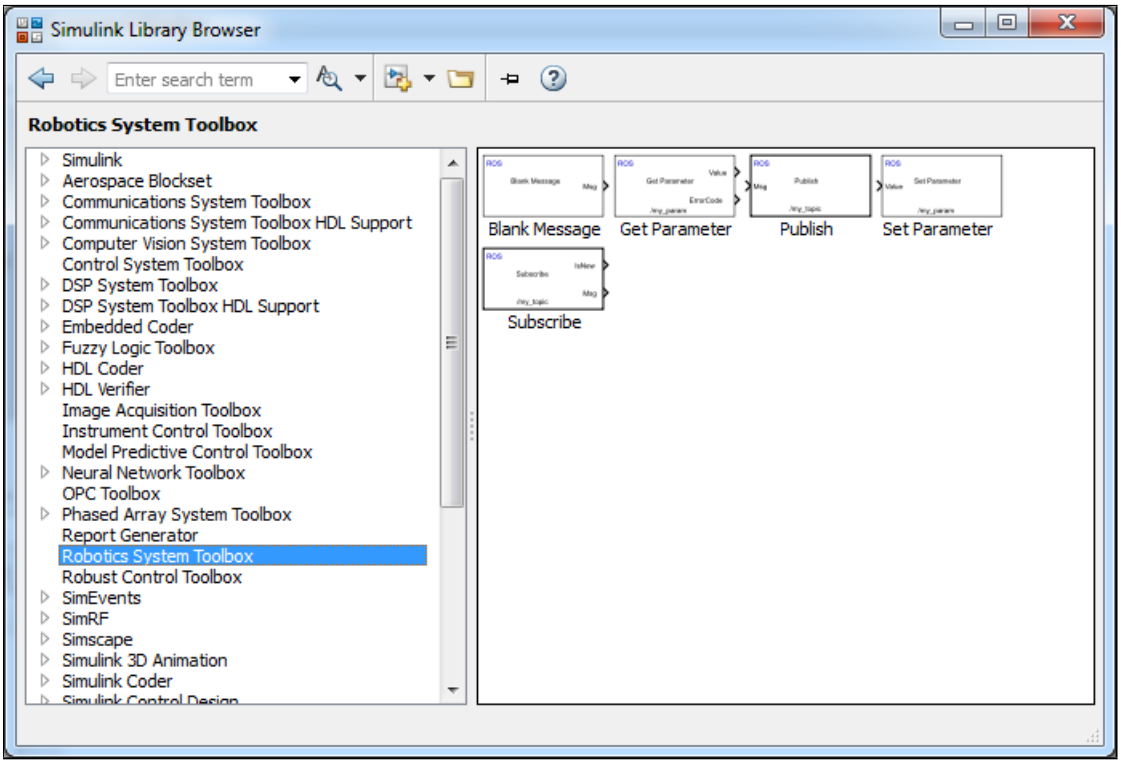

Figura 16: Captura de tela do aplicativo MATLAB com destaque para o conjunto de ferramentas *Robotics System Toolbox*.

A primeira parte desse exemplo cria o nó que publica as coordenadas do robô. Para isso, basta arrastar o bloco "Publish" da biblioteca para o modelo. Um duplo clique com o mouse sobre a caixa, recém adicionada, abre uma tela de configurações igual à mostrada na Figura 17. Na janela "Sink Block Parameters: Publish", destacam-se três configurações:

- *Topic Source*: Oferece duas opções para definição do parâmetro "Topic". A primeira é "Select from ROS network" que, quando selecionada, permite escolher o nome do tópico a partir de uma lista de tópicos na rede ROS. A segunda é "Specify your own" que permite a livre definição do nome tópico;
- *Topic*: Define o nome do tópico que será criado para publicação de dados;
- *Message Type*: Define o tipo de dados que será gerado por esse tópico. A Figura 17 destaca uma janela, intitulada "Select ROS Message Type", que permite selecionar o tipo da mensagem.

Para este exemplo, foram definidos “Topic Source” como “Specify your own”, “Topic” igual a “/location” e “Message Type” igual a “geometry_msgs/Point”.

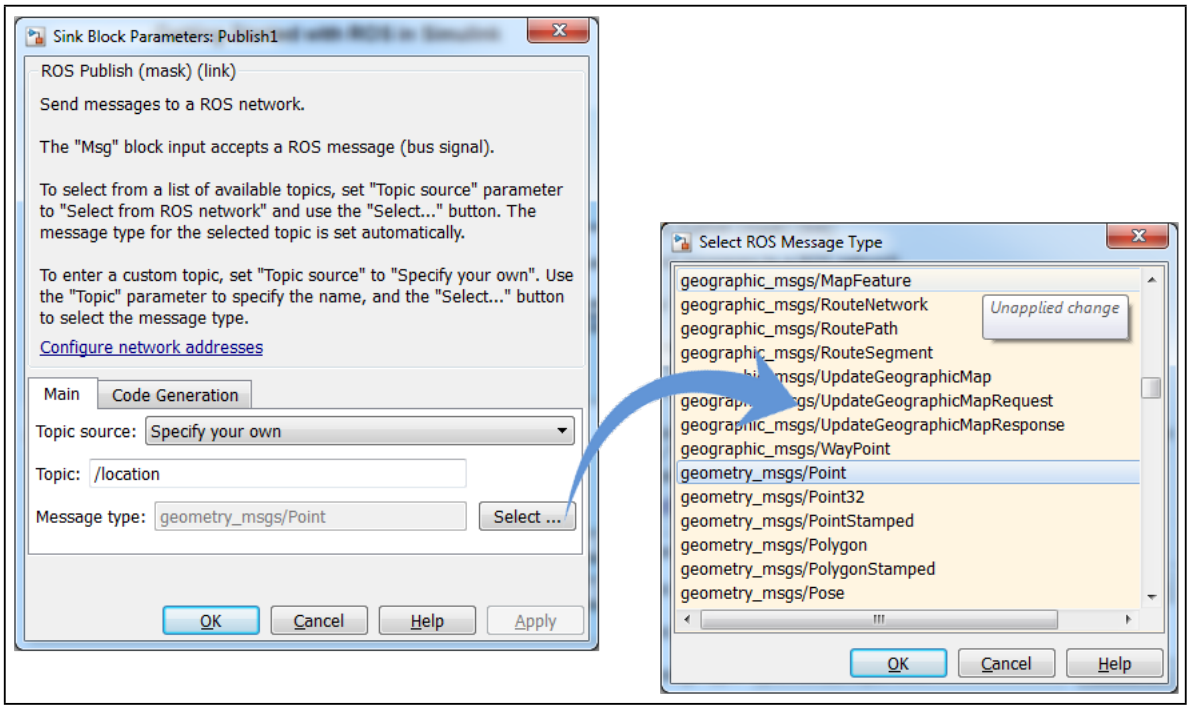

Figura 17: *Robotics System Toolbox* - Tela de configurações de parâmetros do bloco “Publish”.

Em seguida, cria-se a parte do modelo responsável por gerar dados que serão transmitidos pelo “Publish”. Para isso, adiciona-se mais um bloco ROS no modelo (“Blank Message”) e configura-se o parâmetro “Message Type” como “geometry_msgs/Point”. A Figura 18 mostra todos os blocos interconectados. É possível observar que o bloco “Blank Message” está ligado a um barramento de sinais, pois, no Simulink, mensagens ROS são tratadas como barramentos. Nota-se, ainda, que foram adicionados dois geradores de sinais ligados às portas de entrada X e Y do barramento. Eles foram defasados em 90 graus (o gerador de sinal da coordenada X foi configurado com fase -pi/2).

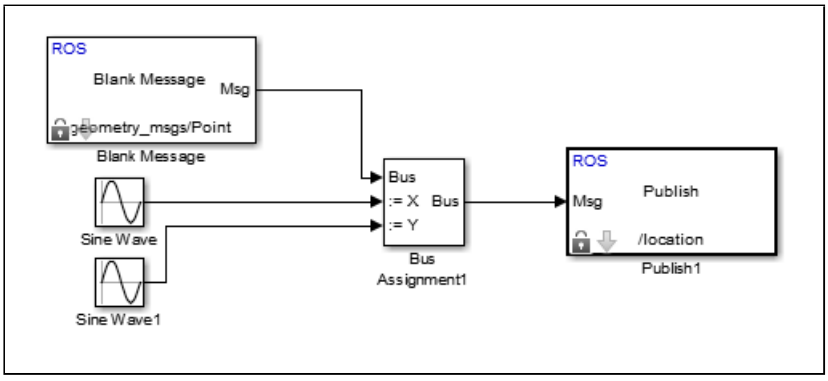

Figura 18: Modelo MATLAB Simulink para envio de coordenadas fictícias de um robô no sistema ROS.

A segunda parte do exemplo cria o conjunto de blocos que irá receber as coordenadas publicadas pelo robô e gerar um gráfico. A Figura 19 mostra o modelo da segunda parte completo (todos os blocos adicionados e interconectados). É possível observar que foi adicionado o bloco "Subscribe". Um duplo clique nesse bloco abre uma janela de configurações muito parecida com a aberta para o bloco "Publish", porém com um campo adicional que permite definir a frequência de amostragem. Deve ser especificado o nome do tópico que publica as coordenadas do robô (*/location*). Além disso, o bloco "Subscribe" possui duas saídas:

- *IsNew*: Sinaliza quando uma nova mensagem foi processada pelo bloco (muito útil para sincronizações);
- *Msg*: Disponibiliza a mensagem recebida do tópico ROS. Nesse exemplo será uma mensagem do tipo "geometry_msgs/Point".

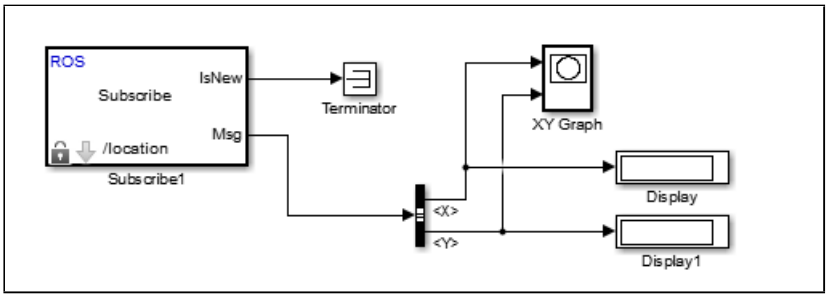

Figura 19: Modelo MATLAB Simulink para processamento de coordenadas fictícias de um robô no sistema ROS.

Para finalizar essa etapa, foi adicionado o bloco "XY Graph", para criar um gráfico, e dois blocos do tipo "Display", para mostrar na tela os valores recebidos. A Figura 20 mostra o modelo Simulink completo. Observa-se que o conjunto de blocos identificado por "Receive messages sent to /location topic" possui uma alteração: Surgiu o bloco *Enabled Subsystem* que agregou o bloco *XY Graph* e passou a usar o sinal *IsNew* do bloco *Subscribe* para sincronização.

Ao iniciar a simulação do modelo Simulink, serão iniciados o simulador ROS *Master* e o nó de publicação do tópico "/location". Esse mesmo nó passará a receber os dados com auxílio do bloco *Subscribe*. Em seguida, será exibido o gráfico de coordenadas, conforme Figura 21.

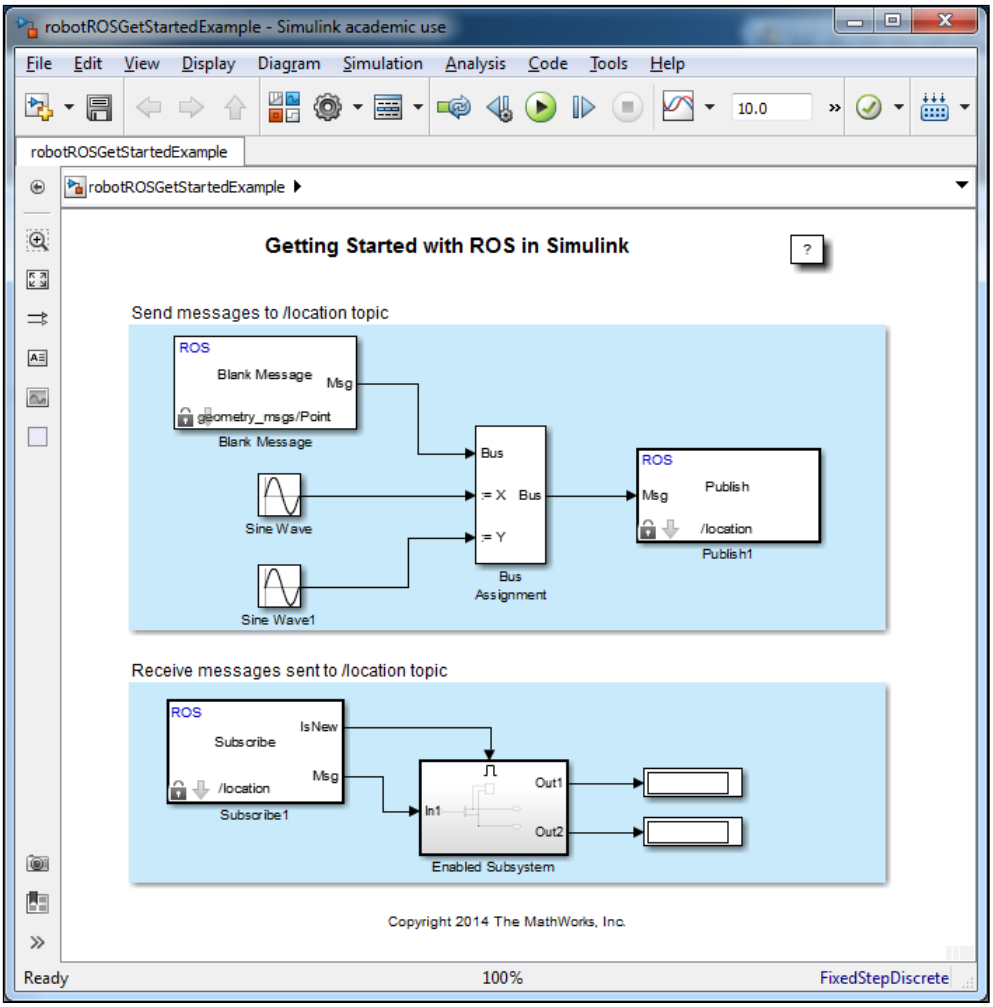

Figura 20: Projeto MATLAB Simulink que exemplifica o uso do sistema ROS.

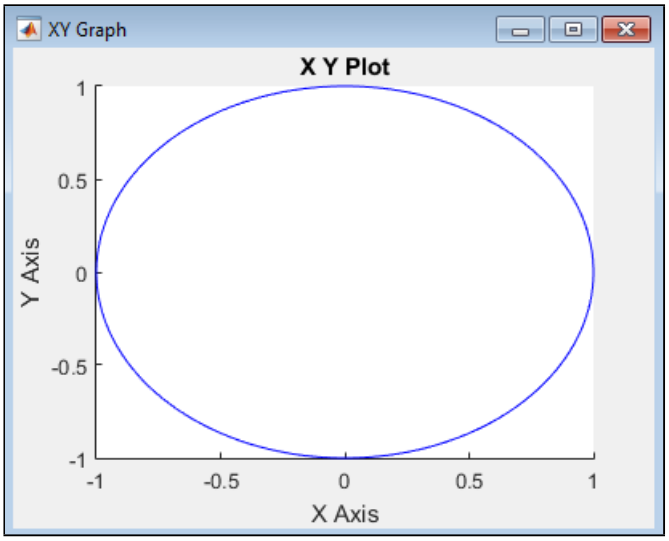

Figura 21: Gráfico das coordenadas (x,y) de um robô fictício criado para demonstrar as ferramentas Simulink para sistemas ROS.

4 Materiais e Métodos

Conforme apresentado na Seção 2, muitos projetos acadêmicos têm se beneficiado da simulação robótica em ambientes virtuais. Os artigos Bayar *et al.* (2014) e Ozcelikors *et al.* (2014), por exemplo, criaram um modelo de cadeira de rodas inteligente para Gazebo mesmo possuindo um equipamento real (ATEKS). Isso mostra que as vantagens da prototipagem de algoritmos em ambientes virtuais em relação a outras técnicas têm despertado grande interesse na comunidade científica. Deseja-se mostrar que o conjunto ROS, simulador GAZEBO e MATLAB pode ser uma boa opção para prototipagem de algoritmos de controle para cadeiras de rodas inteligentes. Diante disso, a validação deste trabalho consistiu em:

- Criar uma cadeira de rodas motorizada virtual;
- Desenvolver um algoritmo de controle anticolisão utilizando MATLAB Simulink;
- Submeter a cadeira de rodas a diferentes cenários de teste para validar a técnica de prototipagem de algoritmos utilizando ROS, GAZEBO e MATLAB.

O material de *hardware* e *software* utilizados foram:

- Dois Computadores: Um para executar a máquina virtual Linux com ROS e Simulador Gazebo e outro para executar o MATLAB;
- *Software Oracle VirtualBox* versão 5.0.10 r104061: Usado como plataforma da máquina virtual Linux;
- *Software Sketchup Make* versão 15.3.331 64-bit: Ferramenta para desenvolvimento da cadeira de rodas em 3D;
- *Software MeshLab* versão 1.3.3 32-bit: Ferramenta para determinação do centro de massa e matriz inercial a partir do modelo 3D da cadeira de rodas;
- MATLAB versão 2015b: Ferramenta para desenvolvimento do algoritmo de controle;
- ROS Indigo;
- Simulador Gazebo versão 2.2;
- Roteador: Usado para conexão entre os dois computadores e comunicação entre os nós ROS.

4.1 Instalação e Configuração do Ambiente com ROS e Simulador Gazebo

A versão mais atual do sistema ROS é a de nome *Jade Turtle*. Por estar em desenvolvimento, ela pode apresentar problemas e não é a versão indicada para usuários que necessitam de uma versão estável. O site de documentação e ajuda do ROS possui uma lista com todas as versões disponíveis (ROS DISTRIBUTIONS, 2015) e recomenda o uso da versão *Indigo Igloo* que conta com suporte de longo prazo (até Abril de 2019). Como não foi identificado uma necessidade técnica que justificasse adotar a versão *Jade*, optou-se por utilizar a versão *Indigo* mais estável e, dessa forma, diminuir o risco de ter os resultados deste trabalho afetados por problemas da versão em desenvolvimento.

Antes de instalar o ROS *Indigo*, porém, foi necessário preparar um ambiente para sua instalação. Decidiu-se criar uma máquina virtual para facilitar sua manutenção e escalabilidade. Além disso, essa estratégia permite facilmente criar cópias de segurança e compartilhar esses recursos com outros interessados e pesquisadores. A virtualização computacional foi feita com o *VirtualBox*, pois ele é um *software* de código fonte aberto que está disponível sob a licença *General Public License (GPL)* versão 2 que permite usar, estudar, compartilhar ou modificar o *software* de maneira livre por indivíduos, organizações e empresas.

Criou-se uma máquina virtual para sistemas Linux com as seguintes características:

- Sistema Linux de 64 bits;
- Dois Processadores (limitados a 90% de utilização);
- 2048 MB de memória RAM;
- 20 GB de disco;
- 128 MB de memória de vídeo;
- Uma interface de rede.

A Figura 22 mostra uma captura de tela das configurações adotadas para a máquina virtual onde o sistema ROS foi instalado.

Com a máquina virtual criada, instalou-se o sistema operacional Linux Ubuntu versão 14.04.3, também conhecido pelo nome *Ubuntu Trusty Tahr*. Alternativamente poderia ter sido instalada a versão Ubuntu 13.10, pois o ROS Indigo é compatível com ambas versões. Entretanto, a versão 14.04.3 foi escolhida por ser recomendada e ainda estar dentro do período de suporte que se encerra em Agosto de 2016.

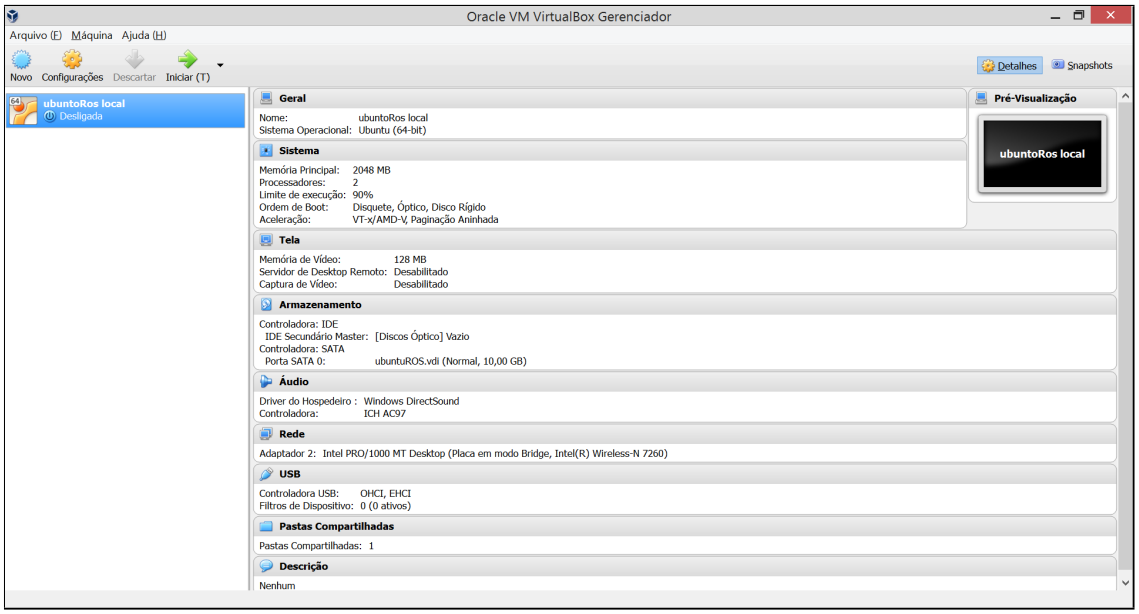

Figura 22: Captura de tela que mostra as configurações da máquina virtual utilizada para instalação dos sistemas Linux Ubuntu e ROS Indigo.

Durante o processo de instalação do Ubuntu, foram escolhidas as opções padrão de instalação.

Após a instalação do sistema operacional Linux, seguiu-se o guia de instalação do ROS (Ubuntu Install of ROS Indigo (2015)). Existem 4 opções de instalação:

- Estação de Trabalho Completa (*Desktop-Full*): É a opção de instalação recomendada. Ela inclui ROS, rqt (um *framework* para desenvolvimento de aplicações com interface gráfica baseada em QT), rviz (uma ferramenta ROS para visualização de modelos 3D), bibliotecas de robótica, Simulador 2D/3D, bibliotecas que auxiliam na implementação de navegação e sensoriamento em 2D/3D;
- Estação de Trabalho (*Desktop*): Inclui ROS, rqt, rviz, e bibliotecas de robótica;
- ROS Básico (*ROS-Base*): Inclui apenas bibliotecas essenciais do sistema ROS necessárias para compilação e comunicação. Não inclui ferramentas gráficas e simuladores;
- Pacotes Individuais (*Individual Package*): É possível realizar instalações personalizadas selecionando apenas os pacotes de interesse.

Optou-se por instalar a versão completa (*Desktop-full*) que já inclui o simulador de ambientes em 3D, Gazebo, na versão 2.

Para funcionar corretamente, o simulador Gazebo requer que a máquina hospedeira suporte a biblioteca de aceleração gráfica 2D e 3D *Open Graphics Library* (OpenGL).

Para isso é preciso possuir uma unidade de processamento gráfico (*Graphics Processing Unit - GPU*) com suporte ao OpenGl e, também, que seu *software* básico (*device driver*) esteja corretamente instalado. Em uma máquina virtual, é preciso que o *software* de virtualização suporte tal funcionalidade e que a máquina hospedeira também tenha o *hardware* GPU funcionando. O VirtualBox permite habilitar a aceleração em 3D, entretanto, não foi possível fazê-lo funcionar. Para solucionar o problema, optou-se por habilitar a API OpenGL simulada por *software*. Para isso, foi necessário executar o comando "export LIBGL_ALWAYS_SOFTWARE = 1".

4.2 Instalação do MATLAB

A versão do *software* MATLAB escolhida para este trabalho foi a 2015b, pois, conforme apresentado na Seção 3, as ferramentas para trabalho com robótica (*Robotics Systems Toolbox*) já possuem suporte ao sistema ROS. Assim como feito para a instalação do ROS e simulador GAZEBO, poderia ter sido criado uma máquina virtual para a instalação do MATLAB. Entretanto optou-se por instala-lo em um computador dedicado pelos seguintes motivos:

- Atender os requisitos mínimos recomendados: Segundo o fabricante, o MATLAB necessita de, pelo menos, 2 GB de memória RAM disponíveis. Como já foram cedidos 2 GB de memória para a máquina virtual Linux, não seria possível reservar outros 2 GB para a máquina com MATLAB, pois o computador hospedeiro possui apenas 4 GB;
- Demonstrar o controle da cadeira via rede: Mesmo sendo possível demonstrar que duas máquinas virtuais utilizam a interface de rede para se comunicar, os testes sendo realizados em computadores diferentes destacam essa característica.

Diante disso, o MATLAB foi instalado em um computador com processador AMD Turion X2 1600MHz, 2 GB de memória, disco rígido com 120 GB de capacidade de armazenamento e sistema operacional Windows 7 64-bit com *Service Pack 1*.

4.3 Desenvolvimento da Cadeira de Rodas Virtual

A primeira etapa do processo de desenvolvimento da cadeira de rodas virtual consistiu na criação de uma área de trabalho ROS (*ROS workspace*). Para isso, utilizou-se a ferramenta de construção oficial do ROS chamada *Catkin*. Ela combina *scripts* criados na linguagem de programação Python com macros do sistema de construção automatizada CMake. Seguiu-se as instruções descritas no guia Creating a Workspace for Catkin (2015).

A sequência de comandos, listados a seguir, cria uma estrutura de diretórios e arquivos básicos necessários a uma área de trabalho ROS.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws/
$ catkin_make
```

Entretanto, para que o ROS possa encontrar o conteúdo criado nessa área de trabalho, é necessário configurar a variável de ambiente "ROS_PACKAGE_PATH" para que ela inclua o caminho desse diretório. Isso pode ser feito com auxílio do seguinte comando:

```
$ source ~/catkin_ws/devel/setup.bash
```

Em seguida, foi necessário criar um pacote ROS (*ROS Package*). Para isso, seguiu-se as instruções do tutorial Creating a ROS Package (2015) que explica toda a estrutura de diretórios e arquivos de um ou mais pacotes ROS dentro de uma área de trabalho. O comando, mostrado a seguir, executa o *script* "catkin_create_pkg" para a criação do pacote chamado "cadeira_gazebo". O primeiro parâmetro é o nome do pacote e os demais são suas dependências.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg cadeira_gazebo std_msgs rospy roscpp
```

O pacote "cadeira_gazebo" depende das bibliotecas "rospy", "roscpp" e das mensagens padrão "std_msgs". Para concluir a criação do pacote foi utilizada a ferramenta "catkin_make", conforme mostrado a seguir:

```
$ cd ~/catkin_ws
$ catkin_make
```

Editou-se o arquivo "~/catkin_ws/src/cadeira_gazebo/Package.xml" para personalizar o pacote. O conteúdo desse arquivo é mostrado a seguir.

```
<?xml version="1.0"?>
<package>
  <name>cadeira_gazebo</name>
  <version>1.0.0</version>
  <description>Trabalho de mestrado - Prototipagem de Algoritmos para Controle
      de Cadeiras de Rodas Motorizadas Utilizando Arquitetura Multi-Agente,
      Simulador GAZEBO e MATLAB</description>
  <maintainer email="luciano.laranjeira@gmail.com">Luciano Laranjeira</
      maintainer>
  <license>BSD</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>
```

```
  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

4.3.1 Desenvolvimento de um Robô Intermediário

A definição de robôs para o simulador GAZEBO é feita a partir da criação de arquivos no formato SDF, conforme apresentado na Seção 3.2.1.2. Para se conseguir criar um robô com sucesso, foi necessário, inicialmente, compreender a estrutura desse arquivo (sintaxe) e seus efeitos no GAZEBO. Foi preciso definir atributos como atrito, peso, inércia, colisão, posicionamento e relação entre as partes móveis e fixas do robô. Além disso, a inclusão de sensores e motores exigiu conhecimento adicional nas extensões de *software* (*plugin*).

Para simplificar o processo de aprendizado, decidiu-se seguir o tutorial Make a Mobile Robot (2015) e criar um veículo mais simples construído a partir de formas básicas como retângulo (chassi), esfera (roda dianteira) e cilindro (rodas traseiras). Outra vantagem dessa abordagem foi o desempenho. Como utilizou-se máquina virtual sem aceleração de *hardware* para processamento gráfico, a adoção desse robô permitiu observar os efeitos das alterações no arquivo SDF e, também, realizar os primeiros testes exigindo menos processamento.

Além da criação do arquivo SDF, foi necessário instalar um pacote de extensão do ROS chamado "hector_gazebo_plugins". Disponível sob licença de uso BSD, ele oferece controle diferencial para robôs de seis rodas, sensor de movimento (*Inertial Measurement Unit - IMU*), sensor de campo magnético terrestre (*Earth Magnetometer*), sensor GPS e sensor sonar, sendo este último o sensor de interesse para uso no robô.

O robô desenvolvido possui as seguintes características:

- Um Chassi Retangular: Definido por um nó *link* no documento SDF, o chassi é um retângulo com 5 Kg de peso, 0,4 m de comprimento, 0,2 m de largura e 0,1 m de altura;
- Duas Rodas Traseiras: Construídas a partir de um cilindro de raio 0,1 m e altura 0,05 m, elas foram definidas por um nó *link*. Foi necessário rotacioná-las em 90 graus nos eixos Y e Z para que se posicionassem como rodas de um veículo. Além disso, houve deslocamento nos três eixos (x,y,z) para posicioná-las na parte traseira.

As rodas tiveram, também, um nó *joint* do tipo *revolute* para associá-las ao chassi e permitir sua rotação no eixo Y;

- Uma Roda Frontal: Idealizada de forma bastante simples, essa roda foi definida por um nó *visual*, filho do nó *link* do chassi. A forma geométrica utilizada foi uma esfera de raio 0,05 m com deslocamento nos eixos X e Z para que ela se posicionasse na parte frontal do veículo;

- Três Sensores Sonar: Definidos por um nó *sensor*, filho do nó *link* do chassi, eles carregam a biblioteca "libhector_gazebo_ros_sonar.so". Seguiu-se o tutorial Using Gazebo Plugins with ROS (2015) para aprender a usar extensões ROS GAZEBO. Os sensores foram configurados para ter um alcance de 2 m e um ângulo de abertura de 12 graus na vertical e horizontal. Todos foram posicionados na parte frontal do veículo, porém apontando para direções diferentes (frente, esquerda e direita). Outra definição importante foi o nome dos tópicos ROS para os sensores. Conforme apresentado na Seção 3.1.2, a troca de informações na arquitetura ROS é feita através de tópicos, então os dados dos sensores são transmitidos dessa maneira. Cada sensor cria um tópico diferente com os nomes "front_sonar_data", "right_sonar_data" e "left_sonar_data" e atualiza os dados a uma frequência de 10 Hz;

- Um Controle Diferencial: As rodas traseiras são tracionadas por um controle diferencial carregado como uma extensão de *software*. O nó *plugin* carrega o arquivo "libgazebo_ros_diff_drive.so" e define seus atributos. Os nós "leftJoint" e "rightJoint" relacionam esse controle às rodas. Há, também, definições sobre o diâmetro das rodas e a distância entre elas. Dois tópicos foram criados. O primeiro é definido pelo nó "commandTopic" que permite ao controle receber instruções. O segundo, definido pelo nó "odometryTopic", permite obter dados do odômetro.

A Figura 23 mostra uma captura de tela onde o robô intermediário foi carregado em um mundo vazio (*empty world*) do simulador GAZEBO. O alcance dos sensores sonar está representado pelas faixas em azul.

4.3.2 Modelo em 3D da Cadeira de Rodas

Assim como feito para o robô intermediário, a cadeira de rodas poderia ser criada a partir da composição de formas básicas diretamente inseridas no arquivo SDF. Entretanto, esse método é bastante trabalhoso quando se deseja criar objetos mais complexos. A definição do arquivo SDF é feita de forma manual com auxílio de, apenas, um editor de textos.

O GAZEBO versão 6.0 já inclui recursos que permitem editar modelos visualmente através da interface gráfica cliente (descrita na Seção 3.2.1.5). Entretanto, esse trabalho

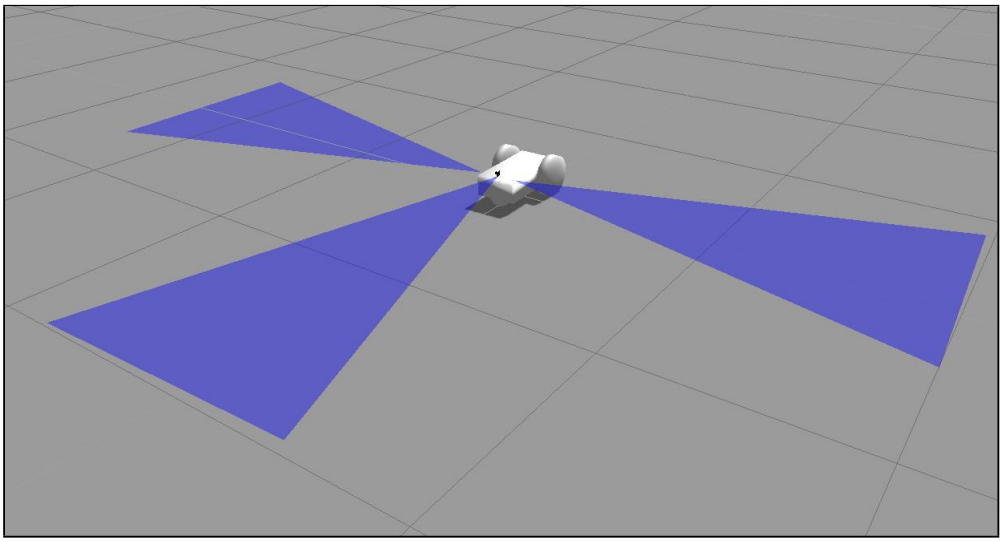

Figura 23: Captura de tela que mostra o robô intermediário construído a partir de formas básicas como retângulo (chassi), esfera (roda dianteira) e cilindro (rodas traseiras).

utiliza a versão 2. Explorou-se, então, a capacidade que o GAZEBO tem de utilizar modelos em 3D que estejam no formato de arquivo COLLADA (*Collaborative Design Activity*). Identificados pela extensão de arquivo ".dae", acrônimo de *digital asset exchange*, os arquivos em formato COLLADA são suportados por muitos aplicativos CAD de modelagem em 3D como, por exemplo, Blender, FreeCad, MeshLab, 3ds Max e SketchUp Make. Para criar o modelo da cadeira, optou-se por utilizar o SketchUp versão 15.

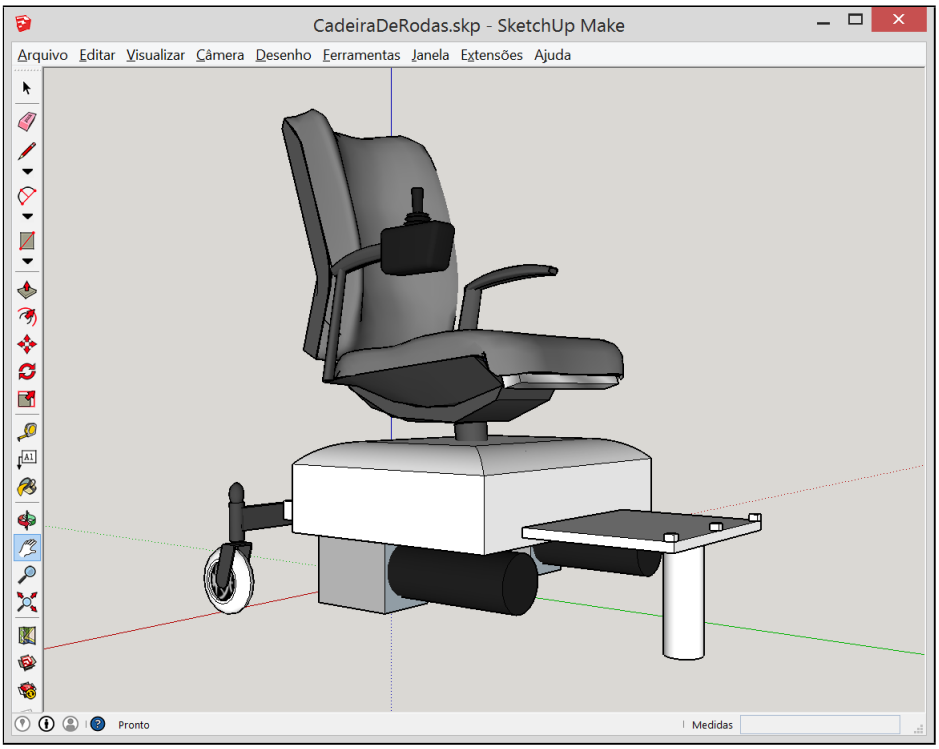

Figura 24: Modelo 3D da cadeira de rodas criado com o aplicativo SketchUp Make.

A Figura 24 mostra a cadeira de rodas em edição no aplicativo SketchUp Make. É possível notar que cadeira possui apenas rodas de apoio na parte traseira. As rodas laterais e a dianteira foram omitidas intencionalmente. Elas foram acrescentadas diretamente no arquivo SDF como formas geométricas básicas. Dessa forma, foi possível definir suas propriedades dinâmicas e também relacioná-las ao *plugin* de controle diferencial.

4.3.3 Cálculo da Matriz Inercial para a Cadeira de Rodas

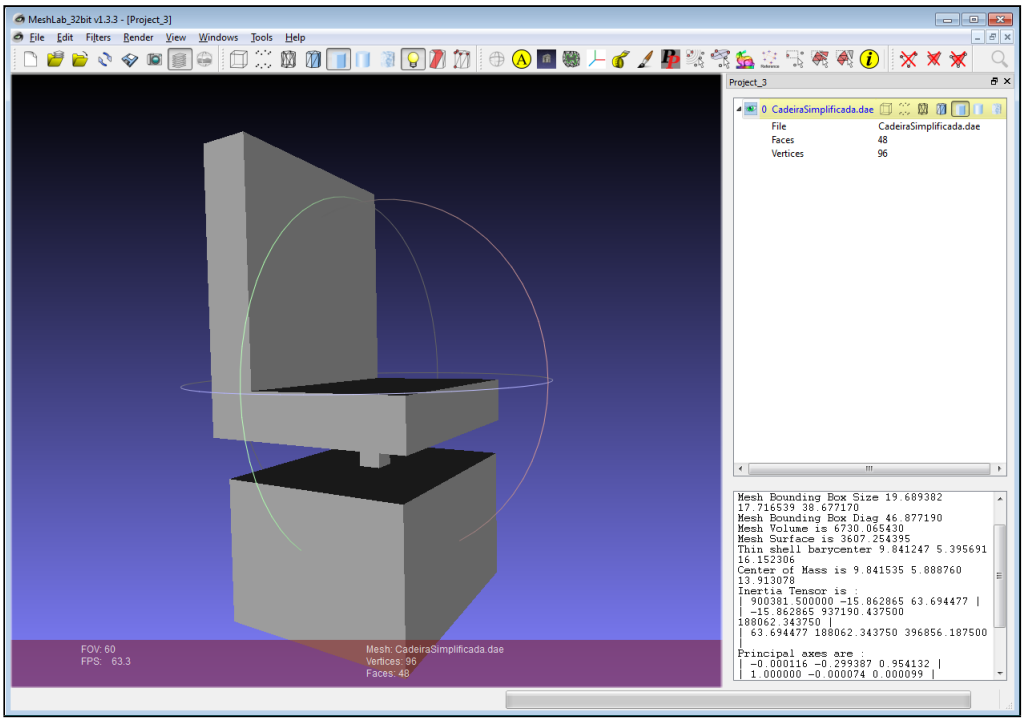

Figura 25: Obtenção da matriz inercial e do centro de massa da cadeira de rodas a partir do aplicativo MeshLab.

Conforme apresentado na Seção 3.2, o simulador GAZEBO permite realizar testes em condições realistas. Mas, para isso, é necessário fornecer informações físicas completas a respeito do modelo. Buscou-se, então, acrescentar dados sobre a matriz inercial, o centro de massa e o peso de todos os *links* (partes do corpo do modelo). Para isso, seguiu-se o tutorial Find Out Inertial Parameters (2015) que apresenta uma técnica para obtenção desses dados a partir de um modelo em 3D. O procedimento descrito utiliza-se de um *software* de uso gratuito chamado MeshLab.

A Figura 25 mostra uma captura de tela do *software* MeshLab onde é possível ver uma representação simplificada da cadeira de rodas. Segundo o tutorial, o MeshLab tem limitações para calcular os parâmetros inerciais de figuras complexas ou com formas côncavas. O texto sugere, então, simplificar o modelo. Além disso, o MeshLab não permite indicar a massa do corpo e há limitações quanto a definição de unidade de medida. Diante disso, o tutorial sugere fazer ajustes, dimensionando o modelo, até que a matriz gerada

tenha resolução de 6 casas decimais. No canto inferior direito da Figura 25 é possível observar os valores calculados pela ferramenta. A matriz inercial gerada foi:

$$\begin{vmatrix} 900381.500000 & -15.862865 & 63.694477 \\ -15.862865 & 937190.437500 & 188062.343750 \\ 63.694477 & 188062.343750 & 396856.187500 \end{vmatrix} = \begin{vmatrix} ixx & ixy & ixz \\ ixy & iyy & iyz \\ ixz & iyz & izz \end{vmatrix}$$

As coordenadas x, y e z para o centro de massa foram (9.841535, 5.888760, 13.913078) e o volume calculado foi 6730.065430 (sem unidade de medida). Diante dessa limitação em identificar a unidade de medida utilizada, adotou-se o seguinte critério: Dividiu-se o volume por 100 e o resultado foi considerado o peso do veículo (67Kg). Utilizou-se o mesmo divisor para o centro de massa.

Para a obtenção dos valores da matriz inercial, porém, foi necessário utilizar um fator de divisão maior (100.000). Esse ajuste foi realizado experimentalmente. O procedimento adotado foi multiplicar o divisor por 10 até que o resultado, observado visualmente através da interface gráfica cliente do GAZEBO, formasse um retângulo na região que compreende o corpo da cadeira de rodas.

A Figura 26 destaca o centro de massa e inércia. Cada *link* possui uma caixa rosa com linhas verdes. O centro da caixa está alinhado com o centro de massa. Seu tamanho e orientação correspondem, respectivamente, à massa e ao comportamento inercial. A partir dessas informações, criou-se a seguinte configuração no arquivo SDF para o chassi da cadeira de rodas:

```
<inertial>
        <mass>67</mass>
        <pose>0.098415 0.409997 0.139130 0 0 0</pose>
        <inertia>
                <ixx>9.00381</ixx>
                <ixy>-0.000158</ixy>
                <ixz>0.000636</ixz>
                <iyy>9.371904</iyy>
                <iyz>1.880623</iyz>
                <izz>3.968561</izz>
        </inertia>
</inertial>
```

4.3.4 Definição do Arquivo SDF para a Cadeira de Rodas

O arquivo modelo, no formato SDF, criado para o robô intermediário serviu como base para o desenvolvimento da cadeira de rodas virtual, pois mudou-se, apenas, o chassi. A Figura 27 mostra o resultado da renderização do arquivo modelo da cadeira de rodas no simulador GAZEBO. É possível observar que as rodas e os sensores da cadeira são iguais aos utilizados no robô intermediário, mas o chassi foi substituído pelo modelo em

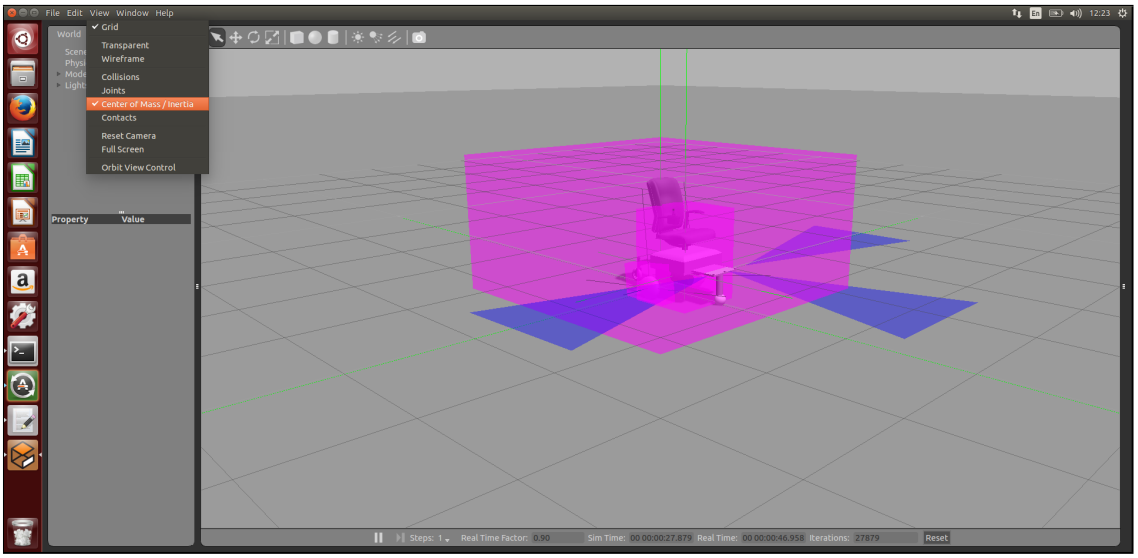

Figura 26: Representação visual do centro de massa e inércia da cadeira de rodas.

3D criado no Skecthup. Para carregá-lo, referenciou-se o arquivo ".dae" como forma geométrica do chassi. O seguinte trecho de código xml mostra como isso foi feito:

```
<visual name='chassi_visual'>
        <geometry>
                <mesh>
                        <uri>model://cadeira/meshes/CadeiraDeRodas.dae</uri>
                </mesh>
        </geometry>
</visual>
```

Adicionalmente ao arquivo SDF criado para definição da cadeira de rodas, criou-se, também, três outros:

- Arquivo Descritivo de Ambientes: Criou-se um arquivo *.wolrd* que define um mundo virtual vazio;
- Arquivo de Partida (*launch*): Arquivos com extensão *.launch* passam parâmetros para a ferramenta *roslaunch* que carrega arquivos *.world* e os popula com robôs. O arquivo criado adiciona a cadeira de rodas no centro do mundo vazio;
- *Bash Script* de Inicialização: Arquivo criado para facilitar a configuração das variáveis de ambiente e o carregamento das aplicações (ROS e Simulador GAZEBO).

4.4 Desenvolvimento do Sistema de Controle Anticolisão Utilizando MATLAB Simulink

Para validar este trabalho, foi desenvolvido um sistema de controle anticolisão utilizando MATLAB Simulink. Para isso, tomou-se como base um projeto exemplo dis-

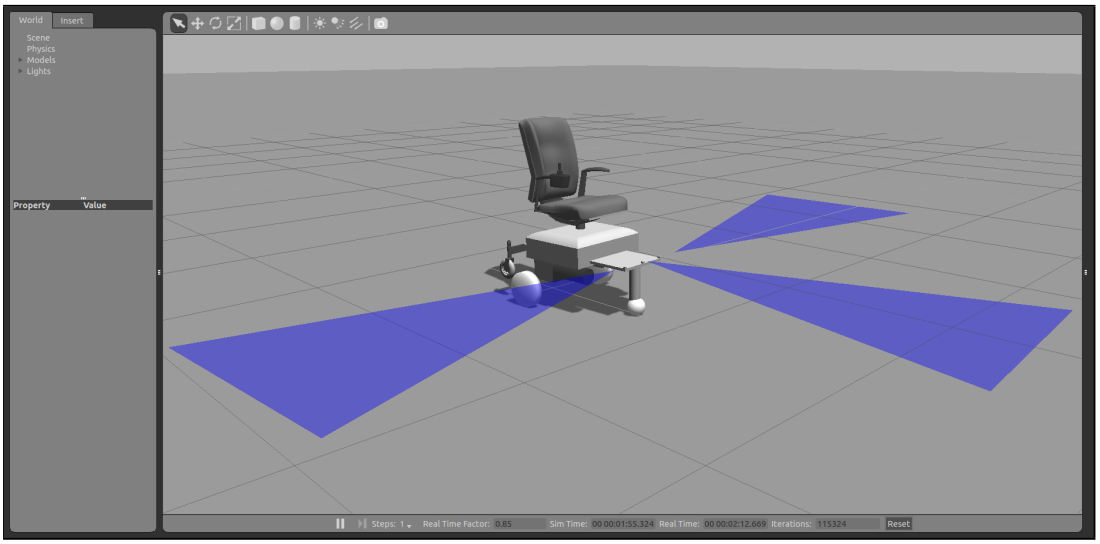

Figura 27: Cadeira de rodas virtual carregada no *software* cliente do simulador GAZEBO.

tribuído com o MATLAB 2015b para controle de robôs em ambientes ROS. Esse sistema monitora as coordenadas de um robô e compara com valores de entrada. Caso sejam diferentes, o robô recebe mensagens para se deslocar até atingir o ponto desejado no mapa.

Esse sistema de controle, entretanto, não é capaz de impedir colisões durante o trajeto. Caso haja um obstáculo à frente, o robô irá colidir e, talvez, ficar preso. Os motores continuarão acionados indefinidamente. Criou-se, então, um subsistema anticolisão especializado que monitora os sensores sonar da cadeira de rodas e, baseado nas informações dos sensores, atua de forma a desviar a cadeira dos objetos em seu caminho. Pode, inclusive, parar o veículo se não houver direção livre.

A Figura 28 apresenta o projeto de controle original cujos principais blocos da programação são:

- *Subscribe*: Bloco especializado para programação de sistemas ROS (conforme descrito na Seção 3.3.2.1). Ele recebe mensagens do tipo *nav_msgs/Odometry* do tópico */odom* (mesmo nome utilizado pela cadeira de rodas virtual para publicar seus dados odométricos). As mensagens recebidas são enviadas para um barramento de dados onde ocorre uma seleção: O bloco *Proportional Controller* recebe a posição atual do veículo em forma de coordenadas x e y (a coordenada z não é usada). O bloco *Conversion* recebe a orientação espacial na forma de quaterniões;

- *Desired Position*: Bloco para definição de um valor constante. Usado para indicar a posição (x,y) da cadeira de rodas no plano;

- *Conversion*: Bloco que realiza a conversão da informação espacial no formato quaterniões para ângulos de Euler;

- *Proportional Controller*: A Figura 30 apresenta o conteúdo desse bloco em detalhes. Ele implementa um controle proporcional. Baseado na diferença entre a posição atual e a desejada, ele define a orientação e velocidade do veículo;

- *Command Velocity Publisher*: Recebe as velocidades angular e linear que devem ser enviadas à cadeira de rodas. A Figura 31 apresenta os detalhes desse subsistema. O bloco *Blank Message* cria mensagens do tipo *geometry_msgs/Twist* que são relacionadas aos valores de entrada. O bloco *Publish* envia a mensagem para o tópico */cmd_vel* na rede ROS.

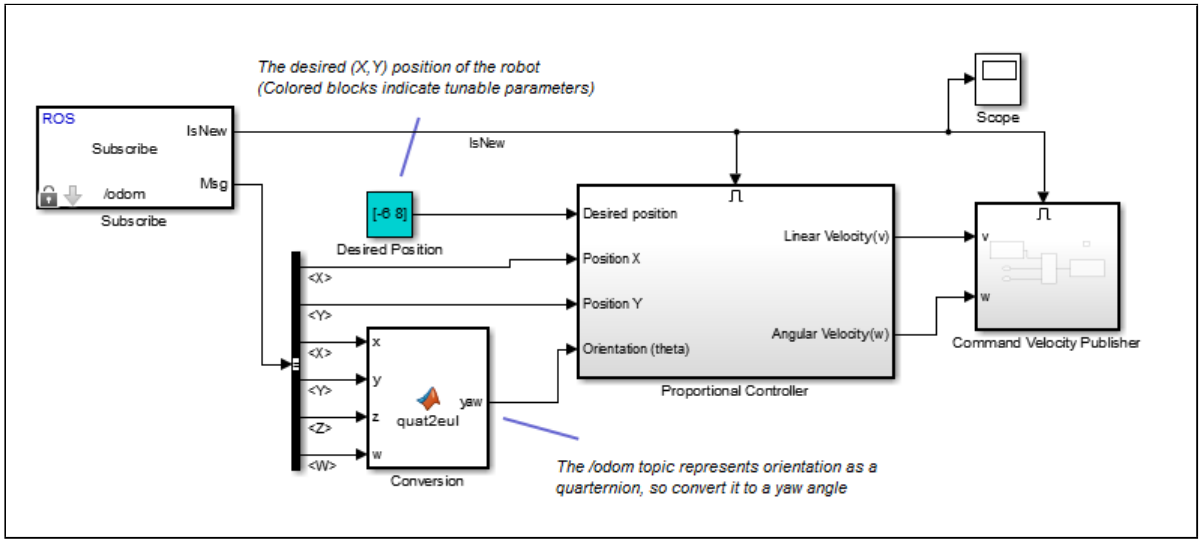

Figura 28: Modelo Simulink distribuído como exemplo para posicionamento (x,y) de robôs em sistemas ROS.

A Figura 29 mostra o modelo Simulink de controle alterado. Foi acrescentado um bloco a mais denominado *Collision Avoidance* que contém a programação para considerar os dados dos sensores sonar e atuar nos comandos de saída (velocidades angular e linear).

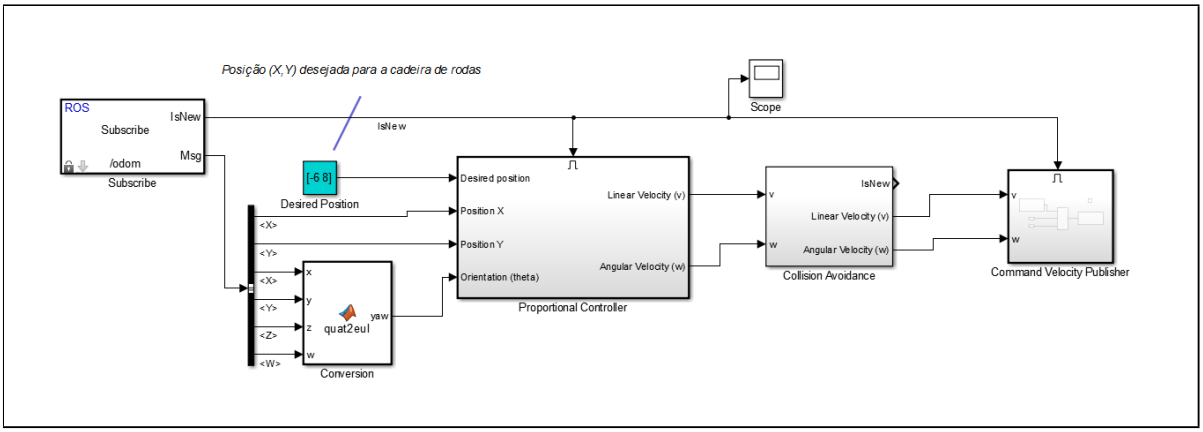

Figura 29: Modelo Simulink para controle de posicionamento da cadeira de rodas motorizada com sistema anticolisão.

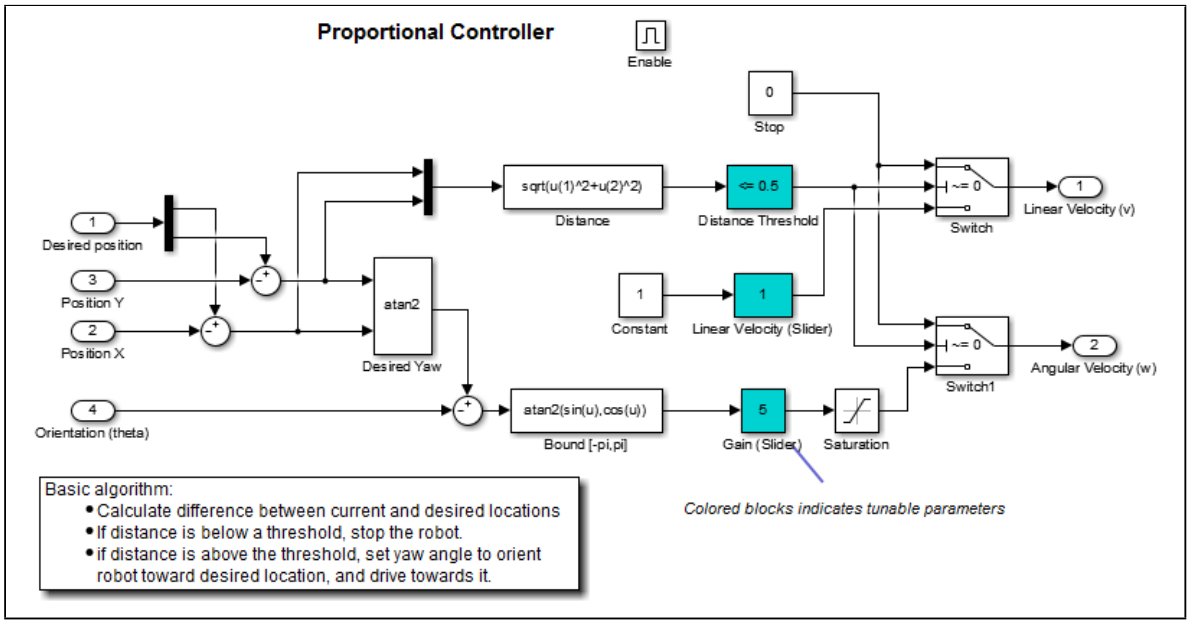

Figura 30: Controle Proporcional que, baseado diferença entre as posições atual e desejada (x,y), define a orientação angular de Euler e a velocidade linear.

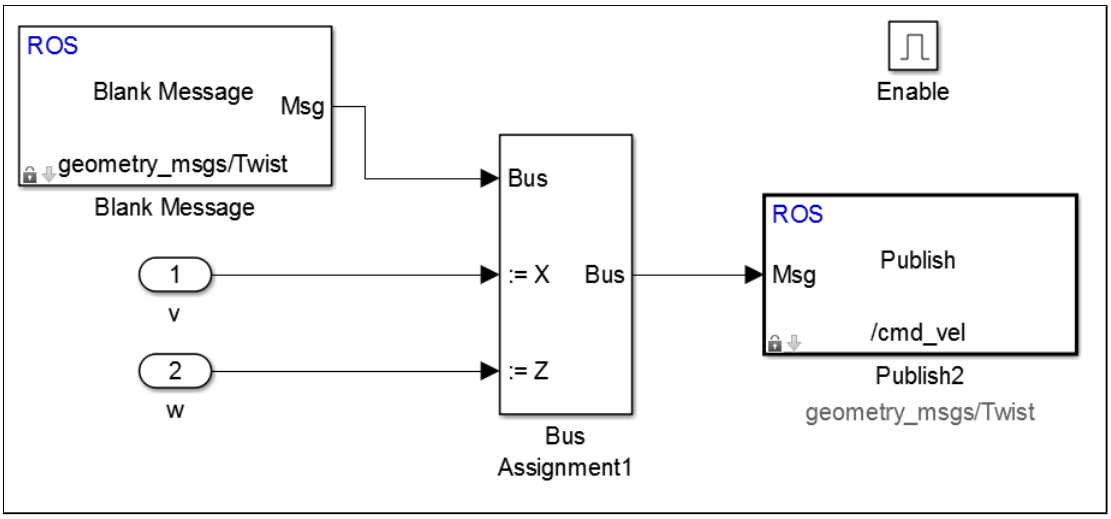

Figura 31: Subsistema *Command Velocity Publisher* criado para publicar mensagens do tipo *geometry_msgs/Twist* para o tópico */cmd_vel* em uma rede ROS.

A Figura 32 mostra a implementação do subsistema *Collision Avoidance*. Ele implementa três blocos Simulink ROS para receber dados do tipo *sensor_msgs/Range* gerados pelos sensores sonar:

- *Subscribe Front Sonar*: Registrado para receber dados do sensor frontal a partir do tópico */front_sonar_data*;
- *Subscribe Right Sonar*: Registrado para receber dados do sensor lateral direito a partir do tópico */right_sonar_data*;

- *Subscribe Left Sonar*: Registrado para receber dados do sensor lateral esquerdo a partir do tópico */left_sonar_data*.

O algoritmo anticolisão implementa a seguinte lógica: Monitora o sensor frontal. Caso seja detectado um obstáculo a uma distância menor ou igual a um metro e meio, o sistema força um desvio à esquerda. Porém, os sensores laterais têm prioridade na definição da direção. Se qualquer um deles identificar um objeto a menos de um metro de distância, então vale a lógica implementada para os sensores laterais. Se o sonar direito reportar distância inferior a um metro, então instrui virar à esquerda. Se o sonar esquerdo reportar distância inferior a um metro, então instrui virar à direita. Se os três sensores identificarem objetos muito próximos (dentro do limite de segurança), então o veículo para. Quando não há objetos à frente da cadeira de rodas, a atuação dos sensores laterais é somada à atuação do controle proporcional. Isso impede que a cadeira colida com algum objeto nas laterais, pois se o controle proporcional instruir mudança de direção sobre o objeto, ela será anulada pela instrução resultante da leitura dos sensores. Além disso, esse algoritmo inverte a direção da cadeira caso o sensor frontal esteja indicando objeto a uma distância inferior a 0,2 m.

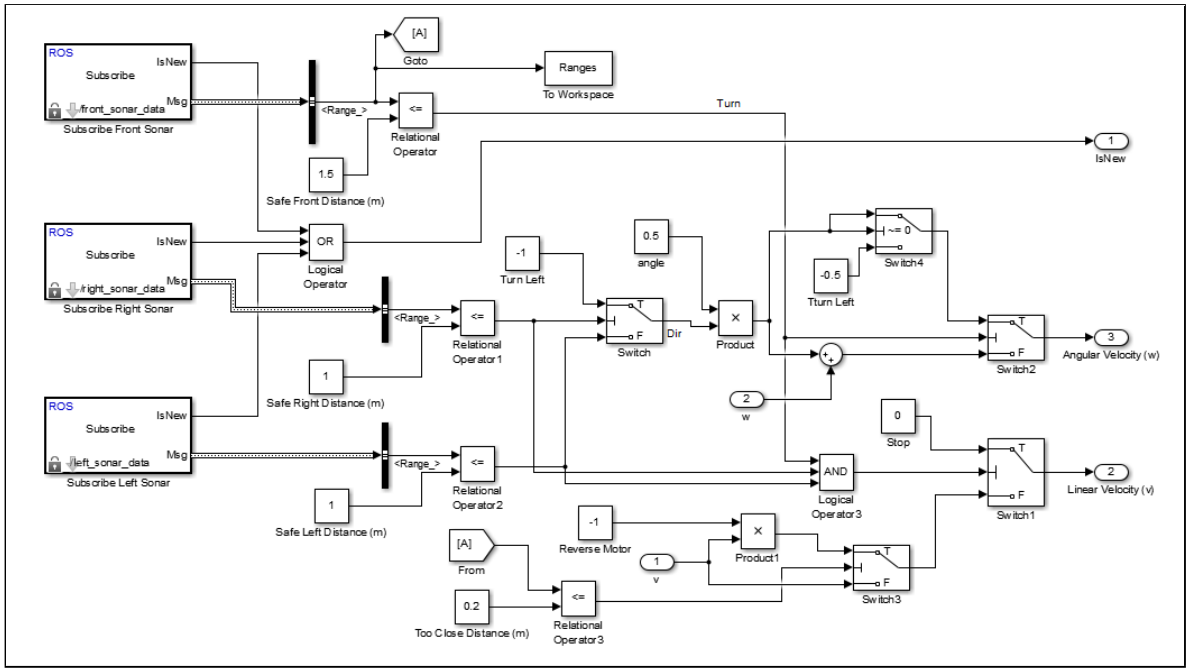

Figura 32: Algoritmo anticolisão programado com MATLAB Simulink.

5 Resultados

A cadeira de rodas virtual e o algoritmo de controle anticolisão foram submetidos a diferentes cenários de teste com o intuito de validar a metodologia apresentada. Conforme apresentado na Seção 4, foram criados ambientes diferentes para executar o algoritmo de controle e o conjunto ROS e simulador Gazebo. A Figura 33 ilustra como estão organizados os principais elementos envolvidos. Foram destacados os tópicos ROS utilizados para envio de comandos e leitura de sensores.

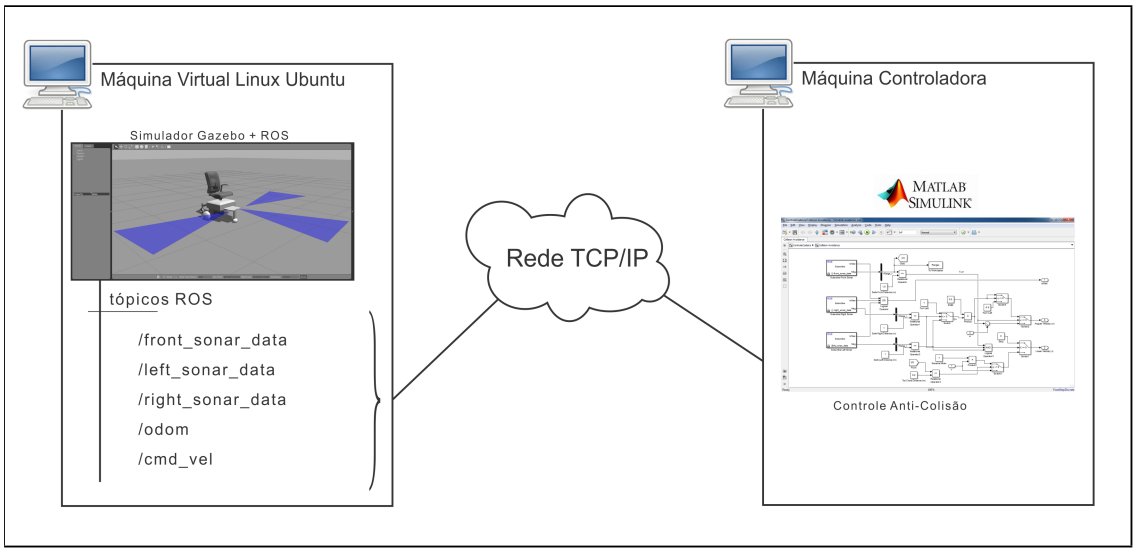

Figura 33: Visão geral do ambiente de testes.

Para cada experimento, foram coletadas imagens sequenciais que mostram o percurso do veículo. Também são apresentados gráficos que ajudam a compreender o comportamento da cadeira e a atuação dos controles. A legenda apresentada nos gráficos possui o seguinte significado:

- Sensor Frontal: Resultado da leitura do sensor sonar frontal. Quando não há obstáculo, esse sensor retorna valor constante igual a 2 que é o alcance máximo do sensor (2 m). Esse número diminui de acordo com a proximidade de obstáculos (podendo chegar a 0);
- Sensor Direito: Funcionalidade e alcance iguais as do sensor frontal, porém com rotação de 90 graus (medindo distâncias à direita da cadeira de rodas);
- Sensor Esquerdo: Funcionalidade e alcance iguais as do sensor frontal, porém com rotação de -90 graus (medindo distâncias à esquerda da cadeira de rodas);

- Velocidade Angular: Componente de rotação, em radianos, que compõe o comando de deslocamento da cadeira. Está limitado em -0,5 e 0,5 radianos nos sistemas de controle criados no MATLAB Simulink;
- Diferença X: É a diferença entre as coordenadas X da origem e do destino da cadeira de rodas;
- Diferença Y: É a diferença entre as coordenadas Y da origem e do destino da cadeira de rodas.

5.1 Deslocamento da Cadeira sem Obstáculos

O primeiro cenário de testes ao qual a cadeira de rodas foi submetida não possui obstáculos no caminho. Dessa forma, apenas o controle proporcional atuou sobre a direção do veículo. A Figura 34 mostra as posições inicial e final da cadeira motorizada.

As configurações do teste com a cadeira foram:

- Origem: x = 0, y = 0;
- Destino: x = 7, y = 8;
- Obstáculos: Não há.

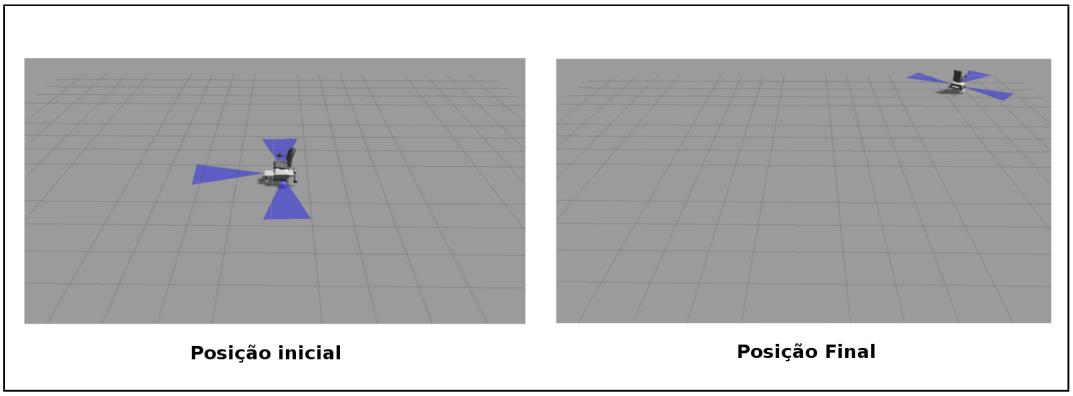

Figura 34: Cenário de teste sem obstáculos.

A Figura 35 apresenta um gráfico com valores capturados durante a execução do teste sem obstáculos. O sensor sonar frontal manteve o valor da distância medida igual ao seu alcance máximo (2 m) durante todo o percurso. A diferença entre as coordenadas x e y de origem e destino foram diminuindo até atingir 0,5 m (margem de erro aceito pelo controle proporcional). A componente angular do comando de deslocamento oscilou entre -0,5 e 0,5 radianos (limite imposto no algoritmo).

Esse teste foi concluído com sucesso. A cadeira de rodas atingiu a posição desejada.

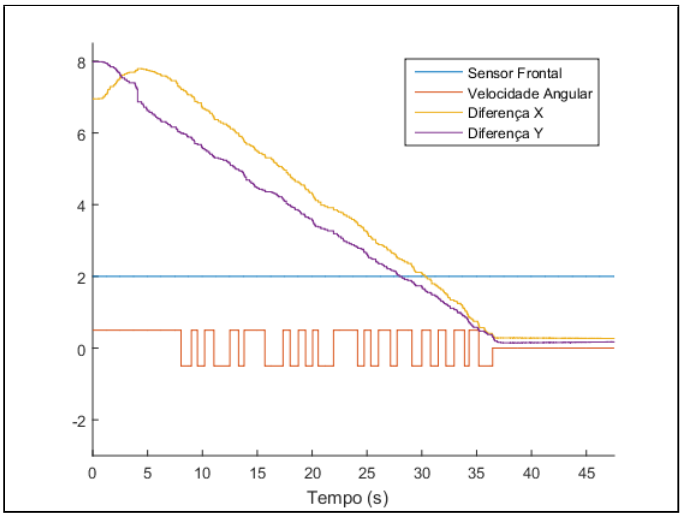

Figura 35: Gráfico com resultados do cenário de teste sem obstáculo.

5.2 Deslocamento da Cadeira com Obstáculo Horizontal

Neste cenário de teste, colocou-se uma barreira de blocos do tipo *New Jersey* entre as posições inicial e final. Isso forçou o desvio realizado pelo sistema anticolisão. A Figura 36 é uma composição de imagens capturadas durante o percurso da cadeira de rodas.

As configurações do teste com a cadeira foram:

- Origem: x = -5, y = 0;
- Destino: x = 5, y = 0;
- Obstáculos: Barreira de passagem formada por uma fila de blocos do tipo *New Jersey* colocados na posição horizontal entre a origem e o destino do veículo.

A Figura 37 apresenta um gráfico com valores capturados durante a execução do teste com barreira na horizontal. Após 10 segundos, aproximadamente, o sensor sonar frontal passou a detectar obstáculo. O algoritmo anticolisão começou a atuar, pois observa-se que a diferença das coordenadas Y, que estava estável, passou a aumentar (o veículo se deslocou para a esquerda). A diferença das coordenadas X, que estava diminuindo, teve um pequeno aumento e só voltou a diminuir depois que a barreira foi ultrapassada. Além disso, é possível notar que a velocidade angular ficou em 0 enquanto a "Diferença X" deixou de diminuir, ou seja, o veículo percorreu paralelamente a barreira para contorná-la.

O teste foi concluído com sucesso. A cadeira de rodas motorizada conseguiu desviar do obstáculo e chegou à posição desejada.

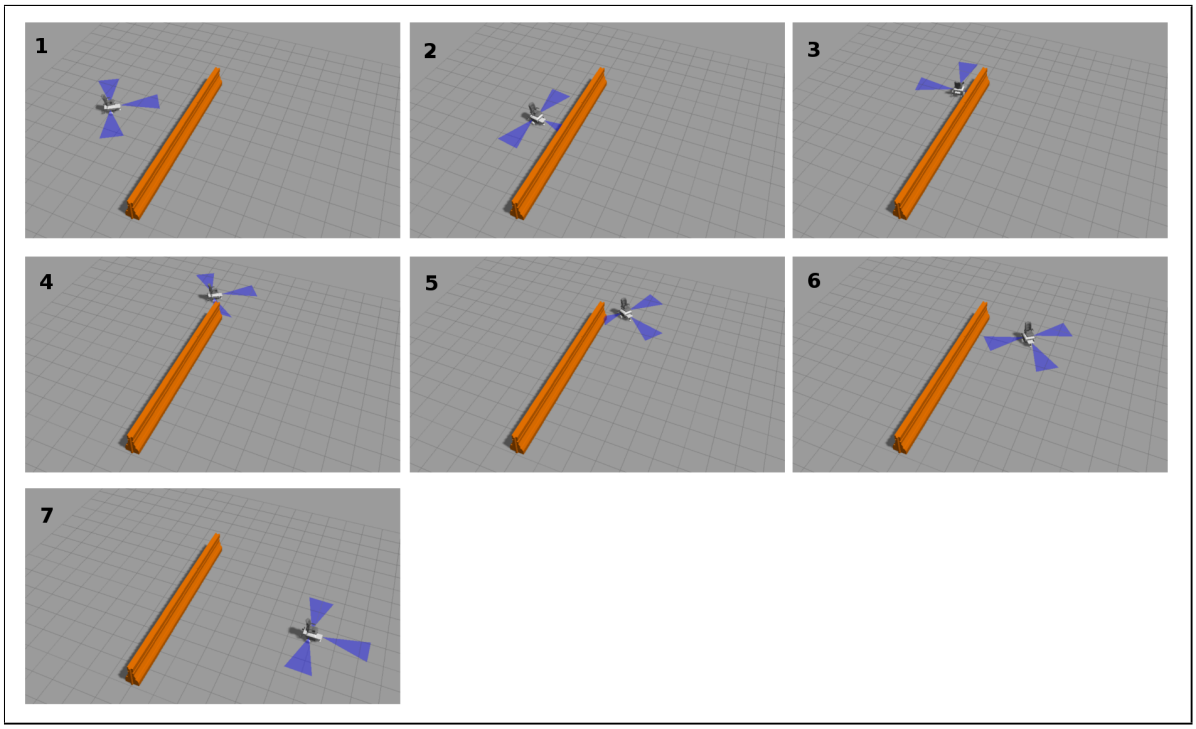

Figura 36: Cenário de teste sem obstáculos.

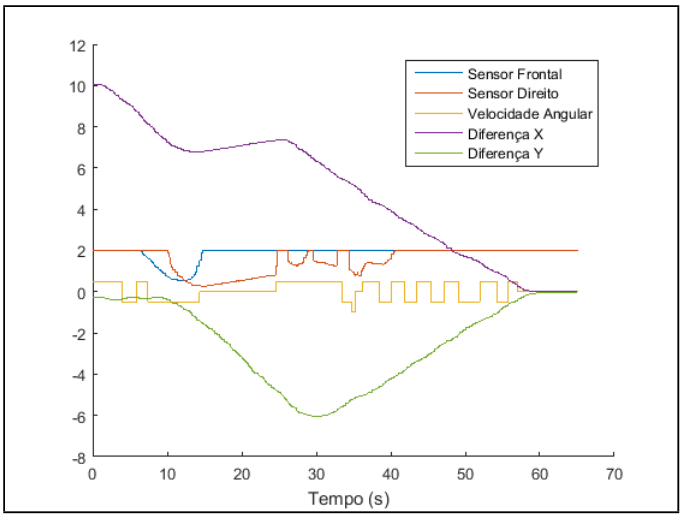

Figura 37: Gráfico com resultados do cenário de teste com obstáculo na horizontal.

5.3 Deslocamento da Cadeira com Obstáculo Diagonal

Este cenário de teste possui uma barreira de blocos, do tipo *New Jersey*, na diagonal entre as posições inicial e final da cadeira de rodas motorizada. Isso acionou sistema anticolisão. A Figura 38 é uma composição de imagens capturadas durante o percurso da cadeira de rodas.

As configurações do teste com a cadeira foram:

- Origem: x = 5, y = 2;

- Destino: x = -5, y = -2;
- Obstáculos: Barreira de passagem formada por uma fila de blocos do tipo *New Jersey* colocados na posição diagonal entre a origem e o destino do veículo.

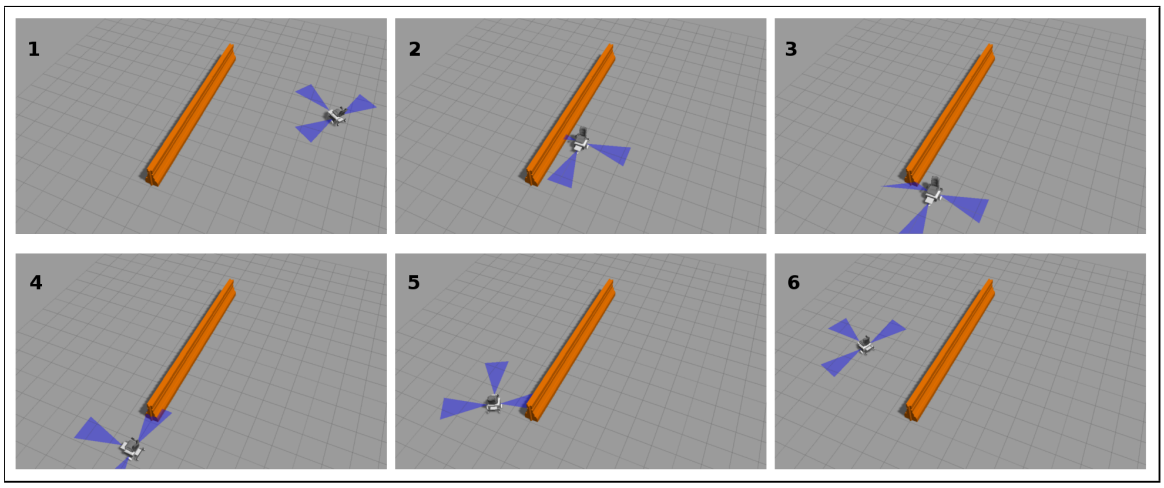

Figura 38: Cenário de teste com obstáculo na posição diagonal entre a origem e o destino da cadeira de rodas.

A Figura 39 apresenta um gráfico com valores capturados durante a execução do teste com barreira na diagonal. Após 15 segundos, aproximadamente, o sensor sonar frontal passou a detectar obstáculo. Após 18 segundos o sensor lateral direito também detectou barreira. Após 22 segundos a velocidade angular ficou estável em 0 (resultado da ação do sistema anticolisão). Nesse momento, observa-se que a diferença das coordenadas X, que estava diminuindo, teve um pequeno aumento e só voltou a diminuir depois que a barreira começou a ser ultrapassada. Pelo gráfico, conclui-se que a barreira foi completamente ultrapassada no tempo de 42 segundos, aproximadamente, pois a diferença Y começou a diminuir e o sensor direito também passou a indicar um aumento da distância para a barreira.

O teste foi concluído com sucesso. A cadeira de rodas motorizada conseguiu desviar do obstáculo e chegou à posição desejada.

5.4 Deslocamento da Cadeira com Obstáculo Lateral

Este cenário de teste possui uma barreira de blocos, do tipo *New Jersey*, paralelo ao deslocamento da cadeira de rodas. O sistema anticolisão impede aproximações à direita. A Figura 40 é uma composição de imagens capturadas durante o percurso da cadeira de rodas.

As configurações do teste com a cadeira foram:

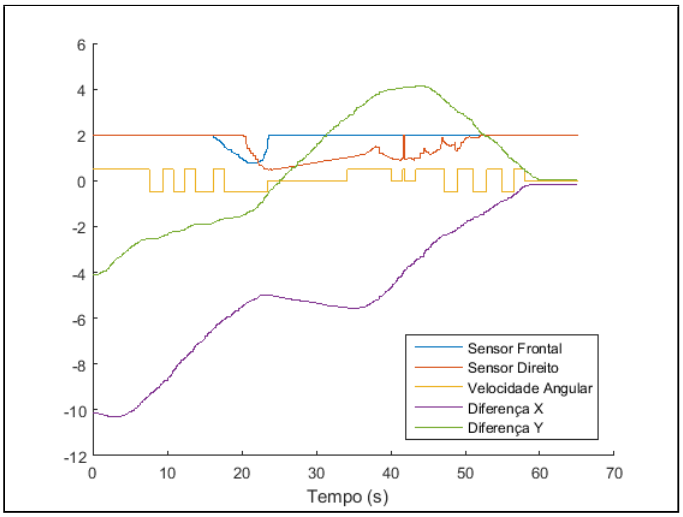

Figura 39: Gráfico com resultados do cenário de teste com obstáculo na diagonal.

- Origem: x = 0, y = 7;
- Destino: x = 0, y = -7;
- Obstáculos: Barreira de passagem formada por uma fila de blocos do tipo *New Jersey* colocados paralelamente ao deslocamento do veículo.

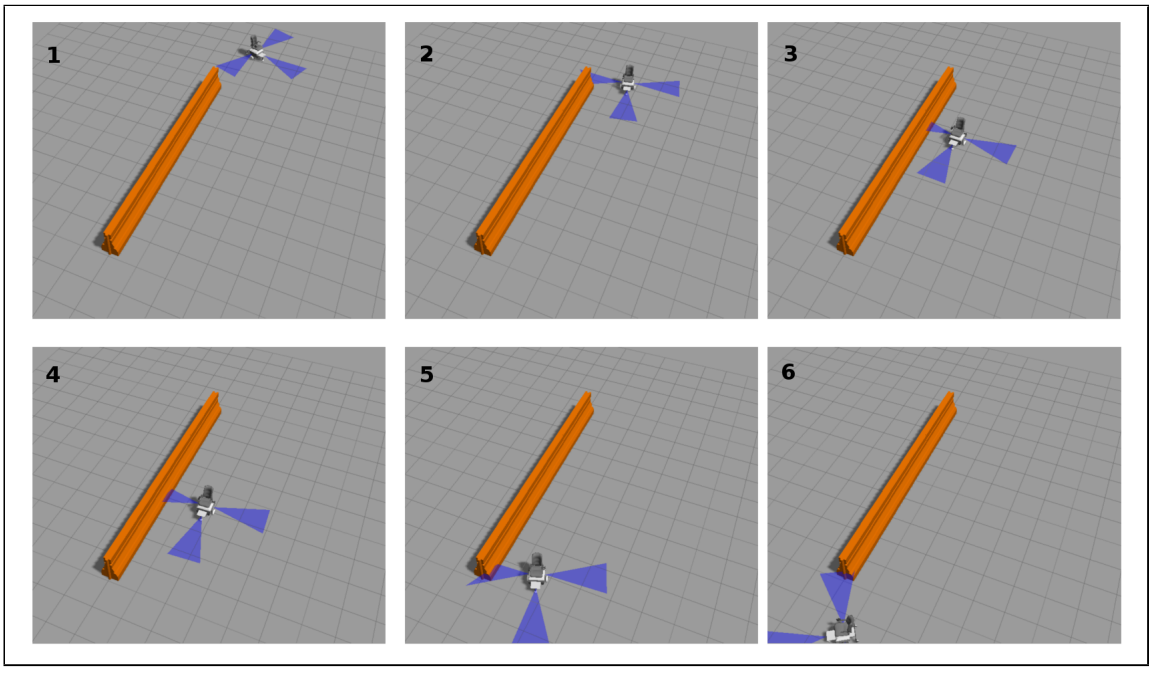

Figura 40: Cenário de teste com obstáculo colocado paralelamente ao deslocamento da cadeira de rodas.

A Figura 41 apresenta um gráfico com valores capturados durante a execução do teste com barreira paralela ao percurso da cadeira. Após 9 segundos, aproximadamente, o sensor sonar direito passou a detectar obstáculo continuamente. Nesse momento, houve

um aumento na diferença das coordenadas X como resultado da ação do algoritmo anticolisão que manteve uma distância mínima de 1 m da barreira. O sensor sonar frontal reportou obstáculo apenas quando a cadeira iniciou seu movimento. Porém essa detecção foi muito pequena e não ocasionou ação do sistema anticolisão. O trajeto configurado para esse teste resultou na variação da coordenada Y apenas. Isso pode ser observado pela variação da "Diferença Y".

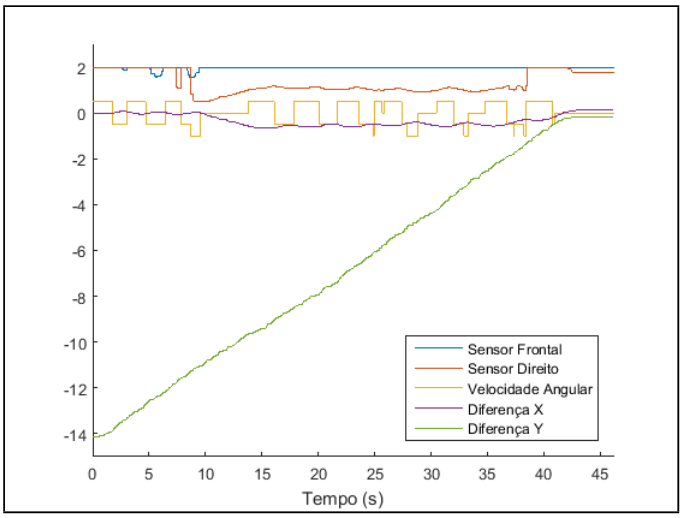

Figura 41: Gráfico com resultados do cenário de teste com obstáculo colocado paralelamente ao deslocamento da cadeira motorizada.

O teste foi concluído com sucesso. A cadeira de rodas motorizada conseguiu chegar à posição desejada sem colidir com a barreira lateral.

5.5 Deslocamento da Cadeira com Obstáculo em U

Este cenário de teste possui uma barreira de blocos, do tipo *New Jersey*, dispostos de maneira a formar uma forma concava, como a letra "U". O sistema anticolisão irá impedir a colisão frontal com a barreira, mas ao desviar, encontrará uma nova barreira. A Figura 42 é uma composição de imagens capturadas durante o percurso da cadeira de rodas.

As configurações do teste com a cadeira foram:

- Origem: x = 0, y = 0;
- Destino: x = -8, y = 0;
- Obstáculos: Barreira de passagem formada por blocos do tipo *New Jersey* dispostos em forma de "U".

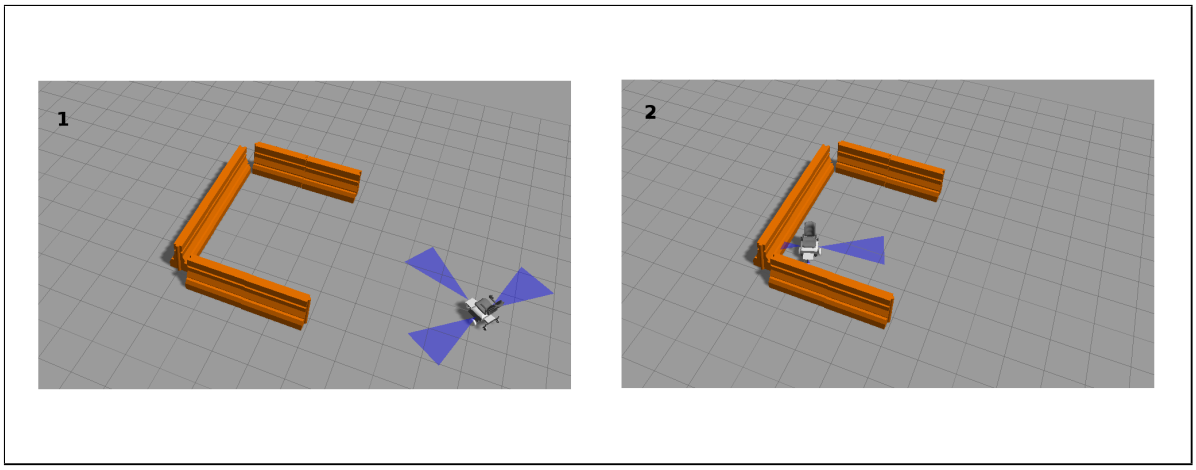

Figura 42: Cenário de teste com obstáculo colocado em forma de “U”.

A Figura 43 apresenta um gráfico com valores capturados durante a execução do teste com barreira dispostos em forma de “U”. Após 10 segundos, o sensor sonar frontal detectou obstáculo à frente. O algoritmo anticolisão atuou direcionando a cadeira para a esquerda. Isso pode ser observado pela “Diferença Y” que passou a aumentar. Mesmo após virar à esquerda, o sonar frontal detectou obstáculo (agora medindo a distância para o obstáculo lateral). O sensor sonar direito também passou a indicar obstáculo. A “Diferença X” diminuiu até encontrar a barreira e depois aumentou, pois o veículo retornou. Entretanto, a cadeira virou para a esquerda novamente, reiniciando o trajeto. Após essa tentativa o veículo acabou enroscado na lateral esquerda e as diferenças X e Y ficaram com pouca variação.

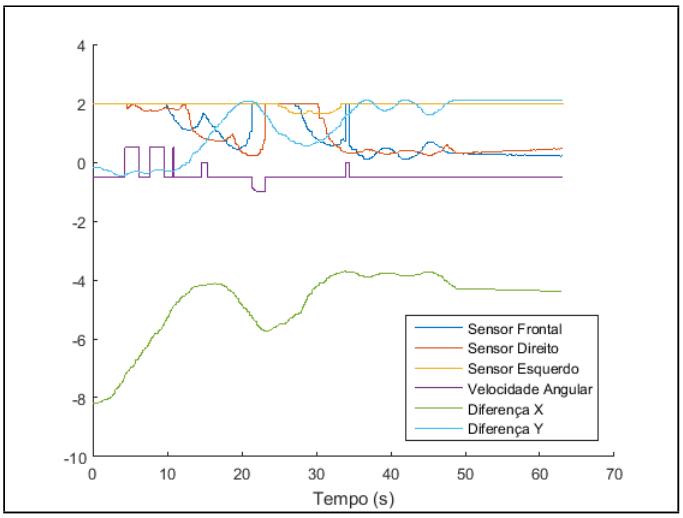

Figura 43: Gráfico com resultados do cenário de teste com obstáculo forma de “ U”.

O teste falhou. A cadeira de rodas motorizada foi incapaz de contornar a barreira e não chegou ao destino desejado.

Para que a cadeira fosse capaz de contornar o obstáculo seria necessário aperfeiçoar

o algoritmo anticolisão. O movimento angular do veículo esta limitado a um ângulo de 0.5 radianos e isso exige um espaço maior para a manobra. A capacidade de girar sobre o próprio eixo poderia melhorar o resultado do teste.

5.6 Deslocamento da Cadeira com Obstáculo em U Estreito

Este cenário de teste possui uma barreira de blocos, do tipo *New Jersey*, dispostos de maneira a formar uma forma concava, como a letra "U" formando um corredor estreito. O sistema anticolisão irá impedir a colisão frontal com a barreira, mas não terá espaço para realizar o desvio. A Figura 44 é uma composição de imagens capturadas durante o percurso da cadeira de rodas.

As configurações do teste com a cadeira foram:

- Origem: x = 0, y = 0;
- Destino: x = -7, y = 0;
- Obstáculos: Barreira de passagem formada por blocos do tipo *New Jersey* dispostos em forma de "U" e corredor estreito.

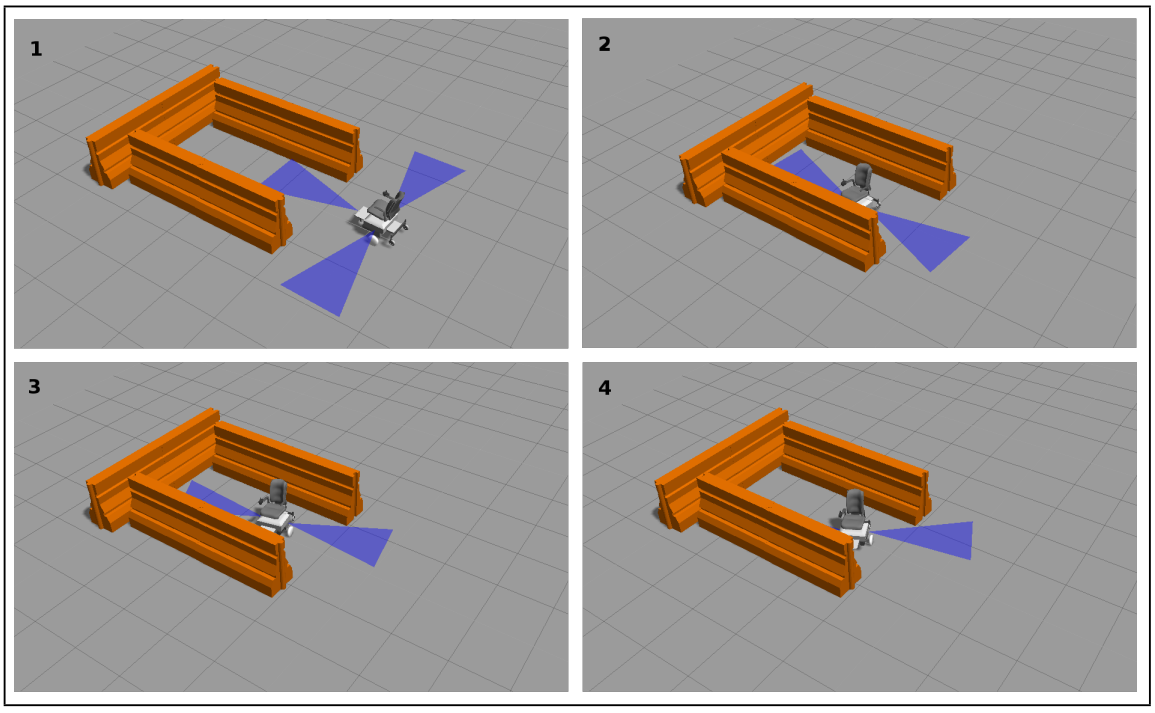

Figura 44: Cenário de teste com obstáculo colocado em forma de "U" estreito.

A Figura 45 apresenta um gráfico com valores capturados durante a execução do teste com barreira dispostos em forma de "U". Após 10 segundos, o sensor sonar frontal

detectou obstáculo à frente. O algoritmo anticolisão atuou direcionando a cadeira para a esquerda. Isso pode ser observado pela "Diferença Y" que passou a aumentar. Mesmo após virar à esquerda, o sonar frontal detectou obstáculo (agora medindo a distância para o obstáculo lateral). O sensor sonar direito também passou a indicar obstáculo. A "Diferença X" diminuiu até encontrar a barreira e depois aumentou, pois o veículo retornou. Entretanto, a cadeira virou para a esquerda novamente, reiniciando o trajeto. Após essa tentativa o veículo acabou enroscado na lateral esquerda e as diferenças X e Y ficaram com pouca variação.

O algoritmo anticolisão prevê a situação em que os três sensores (frontal, lateral esquerdo e lateral direito) estão detectando obstáculo. Nesse caso, a cadeira deveria parar. Isso não ocorreu, entretanto, devido à imprecisão do algoritmo proporcional que muda a direção da cadeira durante o trajeto. O gráfico da Figura 45 mostrou que o sensor frontal detectou obstáculo à frente prematuramente e isso resultou em tentativa de desvio.

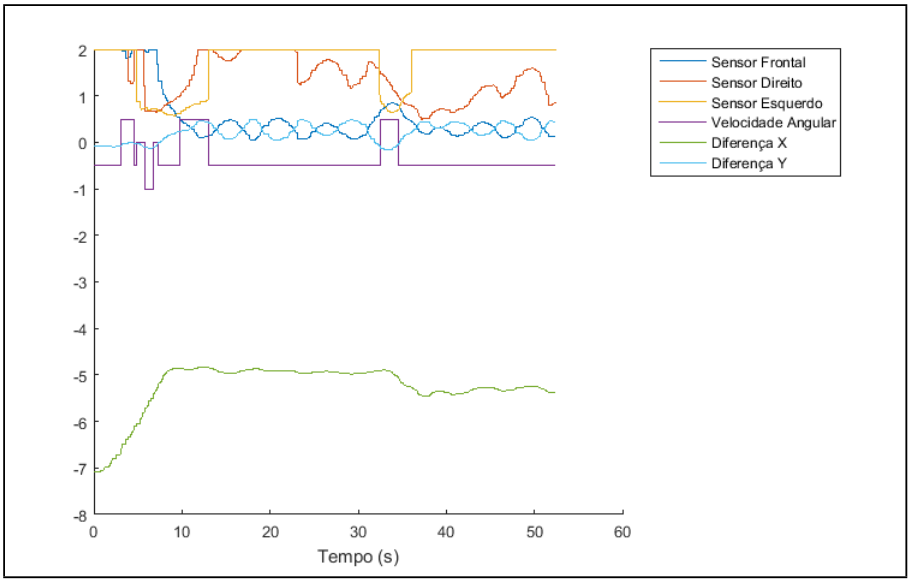

Figura 45: Gráfico com resultados do cenário de teste com obstáculo forma de " U" estreito.

O teste falhou. A cadeira de rodas motorizada não parou, foi incapaz de contornar o obstáculo e não chegou ao destino desejado.

6 Discussão e Conclusão

Conclui-se, através dos experimentos realizados com a cadeira de rodas em ambiente virtual, que o método proposto permite criar situações realistas a fim de validar algoritmos de controle para cadeira de rodas motorizadas.

A integração dos sistemas ROS e simulador GAZEBO mostrou-se bastante vantajosa, pois não houve dificuldade em utilizá-los em conjunto. O processo de configuração limitou-se em uma simples instalação de pacotes de *software*.

A criação do robô virtual foi a maior dificuldade encontrada. As informações e os tutoriais disponíveis no site do ROS e do simulador GAZEBO ajudaram bastante na compreensão desses sistemas. A documentação do ROS possui detalhes sobre a arquitetura e exemplos de como utilizá-lo. O GAZEBO também tem um bom tutorial. Entretanto, a criação do arquivo modelo (SDF), para descrição do robô, mostrou ser um trabalho complexo devido a falta de mais exemplos. A existência de um site dedicado à especificação do formato SDF não supre a escassez de exemplos e melhores explicações sobre os atributos do robô neste formato.

A definição de propriedades mecânicas e físicas do modelo mostrou ser a etapa que requer maior aprofundamento. O tutorial *Find Out Inertial Parameters (2015)* ajudou a obter a matriz inercial do modelo em 3D, mas sem precisão e com muitos ajustes intuitivos.

Diante disso, a disponibilidade de uma cadeira de rodas no banco de modelos da comunidade de usuários do ROS e do GAZEBO seria um facilitador para o desenvolvimento de sistemas de controle para cadeira de rodas. Essa ideia está alinhada com uma das principais vantagens oferecidas pelo ROS: A colaboração.

A adoção do aplicativo MATLAB facilitou o trabalho. A disponibilidade de exemplos e a facilidade em se criar algoritmos de controle, com auxílio dos blocos de programação do Simulink, permitiu realizar os primeiros testes rapidamente.

Este trabalho limitou-se em submeter a cadeira de rodas a situações que pudessem exercitar o algoritmo anticolisão como forma de demonstrar a viabilidade da prototipagem de algoritmos para cadeira de rodas utilizando ROS, simulador GAZEBO e MATLAB. Entretanto, a metodologia proposta pode ser aplicada para desenvolvimento de sistemas de controle que atuem em diferentes situações do cotidiano como trajetos por corredores, passagem por portas estreitas e deslocamento em terrenos irregulares. Para isso, basta reproduzir os cenários de teste virtualmente.

O movimento angular do veículo esta limitado a um ângulo de 0.5 radianos (aproximadamente 28,6 graus). Isso impede que ele faça curvas fechadas e, portanto, tem

dificuldades em desviar de obstáculos quando há barreiras em forma de "U". A capacidade de girar sobre o próprio eixo poderia melhorar os resultados dos testes. Isso poderia ser feito com o aperfeiçoamento do algoritmo anticolisão.

A característica colaborativa do ROS pode ser observada com o uso dos *plugins* para motor e sensor sonar. A reutilização do controle proporcional para posicionamento da cadeira também é um exemplo de reutilização de código. Entretanto, essa característica pode ser ainda mais explorada ao se adotar outros módulos de controle e sensores.

Dentre as ferramentas de *software* adotadas, apenas o MATLAB não é gratuito. Sua licença de uso varia entre 45 e 250 dólares para uso estudantil e acadêmico. O *hardware* utilizado é bastante comum (dois computadores do tipo *Personal Computer* com processadores x86-64) e com custo total aproximado em R$ 3.000,00. O investimento total é considerado baixo. Outra vantagem financeira dessa solução é a possibilidade de se criar *software* embarcado com o ROS sem necessidade de pagamento de licenças de uso e distribuição. Diante disso, conclui-se, também, que o método proposto neste trabalho reduz custos no desenvolvimento de cadeira de rodas inteligentes e pode viabilizar o surgimento de equipamentos a preços mais baixos (importante contribuição social em países em desenvolvimento).

6.1 Proposta para Trabalhos Futuros

Para trabalhos futuros pode-se criar um sistema gerenciador de subsistemas de controle para cadeira de rodas motorizadas capaz de integrar os diferentes algoritmos como, por exemplo, controle de tração, sistema anticolisão, controle por gestos e sistemas de mapeamento de ambientes. Isso estimularia a colaboração entre diferentes pesquisadores, pois permitiria a integração de diferentes trabalhos. Além disso, a disponibilidade de sistemas, facilmente integráveis, fomentaria estudos com cadeiras de rodas em situações peculiares a diferentes culturas e países.

Outra proposta é a definição de uma cadeira de rodas motorizada que possa ser produzida para, então, criar uma cadeira de rodas virtual. O arquivo modelo de robôs, no formato SDF, permite definir propriedades físicas como atritos, momento de inércia, torque, frequencias de amostragem e alcance de sensores e outras propriedades. Para que a simulação seja realista, todas essas informações dependem do projeto da cadeira de rodas real. O tamanho das rodas, a escolha dos motores e redutores, a quantidade e variedade dos sensores disponíveis e o desenho do corpo da cadeira, são exemplos de escolhas de projeto que afetam diretamente a definição da cadeira de rodas virtual.

Referências

BAYAR, V. *et al.* Fuzzy logic based design of classical behaviors for mobile robots in ros middleware. In: *Innovations in Intelligent Systems and Applications (INISTA) Proceedings, 2014 IEEE International Symposium on*. [S.l.: s.n.], 2014. p. 162–169.

CERES, R. *et al.* A robotic vehicle for disabled children. *Engineering in Medicine and Biology Magazine, IEEE*, v. 24, n. 6, p. 55–63, Nov 2005. ISSN 0739-5175.

CHIPAILA, C. *et al.* Hardware and software solutions for a conventional electric-powered wheelchair. In: *System Theory, Control and Computing (ICSTCC), 2012 16th International Conference on*. [S.l.: s.n.], 2012. p. 1–6.

CONNELL, J.; YOUNG, U. K. *Comprehensive scoping study on the use of assistive technology by frail older people living in the community*. [S.l.]: Urbis, 2008.

CREATING A ROS PACKAGE. 2015. Disponível em: <http://wiki.ros.org/catkin/Tutorials/CreatingPackage>. Acesso em: 11 set. 2015.

CREATING A WORKSPACE FOR CATKIN. 2015. Disponível em: <http://wiki.ros.org/catkin/Tutorials/create_a_workspace>. Acesso em: 11 set. 2015.

DU, Z. *et al.* A ros/gazebo based method in developing virtual training scene for upper limb rehabilitation. In: *Progress in Informatics and Computing (PIC), 2014 International Conference on*. [S.l.: s.n.], 2014. p. 307–311.

FIND OUT INERTIAL PARAMETERS. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=inertia&cat=>. Acesso em: 14 set. 2015.

FRISOLI, A. *et al.* Robotic assisted rehabilitation in virtual reality with the l-exos. *Stud Health Technol Inform*, v. 145, p. 40–54, 2009.

FUJIMURA, K.; XU, L. Sign recognition using constrained optimization. In: *Computer Vision–ACCV 2007*. [S.l.]: Springer, 2007. p. 32–41.

GAZEBO. 2015. Disponível em: <http://gazebosim.org>. Acesso em: 23 fev. 2015.

GAZEBO COMPONENTS. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=components&cat=get_started>. Acesso em: 20 mar. 2015.

GETTING STARTED WITH ROS IN SIMULINK. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=inertia&cat=>. Acesso em: 21 out. 2015.

GRIGORESCU, S. M. *et al.* A bci-controlled robotic assistant for quadriplegic people in domestic and professional life. *Robotica*, v. 30, p. 419–431, 5 2012. ISSN 1469-8668. Disponível em: <http://journals.cambridge.org/article_S0263574711000737>.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATíSTICA. *Censo Demográfico 2010*. 2010. Disponível em: <http://www.ibge.gov.br/home/estatistica/populacao/censo2010/default.shtm>. Acesso em: 19 Nov 2013.

JIA, S. *et al.* Intelligent home service robotic system based on robot technology middleware. In: IEEE RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS. *IROS*. [S.l.]: IEEE, 2006. p. 4478–4483.

JUN, W.; JIAN, H.; YONGJI, W. Upper limb rehabilitation robot integrated with motion intention recognition and virtual reality environment. In: IEEE. *Control Conference (CCC), 2010 29th Chinese*. [S.l.], 2010. p. 3709–3715.

LAFFERTY, J.; MCCALLUM, A.; PEREIRA, F. C. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

LANE, I. *et al.* Hritk: The human-robot interaction toolkit rapid development of speech-centric interactive systems in ros. In: *NAACL-HLT Workshop on Future Directions and Needs in the Spoken Dialog Community: Tools and Data*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2012. (SDCTD '12), p. 41–44. Disponível em: <http://dl.acm.org/citation.cfm?id=2390444.2390467>.

LEE, A.; KAWAHARA, T. Recent development of open-source speech recognition engine julius. *Proceedings: APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference*, Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference, International Organizing Committee, p. 131–137, 2009.

LI, R.; OSKOEI, M.; HU, H. Towards ros based multi-robot architecture for ambient assisted living. In: *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. [S.l.: s.n.], 2013. p. 3458–3463.

LI, R. *et al.* Ros based multi-sensor navigation of intelligent wheelchair. In: *Emerging Security Technologies (EST), 2013 Fourth International Conference on*. [S.l.: s.n.], 2013. p. 83–88.

MAKE A MOBILE ROBOT. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=build_robot&cat=build_robot>. Acesso em: 8 set. 2015.

MAKE A MODEL. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=build_model>. Acesso em: 26 mar. 2015.

MARTINEZ, E. F. A. *Learning ROS for Robotics Programming*. [S.l.]: Packt Publishing Ltd, 2013. ISBN 978-1-78216-144-8.

MIRO, J.; POON, J.; HUANG, S. Low-cost visual tracking with an intelligent wheelchair for innovative assistive care. In: *Control Automation Robotics Vision (ICARCV), 2012 12th International Conference on*. [S.l.: s.n.], 2012. p. 1540–1545.

O'KANE, J. M. *A Gentle Introduction to ROS*. [S.l.]: Independently published, 2013. Available at <http://www.cse.sc.edu/~jokane/agitr/>. ISBN 978-1492143239.

ONO, Y.; UCHIYAMA, H.; POTTER, W. A mobile robot for corridor navigation: A multi-agent approach. In: *Proceedings of the 42Nd Annual Southeast Regional Conference*. New York, NY, USA: ACM, 2004. (ACM-SE 42), p. 379–384. ISBN 1-58113-870-9. Disponível em: <http://doi.acm.org/10.1145/986537.986629>.

OZCELIKORS, M. *et al.* Kinect based intelligent wheelchair navigation with potential fields. In: *Innovations in Intelligent Systems and Applications (INISTA) Proceedings, 2014 IEEE International Symposium on.* [S.l.: s.n.], 2014. p. 330–337.

PASTEAU, F.; KRUPA, A.; BABEL, M. Vision-based assistance for wheelchair navigation along corridors. In: *IEEE Int. Conf. on Robotics and Automation, ICRA'14.* Hong-Kong, Hong-Kong: [s.n.], 2014. Disponível em: <http://hal.inria.fr/hal-00949188>.

PETRY, M. R. *et al.* Intellwheels: Intelligent wheelchair with user-centered design. In: IEEE. *e-Health Networking, Applications & Services (Healthcom), 2013 IEEE 15th International Conference on.* [S.l.], 2013. p. 414–418.

ROS DISTRIBUTIONS. 2015. Disponível em: <http://wiki.ros.org/Distributions>. Acesso em: 25 set. 2015.

ROS.ORG. 2015. Disponível em: <http://wiki.ros.org>. Acesso em: 12 jan. 2015.

SDF FORMAT SPECIFICATION. 2015. Disponível em: <http://sdformat.org/spec>. Acesso em: 8 set. 2015.

SIMPSON, R. C.; LOPRESTI, E. F.; COOPER, R. A. How many people would benefit from a smart wheelchair? *Journal of rehabilitation research and development*, v. 45, n. 1, p. 53–71, 2008. Disponível em: <http://d-scholarship.pitt.edu/15843/>.

The MathWorks Inc. *ROS I/O Package Getting Started Guide R2014a.* 2014. 29 p. Documento eletrônico distribuído em conjunto com o software de instalação da extensão "ROS I/O Package"para MATLAB.

TREFLER, E. *et al.* Outcomes of wheelchair systems intervention with residents of long-term care facilities. *Assist Technol*, University of Pittsburgh, SHRS, Department of Rehabilitation Science and Technology, USA., v. 16, n. 1, p. 18–27, 2004. ISSN 1040-0435. Disponível em: <http://view.ncbi.nlm.nih.gov/pubmed/15357146>.

UBUNTU INSTALL OF ROS INDIGO. 2015. Disponível em: <http://wiki.ros.org/indigo/Installation/Ubuntu>. Acesso em: 2 set. 2015.

USING GAZEBO PLUGINS WITH ROS. 2015. Disponível em: <http://gazebosim.org/tutorials?tut=ros_gzplugins>. Acesso em: 30 set. 2015.

WASSON, G. *et al.* Intelligent mobility aids for the elderly. In: *Eldercare Technology for Clinical Practitioners.* [S.l.]: Springer, 2008. p. 53–76.

WEHRMEISTER, M.; PEREIRA, C.; BECKER, L. Object-oriented methodology to the development of embedded real-time systems. In: *Industrial Informatics, 2005. INDIN '05. 2005 3rd IEEE International Conference on.* [S.l.: s.n.], 2005. p. 68–73.

ZIHERL, J. *et al.* Evaluation of upper extremity robot-assistances in subacute and chronic stroke subjects. *Journal of NeuroEngineering and Rehabilitation*, v. 7, n. 1, p. 52, 2010. ISSN 1743-0003. Disponível em: <http://www.jneuroengrehab.com/content/7/1/52>.