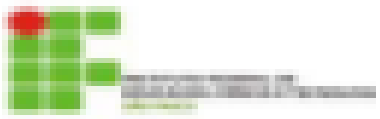


INSTITUTO FEDERAL DE EDUCAÇÃO CIÊNCIA E TECNOLOGIA – IFSP

RAPHAEL DE ABREU ALVES E SILVA

DESENVOLVIMENTO DE AMBIENTE OPERACIONAL ROS PARA QUAD-ROTORES

São Paulo - Brasil

2017

Raphael de Abreu Alves e Silva

DESENVOLVIMENTO DE AMBIENTE OPERACIONAL ROS PARA QUAD-ROTORES

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 Controle e Automação de Processos

Orientador: Prof. Dr. Alexandre Simião Caporali

São Paulo - Brasil

2017

Agradecimentos

Agradeço ao meu orientador Prof. Dr. Alexandre Simião Caporali, um amigo que sempre me estimulou a buscar excelência no que fazia e que me apresentou o mundo acadêmico despertando em mim o desejo de fazer pesquisa científica.

A Deus por me dar as oportunidades e sabedoria para aproveitá-las.

A minha esposa e filha que me apoiaram e possibilitaram que este trabalho fosse realizado.

Aos amigos pela ajuda essencial durante a realização desse trabalho.

Aos meus colegas de trabalho que me auxiliaram durante todo curso, tanto com conselhos quanto com soluções.

A minha mãe Rita de Cássia Abreu Erra pela ajuda com a revisão, nos últimos momentos.

Ao Instituto Federal de São Paulo pelo apoio na minha formação e oportunidades de aperfeiçoamento.

RESUMO

A concepção de robôs tem se tornado cada vez mais comum tanto em ambiente acadêmico como em setores profissionais e amadores, isto é alavancado pela redução dos custos de componentes e da elaboração de ferramentas gratuitas, tais como: ROS; Arduíno; e Linux. Um dos grandes obstáculos dessa tarefa de elaborar sistemas robóticos é a integração dos diversos componentes construtivos, estes fabricados por diferentes empresas e com metodologias de uso diversos. Este trabalho visa a elaboração de um sistema operacional utilizando o ambiente de desenvolvimento ROS (*Robot Operating System* – Sistema Operacional para Robôs), o qual propõe a melhora da integração de subsistemas e componentes pertencentes a robôs, sejam estes, sensores, computadores ou um grupo de atuadores, com um ambiente que favorece a concepção em diversas frentes concomitantes, mantendo a coesão do sistema. Outro fator fundamental é que ROS é um ambiente gratuito com grande comunidade colaborativa, oferecendo diversos módulos de funcionalidades específicas (operação remota, controle de posição) e exemplos disponíveis para o uso. A maior contribuição desse trabalho é a elaboração de um sistema que possa auxiliar na expansão de tecnologias aplicadas para robôs do tipo quad-rotor, possibilitando avanços mais rápidos e em harmonia com o que tem sido utilizado por outros pesquisadores.

Palavras-chave: Desenvolvimento de sistemas robóticos– Quad-rotor – ROS – Simulador de quad-rotores - Telemetria.

ABSTRACT

The robot development nowadays is a common issue in the academic and non-academic and hobbyist sphere, due to cost reduction of components and open source tool. One of the biggest barriers to robot development is integration between many sources of components and systems, due to several manufacturers and many methodologies of implementation. With an environment that favors the development. This paper goal is developing an operating system using ROS, which add new capabilities for improvement of integration of sub-systems, groups of robots, sensors and computers. ROS is an ambient for robot development, with this tool is possible work in several parts of the main project at the same time and still have full integration between sub-systems. Furthermore, as ROS is an open source tool with collaborative community which offer many samples with specific functionalities for robot development (tele operation, position controlling) that are already tested and give support to new projects. The major contribution of this paper is helping the development process of technologies for quadcopters robots that make possible faster advances and harmony between this work and the other's researchers.

Keywords: Robot systems development – Quadcopter – ROS – Quadcopter simulator - Telemetry

Lista de figuras

Lista de abreviaturas

DDD: D-cubo – *Dangerous, Dirty and Dull* – perigoso, sujo e maçante.

ESC: *Electronic speed controller* – Controlador de velocidade eletrônico.

GDL: Graus de Liberdade.

IMU: *inertial measurement unit* - Unidade de medição inercial.

ISA: *International Standard Atmosphere* - Padrão Internacional Atmosférico.

MEMS*: Micro-Electro-Mechanical Systems* - Sistemas Micro-eletromecânicos.

OSRF: *Open Source Robotics Foundation* – Fundação de programas computacionais livres para robótica.

PID: Proporcional Integrativo Derivativo.

RF: rádio frequência.

ROS: *Robot Operating System* – Sistema Operacional para Robôs.

Rviz: ROS *visualization* – Visualizador de ROS.

SDF: *Simulation Description Format* – formato de descrição de simulação.

SLAM: *Simultaneous Localization and Mapping* - Localização e mapeamento simultâneo.

URI: *Uniform Resource Identifier* – Identificador de recursos uniforme.

VANT's: Veículos Aéreos Não Tripulados.

Sumário

1 INTRODUÇÃO

1.1. Justificativa

Nos últimos anos, foi observado um crescimento do interesse em robótica. Diversas áreas industriais (automobilística, manufatura e espacial) têm se utilizado de robôs para substituir homens em locais perigosos, trabalho repetitivo e situações onerosas. Uma grande parte das pesquisas dessa área está em equipamentos de plataforma aérea (aviões com asas fixas, dirigíveis, helicópteros, quad-rotores). Cada modelo tem suas vantagens e desvantagens, tornando necessária uma avaliação desses benefícios para a aplicação (BRESCIANI, 2008).

Equipamentos robóticos dependem da utilização de sensores para garantir sua operação, dessa forma conseguem executar suas solicitações de maneira precisa e segura, reduzindo riscos e retrabalhos. Com o crescimento de pesquisas relacionadas a MEMS (*Micro Electro Mechanical Systems*) Sistemas Micro eletromecânicos, os dispositivos se tornaram menores, adquirindo a capacidade de serem utilizados em equipamentos de tamanho reduzido. Acelerômetros, giroscópios, barômetros, dentre outros, passaram a ter dimensões milimétricas (encapsulados) e massa de miligramas. Com isso, a utilização desses componentes tornou-se mais comum e sua produção de larga escala, o que reduziu seu custo (MELO, SALLES e ALMEIDA, 2010).

A partir dessa série de mudanças de cenário favorecendo as pesquisas com VANT's (Veículos Aéreos Não Tripulados) de tamanho reduzido, houve um crescimento significativo desse segmento, pois obtiveram uma melhoria na relação carga-massa das baterias e melhor controle de voo. Em especial os equipamentos elétricos, que por sua simplicidade mecânica e não carregarem combustível inflamável a bordo, foram os mais focados nesse tipo de pesquisa (MELO, SALLES e ALMEIDA, 2010).

Os aeromodelos podem ser denominados como VANT's, o qual é um termo genérico que identifica uma aeronave que pode voar sem tripulação. Ela é normalmente projetada para operar em situações perigosas, repetitivas, em condições hostis ou de difícil acesso ou onerosas. Existe uma grande variedade de VANT's que tem se tornado cada vez mais presente nas áreas civis, sendo uma opção muito utilizada comercialmente (BOUABDALLAH; MURRIERI e SIEGWART, 2004);

Sistemas robóticos podem ser classificados, também, pelos seus níveis de autonomia e inteligência, isso é definido pelos sistemas de controle integrados ao equipamento. Sistemas nos quais os níveis de robustez, autonomia e inteligência são reduzidos, não são capazes de suportar falhas ou desviar de obstáculos para retomar sua tarefa primordial. O estudo de robótica autônoma tem avançado na direção de propor sistemas que tenham "comportamento Inteligente" que torna o robô mais confiável, robusto e menos dependente da intervenção de humanos.

Para inserir "inteligência" a um sistema, foram utilizadas técnicas de controle que relacionam a resposta dos sensores com o que se espera que o dispositivo realize. Os sistemas de controle devem entre outras funções exercer as seguintes tarefas, conforme Ogata (2000):

- Garantir a preservação da integridade física de humanos e do ambiente inserido;
- Garantir a preservação da integridade do robô;
- Garantir a atualização de informações que possam gerar melhoria na execução das tarefas;
- Integrar informações de diversos sensores, tratá-las e interpretá-las;
- Gerar comandos na sequência correta;
- Gerar soluções alternativas com situações não previstas;

- Habilidade de adaptação.

Para se desenvolver controladores adequadamente, é necessário: Modelar o comportamento do sistema de maneira confiável; simular o sistema; integrar o controlador; e realizar testes experimentais (JUNG; OSÓRIO e ROBERTO, 2005).

Quando VANT´s têm equipamentos embarcados que possibilitam aquisição de imagens, se tornam ferramentas poderosas no uso de exploração de ambientes, redução de custo em filmagens ou fotos panorâmicas (*Perspectives Aerials*, 2013), segurança e perseguições (Draganflyer, 2013), inspeções de linhas elétricas de difícil acesso, monitoração de animais, plantações e florestas (MELO, SALLES e ALMEIDA, 2010).

Segundo ROS (2015), ROS (*Robot Operating System* – Sistema Operacional para Robôs) é o um ambiente flexível para desenvolvimento de programas de computador para robôs, um conjunto de ferramentas, bibliotecas (programas que criam funções específicas de programação) e convenções que têm por objetivo principal simplificar a tarefa de desenvolver robôs com grandes níveis de complexidade e comportamento robusto, para uma grande variedade de plataformas robóticas.

Outro fator importante para o uso do ROS é o fato de ser uma plataforma com uma ampla comunidade, que foi criada com o objetivo de ser colaborativa. Isso traz o benefício de já existir diversas soluções disponíveis, desenvolvidas por especialistas de diversos lugares do mundo, gerando ainda mais confiabilidade (DEMARCO; WEST e COLLINS, 2011).

Além disso, devido a alta complexidade dos sistemas robóticos, são oferecidas diversas plataformas físicas com sensores embarcados ou kits, direcionando a concentração das atividades no progresso de códigos de programação e aplicativos para adicionar funcionalidades ao protótipo. (ROS, 2015).

Utilizando a ferramenta ROS, também, se torna possível o desenvolvimento desarticulado de um mesmo robô, tornando o projeto de pesquisa integrador, possibilitando inserir novas capacidades ao robô sem interferir em outras, dessa forma diversas pessoas podem trabalhar no mesmo projeto e manter a fácil integração (COUSINS *et al.*, 2010).

Segundo Cousins *et al.* (2010), o compartilhamento de códigos tem se tornado uma prática cada vez mais recorrente. Essa técnica visa ao aumento da velocidade de desenvolvimento para pesquisadores e permite que a comunidade em geral possa, também, replicar e melhorar os resultados de projetos de determinados grupos. Esse é um fator de grande importância para o uso de ROS na concepção de quad-rotores. Ao utilizar essa técnica, pode-se concentrar em inovações pontuais do projeto e resolução de problemas específicos.

1.2 Objetivo Geral

O objetivo geral deste trabalho é desenvolver um sistema operacional para um VANT, criando a possibilidade da implantação de diversas funcionalidades com integração simplificada utilizando a ferramenta ROS, linguagem de programação C++ e Linux.

1.3 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Implementar comunicação entre o quad-rotor que utiliza myRIO e o computador;
- Implementar comunicação entre o quad-rotor, Arduino e o computador;
- Programar o sistema computacional em ROS que interprete dados recebidos e possa enviar comandos ao quad-rotor;

- Implementar telemetria da plataforma real, possibilitando a validação de experimentos com VANT's.

1.4 Contribuições do trabalho

A proposta desse trabalho é oferecer um ambiente para concepção de robôs do tipo quad-rotor (LI *et al.*, 2013). Como continuação do trabalho de modelagem e controle de estabilidade de um quad-rotor (SILVA, 2014), elaborar um sistema operacional utilizando ROS para um robô que gerará a possibilidade de expandir o leque de funcionalidades do sistema. Esse poderá ter um sistema integrado dos sensores, atuadores e computador, telemetria entre outras diversas possibilidades de aplicação de um VANT (DEMARCO; WEST e COLLINS, 2011; GRABE *et al.*, 2013; MONTUFAR e MUNOZ, 2014; ZAMAN; SLANY e STEINBAUER, 2011).

Abaixo as colaborações deste trabalho:

- Desenvolver sistema operacional para robô VANT (GRABE *et al.*, 2013; MONTUFAR e MUNOZ, 2014);
- Agregar as diversas partes do projeto (DEMARCO; WEST e COLLINS, 2011; GRABE *et al.*, 2013; LEVI *et al.*, 2013; ZAMAN; SLANY e STEINBAUER, 2011);
- Possibilitar o desdobramento independente de funcionalidades para um mesmo robô (COUSINS *et al.*, 2010);
- Compartilhar códigos (COUSINS *et al.*, 2010);
- Oferecer ambiente de simulação (DEMARCO; WEST e COLLINS, 2011);
- Integrar o robô com *softwares* diversos (Linux, Windows, Labview) (HOLD-GEOFFROY *et al.*, 2013; LI *et al.*, 2013);

- Associar quad-rotor que utiliza myRIO ao ROS;
- Operar o robô pelo computador (controles, teclado ou comando especial) (GRABE *et al.*, 2013).

1.5 Organização do Trabalho

O capítulo 2 apresenta o estado da arte da área pesquisada nesse trabalho, onde através de citações de textos, pode-se contextualizar esse estudo aos demais encontrados na comunidade acadêmica.

O capítulo 3 analisa os diversos conceitos teóricos necessários para o desenvolvimento deste trabalho, nele são descritas as ferramentas utilizadas de ROS, programação e prototipagem.

O capítulo 4 apresenta os materiais e métodos utilizados na pesquisa, mostrando a forma como as ferramentas disponíveis de ROS foram aplicadas no projeto e suas utilizações.

No capítulo 5 são apresentados os resultados alcançados e dificuldades encontradas durante a realização deste trabalho, apontando a direção que este trabalho seguirá.

O capítulo 6 apresenta a conclusão obtida neste trabalho e discute o que foi obtido, o que pode ser alterado e propostas para trabalhos futuros.

2 REVISÃO DA LITERATURA

Este capítulo apresenta, por meio de uma revisão analítica, o que é encontrado na comunidade acadêmica relacionado aos sistemas de quad-rotores. Foram estudados projetos nos quais foram utilizados a plataforma de desenvolvimento ROS ou similares e, também, outros projetos que propõe uma arquitetura computacional e protótipo próprio.

Nos últimos anos, o estudo de VANT's teve grande importância no mundo. Diversos usos e propostas foram desenvolvidos e a estrutura de quad-rotor foi focada grandemente na atenção sobre os outros tipos de VANT's.

2.1 Tipos de Aeromodelos

Neste capítulo, serão mostrados os tipos de aeromodelos existentes atualmente, sendo eles:

a) Dirigíveis: por serem classificados como uma aeronave mais leve que o ar, não precisam de propulsores para a estabilidade. São utilizados apenas para locomoção, silenciosos, fáceis de controlar, ocupam maior volume e menor relação carga e autonomia, podem ser utilizados em ambientes fechados, boa autonomia (BOUABDALLAH; MURRIERI e SIEGWART, 2004);

b) Aviões: possuem boa estabilidade em virtude de sua sustentação aerodinâmica (após atingir velocidade mínima). Necessita de área grande de aterrissagem e decolagem, não podem se aproximar muito de objetos (reduz muito a capacidade de uso em ambiente fechado), melhor alcance em ambiente aberto (podem ser utilizados em

longas distâncias), boa relação carga e autonomia (BOUABDALLAH; MURRIERI e SIEGWART, 2004);

c) Helicópteros: apresentam vantagens sobre os demais modelos, pois possuem 6 GDL (Graus De Liberdade), grande manobrabilidade, capacidade de miniaturização (Kroo e Prinz, 2001), têm menor autonomia em relação aos demais modelos, porém com o avanço da tecnologia das baterias elétricas, a desvantagem vem se tornando menos expressiva (KROO e PRINZ, 2001);

d) Quad-rotor: topologia similar à do helicóptero (praticamente mesmas vantagens e desvantagens), com a diferença de utilizar mais motores e hélices. Maior simplicidade dos sistemas mecânicos. Apesar de ter surgido em 1920, ficou esquecido por muito tempo, pela dificuldade de estabilidade por um piloto, porém com o advento de novas tecnologias de MEMS, passou a ser um bom substituto do helicóptero, com menor custo e possui a habilidade de se aproximar mais dos objetos com segurança. Mesmo caso um dos rotores tenha algum problema, pode continuar estável (BOUABDALLAH; MURRIERI e SIEGWART, 2004).

As aplicações de VANT's são inúmeras, exemplos delas são: (PASTOR-MORENO, 2015):

a) DDD/D-cubo (Dangerous-Dirty-Dull), missões onde há perigo, sujas ou repetitivas;

b) Pesquisa ambiental remota;

c) Monitoração e certificação de poluição;

d) Gerenciamento de queimadas;

e) Segurança;

f) Monitoração de fronteira;

g) Oceanografia;

h) Agricultura e aplicações de pesca;

i) Aplicações de segurança;

j) Aplicações de comunicação;

k) Aplicações de monitoramento;

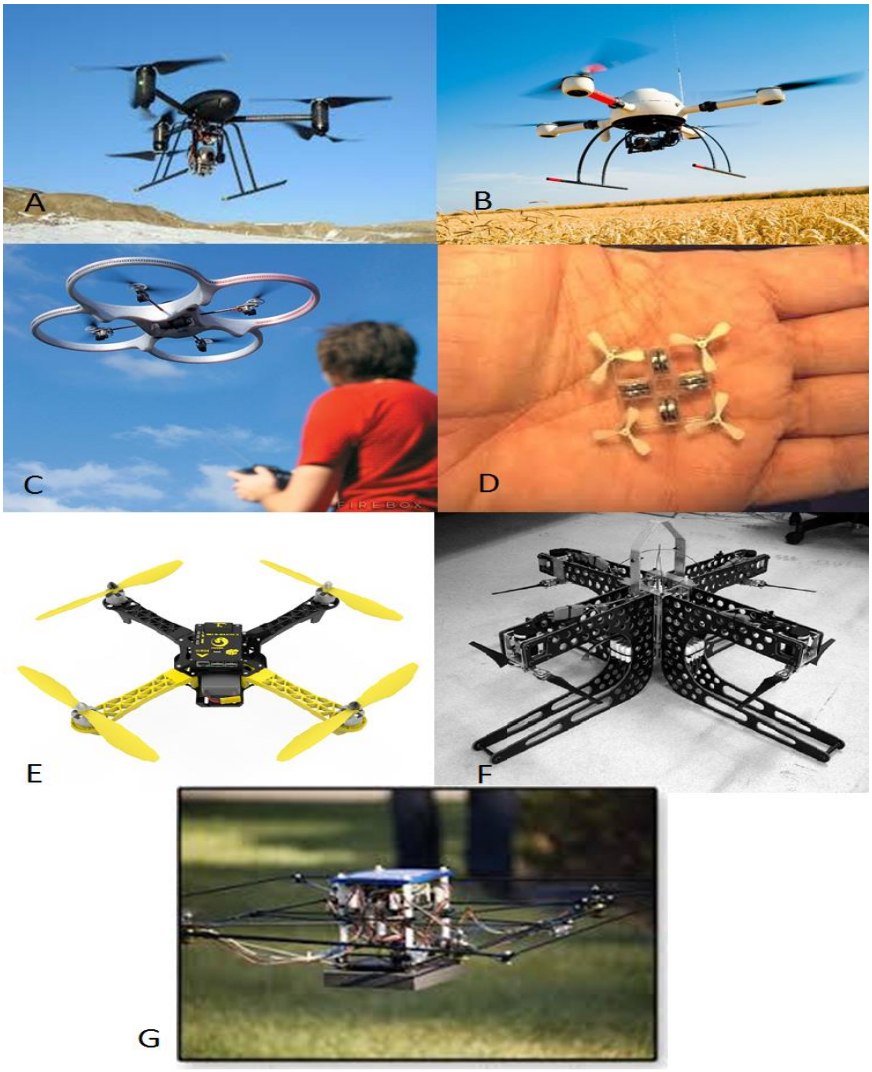
l) Redução de custos de operação.

Atualmente, depois de um extenso ciclo de uso exclusivo de áreas militares, os VANT's têm ganhado espaço em áreas civil e/ou comercial, tendo controle de forma remota, utilizando pilotos em estações de controle, controles embarcados (autônomos). No entanto há, no Brasil, uma precariedade de suporte de *hardware, software* e políticas de segurança para usar os VANT's com potencial máximo (FURTADO *et al.*, 2008).

Podem-se encontrar diversos modelos de quad-rotores disponíveis no mercado, por exemplo: Draganflyer (R C Toys, 2014), X-UFO (Silverlit, 2014), MD4-200 (Microdrones, 2014) e Erle-Copter Drone (ERLE-COPTER DRONE, 2016), inclusive o último com a mesma proposta que será estudada nest4e trabalho. Enquanto, alguns acadêmicos preferem desenvolver uma plataforma própria, observados em alguns exemplos como o mesicopter, X4-Flyer e STARMAC como pode–se ver na figura 1:

São apresentadas configurações diversas, por exemplo, estrutura de direção de rotação não-simétrica ou com rotores direcionais (BEJI, ZEMALACHE e MARREF, 2005; ABICHOU;

Figura 1: A- Draganflyer (R C Toys, 2014); B - X-ufo (Silverlit, 2014); C- MD4-200 (Microdrones, 2014); D- mesicopter (Microdrones, 2014); E- Erle-Copter Drone(2016); F- X4-Flyer (Silverlit, 2014); G- STARMAC (Silverlit, 2014).



FONTE: (AUTOR, 2016)

BEJI e ZEMALACHE, 2005), outros trabalhos apresentam configurações eficientes (Achtelik *et al.*, 2007) e configurações que visa aumento de estabilidade para aplicações específicas (ARGYLE *et al.*, 2014; BRISTEAU *et al.*, 2009; HAMAMOTO *et al.*, 2010; OSA; UCHIKADOe TANAKA, 2014).

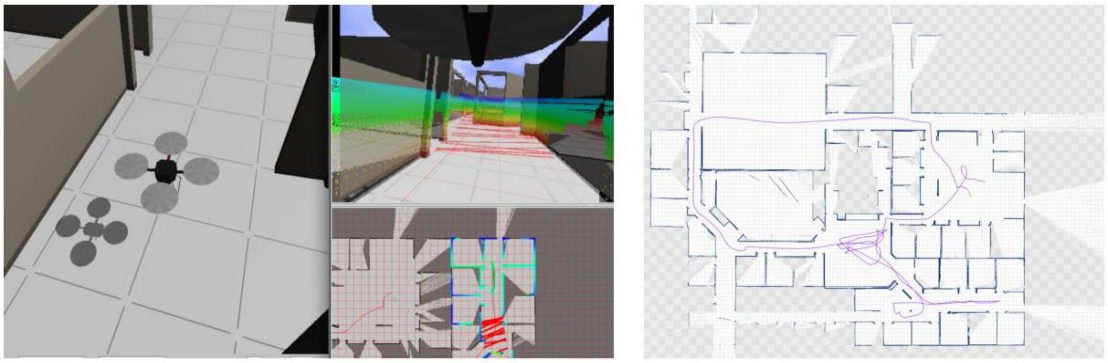
2.2 Quad-rotor ROS

Neste capítulo, foi utilizada como metodologia de pesquisa para elaboração do estado da arte de quad-rotores que utilizam ROS o algoritmo de pesquisa da plataforma Google Acadêmico. Foi pesquisada a expressão "*ROS quadrotor*" nesta plataforma e ordenado por relevância. Foram observados inicialmente aproximadamente 600 artigos científicos, destes artigos foram identificados os 19 artigos listados a seguir que tem compatibilidade com o tema apresentado neste trabalho mantendo a ordem de relevância apresentada pela plataforma.

MEYER *et al.* (2012) propõe um ambiente de simulação de quad-rotores utilizando ferramentas do ROS. Dessa forma, possibilitando um melhor aproveitamento no desenvolvimento de tecnologias que usem VANT's desse tipo. No projeto, foi utilizada a ferramenta Gazebo/ROS, que oferece um ambiente confiável para simulação, que considera as diversas interações físicas do objeto simulado e possibilita ao usuário modificar condições de simulação, um exemplo, os parâmetros do controlador.

Foi utilizado um modelo 3D, feito em um programa de computador chamado Blender (MEYER *et al.,* 2012), para a simulação geométrica e modelamento matemático para simulação cinemática e dinâmica. Para simulação dos sensores, foram criados programas adicionais independentes do Gazebo, que podem ser ativados ou desativados, conforme a necessidade. Foram realizados experimentos que consistiam de trajetórias e transições de velocidades, tanto no modelo simulado, como no quad-rotor real, onde foi verificado um resultado satisfatório, que repetia o sistema real no ambiente computacional. Pode-se observar uma imagem do sistema na figura 2.

Figura 2: Simulação SLAM (*Simultaneous Localization and Mapping* - Localização e mapeamento simultâneo) visual: (a) calibração da câmera do sistema na simulação. (b) Imagem obtida pelo quad-rotor voando no ambiente simulado



(a) (b)

FONTE: MEYER *et al.* (2012)

GRABE *et al.* (2013) apresentam um sistema que comanda diversos VANT's de maneira simultânea e com comunicação bilateral entre operação homem-máquina e máquina-máquina. O TeleKyb reúne uma série de grupos de programas que realizam determinadas operações, por exemplo:

a) *Human Interface*: que apresenta funcionalidades para a operação do VANT por uma pessoa;

b) TeleKyb Base: Oferece suporte para desenvolvedores de robôs;

c) TeleKyb Core: oferece uma biblioteca de controladores, estimadores e outras ferramentas;

d) ROS-Simulink Bridge: Oferece uma ferramenta para conexão do sistema com MATLAB.

Este projeto apresentou experimentos que utilizaram a ferramenta com bons resultados mostrando que ela é confiável e de boa integração com projetos diversos.

DUNKLEY *et al.* (2014) apresentam um nanoquad-rotor equipado com uma câmera e transmissão sem fio, comunicando com um computador no solo, o quad-rotor mais leve capaz de gerar imagens com baixo custo, robustez e fácil reconfiguração, que pode ser visto na figura 3.

Figura 3: Crazyflie nano-quad-rotor com câmera embarcada. 25g de peso total, 3,5 minutos de autonomia e tamanho de 9cm.

FONTE: DUNKLEY *et al.* (2014)

Quad-rotores com tamanho e peso reduzido têm se mostrado úteis para experimentos em locais onde o espaço e o risco de impacto, tanto para o robô, quanto para pessoas é evidente. O sistema é disponibilizado para reprodução dos experimentos e totalmente compartilhado com a comunidade para o uso livre.

Utilizou-se ROS no trabalho por ser uma plataforma que favorece o compartilhamento do que foi desenvolvido e dos resultados e é um ambiente de fácil evolução. A plataforma tem baixo custo, 7 minutos de autonomia de voo e 20 minutos de tempo de recarga e mostrou-se muito resistente a quedas e choques. Através do ROS, é possível fazer telemetria dos estados

do quad-rotor, com atraso de 8 ms, ferramenta de visão computacional confiável. Foram feitos testes de voo com e sem câmera, na plataforma com bons resultados em ambos.

ALEJO *et al.* (2014) apresentam uma proposta para prevenção de colisão com obstáculos para sistemas multi-VANT. O uso de grupos de robôs tem sido muito estudado, por apresentar vantagens em relação ao uso de apenas um robô em situações de vigilância, montagem de estruturas e detecção de incêndios.

A coordenação e a prevenção de colisões têm papel principal nestes tipos de aplicações, conforme o número de robôs aumenta, é necessário planejamento de trajetórias e correções em tempo real. O projeto apresenta um algoritmo para trajetória livre de colisão, que oferece uma solução rápida, mas não ideal e evolui para uma solução ótima. A técnica utilizada de evasão de colisões recíproca é utilizada para que cada componente auxilie na tarefa de desvio dos outros, evolui para a solução ótima da mesma técnica.

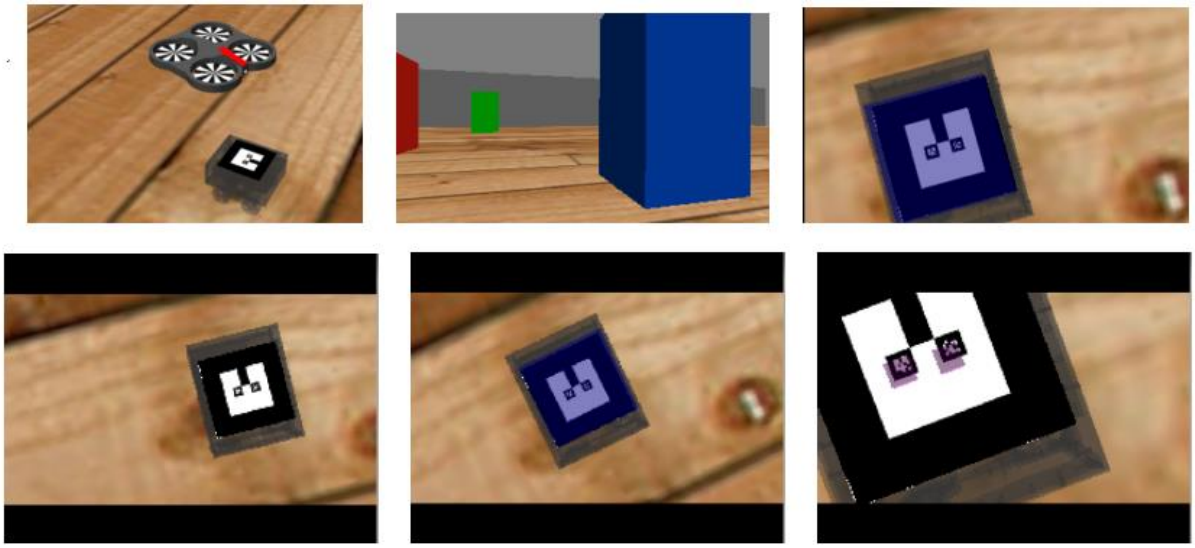
Os algoritmos foram implementados em ROS, para coordenação dos dados, controle dos VANT's e simulação. As diversas fontes de informação em conjunto geram um mapa de obstáculos e consideram a posição de cada membro do grupo para evitar colisões. O uso de ROS possibilita ao mesmo tempo boa base de simulação para desenvolvimento e a fácil transposição do ambiente de simulação para real. Essa plataforma teve resultados satisfatórios e mostrou que o custo computacional não evolui da mesma maneira que a quantidade de robôs do grupo, sendo assim, é possível o aumento do grupo significativamente e continuar utilizando o mesmo sistema para evitar colisões.

BENAVIDEZ e LAMBERT (2014) apresentam um controlador difuso que proporciona estabilidade no voo, pouso e controle de altura de um quad-rotor do tipo Parrot AR.Drone 2.0, utilizando imagens para gerar os sinais de controle, sendo gerenciados por meio de ROS. Devido ao uso intensivo das baterias, um problema de autonomia foi identificado em quad-

rotores. Como proposta de solução, foi desenvolvido um mecanismo que segue o VANT para oferecer uma superfície de pouso e troca de bateria rápida.

Foram utilizadas câmeras de baixo custo para o problema de seguir o VANT pela plataforma móvel de pouso, que gera mapas do ambiente e desvio de obstáculos, uma imagem esquemática pode ser observada na figura 4.

Figura 4: Imagens do simulador que utiliza câmera pra localizar imagem referência para pouso.



FONTE: BENAVIDEZ e LAMBERT (2014)

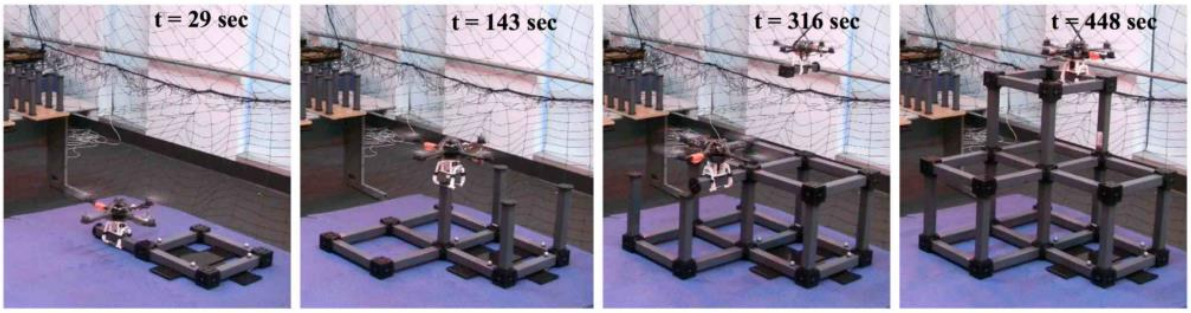
Os controladores foram desenvolvidos em ROS, utilizando um pacote de programas computacionais disponíveis na comunidade chamada QtFuzzyLite, ROS Gazebo com TUM AR.Drone Simulator que é baseado no pacote Hector_quadrotor (TUM_SIMULATOR, 2017) e ar_track_alvar para simulação 3D e identificação de marcações.

Os resultados obtidos nas simulações e experimentais foram satisfatórios, nos quais o sistema identificou as marcações na base móvel de pouso e as seguiu. Em alguns casos na simulação, quando as situações eram fora dos limites impostos inicialmente, houve alguns problemas na detecção das marcações, devidos a distância da base para o VANT e no

experimento o sonar apresentou algumas inconsistências, dando entradas de altura erradas devido a passagem de objetos próximos ao VANT.

LINDSEY, MELLINGER e KUMAR (2011) apresentam uma proposta de grupos de quad-rotores, sendo utilizados para montar estruturas cúbicas especiais de maneira autônoma, como pode ser visto na figura 5.

Figura 5: Fotos de diversos momentos da montagem de uma pirâmide realizada por 3 quad-rotores



FONTE: LINDSEY, MELLINGER e KUMAR (2011)

Aplicações nas quais robôs são usados para realizar montagens e construir estruturas têm crescido. Essas aplicações, normalmente, são altamente estruturadas e de alto custo. Esse paradigma é baseado em posições absolutas que dão a possibilidade de programações específicas para a realização da tarefa. Existem diversas aplicações de montagem, nas quais o ambiente não é tão estruturado, com posições absolutas que utilizam aproximadamente a mesma metodologia.

Montagens empregando equipamentos com hélices são compostas de tarefas de elevar e transportar objetos para lugares difíceis de serem alcançados, porém são operados por humanos nessa movimentação e esse trabalho propõe que VANT's podem executar essa empreitada melhor que humanos. Para o experimento, foram utilizados quad-rotores comerciais

chamados Hummingbird equipados com uma garra especialmente projetada para desenvolver a tarefa.

As barras das estruturas tinham ímãs para auxiliar no posicionamento de montagem e formato apropriado para facilitar o reconhecimento e montagem pelos VANT's. Para o sistema de posicionamento, foi utilizado o VICON (sistema de captura de vídeo em movimento), que forneceu uma precisão de milímetros em uma área de 6,7m x 4,4m x 4,0m.

Para o controle dos VANT's, foi utilizada a plataforma ROS em conjunto com o VICON e MATLAB, que assim podiam fornecer parâmetros para controle de estabilidade, posicionamento e trajetória, respectivamente. Das estruturas propostas para os testes, algumas obtiveram sucesso e outras não, devido a interpretação do robô em relação ao posicionamento das partes, porém os erros foram considerados aceitáveis aos padrões de tolerância da tarefa.

Quanto ao uso de mais VANT's, foi detectada uma relevante melhoria da eficiência até o número de 3 VANT's, permanecendo constante em números maiores. Do experimento, os resultados foram satisfatórios e foi sugerida uma melhoria em relação à autonomia dos VANT's que poderia ser aumentada se fosse integrada uma base para recarga das baterias durante a tarefa.

Quanto a análise da estrutura em ambientes de uso real, as juntas precisarão ser mais resistentes e, para resolver esse problema, é necessário que sejam desenvolvidas juntas inteligentes que possam geram uma maior resistência de fixação.

PESTANA e SANCHEZ-LOPEZ (2014) apresentam o resultado do desenvolvimento de um quad-rotor para uma competição. Foi buscada a concepção de baixo custo do protótipo e facilidade de projeto de sistemas multirrobóticos com desvio de obstáculos e reconhecimento de companheiros. Foi utilizada a plataforma comercial Parrot AR.Drone 2.0 comunicando via Wi-Fi com um computador no solo e ROS.

O grupo de robôs é autônomo, sem alto nível de inteligência embarcado, cada VANT consegue cumprir a missão de seguir a trajetória, desviando de obstáculos independentemente e compartilha sua posição para auxiliar o desvio de obstáculo entre o grupo. O controle de posição é feito com visão embarcada e externa, que se comunica para melhorar a precisão, considerando as dificuldades de definir posição em ambiente fechado com um VANT de baixo custo sem sensores especiais para esta função (GPS, LASER). O ROS gerencia as mensagens de posicionamento dos módulos e geração de trajetórias e mapa de obstáculos.

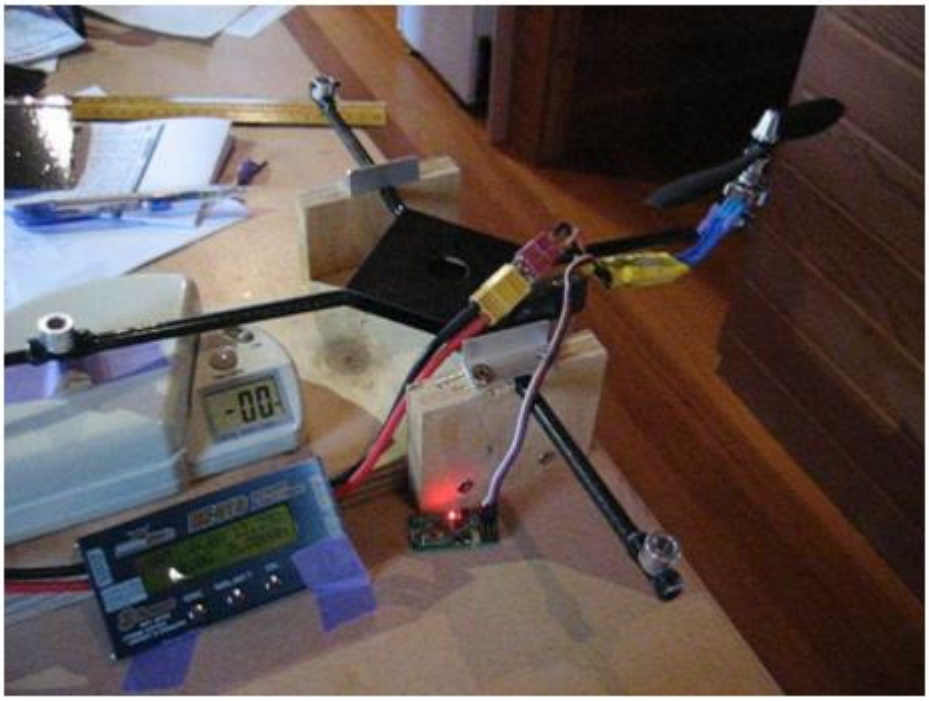
KOVAL, MANSLEY e LITTMAN (2016) apresentam uma proposta não usual de controle para quad-rotores, a aprendizagem por reforço. Através dessa técnica, é possível uma otimização do tempo de desenvolvimento de programação do robô.

A aprendizagem por reforço é um subcampo de aprendizado de máquinas que consiste das relações entre a interação e a resposta num ambiente estocástico. Nele, uma quantidade limitada de estados, ações e transições são definidas e as relações entre elas são observadas e recebem recompensas convenientemente.

Utilizando as ferramentas do ROS, pôde-se potencializar a evolução do robô e identificar melhor as relações que o trabalho pretendia discutir. Foi observado que utilizar ROS para padronizar o formato de transmissão de dados e interação com o quad-rotor, a disponibilidade de códigos já desenvolvidos e compartilhados se mostrou útil e um vetor facilitador do projeto. Utilizar o quad-rotor comercial AR.Drone se mostrou como uma forma barata e confiável e facilmente integrável ao ROS de discutir técnicas envolvidas com quad-rotores.

DAVIS, NIZETTE e YU (2013) apresentam o desenvolvimento de uma plataforma de quad-rotor para experimentos em grupos de VANT's com baixo custo, utilizando Arduíno e ROS, que pode ser visto na figura 6.

Figura 6: Proposta final de quad-rotor de baixo custo



FONTE: DAVIS; NIZETTE; YU (2013)

O objetivo inicial era construir um quad-rotor que tivesse custo inferior a USD$500,00 com autonomia de voo de 10 min e com menos de 500g. Foram obtidas essas características utilizando um quadro de tubos de fibra de carbono e duas placas de fibra de carbono ligadas por cola epóxi atingindo o peso de 43g e fácil manutenção. Foram utilizadas hélices 8 x 4,5 e ESC (*Electronic speed controller* – Controlador de velocidade eletrônico) comercial. Para a parte de aviônica foi utilizado APM2.0, plataforma de comunicação e aviônica, desenvolvida pela comunidade do projeto ArduPilot (DAVIS, NIZETTE e YU, 2013). Para comunicação, foi utilizado rádio frequência e um Arduíno ligado na porta USB do computador.

Em ROS, foi feito o sistema que gerencia posição e controle de velocidade e, em MATLAB, planejamento de trajetória. O ambiente de teste possui 12 câmeras. Foi observado

um resultado satisfatório em relação à qualidade da plataforma desenvolvida, gerar trajetórias e controle simplificado de posição com 3 VANT's ao mesmo tempo, foi identificada a necessidade de um espaço de testes maior para o experimento de trajetórias complexas.

ERMACORA *et al.* (2014) propõe um sistema que utilize diversos quad-rotores disponibilizando informações para usuários, por exemplo, vídeos aéreos utilizando plataforma de nuvem em cidades inteligentes. O monitoramento por câmeras se mostrou uma solução ineficiente para a redução de crimes, visto que os crimes acontecem em partes não monitoradas.

O objetivo é utilizar uma arquitetura de nuvem, baseada em ROS, para prevenção de crimes. Caso ocorra um incidente, o usuário solicita o serviço de emergência pela rede que indica a posição do evento e o sistema envia um VANT para o local, para acompanhá-lo ao evento e ser monitorado pelas autoridades.

O sistema tem duas formas de operação à plataforma do usuário comum e uma para a polícia, sendo a primeira para solicitar ajuda e a segunda monitorar tendo acesso a todas as informações disponíveis e controlar o sistema. O sistema pode utilizar diversos quad-rotores compatíveis com ROS. Foram realizados alguns testes, que identificaram dificuldades em relação ao sinal de internet para comunicação, porém indicaram uma boa capacidade de que com as melhorias de sinal de comunicação e aperfeiçoamento do sistema multi-VANT vão oferecer o serviço de maneira satisfatória.

CASTIBLANCO e RODRIGUEZ (2014) apresentam a proposta de utilizar quad-rotores de baixo custo para detecção de minas terrestres e objetos relacionados visualmente. Analisando especificamente o caso da Colômbia, lugar onde nos últimos anos houve 10.253 vítimas desde 1990, que tem características peculiares ao material explosivo tanto na construção e quanto na exposição de partes do artefato.

Foi possível propor uma solução de baixo custo para a detecção. O protótipo é baseado na plataforma ROS utilizando pacotes de programas com algoritmos de visão, as imagens são de uma câmera embarcada no VANT, montada na parte de baixo do equipamento, foi utilizada o Parrot AR.Drone 2.0 como VANT utilizado no projeto por ser adequado a proposta.

Foram desenvolvidos algoritmos de visão computacional que são gerenciados pelo ROS utilizando a biblioteca openCV para detectar as possíveis ameaças no solo. Pela característica do país, as bombas, normalmente, são feitas de latas de atum ou garrafas plásticas que ficam parcialmente enterradas. Focar nesses objetos para identificação é a proposta de solução.

Realizaram-se dois experimentos para a detecção em diferentes tipos de terrenos e com as minas parcialmente enterradas. No primeiro experimento, a porcentagem de detecção variou entre 73% dos casos nas piores condições e maior que 88% nas melhores condições. No segundo experimento, considerando a exposição entre 71% a 90% como visível, de 30% a 70% como parcialmente visível, sendo 30% de exposição o mínimo aceitável no experimento.

Nos experimentos, o resultado obtido foi a detecção de 87% dos objetos visíveis e nos casos com menos de 70% exposto houve muitas incertezas, classificando diversos objetos como possíveis minas e, na verdade, não o sendo, obtendo uma detecção de aproximadamente 80% dos casos. A proposta se mostrou viável, por se tratar de um equipamento de baixo custo, que pode prevenir acidentes e ser operado por qualquer pessoa com pouco ou nenhum conhecimento de robótica, após o devido treinamento.

OLIVARES-MENDEZ (2012) apresenta um projeto de quad-rotor que detecta obstáculos e se evade deles com sistema otimizado por entropia cruzada. A proposta deste artigo é discutir o uso de uma metodologia pouco explorada, para otimizar um controlador difuso e processamento de imagem.

Neste projeto, foi utilizada uma câmera embarcada direcionada para a parte frontal do quad-rotor, que tem as imagens processadas em um computador que se comunica com o quad-rotor usando comunicação 802.11n, que visa a identificação e perseguição de objetos. O controlador é do tipo difuso, que gera comandos de guinada para o VANT.

A metodologia de entropia cruzada é uma nova abordagem para otimização estocástica e simulação. Foi desenvolvida como um método eficiente para estimação de eventos com probabilidade rara. Foi utilizada a plataforma ROS-Gazebo através do pacote Starmac para desenvolvimento de um ambiente de simulação utilizando linguagem de programação C++ e a metodologia de entropia cruzada para a evasão de obstáculos, obtendo, após uma série de simulações, parâmetros para validação com um experimento real.

Foi utilizado um quad-rotor comercial Parrot AR.Drone, com 2 câmeras direcionadas para frente e conectadas a estação de processamento via *Wi-Fi*. O projeto obteve um controlador após 330 simulações e que, quando utilizado em quad-rotor real, apresentou resposta rápida e bom comportamento, com erro pequeno durante os testes, mesmo o VANT simulado não ser exatamente o mesmo utilizado no experimento.

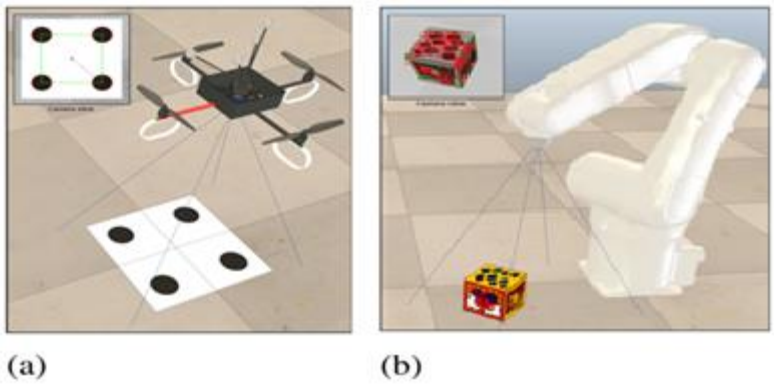
SPICA *et al.* (2013) apresenta uma proposta de quad-rotor de baixo custo, com tecnologia aberta altamente customizável, que possa ser utilizado para pesquisas com VANT's. Foram apresentadas soluções comerciais para cada componente do quad-rotor, respeitando os objetivos de menor custo possível, fácil aquisição dos componentes em qualquer lugar, boa qualidade de processamento embarcado e integrabilidade com a plataforma de desenvolvimento ROS.

Os programas computacionais e controladores foram todos desenvolvidos utilizando a plataforma ROS, empregando a ferramenta TeleKyb. Foi desenvolvido, também, um estimador de estado, que utilizou filtros para melhorar seus resultados. Foram realizados dois experimentos: seguir trajetória com auxílio de sensores externos e internos; e apenas com

equipamento embarcado. No primeiro, houve um problema na estimativa da massa do robô que requisitou um empuxo menor que o necessário para alçar voo, que, em determinado ponto, conseguiu alçar voo após ter aumentado internamente a estimativa de peso do VANT. No segundo, foi possível estimar utilizando apenas os sensores embarcados a trajetória realizada pelo VANT de maneira satisfatória. Sendo assim, o protótipo se mostrou uma plataforma viável e de baixo custo para pesquisas relacionadas a quad-rotores.

SPICA *et al.* (2014) apresentam a utilização da ferramenta V-REP do ROS, para simulação integrada com Matlab/Simulink como uma opção de fácil concepção de algoritmos de controle para plataformas robóticas. Os robôs podem ser vistos na figura 7.

Figura 7: Extraído da simulação no V-REP: (a) Quad-rotor. (b) Manipulador Adept Viper s850



FONTE: SPICA *et al.* (2014)

A ferramenta de MATLAB/Simulink foi considerada boa para os testes de algoritmos complexos e um bom ambiente de avaliação de resultados de progresso. Entretanto, essa ferramenta não tem um bom ambiente de simulação 3D desenvolvido para aplicações robóticas.

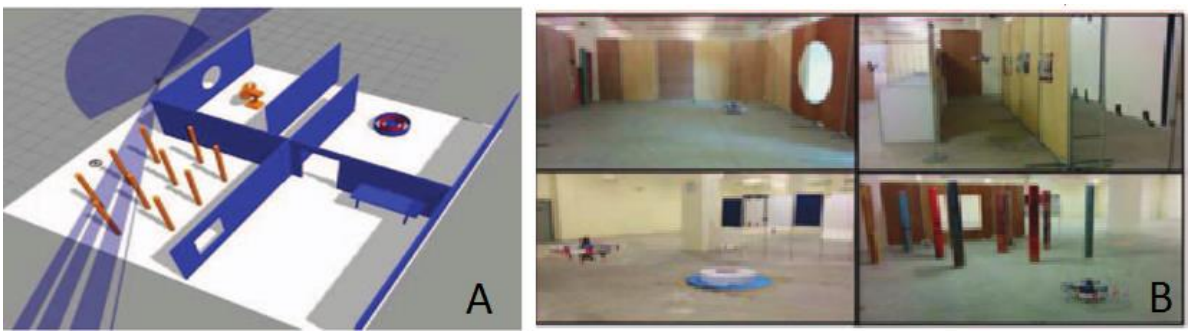
Partindo dessa necessidade, a ferramenta V-REP tem sido apresentada como uma boa solução, por ser gratuita e com grande comunidade colaborativa. Unindo a versatilidade dos dois ambientes com a facilidade de comunicação de ROS, foi possível realizar algumas

simulações. Primeiramente, o controle visual de um quad-rotor e o segundo controle visual de um manipulador industrial. Nos dois casos, as imagens foram geradas no V-REP, préprocessadas em ROS, a localização dos objetos e os estados dos robôs é passada para o MATLAB/Simulink, que aplica as leis de controle e envia para o sistema os sinais para os atuadores. Foi constatado o fácil desenvolvimento de sistemas robóticos, aproveitando essa proposta, e bons resultados para as simulações.

ZHANG *et al.* (2015) apresentam uma proposta de ambiente de simulação completo para quad-rotores utilizando ROS. O estudo de VANT's tem crescido exponencialmente ultimamente, logo, novas aplicações e áreas de estudo estão sendo desenvolvidos para eles. Um ambiente de simulação com grande fidelidade é uma boa ferramenta para desenvolvedores, pois diminui o risco de dano ao protótipo, possibilitando o teste de diversas configurações, flexibilidade para testes, facilidade de configuração entre outras possibilidades.

Diferentemente da maioria de programas computacionais que oferecem uma plataforma de desenvolvimento e simulação, o ROS além de ser gratuito e ter grande comunidade colaborativa, permite um acesso integral a todos os parâmetros do sistema, o que facilita não só o progresso, mas a integração de outros sistemas por outros pesquisadores ou grupos de pessoas interessadas na expansão do projeto. Foram realizadas simulações baseadas em um ambiente de testes real para verificar a qualidade do VANT. O ambiente foi importado para a ferramenta Gazebo, como pode ser observado na figura 8.

Figura 8: (a) Modelo do ambiente de simulação. (b) Imagem do ambiente real



FONTE: ZHANG *et al.* (2015)

O controle de estabilidade e trajetória é composto de sensores embarcados e câmeras externas que realizam combinações e, com ferramentas específicas de ROS, conseguem traçar mapas e estimar posição e trajetória. O resultado obtido com essa técnica foi que o sistema se mostrou satisfatório. O VANT teve um comportamento bom, podendo ter um controle de estabilidade, mapeamento, controle de posição e desvio de obstáculos. A simulação, também, mostrou que o controle desenvolvido não é ideal para espaços pequenos, *e. g.* corredores, porém o sistema dá uma tolerância de 0,5m de posição. A ferramenta foi considerada boa para teste de protótipos e num próximo estágio será integrada com Unity 3D para simulações ainda mais precisas.

MESTER (2015) apresenta a proposta de robótica de nuvem, que consiste em utilizar robôs por meio da rede de computadores distribuídas. Desenvolvido principalmente através das ferramentas do ROS, foi possível criar diversos aplicativos de internet, possibilitando várias arquiteturas e rotas disponíveis para a computação em nuvem. Este tipo de concepção de robôs, também, possibilita a integração de dispositivos móveis aos quad-rotores tais como: telefone celular; telas; e controles especiais. Pela grande evolução das taxas de transferências de dados ultimamente, a robótica de nuvem tem se tornado cada vez mais viável e amplia as possibilidades de desenvolvimento e utilização de robôs.

COCCHIONI *et al.* (2015) apresentam uma proposta para aumentar a autonomia de voo de VANT, através de plataformas de recarga de bateria, utilizando um sistema de visão computacional que possa detectar e seguir uma marcação como alvo de pouso e estimar a posição do VANT que pode ser observado na figura 9.

Figura 9: Quad-rotor no protótipo de base de recarga

FONTE: COCCHIONI et al. (2015)

Existem duas propostas de aumento de autonomia com paradas em plataformas: ativa e passiva. Na forma ativa, é necessário um sistema eletromecânico complexo para substituição da bateria durante uma pequena pausa. No sistema passivo, não há substituição da bateria e a pausa é aumentada, porém a plataforma é simplificada. Outra problemática é a capacidade de decolar e pousar de maneira segura e rápida.

Foi utilizado um sistema que segue uma marca e pousa numa plataforma passiva que recarrega as baterias do VANT. Utiliza o ambiente ROS para processamento de imagens em tempo real. A plataforma desenvolvida tinha uma aceitabilidade de erro de posicionamento pequena que corrigia a posição final para garantir a posição para carregamento de baterias. Os experimentos detectaram um pequeno erro do sistema de visão, mostrando grande robustez e a identificação de que a mudança de iluminação e presença de sombras diminuem a precisão do sistema e proporcionaram um aumento da autonomia do VANT.

MAS *et al.* (2015) apresentou a discussão sobre perseguição de trajetórias visualmente usando uma formação de VANT's. Usando-os, é possível utilizar essa formação para situações de vigilância e escolta, pois adiciona a possibilidade de detectar marcações visuais e, com isso, identificar objetos, estimar distâncias para o alvo e manter a formação inicial, mantendo redundâncias.

Foi utilizado um quad-rotor comercial Parrot AR.Drone no projeto e ROS como plataforma de desenvolvimento, o que possibilitou a fácil transição entre simulação e testes. A formação era de três quad-rotores, que poderia ser extrapolada para qualquer número, onde um serve de referência para o posicionamento dos outros. Ao mesmo tempo em que mantém a formação, também, seguem um objeto desejado através de imagens geradas pelos VANT's.

Foram feitos experimentos no ambiente de simulação de perseguição de um alvo e formação, foi utilizada uma forma de identificação de alvo inovadora (cilindros coloridos) e redundância em número para aumentar a precisão, obtendo resultados satisfatórios e com possibilidade de implementação futura em protótipo real.

OLIVARES-MENDEZ, KANNAN e HOLGER VOOS (2014) apresentam uma proposta de uso paralelo entre V-REP e ROS para desenvolvimento de controle de estabilidade, usando a técnica de controle difuso. ROS foi apresentado como uma solução de concepção que une baixo custo, fácil interação com protótipos reais, fácil uso, grande comunidade de desenvolvedores e grande quantidade de ferramentas disponíveis. V-REP é uma solução com custo de processamento menor que a disponível em ROS, tornando a simulação 3D menos onerosa ao computador, e tem uma grande disponibilidade de robôs, sensores, atuadores e estruturas disponíveis para simulação e tem boa integração com as demais ferramentas de ROS.

Neste trabalho, é descrito o processo de configuração do V-REP e ROS para elaboração de controladores difusos. Foram feitos experimentos de decolagem em plataforma estática e com movimentação, que mostraram que o controlador teve resultado satisfatório.

3 – MATERIAIS E MÉTODOS

Neste capítulo serão abordadas as ferramentas necessárias para o progresso do trabalho. O projeto é fundamentado na ferramenta gratuita de livre distribuição ROS, que basicamente é um *framework* que auxilia no desenvolvimento de robôs.

Algumas ferramentas integráveis ao ROS, também importantes no trabalho, serão utilizadas. Gazebo é um ambiente que possibilita simular robôs e comparar resultados com os de protótipos reais. Será utilizado o Hector_quadrotor como ponto de início da concepção do sistema operacional para os quad-rotores. Esse é um projeto que apresenta funcionalidades similares a este trabalho e que compactua com o *framework* do ROS de reutilização de código e, também, os objetivos deste trabalho visam complementar esse pacote adicionando novas funções: Telemetria e integração com VANT's reais.

Segundo Gil (2002), a metodologia deve ser definida relacionando o problema a ser estudado. Ela deve ser adequada ao problema de pesquisa, objeto da pesquisa, recursos financeiros, equipe humana, hipóteses levantadas que se almeja confirmar e ao tipo de informantes com que vai se entrar em contato.

Nesse trabalho, a metodologia de pesquisa é de natureza explicativa. Segundo Gil (2008), essa modalidade tem como objetivo primeiro identificar os fatores determinantes ou que contribuam para a ocorrência dos fenômenos estudados. É o tipo que mais aprofunda o conhecimento da realidade, geralmente associada a métodos experimentais. Um estilo subjetivo, mas de grande utilidade, por sua aplicação prática, assim, a pesquisa explicativa toma muitas vezes a forma de uma pesquisa aplicada (ou pesquisa experimental), ou pode também se utilizar de dados e informações de uma pesquisa *Ex-post facto*.

O universo a ser estudado é o dos robôs que têm sido extensivamente estudados por sua aplicabilidade para redução de custos e substituição de humanos em ambientes hostis.

Foi utilizado um modelo de quad-rotor para realizar o desenvolvimento deste trabalho. Este tema foi selecionado em virtude da recorrência do tema tanto na academia quanto em discussões de leigos, onde o objeto da pesquisa se mostra presente no cotidiano e como uma ferramenta poderosa.

A pesquisa consistirá em desenvolver um ambiente operacional em computador para robôs, ambiente de simulação virtual e testes. O trabalho foi realizado nas instalações do Instituto Federal de São Paulo - IFSP campus São Paulo, pela disponibilidade de equipamentos e espaço para testes.

O trabalho foi desenvolvido em três partes: adaptação do projeto Hector_quadrotor às características do projeto; simulação virtual; e integração com modelo real. Para a elaboração do projeto, foi necessário um computador com sistema operacional LINUX, ROS, Gazebo, rviz, LabVIEW sistema de comunicação sem fio e um protótipo de quad-rotor utilizando myRIO ou Arduino.

Os aplicativos foram elaborados em linguagem de programação C++/ROS. O sistema e seus subsistemas serão programados de maneira independente e serão integrados posteriormente ao ambiente principal. Essa estratégia permite o progresso descentralizado, permitindo que mais de um pesquisador trabalhe no projeto ao mesmo tempo. Serão utilizados programas de licença aberta, gratuita ou já disponíveis ao grupo de pesquisa pois foram adquiridas anteriormente e utilizadas em outros projetos não gerando nenhum novo custo ao projeto seguindo orientações padrão de instalação. Deseja-se, conforme apresentado na introdução, mostrar que a evolução descentralizada utilizando ROS otimiza o projeto e integração de subsistemas de quad-rotores.

Conforme apresentado no capítulo 2, a concepção de plataformas do tipo quad-rotor tem se mostrado um tema recorrente e utilizar um sistema operacional que auxilia na evolução de robôs se mostra uma ferramenta que pode catalisar este processo. Em diversos artigos, é exaltado o uso de ROS, e um ambiente de simulação para o desenvolvimento de quad-rotores (MEYER *et al.*, 2012; DUNKLEY *et al.*, 2014; ALEJO *et al.*, 2014). Este trabalho busca mostrar que a utilização de ROS, e simulação em ambiente virtual favorecem a concepção de plataformas do tipo quad-rotor, isto pode ser observado através:

a) Criação de ambiente operacional ROS utilizando pacotes disponíveis da comunidade, que oferecem ambiente de simulação e estruturação do fluxo de dados do sistema;

b) Comunicação dos diversos pacotes com o protótipo;

c) Simulações e testes do sistema;

Pode ser observado na figura 10 um esquema de como o sistema foi disposto. O Material utilizado foi:

a) Dois computadores: Um para executar o sistema operacional Linux e sistema ROS e outro com sistema operacional Windows para utilizar o LabVIEW na elaboração da integração entre o sistema ROS e o myRIO utilizado no protótipo;

b) LabVIEW: Ferramenta para integração do myRIO ao ROS;

c) ROS *Indigo Igloo*;

d) Gazebo 2;

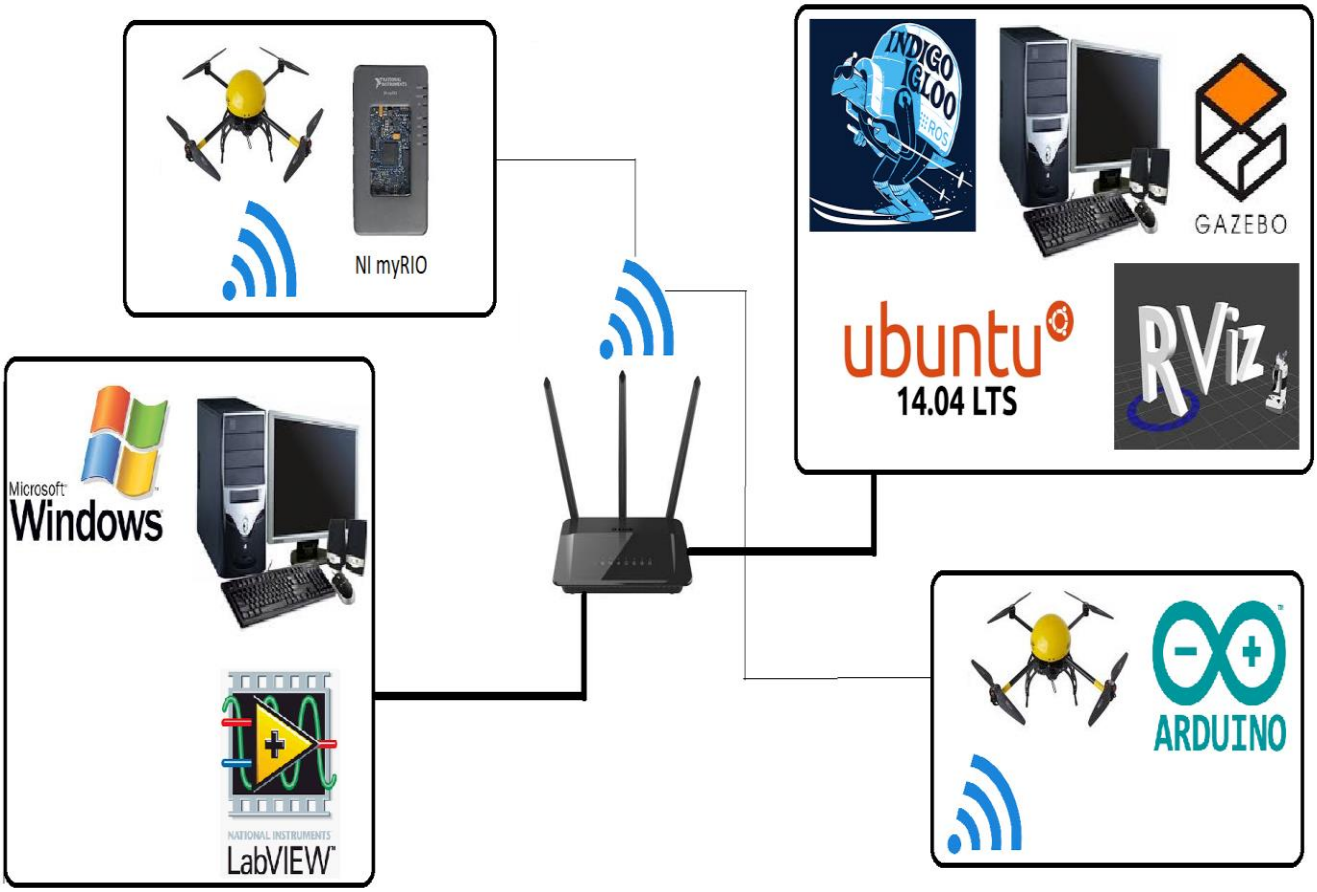
e) Rviz;

f) Pacotes ROS: Hector_quadrotor, “ROS for LabVIEW Software”, Rosserial e teleop_twist_keyboard;

g) Roteador: Utilizado para conexão entre o myRIO ou Arduino e o computador com Linux ou Windows;

h) Quad-rotor que utiliza myRIO ou Arduino;

Figura 10: Esquema do sistema desenvolvido neste trabalho.



FONTE: (AUTOR, 2017)

3.1 ROS

Segundo MARTINEZ e FERNÁNDEZ (2013), ROS é um *framework* de programas de computadores, que consistem em uma série de paradigmas e orientações para resolver uma das maiores dificuldades de sistemas robóticos complexos e a integração destes sistemas. Utiliza-

se de práticas e metodologias definidas para o desenvolvimento dos sistemas. Não é um programa executável, mas sim um conjunto de classes que são utilizadas para auxiliar a elaboração de programas de computador executáveis. Não diferente de um *framework* convencional, ROS tem por característica principal o uso de pequenas porções de código que possam funcionar em outros robôs apenas necessitando de pequenas alterações.

O ROS foi desenvolvido em 2007 pelo SAIL (*Stanford Artificial Intelligence Laboratory* – Laboratório de Inteligência Artificial de Stanford) para dar suporte aos projetos de inteligência artificial em robôs do laboratório. Em 2008, o desenvolvimento continuou na Willow Garage, instituto de robótica com mais de vinte instituições colaborando nos projetos.

Com o passar dos anos, a comunidade colaborativa foi aumentando, conforme pesquisadores e instituições começaram a desenvolver projetos em ROS adicionando protótipos e compartilhando porções de código. Com isso, diversas empresas começaram a adaptar seus produtos ao ROS. Na figura 11, podem-se ver alguns produtos destes citados. Essas plataformas passaram a ter publicações com muitos exemplos, simuladores e código que possibilitaram que mais desenvolvedores pudessem ter seus trabalhos potencializados.

O ROS possibilita uma série de vantagens no desenvolvimento de robôs. Uma maior abstração sobre as partes físicas do projeto, facilidade de controle e uso de componentes de baixo nível, reuso de funções comuns, comunicação entre os processos através de mensagens e a gestão através de pacotes. Ele é baseado numa topologia gráfica, na qual o processamento é feito em nós que podem tanto enviar como receber informação, tais como: sensores; atuadores; controle; estado; e planejamento de atitude. O sistema operacional preferencial é o Ubuntu Linux.

O repositório da comunidade também chamado de pacote (*.ros-pkg) é um conjunto de bibliotecas que contém diversas soluções compartilhadas por usuários. Por exemplo, para

navegação ou para visualização, a biblioteca rviz. Dentre as contribuições mais importantes, podem-se ressaltar as ferramentas de visualização, simulação e revisão.

Figura 11: Produtos desenvolvidos utilizando ROS

FONTE: MARTINEZ; FERNÁNDEZ (2013, p. 8 e 9)

O ROS incentiva a reutilização de códigos para agilizar o desenvolvimento de projetos sob a licença BSD (Berkeley Software Distribution – Distribuição de programas computacionais de Berkeley) que é de uso aberto e gratuito para uso comercial e pesquisa.

A arquitetura de ROS é dividida em 3 níveis (MARTINEZ e FERNÁNDEZ,2013):

a) Sistema de Arquivos;

b) Computação gráfica; e

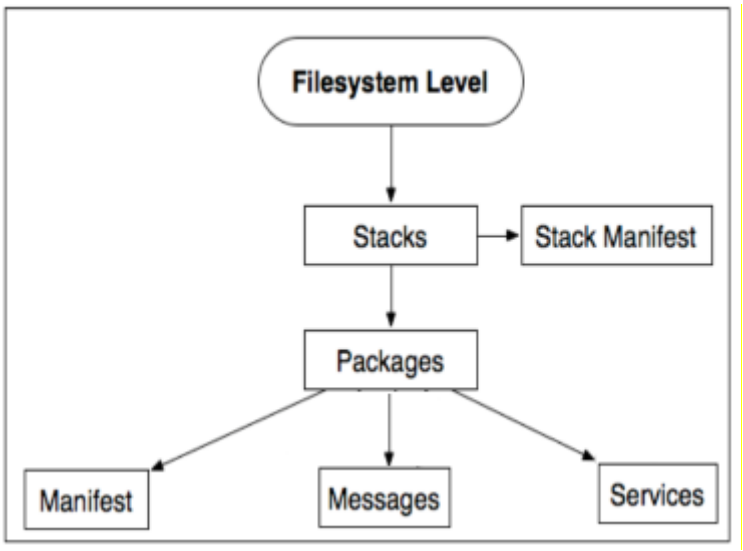
c) Comunidade colaborativa.

O Sistema de arquivos, o primeiro nível, é um grupo de conceitos que explicam como o ROS é formado internamente, estruturado em pastas e os arquivos essenciais para o funcionamento. O segundo nível, computação gráfica, é o local no qual a comunicação entre os processos e sistemas ocorre. Para o ROS interagir com os sistemas, é necessária uma configuração específica que é realizada nesse nível. No terceiro nível, comunidade colaborativa, os conceitos de compartilhar o conhecimento, algoritmos e código para qualquer desenvolvedor. Este nível é muito importante, pois possibilita o crescimento de ROS e a possibilidade de suporte vindo da comunidade.

3.1.1 Sistema de Arquivos

O sistema de arquivos pode ser visto esquematicamente na figura 12. Neste nível, o ROS é organizado de maneira parecida com o de um sistema operacional convencional: subdividido em pastas, sendo que cada pasta tem sua funcionalidade.

Figura 12: Esquema do nível sistema de arquivos.



FONTE: ADAPTADO DE MARTINEZ e FERNÁNDEZ (2013, p. 26)

a) Pacotes/*Packages*: é a estrutura e conteúdo mínimo necessário para desenvolver um programa em ROS. Ele deve conter nós (processos executáveis do ROS), arquivos de configuração entre outros itens.

b) Manifestos/*Manifests:* Oferecem informações sobre os pacotes, licença, dependências e indicadores de compilação. Essas informações são gerenciadas em um arquivo com nome manifests.xml.

c) Fila/*Stack*: A partir do momento em que vários pacotes com diversas funcionalidades são acumulados, é criada uma fila. No ROS, diversas filas para diversas funcionalidades são criadas, por exemplo, fila de navegação.

d) Fila de manifestos/*Stack manifests*: Da mesma forma que os manifestos, oferecem uma série de informações sobre as filas, licenças e dependências das filas utilizadas no projeto. É encontrada no arquivo stack.xml.

e) Mensagens (msg)/ *Message type*: As mensagens (msg) são as informações que os processos enviam e recebem entre eles. Em ROS existem vários tipos de mensagens. Elas são descritas e armazenadas no arquivo my_package/msg/MyMessageType.msg.

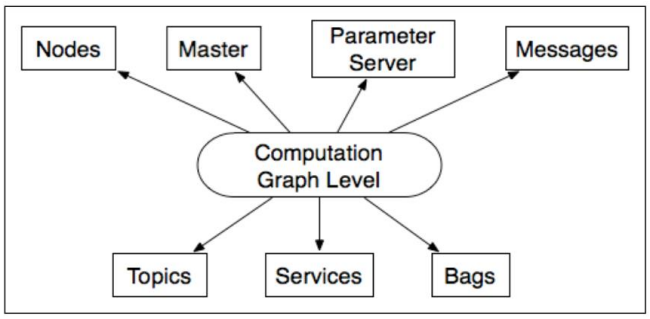
f) Serviços (srv)/ *Service types*: Os serviços (srv) são as informações estruturadas de requerimento e resposta em ROS. Elas são descritas e armazenadas no arquivo my_package/srv/MyServiceType.srv.

3.1.2 Arquitetura Computacional Gráfica

Nesse nível, é possível observar como os processos são conectados através de uma rede criada pelo ROS. Qualquer nó do sistema pode acessar essa rede de comunicação, interagir com

outro nó, visualizar as informações que são enviadas e transmitir dados para a rede. Um esquema desse nível pode ser visto na figura 13.

Figura 13: Esquema do nível Arquitetura Computacional Gráfica



FONTE: MARTINEZ e FERNÁNDEZ (2013, p. 32)

a) Nós/N*odes*: O nó é o lugar no qual as ações são realizadas, através dele é possível interagir com outros nós e conectar-se à rede de informações do ROS. Um sistema, normalmente, possui diversos nós para diversas funções, pois é melhor utilizar vários nós com funções únicas do que poucos nós com muitas funcionalidades. Os nós podem ser escritos em ROS utilizando a biblioteca *roscpp* ou *rospy*.

b) Mestre/M*aster*: O mestre gerencia o registro de nomes e relaciona os nós. Ele é necessário para a comunicação entre nós, serviços, mensagens e outros, porém não é necessário que eles estejam no mesmo computador no qual os nós são executados.

c) Servidor de Parâmetros/*Parameter Server*: O servidor de parâmetros gerencia em lugar central os parâmetros dos nós, possibilitando uma reconfiguração deles mesmo em funcionamento.

d) Mensagens/*Messages*: As mensagens são o meio pelo qual os nós se comunicam. Elas coletam as informações e enviam para outros nós. O ROS tem muitos tipos de

mensagens padronizadas, porém é possível criar tipos próprios para funções específicas.

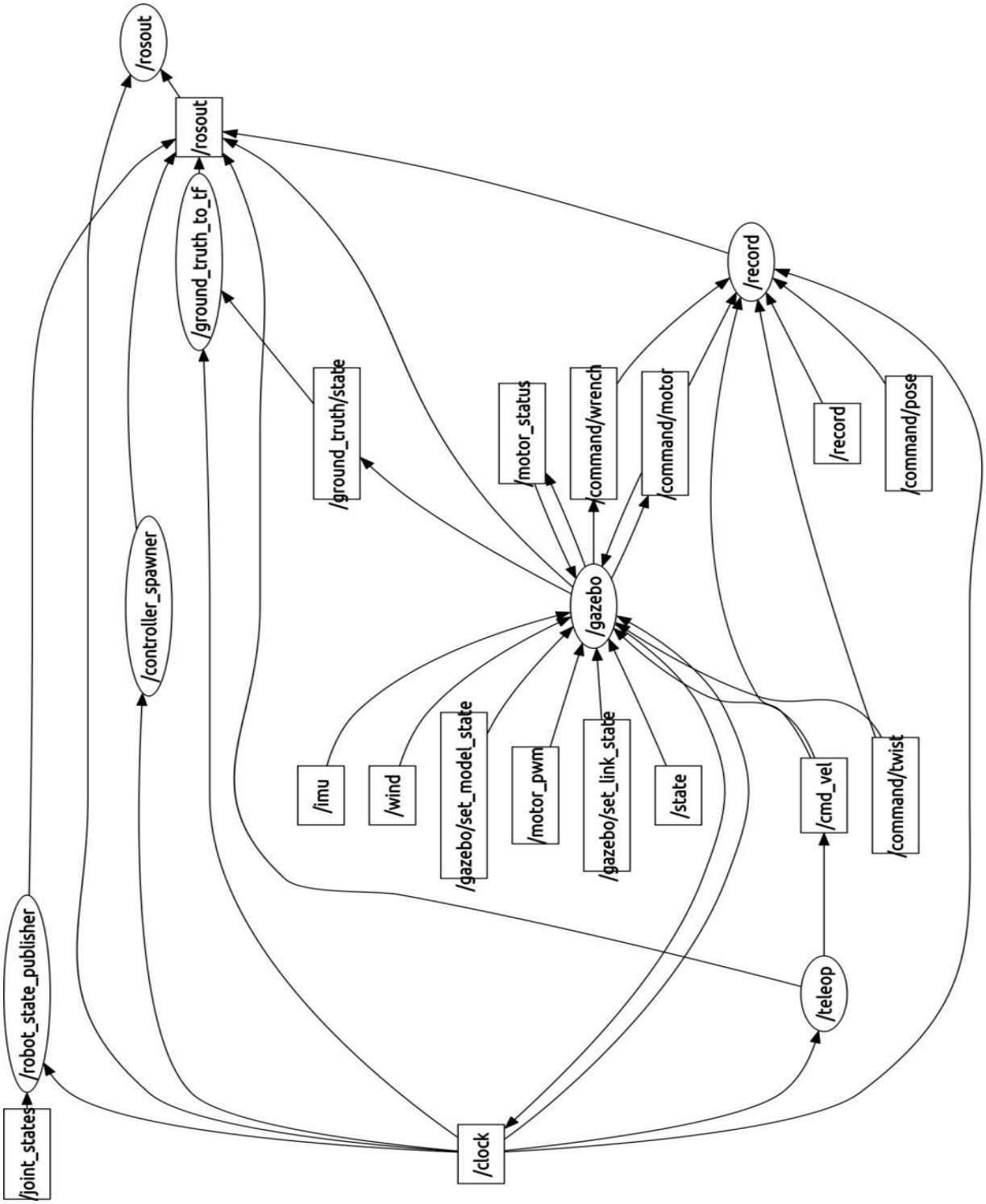
e) Tópicos/*Topics*: As mensagens precisam ter um nome para ser endereçadaS pela rede do ROS. Quando um nó está enviando dados, pode-se dizer que o nó está publicando um tópico. Outros nós podem subscrever a outros tópicos sem necessariamente publicar algo no tópico, e o nó que deveria publicar nesse tópico não precisa existir. Isso permite o desenvolvimento desarticulado, mas é importante que os nomes dos tópicos precisam ser únicos.

f) Serviços/*Services*: É possível enviar dados de diversas maneiras, porém, quando é necessária uma resposta vinda de um nó, não é possível fazer isso através de um tópico. Os serviços dão essa possibilidade de interagir com os nós. Os serviços, também, precisam ter um nome único. Quando um nó tem um serviço, todos os nós podem se comunicar com ele, graças ao cliente de bibliotecas de ROS.

g) Sacolas/Bags: As sacolas são o formato que possibilita salvar e executar as mensagens de dados de ROS. É um mecanismo de armazenamento de informações, por exemplo, no caso dos dados de um sensor, pode ser difícil o processo de coleta, mas é necessário para concepção e teste de algoritmos. Quando trabalhar com robôs complexos, provavelmente, será utilizado esse recurso.

Na figura 14, pode-se ver um exemplo desse nível de ROS. É uma representação gráfica de um robô real trabalhando em condições normais. No gráfico, podem-se observar nós, os tópicos e quem subscreve a quem, porém ele não mostra as mensagens, sacolas, servidor de parâmetros e serviços. A ferramenta utilizada para gerar esse gráfico é a rosgraph.

Figura 14: Representação gráfica do sistema utilizando a ferramenta rosgraph.



FONTE: (AUTOR, 2016)

3.1.3 Comunidade colaborativa

O nível Comunidade Colaborativa de ROS é um conceito que disponibiliza recursos para compartilhamento de conhecimento e códigos (MARTINEZ; FERNÁNDEZ, 2013).

Esses recursos são:

a) Distribuições/*Distributuions*: As distribuições de ROS são coleções de versões que se podem instalar. O comportamento delas é similar ao visto pelas distribuições de Linux. Isso facilita a instalação dos programas e mantém a consistência das versões do programa como um todo;

b) Repositórios/*Repositories*: O ROS é baseado numa rede de repositórios de códigos, na qual diversas instituições desenvolvem e publicam seus códigos de robôs;

c) ROS Wiki: É o fórum principal para documentação de informação sobre ROS. Qualquer pessoa pode se conectar e colaborar com documentação, correções, atualizações e tutoriais;

d) Listas de Discussão/*Mailing Lists*: É o canal de comunicação sobre atualizações de ROS e um fórum onde é possível compartilhar problemas e obter soluções de maneira colaborativa.

3.2 Rviz (ROS *visualization* – Visualizador de ROS)

Segundo MARTINEZ e FERNÁNDEZ (2013), para observar alguns tipos de dados oriundos de sensores específicos (câmeras estéreas, Lasers 3D e Kinect) que fornecem informações 3D, normalmente na forma de *point clouds* (organizados ou não). Logo, para observar esse tipo de dado, uma ferramenta apropriada é necessária. Em ROS, o rviz é utilizado

para esse tipo de função, entre outras de visualização. Neste aplicativo, é possível integrar bibliotecas de processamento de imagem (e.g. OpenGL) com uma representação de ambiente 3D com sensores modelados para ele. É possível atribuir referenciais para cada sensor e suas transformadas. O rviz é um ambiente de visualização, diferentemente do Gazebo, que é um ambiente de simulação. É possível adicionar uma série de configurações para visualização no rviz:

a) Displays: São objetos que representarão algo no ambiente 3D (e.g. Dados point cloud, estados de um robô, luzes e câmera). RVIZ

   (2016 - http://wiki.ros.org/rviz/UserGuide#Displays);

b) Barra de status: Local onde é possível ter acesso a informações sobre o tempo modos de operação de navegação no ambiente e selecionar e manipular os elementos apresentados (MARTINEZ e FERNÁNDEZ, 2013);

c) Propriedades de ferramentas: Onde podem ser visualizadas as ferramentas utilizadas no ambiente (*e.g.* movimentar câmera, selecionar objetos, adicionar objetivos de navegação e estimação de posição) RVIZ

   (2016 - http://wiki.ros.org/rviz/UserGuide#Displays);

d) Navegação: No Rviz, a navegação é feita por transformadas de referenciais. Sendo assim, é atribuído um referencial para cada componente do sistema, que pode ser observado e manipulado através das ferramentas utilizadas. Inclusive, sendo possível traçar trajetórias, os elementos podem ser exibidos, tornando uma ferramenta facilitadora para detecção de falhas e avaliação de problemas no sistema (MARTINEZ e FERNÁNDEZ, 2013).

Segundo Martinez e Fernández (2013), para utilização no Rviz, os dados de sensores precisam ter uma identificação do referencial que representam quando publicados. Desta forma, é possível relacionar, através de suas transformadas, as diferentes partes do sistema. Por exemplo, um acelerômetro, é necessário ter a transformação entre o referencial da base do objeto e do sensor para então realizar inferências do tipo estimar a velocidade de um robô ou sua pose. É fundamental que as mensagens contendo os dados tenham uma marcação de tempo para sincronização dos dados.

3.3 Gazebo

Segundo O'KANE (2014), Gazebo é uma ferramenta muito eficaz no desenvolvimento de robôs. Ela traz uma grande vantagem para desenvolvedores, visto que torna simples adicionar e remover componentes do projeto, tornando as simulações mais simples e fáceis de serem configuradas. Ele é um simulador de alta fidelidade para robôs, que tem total integração com ROS, tornando o progresso e simulação mais simples e rápido.

Segundo Gazebo (2016), a simulação de robôs é essencial para qualquer desenvolvedor, pois, com um bom simulador, é possível testar de maneira rápida e eficiente algoritmos, projeto de robôs e testes de desempenho em cenários diversos. Essa ferramenta tem uma grande biblioteca de robôs e cenários de simulação que representam, de maneira fidedigna, seus componentes. Possui simulação de cinemática e dinâmica, gráficos de alta qualidade, bom ambiente de programação gráfico e grande comunidade colaborativa. A principal fonte de informações sobre Gazebo é a página da internet oficial da ferramenta (www.gazebosim.org). É possível encontrar tutoriais de todos os níveis de aplicação que auxiliem no desenvolvimento de um simulador.

Essa ferramenta começou a ser desenvolvida em 2002 na Universidade do Sul da Califórnia, Dr. Andrew Howard e seu aluno Nate Koenig foram os criadores. O conceito base do projeto era desenvolver um simulador de alta fidelidade para a necessidade dos desenvolvedores de robôs em diversos ambientes e condições. Por ser frequentemente utilizados para situações em ambiente fechado, acabou recebendo o nome de Gazebo. Em 2009, foi integrado ao ROS e, a partir de 2011, o Willow Garage começou a financiar a evolução dessa ferramenta. Em 2013, foi usado como ambiente de uma competição virtual de robótica chamada DARPA *Robotics Challenge*. E continua em Desenvolvimento pela OSRF (*Open Source Robotics Foundation* – Fundação de programas computacionais livres de robótica) (GAZEBO, 2016).

3.3.1 Elementos Principais

Segundo Gazebo (2016), podem-se separar alguns elementos que compõem a ferramenta. Dessa forma, através de cada item, pode-se entender o funcionamento completo do programa.

3.3.1.1 Arquivo de ambiente (World Files)

Neste arquivo, todas as informações referentes ao ambiente simulado são encontradas, tais como: luzes; sensores; e objetos estáticos. Ele tem formato SDF (*Simulation Description Format* – formato de descrição de simulação) e extensão **.world*. No servidor do Gazebo (gzserver), este arquivo pode ser lido e gera o ambiente a ser simulado. São disponibilizados diversos exemplos de ambientes junto a ferramenta padrão.

3.3.1.2 Arquivo de modelo (*Model Files*)

Este arquivo, usando o mesmo formato SDF e mesma extensão *.*world*, porém, contém apenas um modelo único. O objetivo deste tipo de arquivo é facilitar a reutilização de modelos. Uma grande biblioteca de modelos pode ser encontrada no banco de dados da ferramenta, pode-se inserir o modelo e serão baixados os dados necessários no momento da execução.

3.3.1.3 Variáveis do ambiente (*Environment Variables*)

O Gazebo utiliza arquivo que armazena um certo número de variáveis e que faz a comunicação entre o servidor e cliente. A partir da versão 1.9.0 os valores são compilados automaticamente Isso significa que não é necessário escolher as variáveis. As variáveis são:

a) GAZEBO_MODEL_PATH: diretórios separados por dois pontos no qual Gazebo vai procurar os modelos;

b) GAZEBO_RESOURCE_PATH: diretórios separados por dois pontos no qual Gazebo vai procurar os arquivos do tipo world e de mídia;

c) GAZEBO_MASTER_URI: Especifica o IP e porta na qual o servidor será iniciado, dizendo ao cliente onde conectar-se;

d) GAZEBO_PLUGIN_PATH: diretórios separados por dois pontos no qual Gazebo vai procurar os *plugins* da biblioteca compartilhada no momento da execução;

e) GAZEBO_MODEL_DATABASE_URI: URI (*Uniform Resource Identifier* – Identificador de recursos uniforme) do modelo do banco de dados, no qual Gazebo vai baixar os modelos.

Caso haja necessidade de alterar configurações de comportamento do Gazebo, pode-se alterar o conjunto de variáveis.

3.3.1.4 Servidor Gazebo (*Gazebo Server*)

O servidor do Gazebo é o ambiente de funcionamento do sistema. Nele, os arquivos são analisados e simulados usando modelos físicos e de sensores.

3.3.1.5 Ambiente Gráfico Cliente (*Graphical Client*)

Esse ambiente conecta o servidor em funcionamento e visualiza os elementos. Essa ferramenta, também, permite alterar configurações de simulação durante a execução.

3.3.1.6 *Plugins*

São um mecanismo simples e conveniente que potencializa o Gazebo. São modelos pré-desenvolvidos que auxiliam na reutilização de componentes. Podem ser executados pelo servidor e pelo Ambiente Gráfico Cliente, facilitando a operação do usuário.

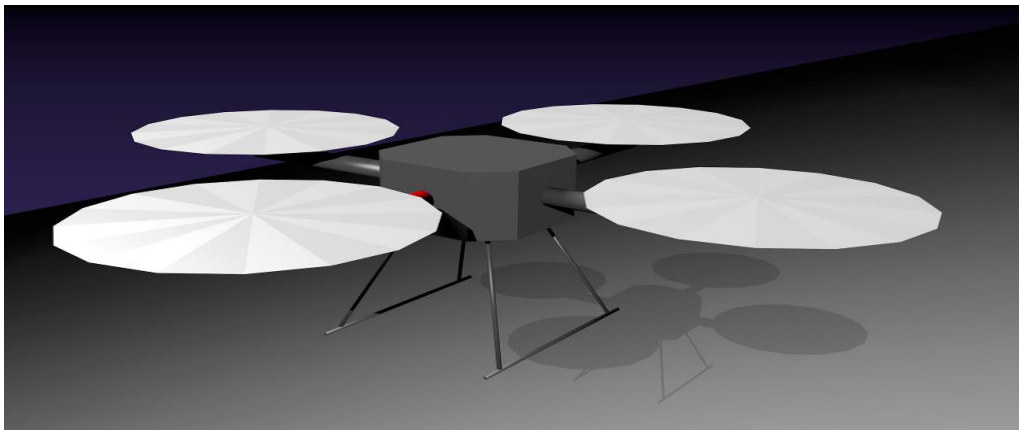
3.4 Hector_quadrotor

Segundo Wiki ROS (2016), Hector_quadrotor é um aglomerado de pacotes relacionados a modelagem, controle e simulação de quad-rotores VANT's. MEYER *et al.* (2012) propõe um ambiente de simulação que utiliza ROS e Gazebo e tem a intenção de acelerar o progresso de aplicações com quad-rotores. O projeto desenvolvido por eles é disponibilizado para o uso de maneira aberta. Pode-se separar o projeto em quatro partes que, juntas ou de maneira isolada, fornecem diversas funcionalidades para a concepção de quad-rotores. Sendo elas:

3.4.1 Hector_quadrotor_description

Esse pacote oferece uma amostra genérica de quad-rotor VANT. A geometria visual é baseada no modelo COLLADA, abreviação do inglês para *COLLAborative Design Activity.* É um padrão de exportação e importação de arquivos e oferece a possibilidade de modelagem 3D. Esse exemplar tem as características necessárias para ser utilizado no Gazebo e pode ser encontrado no pacote hector_quadrotor_gazebo, que disponibiliza o padrão URDF do VANT. Na figura 15 pode-se observar o arquétipo gerado por esse pacote.

Figura 15: Modelo COLLADA renderizado em Blender



FONTE: Hector Quadrotor description (2016)

Segundo ROS (WIKI, 2016), neste pacote, são oferecidos três modelos, quem têm características distintas e representam um mesmo VANT do tipo COLLADA. O primeiro, com nome quadrotor_base.urdf.xacro, é um modelo que tem os parâmetros básicos (formato e referencial inercial) (Quadrotor base Xacro, 2016).

O segundo, com o nome quadrotor.urdf.xacro, é um exemplar mais simples, que tem as características do quadrotor_base.urdf.xacro sem os sensores embarcados no primeiro exemplo. Essa amostra pode ser utilizada como um parâmetro de descrição de robô (Quadrotor base Xacro, 2016).

O último, com o nome quadrotor_hokuyo_utm30lx.urdf.xacro, oferece as características do primeiro padrão, com sensores embarcados e uma câmera apontada para a frente do tipo Hokuyo UTM-30LX LIDAR. Esse tipo pode ser utilizado como um parâmetro de descrição de robô (Quadrotor Hokuyo, 2016).

3.4.2 Hector_quadrotor_gazebo

Esse pacote disponibiliza um modelo de quad-rotor que pode ser utilizado no simulador Gazebo. Ele pode ser comandado por mensagens do tipo geometry_msgs/Twist, funções específicas do programa. Dentre as funcionalidades deste pacote, podem-se listar as seguintes:

a) Modelo 3d colorido do quad-rotor COLLADA (*.dae);

b) Arquivo URDF;

c) Publica a posição do VANT em relação ao solo e simula os dados de uma IMU (*inertial measurement unit* - Unidade de medição inercial);

d) Pode ser criado no ambiente de simulação de maneira visível e com interação com as ferramentas do Gazebo;

e) Controlador dedicado para o uso na simulação em Gazebo. O VANT pode ser controlado em Gazebo utilizando mensagens do tipo ‘geometry_msgs/Twist’ no tópico 'cmd_vel'. (*Geometry Msgs*, 2016).

Neste pacote, também, são oferecidos tutoriais de utilização e suporte para o desenvolvimento de aplicações, sendo um para uso em ambiente ao ar livre e outro utilizando técnicas de SLAM. Através dessa técnica, é possível gerar mapas do ambiente por onde o quadrotor sobrevoou e controle de posição no espaço (Hector Quadrotor Gazebo, 2016).

3.4.3 Hector_quadrotor_teleop

Este pacote oferece uma funcionalidade de poder operar o robô por meio de controle remoto, teclado e controle de videogame. Utilizando mensagens do tipo ‘joy/Joy’ e traduzindo-as para as mensagens do tipo ‘geometry_msgs/Twist’.

Através dessa funcionalidade, pode-se convenientemente simular um modo similar ao “modo 2” de helicópteros RF (rádio frequência) comuns, sendo sua programação similar ao controle de RF. O nó publica mensagens do tipo ‘geometry_msgs/Twist’ no tópico ‘/cmd_vel topic’, dessa forma, alterando os parâmetros desejados de velocidade linear e angular. O pacote, atualmente, tem dois tipos de controles compatíveis da marca Logitech e controles do videogame Xbox, porém, pode ser utilizado com qualquer outro e pode-se encontrar instruções disponíveis para a configuração de outros dispositivos. Também existe uma funcionalidade disponível, que é o uso de teclado comum de computador para controlar o VANT (Hector Quadrotor Teleop, 2016).

3.4.4 Hector_quadrotor_gazebo_plugins

Neste pacote, são disponibilizados diversos *plugins* para a adição de funções ao modelo. Utilizando esse tipo de técnica, o desenvolvimento se torna mais desarticulado e os códigos podem ser reutilizados com maior facilidade (Hector Quadrotor Gazebo Plugins, 2016).

Os principais *plug-ins* são:

a) Barômetro: Nesse *plug-in* é simulado um sensor de pressão que pode, de maneira indireta, medir altura de acordo com a ISA (International Standard Atmosphere - Padrão Internacional Atmosférico);

b) Controle simples: Nesse *plug-in*, um controlador genérico para quad-rotores é oferecido, permitindo o uso de voo por teleoperação (uso de controle remoto) para o voo no Gazebo. São utilizados controladores PID (Proporcional Integrativo Derivativo) para controlar velocidade e orientação do VANT. Ele calcula as forças necessárias e o torque que é aplicado ao corpo. Ele oferece um controle de estabilidade em funções não acrobáticas e por isso é chamado de controlador simples. Ele subscreve ao tópico 'cmd_vel', recebendo informações sobre ângulos e velocidades;

c) Aerodinâmica do Quad-rotor: Neste *plug-in*, são simulados fatores aerodinâmicos do quad-rotor. O sistema de propulsão, composto por motor e hélice, gera arrasto, empuxo e outros coeficientes relacionados a aerodinâmica são abordados por esse *plug-in*.

3.4.5 Parâmetros do ROS

Uma série de parâmetros necessários para a utilização do ROS com o quad-rotor são oferecidos por esse *plug-in*. Esse *plug-in* atualiza parâmetros do servidor durante a inicialização.

4.1 Instalação e configuração do ambiente com ROS, Gazebo e Rviz

A versão mais atual do sistema ROS é a de nome *Kinetic Kame*. Porém, essa é uma versão em desenvolvimento, que pode apresentar problemas, tornando-se uma versão não indicada para usuários que precisam de uma versão estável do sistema. A lista completa com todas as versões disponíveis (ROS DISTRIBUTIONS, 2017) recomenda a versão *Kinetic Kame,* porém, após o início da elaboração do trabalho, foi observada a incompatibilidade de alguns pacotes utilizados. Isto posto, foi escolhida a versão *Indigo Igloo* por ser mais estável e ter suporte até abril de 2019.

Para realizar a instalação do ROS *Indigo Igloo,* foi necessária a preparação prévia do ambiente para sua instalação. Decidiu-se instalar em um computador dedicado para o sistema a ser desenvolvido com as seguintes configurações:

a) Sistema Linux de 64 bits, Ubuntu 14.04.5 LTS (*Trusty Tahr*) indicado para versão do ROS utilizada;

b) Processador Intel® Core™ i5-4300M CPU @ 2.60GHz × 4;

c) 3,8 GB de memória RAM;

d) 342GB de espaço em disco;

e) Placa de vídeo Intel® Haswell Mobile;

f) Interface de rede.

A figura 16 mostra as configurações do Computador por uma captura da tela:

*Figura 16:*Captura de tela que mostra as configurações do computador dedicado ao desenvolvimento do projeto utilizando Linux Ubuntu 14.04.5 e ROS *Indigo Igloo.*

FONTE: (AUTOR, 2016)

A versão do sistema operacional Linux utilizada foi a Ubuntu 14.04.5LTS *Trust Tahr* por ser compatível com o ROS *Indigo Igloo*. Apesar de a versão do Ubuntu ter suporte apenas até agosto de 2016, essa se mostrou a mais apropriada para o sistema e continua tendo suporte pela comunidade Linux de maneira informal.

Durante o processo de instalação, foi escolhida a opção padrão. Após isso, prosseguiu-se para instalação do ROS *Indigo Igloo,* que foi realizada na opção completa *(Desktop-Full).* Nessa opção de instalação, obtém-se: A instalação de ROS, rqt (um *framework* para desenvolvimento de aplicações com interface gráfica baseada em QT), rviz (ferramenta ROS para visualização de modelos 3D), bibliotecas padrão de robótica, simulador Gazebo 2 (ambiente de simulação), bibliotecas de navegação e sensoriamento padrão.

4.2 Instalação dos pacotes utilizados

Conforme apresentado na revisão de literatura, esse projeto de sistema operacional para quad-rotores é amplamente utilizado por diversos desenvolvedores. Com um sistema já estabelecido, o desafio foi a adaptação para este projeto.

O sistema de controle proposto neste trabalho é embarcado, logo, alguns pacotes de Hector_quadrotor não serão utilizados, porém a maior parte das ferramentas será aproveitada, agilizando o desenvolvimento do sistema.

A simulação será realizada no ambiente Gazebo. O projeto Hector_quadrotor, também, já desenvolveu um simulador utilizando essa ferramenta. Para uma visualização do sistema, será utilizada a configuração do pacote Hector_quadrotor para Rviz. Desta forma, em futuras aplicações poderão ser utilizadas as diversas partes já desenvolvidas do pacote.

Outro pacote utilizado para a integração do protótipo é o "ROS for LabVIEW Software", desenvolvido pela universidade *Tufts* no *Mechanical Engineering Department and the Center for Engineering Education and Outreach*. Consiste em uma série de aplicações em LabVIEW para utilização com ROS, permitindo a comunicação e publicação e recebimento de mensagens entre os subsistemas.

Para integração com Arduíno, foi utilizado o pacote rosserial, que tem a função de gerenciar a comunicação via serial entre o microcontrolador e o sistema ROS. Desta forma, é possível a leitura de sensores, execução de ações nos periféricos através do comando do sistema ROS, comunicando através dos tópicos já anteriormente comentados.

Outros códigos foram gerados para integração do sistema e desenvolvidos pelo autor. Desta forma, podem-se integrar os diversos pacotes e operar o sistema como um todo.

4.2.1 Hector_quadrotor

A instalação desse pacote seguiu o tutorial disponível na página dos desenvolvedores (Hector_quadrotor Tutorial, 2014) com os comandos a seguir:

```
$ mkdir ~/hector_quadrotor_tutorial
$ cd ~/hector_quadrotor_tutorial
$ wstool init src https://raw.github.com/tu-darmstadt-ros-pkg/hector_quadrotor/indigo-devel/tutorials.rosinstall
$ catkin_make
$ source devel/setup.bash
```

Foram alterados alguns arquivos, conforme a conveniência, do tipo *launch e package,* utilizados de maneira a adicionar o controlador do tipo pose e adicionar pacotes utilizados em conjunto com o hector_quadrotor.

4.2.2 ROS for LabVIEW

Para utilização deste pacote, é necessária a instalação, no computador com o sistema operacional Linux e ROS, do pacote *rosbridge_suite,* que faz a comunicação entre programas que não são baseados em ROS com o sistema ROS. Utiliza-se a sequência de comandos a seguir:

```
$ sudo apt-get install ros-indigo-rosbridge-server
```

No computador com sistema operacional Windows e LabVIEW, é necessário seguir os procedimentos apresentados em (ROS for LabVIEW Software, 2016), que consistem em:

a) Instalar o toolkit LabVIEW myRIO;

b) Instalar o toolkit ROS for LabVIEW.

Após a instalação, foram desenvolvidos códigos que pudessem comunicar inicialmente o computador com LabVIEW ao sistema ROS no computador principal com Linux e, posteriormente, comunicação entre o myRIO e o sistema ROS.

Foi possível realizar a comunicação de publicadores e leitores simultaneamente entre o LabVIEW e o sistema ROS, porém com algumas limitações:

a) Uma vez interrompida a comunicação, o LabVIEW precisava ser totalmente reinicializado;

b) Havia um atraso entre o envio e recebimento da mensagem que variou entre 2ms a 40ms;

c) Quando não era possível conectar os sistemas, o computador precisava ser reinicializado;

d) A ferramenta ROS for LabVIEW, aparentemente, não oferece mais suporte, visto que, depois de enviar perguntar e esperar mais de 2 meses, nenhuma resposta foi enviada.

4.2.2.1 LabVIEW e myRIO

Segundo HALVORSEN (2016), LabVIEW é um acrônimo para ***Lab****oratory* ***V****irtual* ***I****nstrumentation* ***E****ngineering* ***W****orkbench* - bancada de trabalho do laboratório virtual de instrumentação e engenharia. É uma plataforma de desenvolvimento com linguagem de programação visual.

Essa linguagem recebe o nome de “G” e foi originalmente lançada pela Apple em 1986. LabVIEW é utilizado comumente para aquisição de sinais, controle instrumental e automação industrial em diversas plataformas, incluindo Windows, Linux e Mac. A extensão do arquivo de programação de blocos é “*.vi” que é uma abreviação para *Virtual Instrument* - Instrumento Virtual e são oferecidas *Add-Ons* (extensões) e *Toolkits* (bibliotecas) para diversas aplicações.

Segundo MyRIO (2014), a evolução das tecnologias segue a lei de Moore, podemos observar que sistemas computacionais complexos estão cada vez mais comuns no nosso meio. Para evoluções tecnológicas constantes, foi desenvolvida uma plataforma de desenvolvimento reconfigurável, o myRIO (*reconfigurable I/O*, RIO - entradas e saídas reconfiguráveis). O myRIO apresenta compatibilidade com o programa LabVIEW e se utiliza da programação nesta arquitetura de VI’s e linguagem visual.

As características do myRIO são:

a) Processador dual-core ARM Cortex-A9;

b) FPGA reconfigurável Xilinx Zynq-7010 Artix-7;

c) 40 entradas analógicas e digitais com SPI, I2C, UART e PWM;

d) 10 canais analógicos de entrada e 6 canais analógicos de saída;

e) LED’s disponíveis para utilização do usuário;

f) Wi-FI integrado;

g) Saída de áudio estéreo;

h) Botão disponível para utilização do usuário;

i) Acelerômetro de 3 eixos.

4.2.2.2 Conexão entre ROS master e myRIO

Nesta etapa, buscou-se comunicar a plataforma myRIO com ROS. Essa tarefa é desenvolvida utilizando o pacote "ROS for LabVIEW Software", que propõe formas para comunicação entre as partes envolvidas.

Foi utilizado o computador com o sistema operacional Windows 7 para desenvolver as VI's em Labview necessárias para configuração da comunicação entre o myRIO e o master do ROS, desta forma, possibilitando uma ponte de comunicação para os tópicos que transmitiam os dados do sistema. A comunicação é feita por cabo USB ou TCP/IP via WI-FI ou cabo, configurando os números de IP de cada parte no programa em LabVIEW, seguindo tutoriais do pacote (Clearpath tutorial, 2012).

4.2.3 Rosserial

Para utilização deste pacote, mais especificamente a parte "rosserial_arduino", utilizada para comunicar com o Arduino, é necessária a instalação, no computador, com o sistema operacional Linux e ROS, do pacote, que faz a comunicação entre o microcontrolador e o sistema ROS. É necessária a instalação do programa utilizado para escrever e compilar os códigos da plataforma, chamado Arduino IDE. Utiliza-se a sequência de comandos a seguir:

```
$ sudo apt-get install ros-indigo-rosserial-arduino

$sudo apt-get install ros-indigo-rosserial

$ cd home/raas/sketchbook/libraries

$ rm -rf ros_lib
```

$ rosrun rosserial_arduino make_libraries.py

Após essa série de comandos, é possível observar, no arduino IDE, exemplos de códigos ligados a biblioteca *ros_lib,* que podem ser utilizados para comunicação com o ROS.

Foi desenvolvido, então, um código que pudesse ler informações de uma IMU, calcular uma conversão dos dados do sensor para comandos de movimento do quad-rotor e recebimento de dados do sistema ROS. Esse programa, inicialmente, era composto de dois tópicos de publicação /imu com mensagem do tipo *sensor_msgs/Imu* e /cmd_vel2 com mensagem do tipo *geometry_msgs/Twist*. E um tópico de leitura /cmd_vel com mensagem do tipo *geometry_msgs/Twist*.

No tópico publicado pelo Arduino /IMU, os dados do sensor inercial foram organizados usando o padrão de mensagem, possibilitando a manipulação dos dados pelo sistema, caso conveniente.

No tópico publicado pelo arduino /cmd_vel2, os dados oriundos da IMU eram utilizados para uma estimação de parâmetros utilizados para comandar o movimento do quad-rotor, sendo velocidade angular e velocidade linear nos três eixos x, y e z. Foram utilizados para velocidade angular os valores diretos lidos pelo sensor e para estimação da velocidade linear as seguintes equação:

$$V_{lin}* = \left[AccF_{lin}* + \frac{(AccF_{lin}* - Acci_{lin}*)}{2}\right] \times (t_f - t_i)$$

$V_{lin}*$: Velocidade linear no eixo x, y ou z;

$AccF_{lin}*$: Aceleração linear final no eixo x, y ou z;

$Acci_{lin}*$: Aceleração linear inicial no eixo x, y ou z;

t_f: Tempo final;

t_i: Tempo inicial.

No tópico lido pelo Arduino /cmd_vel, os dados referentes aos diversos modos de envio de comandos de trajetória eram lidos, possibilitando o recebimento de comandos de movimentação do sistema ROS para a plataforma.

Foram utilizados dois microcontroladores Arduino para a implementação dos códigos desenvolvidos anteriormente, o primeiro chamado *duemilanove* e o segundo *due*. O primeiro dispositivo, com configuração que pode ser observada na tabela 1, apresentou a dificuldade em relação ao tamanho das mensagens, que poderiam ser enviadas via porta serial e apenas uma porta de comunicação serial.

Não foi possível o envio de mensagens do tipo sensor_msgs/Imu, devido ao tamanho do buffer. Logo, por não ser fundamental para o sistema, foi enviado apenas o tópico /cmd_vel2, que indiretamente enviava as informações do sensor para o sistema ROS. Foi possível receber os dados do tópico /cmd_vel concomitantemente, porém, foi identificado um atraso no envio das mensagens, gerando distorções entre os comandos e o recebimento das mensagens.

Durante o teste, no qual uma série de comandos de posicionamento era enviado para o microcontrolador e este publicava a mesma mensagem que recebeu, foi detectado um atraso entre o recebimento e o envio de mais de 50s ao final da rotina, indicando uma inadequação da plataforma para o projeto.

Tabela 1: configurações da placa arduino Duemilanove.

Microcontroller	ATmega368
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB of which 2 KB used by bootloader
SRAM	2 KB
EEPROM	1 KB
Clock Speed	16 MHz

FONTE: (https://www.arduino.cc/en/Main/ArduinoBoardDuemilanove)

Foi utilizada uma segunda plataforma Arduino do tipo Due que tinha configurações que podem ser observadas na tabela 2.

Para utilizar essa plataforma, foram consultados tutoriais disponibilizados pela comunidade que a utiliza. Porém, não foi obtido sucesso na utilização, devido à incompatibilidade do sistema e falta de suporte para a plataforma. Era esperado que, com configurações mais potentes seria possível a utilização do código inicial integralmente e redução de possíveis distorções geradas pela comunicação. Contudo, não foi possível a

implementação do sistema, tanto utilizando o sistema operacional Linux quanto em sistema Windows.

Tabela 2: Configurações da placa Arduino Due.

Microcontroller	AT91SAM3X8E
Operating Voltage	3.3V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-16V
Digital I/O Pins	54 (of which 12 provide PWM output)
Analog Input Pins	12
Analog Output Pins	2 (DAC)
Total DC Output Current on all I/O lines	130 mA
DC Current for 3.3V Pin	800 mA
DC Current for 5V Pin	800 mA
Flash Memory	512 KB all available for the user applications
SRAM	96 KB (two banks: 64KB and 32KB)
Clock Speed	84 MHz
Length	101.52 mm
Width	53.3 mm
Weight	36 g

FONTE: (https://www.arduino.cc/en/Main/ArduinoBoardDue)

As possíveis soluções para esse problema de comunicação, que não puderam ser exploradas devido ao tempo disponível são:

a) Elaboração de um código próprio que tratasse os dados de maneira específica para a plataforma utilizada. Desta forma, a mensagem seria adaptada para o tamanho de *buffer* de cada equipamento e teria um novo tratamento dos dados pelo sistema após o recebimento da mensagem;

b) Utilização de outras plataformas Arduino com características diferentes, que, possivelmente, não apresentariam o mesmo problema do tamanho do *buffer* e, após testes, seria verificado se houve atraso entre a transmissão e o recebimento das mensagens satisfatório;

c) Utilização de outras plataformas de microcontroladores disponíveis no mercado e desenvolvimento de códigos próprios para conexão entre estas e o sistema ROS.

4.2.4 Códigos desenvolvidos pelo autor

Para desenvolver os códigos necessários para integração dos sistemas a sequência de comandos para criar uma estrutura de diretórios e arquivos básicos para uma área de trabalho ROS, listados a seguir foi realizada:

```
$ mkdir -p ~/quad_ws/src
$ cd ~/quad_ws/src
$ catkin_init_workspace
$ cd ~/quad_ws/
$ catkin_make
$ source ~/quad_ws/devel/setup.bash
```

Para criar um pacote que foi utilizado no sistema foi utilizada a sequência de códigos a seguir:

```
$ mkdir -p ~/quad_ws/src
$ catkin_create_pkg quad std_msgs rospy roscpp geometry_msgs sensor_msgs tf
    nav_msgs
$ cd ~/quad_ws/
$ catkin_make
$ source ~/quad_ws/devel/setup.bash
```

5 RESULTADOS

Foi utilizado o pacote hector_quadrotor como base para o desenvolvimento do sistema utilizado nesse trabalho. Durante o processo de desenvolvimento, foram identificados os tópicos nos quais poderia ser comandado o quad-rotor simulado pelo pacote hector_quadrotor:

- /cmd_vel: Tópico que tem tipo *geometry_msgs/Twister*. Através dele pode se enviar informações referentes a velocidade angular e linear do quad-rotor. Essa sequência de comandos é similar ao feito por um controle de rádio frequência comumente utilizado em quad-rotores genéricos. Através desse tópico, é possível, também, a teleoperação utilizando o teclado ou controle de *Xbox* do quad-rotor tanto simulado quanto real. Um exemplo da mensagem pode ser observado a seguir:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear:  {x: 1.0,y: 1.0, z: 3.0}, angular: {x: 0.1,y: 0.1,z: 0.1}}'
```

Apesar de existir o tópico */command/twist* que seria a opção de comando mais óbvia com configuração idêntica ao */cmd_vel* o sistema não responde como esperado, sendo necessário o uso do segundo no lugar do primeiro.

Para tanto, incialmente, foram desenvolvidos códigos que publicassem nesses tópicos e pudesse ser observada a resposta. Para alimentar as informações desse tópico, foi identificada a leitura do sensor inercial do quad-rotor e conversão de seus dados para o formato dos tópicos, de maneira a fazer uma estimativa de posição e estado do robô para representação no sistema simulado. Foram utilizados algoritmos para converter o sinal de orientação, velocidade angular, aceleração linear e altura no formato desejado.

Devido às dificuldades encontradas para a comunicação entre as plataformas microcontroladoras e o sistema ROS não foi possível realizar um teste que obtivesse os dados dos sensores e os enviasse ao sistema ROS, após o tratamento do sinal com filtro apropriado, de forma a visualizar a plataforma real em ambiente virtual.

Então, foi desenvolvido na plataforma myRIO um código que gere um publicador nos tópicos anteriormente descritos /cmd_vel, /command/pose e com os dados da IMU utilizados pela plataforma no programa LabVIEW. O código foi feito seguindo tutoriais da comunidade (myRIO Publisher, 2015).

Foi desenvolvido na plataforma myRIO um código que gere um leitor do tópico anteriormente descritos /cmd_vel de maneira que é possível fazer a operação do quad-rotor pelo computador com sistema ROS no programa LabVIEW o código seguindo tutoriais da comunidade (myRIO Subscriber, 2015).

Foi elaborado um código similar ao realizado para os testes utilizando o computador com LabVIEW para embarcar no myRIO e comunicar ao sistema ROS, porém não foi obtido nenhum sucesso e foram observadas as seguintes dificuldades:

a) Não foi possível seguir os tutoriais, pois eles não funcionavam como o apresentado;

b) Aparentemente, a falta de suporte a ferramenta impossibilitou a busca por soluções para os problemas com os tutoriais apresentados;

c) Problemas de comunicação entre o myRIO e o computador com LabVIEW através da WI-FI, que não foi possível ser solucionado, forçando a comunicação somente através do cabo USB;

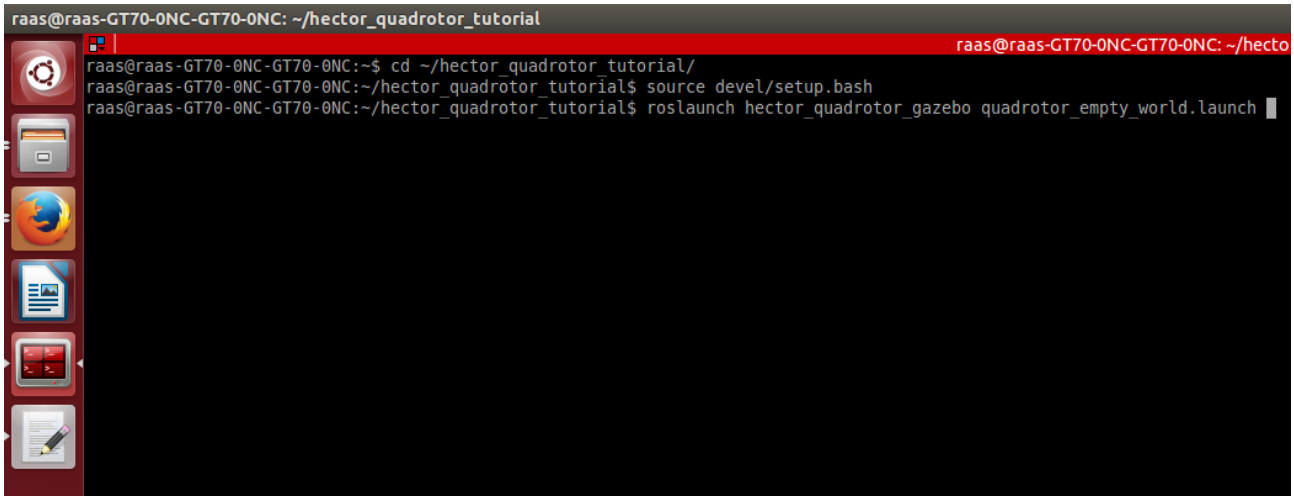
d) Devido a falta de mais tempo, não foi possível explorar outras possíveis maneiras de utilizar a ferramenta seguindo sugestões de colegas com experiência junto a plataforma de desenvolvimento myRIO, sendo esse um possível trabalho futuro.

Com a impossibilidade de comunicação entre o myRIO e ROS, não foi possível, durante esse trabalho, integrar um quad-rotor que utilizava a plataforma myRIO ao sistema ROS.

5.1 Testes em ambiente virtual

O sistema que foi desenvolvido durante esse trabalho pode ser acessado utilizando a sequência de linhas de comando abaixo, que inicializam e dão acesso às funcionalidades desenvolvidas, a figura 17, 18 e 19 mostra o sistema sendo iniciado no computador.

Figura 17: Tela de inicialização do sistema no terminal.



FONTE: (AUTOR, 2017)

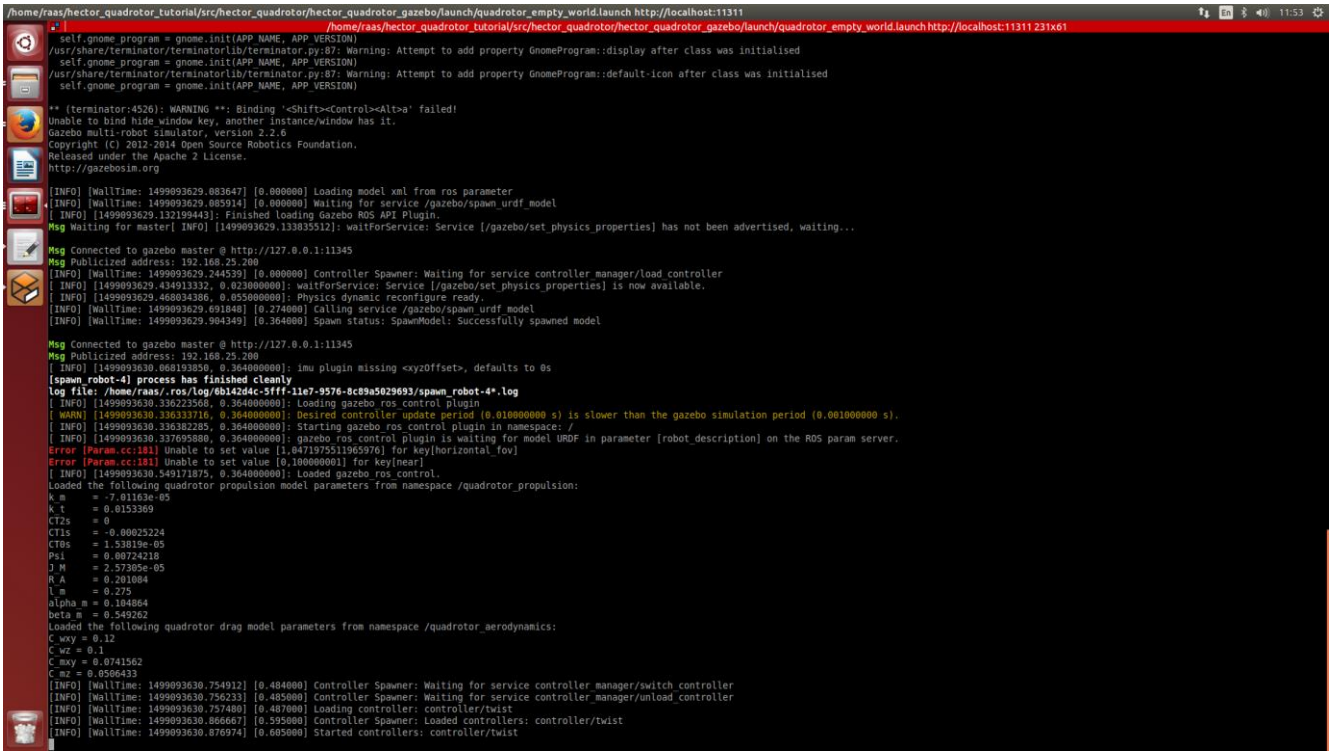
$ cd ~/hector_quadrotor_tutorial/

$ source devel/setup.bash

$ roslaunch hector_quadrotor_gazebo quadrotor_empty_world.launch

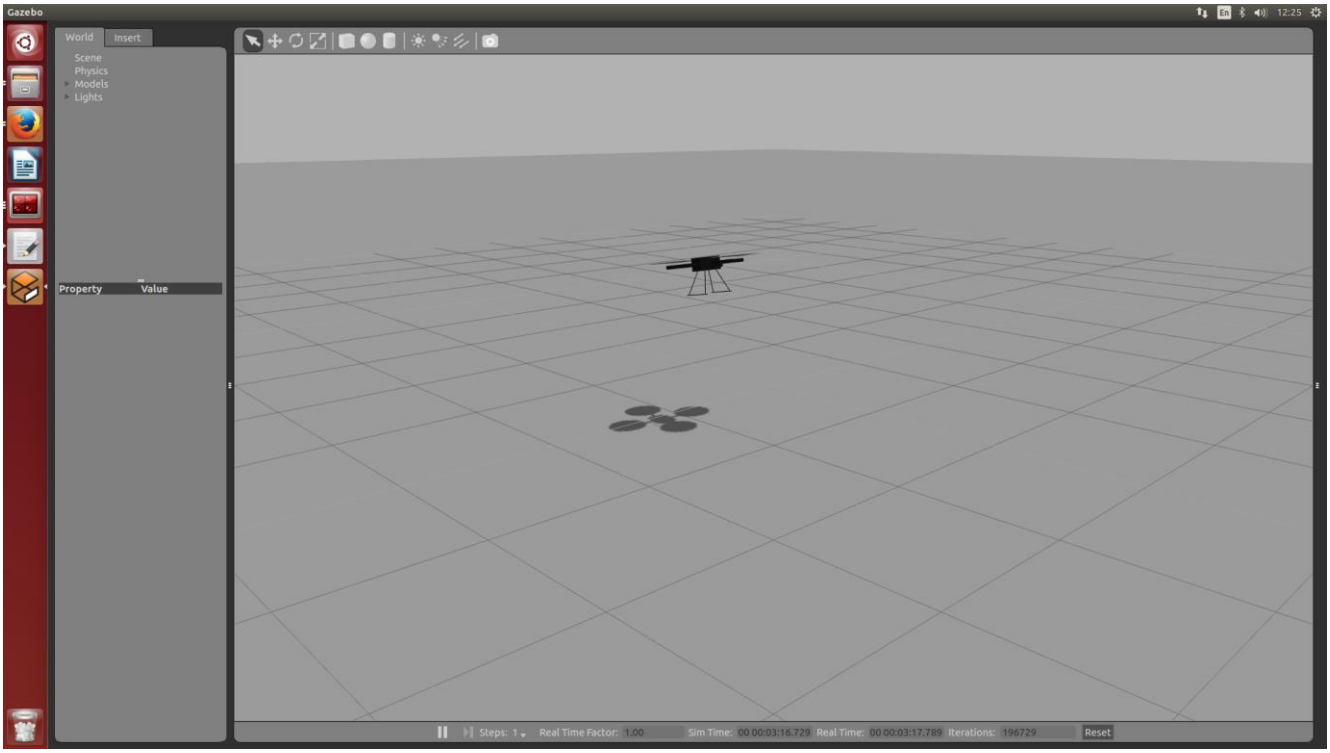
Foi alterado o arquivo de inicialização, adicionando funcionalidades diferentes do padrão (teleoperação, ambiente de simulação simplificado e reprodutor de trajetórias pré-definidas), o código pode ser observado no apêndice A.

Figura 19: Tela após inicialização do sistema no terminal.



FONTE: (AUTOR, 2017)

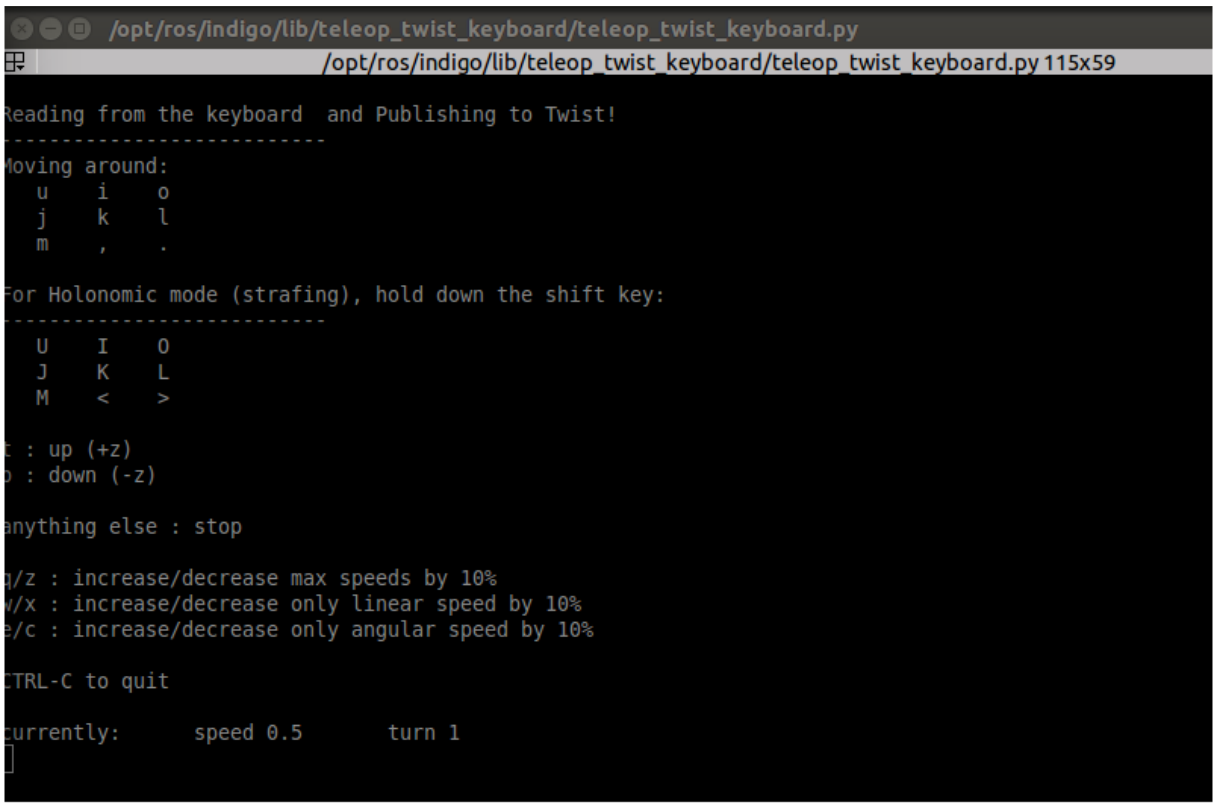
Figura 18: Tela do simulador de voo do quad-rotor.



FONTE: (AUTOR, 2017)

Foram utilizados pacotes auxiliares para realizar a teleoperação do quad-rotor em ambiente virtual. Foi utilizado o pacote *teleop_twist_keyboard,* que permitiu a operação do quad-rotor pelo teclado do computador e se mostrou uma ferramenta simples e com bons resultados para observação e identificação dos parâmetros desejados do sistema, que pode ser observado na figura 20.

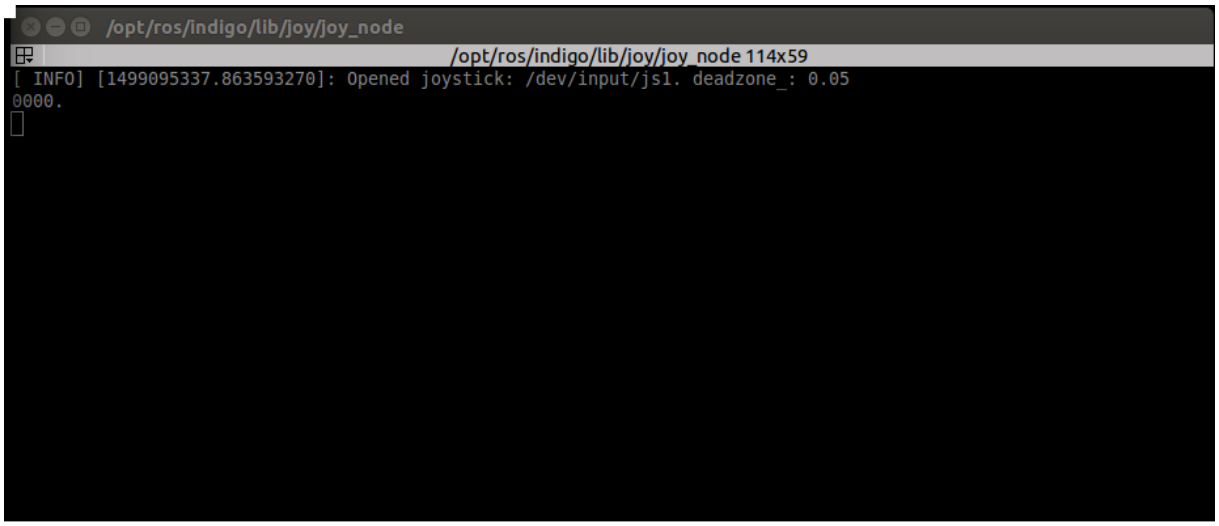
*Figura 20:*Tela de inicialização do comando de teleoperação por teclado do quad-rotor.



FONTE: (AUTOR, 2017)

Outro pacote, mais complexo que o anterior, utilizado para teleoperação, foi *hector_quadrotor_teleop,* que permite o uso de um controle de *Xbox* para operação do sistema. Utilizar um controle desse tipo é mais confortável e possibilita uma maior precisão e melhor operação do sistema, que pode ser observado na figura 21.

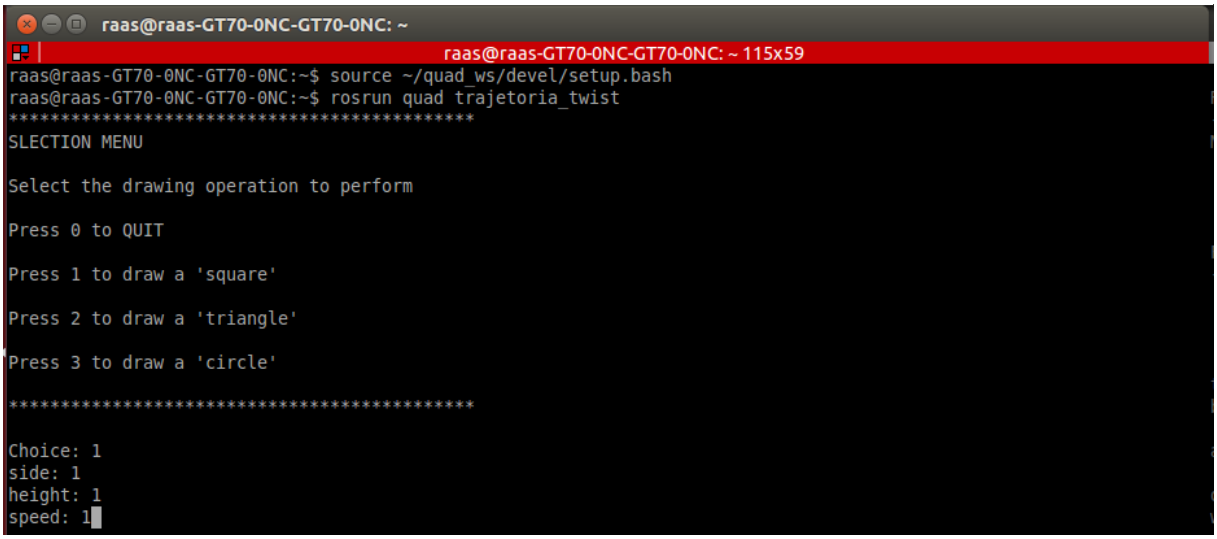
Figura 21: Tela de inicialização do comando de teleoperação por controle de Xbox de quad-rotor.



FONTE: (AUTOR, 2017)

Foi desenvolvido um código que envia uma rotina de comandos de movimentação, gerando trajetórias para o quad-rotor, com o intuito de observar o sistema operando de maneira autônoma, sem uso dos controles apresentados anteriormente. Foi observada uma boa resposta do sistema a esse tipo de controle, possibilitando o uso de planejadores de trajetória que possam alcançar objetivos estipulados pelo usuário de maneira autônoma, a figura 21 mostra o sistema sendo iniciado no computador, o código completo pode ser observado no apêndice B.

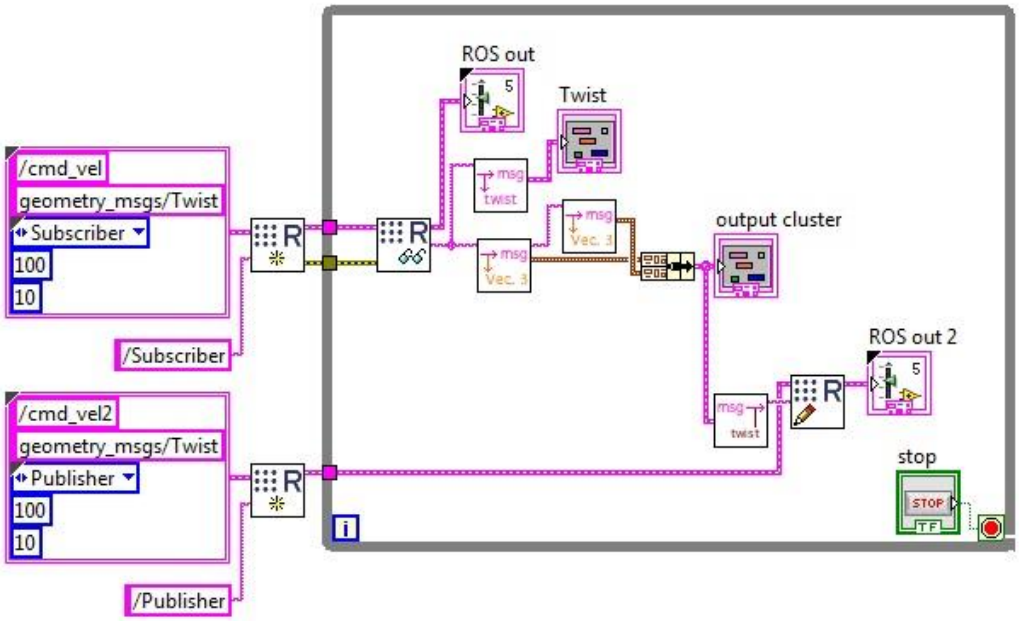
Figura 22: Tela do terminal que inicia programa gerador de trajetórias.



FONTE: (AUTOR, 2017)

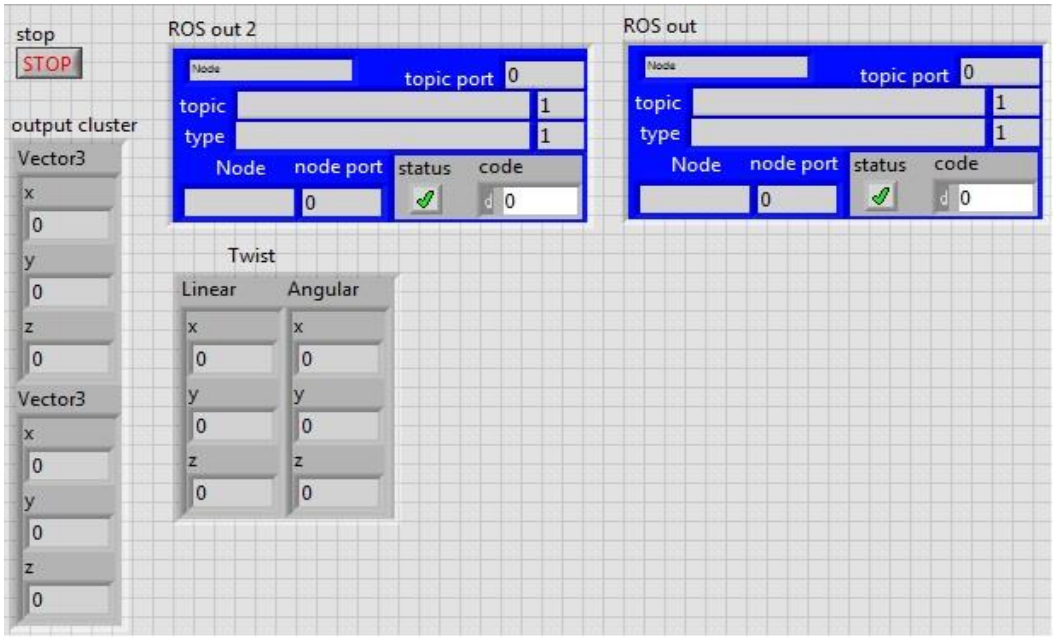
Utilizando a ferramenta disponível para comunicação entre o LabVIEW™ e myRIO com o sistema ROS, foi elaborado um programa que comunicasse as partes. Não foram obtidos os resultados esperados ao utilizar a ferramenta, sendo possível apenas comunicar um computador utilizando LabVIEW™ ao sistema ROS, não sendo possível a conexão entre o myRIO e ROS, o programa utilizado no computador e myRIO pode ser observado nas figuras 22 e 23.

Figura 23: Programa em linguagem de blocos para comunicação entre LabVIEW™ e ROS.



FONTE: (AUTOR, 2016)

*Figura 24:*Painel frontal do programa em LabVIEW™ que se comunica com ROS.



FONTE: (AUTOR, 2016)

Após identificar problemas de comunicação para os quais não foi obtido sucesso em soluciona-los utilizando a plataforma myRIO e LabVIEW, foi proposta como solução para integração a plataforma de prototipagem Arduino com diferentes versões, porém, não foi obtido o resultado esperado, novamente. Foi identificada a falta de compatibilidade da versão due, por ser uma plataforma com arquitetura interna diferente da maior parte das placas, e baixa eficiência do sistema na versão duemilanove. Não foi possível estabelecer uma comunicação que atendesse às necessidades deste trabalho. O programa completo utilizado nos microcontroladores do tipo Arduino pode ser observado no apêndice C.

6 DISCUSSÃO E CONCLUSÃO

Durante o processo de desenvolvimento do sistema, foi observada uma grande dificuldade na reutilização do código ROS de terceiros, devido à falta de suporte. Os pacotes de ROS tinham pouca documentação e elevado nível de conhecimento no *framework* como pré-requisito para o uso. Foi necessária a dedicação da maior parte do tempo disponível para elaboração do trabalho para entender o funcionamento dos pacotes e desenvolver adaptações para o projeto.

Foi necessário identificar por quais caminhos os dados deveriam ser distribuídos e quais seus formatos. Diversos pacotes foram analisados para a escolha final do hector_quadrotor como base para o sistema deste trabalho, devido a uma maior documentação disponível e muitos usuários da comunidade ROS.

Apesar de o pacote hector_quadrotor ter aplicação em sistemas apenas simulados, a integração com um sistema real se mostrou possível, evidenciando a utilidade do pacote para potencializar a concepção de quad-rotores e tecnologias utilizadas nesse tipo de robôs. Através da análise do fluxo dos dados pelos tópicos e a estrutura do sistema pode-se oferecer um sistema de visualização e telemetria para o sistema real e simulação.

Foi necessária a alteração de arquivos de inicialização para carregamento de funcionalidades diferentes do padrão (teleoperação e ambiente de simulação simplificado). Outro desafio foi a integração do sistema ROS que é prioritariamente utilizado em plataforma Linux e o myRIO, que é utilizado principalmente em plataforma Windows. Foi necessária a utilização de um pacote de comunicação do sistema ROS com programas que não são do mesmo *framework*. Ao mesmo tempo, foi necessário o uso de uma ferramenta no LabVIEW que permitia a comunicação e programação que atendessem ao *framework* ROS.

Identificamos que a ferramenta disponível para comunicação entre o LabVIEW e myRIO com o sistema ROS não tem mais suporte do desenvolvedor o que dificulta a sua utilização. Não foram obtidos os resultados esperados ao utilizar a ferramenta sendo possível apenas comunicar um computador utilizando LabVIEW ao sistema ROS.

Como solução para integração, foi utilizada a plataforma de prototipagem Arduíno, com diferentes microcontroladores, porém, não foi obtido o resultado esperado. Devido à falta de compatibilidade ou baixa eficiência do sistema, não foi possível estabelecer uma comunicação que atendesse às necessidades deste trabalho.

Não foi possível realizar a integração entre um quad-rotor genérico e o sistema ROS, devido à falta de tempo para contornar os desafios encontrados com a comunicação entre microcontrolador e sistema ROS.

Foi possível chegar à conclusão, através do desenvolvimento do sistema, que é possível integrar um quad-rotor genérico ao sistema ROS utilizando uma abordagem diferente do que é encontrado na comunidade, utilizando informações do sistema real para alimentar o ambiente virtual e utilizando menos sistemas proprietários e mais aplicações desenvolvidas pelo grupo de trabalho.

O progresso desarticulado de soluções para o sistema foi muito vantajoso, pois possibilitou trabalhar em diferentes partes do projeto sem prejudicar outras. Foi possível gerar códigos com diferentes funcionalidades que, posteriormente, foram integradas ao sistema, devido à metodologia do *framework* ROS.

Foram aproveitados códigos de terceiros, disponibilizados pela comunidade. Empregar porções ou integralmente esses códigos não foi uma tarefa trivial, devido à dificuldade de interpretar e usar códigos com nenhuma ou pouca documentação clara, porém, mostrou-se uma

grande vantagem utilizá-los, após dominar a metodologia do *framework* e a grande quantidade disponível para adaptação.

Foi empregado o ambiente de simulação do pacote hector_quadrotor, que se mostrou adequado para o projeto. Foi possível, nesse ambiente, testar um simulador de trajetórias e sistemas de teleoperação. Quando integrado com o microcontrolador Arduino duemilanove, funcionou de maneira pouco satisfatória, devido às distorções de trajetória oriundas dos atrasos na comunicação.

Foram utilizadas no projeto as plataformas Linux (Ubuntu), Windows 7, LabVIEW 14, Arduino, myRIO e ROS. A utilização das diversas plataformas se mostrou uma grande vantagem do *framework,* possibilitando a utilização de diversas ferramentas e possível integração entre elas. Foi identificado que as ferramentas diponíveis para utilizar o LabVIEW e myRIO não apresentam suporte do desenvolvedor e não apresentaram boas perspectivas de uso futuro. As outras plataformas têm diversas aplicações disponíveis e boa compatibilidade.

Pode-se observar uma boa aplicação dos pacotes de teleoperação, utilizá-los torna o sistema mais relevante, oferecendo a possibilidade de utilizar diversos meios para gerar comandos de movimentação através de teclado, joystick e rotinas geradas por códigos que geram uma sequência de comandos no sistema.

Foi alcançado o objetivo de criar um sistema operacional para quad-rotores genéricos, utilizando porções ou integralmente códigos disponibilizados pela comunidade e adicionando códigos gerados pelo autor. Após vencida a barreira de interpretação dos códigos de terceiros e do método oferecido pelo *framework,* foi observado que esse ambiente pode facilitar as pesquisas futuras com quad-rotores, dando um direcionamento para evolução de novas aplicações para quad-rotores.

Devido à limitação de tempo, não foi possível uma integração com um protótipo de quad-rotor genérico real, mas é uma possibilidade plausível que, após a solução de problemas de comunicação entre a plataforma de microcontrolador utilizada e o sistema ROS, poderá ser alcançada.

Entre os programas de computador utilizados apenas o LabVIEW não era *open source*. Porém, foi utilizada a licença que é oferecida junto ao myRIO. Logo, não foi gerado custo extra. Os equipamentos utilizados já estavam disponíveis, o que fez o projeto não ter gerado ônus financeiro, mostrando que o progresso tem baixo investimento inicial e uma perspectiva de baixo orçamento necessário para futuros projetos que derem prosseguimento a esse sistema.

Diante disso, conclui-se que o desenvolvimento do sistema operacional ROS para quad-rotores genéricos proposto por este trabalho necessita de um extenso período de aprendizado e analise de códigos para reutilização. Vencida essa etapa, reduz-se o tempo necessário para projetar e executar novos protótipos e aplicações de quad-rotores indicando que sua aplicação é ideal quando é prevista a continuação do trabalho, desta forma, diminui-se o tempo de desenvolvimento. Introduz custo mínimo para o projeto viabilizando a utilização deste e podendo potencializar o estudo de VANT's, oferecendo um ambiente desarticulado de concepção, porém, com integração amigável através do *framework* ROS.

6.1 Proposta para trabalhos futuros

Nesta secção, foram exploradas as diversas possibilidades de continuação deste trabalho. A possibilidade de continuidade é um indício de que o tema abordado é relevante e continua abrindo oportunidades para pesquisa e aplicação às situações do cotidiano.

Para trabalhos futuros, pode-se criar um sistema gerenciador que reúna todos os pacotes utilizados em um sistema único, com possível operação e instalação de maneira simplificada, permitindo a utilização de inicialização do sistema utilizando menos comandos.

Outra proposta é a integração do sistema com um planejador de trajetórias e modelamento de mapas e cenários para simulação e validação das trajetórias com o protótipo real com adição de novos sensores, como, por exemplo, GPS e câmera, para realizar a navegação e outras possíveis funções, *e.g.* SLAM.

Desenvolver um pacote próprio para comunicação serial, observando velocidade de transmissão e tamanho de *buffer* que sejam adequados para o microcontrolador utilizado pelo quad-rotor e que não prejudique as rotinas de controle de estabilidade do microcontrolador no VANT.

Explorar mais a ferramenta “ROS for LabVIEW”, encontrando maneiras para integrar e embarcar códigos da linguagem LabVIEW no myRIO e oferecer para comunidade soluções para os problemas encontrados neste trabalho.

7 APÊNDICE A – Arquivo de inicialização do sistema ROS.

```
<?xml version="1.0"?>

<launch>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>

  <include file="$(find hector_quadrotor_gazebo)/launch/spawn_quadrotor.launch" />
   <!-- controle teclado.-->
  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py" name="teleop"
launch-prefix="terminator -x "/>
  <!-- Controle xbox.
  Note that axis IDs are those from the joystick message plus one, to be able to invert axes by
specifiying either positive or negative axis numbers.
  Axis 2 from joy message thus has to be set as '3' or '-3'(inverted mode) below
  -->

  <arg name="joy_dev" default="/dev/input/js0" />

  <node name="joy" pkg="joy" type="joy_node" launch-prefix="terminator -x " >
    <param name="dev" value="$(arg joy_dev)" />
  </node>
  <node name="quadrotor_teleop" pkg="hector_quadrotor_teleop" type="quadrotor_teleop">
    <param name="x_axis" value="5"/>
    <param name="y_axis" value="4"/>
    <param name="z_axis" value="2"/>
    <param name="yaw_axis" value="1"/>
```

```
  </node>
<!-- gerador de trajetória.
<node pkg="quad_ws" type="trajetoria_twist" name="trajetoria_twist" launch-prefix="terminator -x "/>
<include file="$(find quad_ws)/src/trajetoria_twist" />-->


</launch>
```

8 APÊNDICE B – Gerador de trajetórias para operação autônoma.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "turtlesim/Pose.h"
#include <sstream>

using namespace std;
using namespace ros;
using namespace geometry_msgs;
using namespace turtlesim;

//declare publishers and subscribers
Publisher velocity_publisher;

int precisao =3; //quanto maior mais preciso
double d = 1;
const double PI = 3.14159265359;

/**** functions declarations *****/
void move(double speed, double distance, bool isForward);
void move_h(double speed, double height, bool isForward);
void rotate (double speed, double angle, bool clockwise);
void drawSquare(int side, double height, double speed); //side is expressed in unit.
void drawTriangle(int side, double height, double speed); //side is expressed in unit. All sides of triangle are equal
void drawCircle(double radius, double height, bool clockwise); //radius is expressed in unit.
void menu();

/**main function**/
```

```
int main(int argc, char **argv)
{
	//inicializando nó
	ros::init(argc, argv, "shape_drawing_node");
	ros::NodeHandle n;
	velocity_publisher = n.advertise<geometry_msgs::Twist>("/cmd_vel", 100);
	ros::Rate loop_rate(100);

	//variaveis
	int choice=0;
	int side=0;
	double radius=0;
	double height=0;
	double speed=0;
	bool clockwise;
	//roda o programa
	while(ros::ok()){

		menu();
		cout<<"Choice: ";
		cin>>choice;
		ros::spinOnce();
		if (choice ==1){
			cout<<"side: ";
			cin>>side;
			cout<<"height: ";
			cin>>height;
			cout<<"speed: ";
			cin>>speed;
			drawSquare(side, height, speed);

		}else if (choice ==2){
			cout<<"side: ";
			cin>>side;
			cout<<"height: ";
			cin>>height;
			cout<<"speed: ";
			cin>>speed;
```

```
                drawTriangle(side, height, speed);

        }else if (choice ==3){
                cout<<"radius: ";
                cin>>radius;
                cout<<"height: ";
                cin>>height;
                cout<<"Anti-horário: ";
                cin>>clockwise;
                drawCircle(radius, height, clockwise);

        }else {
                return 0;
        }
        ros::spinOnce();
    }
    return 0;
}


/** draw the shape square. Use move and rotate functions defined below **/
void drawSquare(int side, double height, double speed){
    ROS_INFO("\nDraw the shape 'Square'\n");
    move_h(speed, height, 1);
    ros::Duration(d).sleep(); // sleep for half a second
    move(speed,side,1);
    ros::Duration(d).sleep(); // sleep for half a second
    rotate(speed, PI/2,1 );
    ros::Duration(d).sleep(); // sleep for half a second
    move(speed,side,1);
    ros::Duration(d).sleep(); // sleep for half a second
    rotate(speed, PI/2,1 );
    ros::Duration(d).sleep(); // sleep for half a second
    move(speed,side,1);
    ros::Duration(d).sleep(); // sleep for half a second
    rotate(speed, PI/2,1 );
    ros::Duration(d).sleep(); // sleep for half a second
    move(speed,side,1);
```

```
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/2,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move_h(speed, height, 0);
        ros::Duration(d).sleep(); // sleep for half a second
}

/** draw the shape triangle. Use move and rotate functions defined below **/
void drawTriangle(int side, double height, double speed){
        ROS_INFO("\nDraw the shape 'Triangle'\n");
        move_h(speed, height, 1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/6 ,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move(speed,side,1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/3,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move(speed,side,1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/3,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move(speed,side,1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, 2*PI/6,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move_h(speed, height, 0);
        ros::Duration(d).sleep(); // sleep for half a second
}
/** draw the shape circle. Use move and rotate functions defined below **/
void drawCircle(double radius, double height, bool clockwise){
        int speed=1;
        ROS_INFO("\nDraw the shape 'Circle'\n");

        move_h(speed, height, 1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/2,0 );
        ros::Duration(d).sleep(); // sleep for half a second
```

```
move(speed,radius,1);
ros::Duration(d).sleep(); // sleep for half a second
rotate(speed, PI/2,1 );
ros::Duration(d).sleep(); // sleep for half a second


geometry_msgs::Twist vel_msg;

//set a  linear velocity in the z/y-axis
vel_msg.linear.x =radius;
vel_msg.linear.y =0;
vel_msg.linear.z =0;
//set a  angular velocity in the axis
vel_msg.angular.x = 0;
vel_msg.angular.y = 0;
if (clockwise)
        vel_msg.angular.z =-1;
else
        vel_msg.angular.z =1;


double pos = 2*PI*radius;
double current_pos = 0;
double t0 = ros::Time::now().toSec();
ros::Rate loop_rate(1000);

do{
        velocity_publisher.publish(vel_msg);
        double t1 = ros::Time::now().toSec();
        current_pos = radius * (t1-t0);
        ros::spinOnce();
        loop_rate.sleep();
        cout<<"Posição atual: "<<current_pos <<"\nPosição final: "<<pos<<endl;
}while(current_pos<pos);

//force the robot to stop when it reaches the desired angle
vel_msg.linear.x =0;
vel_msg.angular.z =0;
```

```
        velocity_publisher.publish(vel_msg);
        ros::Duration(d).sleep(); // sleep for half a second

        rotate(speed, PI/2,1 );
        ros::Duration(d).sleep(); // sleep for half a second
        move(speed,radius,1);
        ros::Duration(d).sleep(); // sleep for half a second
        rotate(speed, PI/2,0 );
        ros::Duration(d).sleep(); // sleep for half a second
        move_h(speed, height, 0);
        ros::Duration(d).sleep(); // sleep for half a second

}

/************************************
 * PROVIDED HELPER FUNCTIONS        *
 ************************************/

/** display the menu of the application **/
void menu(){

        cout<<"*********************************************"<<endl;
        cout<<"SLECTION MENU\n"<<endl;
        cout<<"Select the drawing operation to perform\n"<<endl;
        cout<<"Press 0 to QUIT \n"<<endl;
        cout<<"Press 1 to draw a 'square' \n"<<endl;
        cout<<"Press 2 to draw a 'triangle' \n"<<endl;
        cout<<"Press 3 to draw a 'circle' \n"<<endl;
        cout<<"********************************************* \n"<<endl;

}

/** move the robot with a certain speed for a certain distance, forward or backward **/
void move(double speed, double distance, bool isForward){
                geometry_msgs::Twist vel_msg;
        //set a linear velocity in the x-axis
        if (isForward)
                vel_msg.linear.x =speed;
```

```
        else
                vel_msg.linear.x =-speed;
        //set a linear velocity in the y/z-axis
        vel_msg.linear.y =0;
        vel_msg.linear.z =0;
        //set a  angular velocity in the axis
        vel_msg.angular.x = 0;
        vel_msg.angular.y = 0;
        vel_msg.angular.z =0;

        double t0 = ros::Time::now().toSec();
        double current_distance = 0.0;
        ros::Rate loop_rate(1000);
        do{
                velocity_publisher.publish(vel_msg);
                double t1 = ros::Time::now().toSec();
                current_distance = speed * (t1-t0);
                ros::spinOnce();
                loop_rate.sleep();
                cout<<"Distancia atual:"<<current_distance<<"\nDistancia Final:"<<distance<<endl;
                }while(current_distance<distance );

        //force the robot to stop when it reaches the desired angle
        vel_msg.linear.x =0;
        velocity_publisher.publish(vel_msg);


}

/** rotate the robot with a certain angular speed for a certain relative angle in radians **/
void rotate (double speed, double relative_angle, bool clockwise){

        geometry_msgs::Twist vel_msg;
        //set a  linear velocity in the axis
        vel_msg.linear.x =0;
        vel_msg.linear.y =0;
        vel_msg.linear.z =0;
        //set a  angular velocity in the x/y-axis
```

```
        vel_msg.angular.x = 0;
        vel_msg.angular.y = 0;
        //set a  angular velocity in the z-axis
        if (clockwise)
                vel_msg.angular.z =-speed;
        else
                vel_msg.angular.z =speed;


        double current_angle = 0.0;
        ros::Rate loop_rate(1000);
        double t0 = ros::Time::now().toSec();
        do{
                velocity_publisher.publish(vel_msg);
                double t1 = ros::Time::now().toSec();
                current_angle = speed * (t1-t0);
                ros::spinOnce();
                loop_rate.sleep();
                cout<< "Angulo atual"<<current_angle <<"\nAngulo final"<<relative_angle<<endl;
        }while(current_angle<relative_angle);

        //force the robot to stop when it reaches the desired angle
        vel_msg.angular.z =0;
        velocity_publisher.publish(vel_msg);


}
/** move the robot with a certain speed for a certain distance, forward or backward **/
void move_h(double speed, double height, bool isForward){
                geometry_msgs::Twist vel_msg;
        //set a  linear velocity in the z-axis
        if (isForward)
                vel_msg.linear.z =speed/precisao;
        else
                vel_msg.linear.z =-speed/precisao;
        //set a  linear velocity in the x/y-axis
        vel_msg.linear.y =0;
        vel_msg.linear.x =0;
```

```
        //set a  angular velocity in the axis
        vel_msg.angular.x = 0;
        vel_msg.angular.y = 0;
        vel_msg.angular.z =0;

        double t0 = ros::Time::now().toSec();
        double current_height = 0.0;
        ros::Rate loop_rate(1000);
        do{
                velocity_publisher.publish(vel_msg);
                double t1 = ros::Time::now().toSec();
                current_height = speed/precisao * (t1-t0);
                ros::spinOnce();
                loop_rate.sleep();
                cout<<"Altura atual:"<<current_height <<"\n Altura final:"<<height<<endl;
        }while(current_height<height);
        //force the robot to stop when it reaches the desired angle
        vel_msg.linear.z =0;
        velocity_publisher.publish(vel_msg);

}
```

9 APÊNDICE C – Sistema Arduino de aquisição de dados de sensores e comunicação com ROS.

```
/*********************************************************************/
/*  Libraries                                                        */
/*********************************************************************/
//#include <Wire.h>
#include <Servo.h>
#include <ros.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Imu.h>

/*********************************************************************/
/*********************************************************************/
/*  Arduino PIN Setup                                                */
```

```
/*********************************************************************/
/*********************************************************************/
/*********************************************************************/
/*  Main objects and global variables                                */
/*********************************************************************/
double x_i=0;
double y_i=0;
double z_i=0;
double t_i=millis();
int seq=0;
/*********************************************************************/
/*********************************************************************/
/*  ROS Callbacks                                                    */
/*********************************************************************/
void callback(const geometry_msgs::Twist& msg){
  // Serial.println(msg.linear.x);
  //Serial.println(msg.linear.y);
  //Serial.println(msg.linear.z);
  //Serial.println(msg.angular.x);
  //Serial.println(msg.angular.y);
  //Serial.println(msg.angular.z);


}
/*********************************************************************/
/*********************************************************************/
/*  ROS Setup                                                        */
/*********************************************************************/
ros::NodeHandle nh;
ros::Subscriber<geometry_msgs::Twist> sub("cmd_vel", callback);

/*sensor_msgs::Imu imu_msg;*/
geometry_msgs::Twist twist_msg;
ros::Publisher pub_twist("/cmd_vel2", &twist_msg);
/*********************************************************************/
/*********************************************************************/
/*  Arduino Setup function                                           */
/*********************************************************************/
```

```
void setup() {

  /*****************************************************/
  /* Setup ROS
  /*****************************************************/
  nh.getHardware()->setBaud(115200);
  nh.initNode();
  nh.subscribe(sub);
  nh.advertise(pub_twist);
  //nh.advertise(pub_imu);
  /*****************************************************/
  delay(1500);
}
/*********************************************************************/
/*********************************************************************/
/* Arduino loop function                                             */
/*********************************************************************/
void loop() {
  nh.spinOnce();
  randomSeed(analogRead(5)); // randomize using noise from analog pin 5
  imu_msg.orientation.x = (random(2)-1)/2;
  imu_msg.orientation.y = (random(2)-1)/2;
  imu_msg.orientation.z = (random(2)-1)/2;
  imu_msg.linear_acceleration.x = (random(2)-1)/2;
  imu_msg.linear_acceleration.y = (random(2)-1)/2;
  imu_msg.linear_acceleration.z = (random(2)-1)/2;
  imu_msg.angular_velocity.x = (random(2)-1)/2;
  imu_msg.angular_velocity.y = (random(2)-1)/2;
  imu_msg.angular_velocity.z = (random(2)-1)/2;
  imu_msg.orientation.w = random(30);//levando a msg de altura no w do quaternion
  double t_f=millis();
  twist_msg.linear.x =imu_msg.angular_velocity.x;
  twist_msg.angular.x=(imu_msg.linear_acceleration.x +(imu_msg.linear_acceleration.x-x_i)/2)*(t_f-
t_i);
  twist_msg.linear.y =imu_msg.angular_velocity.x;
  twist_msg.angular.y=(imu_msg.linear_acceleration.y +(imu_msg.linear_acceleration.y-y_i)/2)*(t_f-
t_i);
  twist_msg.linear.z =imu_msg.angular_velocity.x;
```

```
            twist_msg.angular.z =(imu_msg.linear_acceleration.z +(imu_msg.linear_acceleration.z-z_i)/2)*(t_f-
t_i);
            pub_twist.publish(&twist_msg);
            x_i=imu_msg.orientation.x;
            y_i=imu_msg.orientation.y;
            z_i=imu_msg.orientation.z;
            t_i=t_f;
            delay(50);
          }
          /*********************************************************************/
          /*********************************************************************/
          /* END OF CODE                                          */
/*********************************************************************/
```

10 REFERÊNCIAS

ABICHOU, A.; BEJI, L. e ZEMALACHE, K. M. **Smooth control of an x4 bidirectional rotors flying robot**. Fifth International Workshop on Robot Motion and Control, pages 181 – 186, 2005.

ACHTELIK, M. *et al.* **Energy-efficient autonomous four-rotor flying robot controlled at 1 khz**. IEEE International Conference on Robotics and Automation, pages 361 – 366, 2007.

ALEJO, D. *et al.* **Optimal Reciprocal Collision Avoidance with mobile and static obstacles for multi-UAV systems**. 2014 International Conference on Unmanned Aircraft Systems, ICUAS 2014 - Conference Proceedings, p. 1259–1266, 2014.

ARGYLE, M. E. *et al.* **Quaternion based attitude error for a tailsitter in hover flight.** Proceedings of the American Control Conference, p. 1396–1401, 2014.

BEJI, L.; ZEMALACHE, K. M. e MARREF, H. **Control of an under-actuated system: Application to a four rotors rotorcraf**t. IEEE International Conference on Robotics and Biomimetics, pages 404 – 409, 2005.

BENAVIDEZ, P.; LAMBERT, J. **Landing of a quadcopter on a mobile base using fuzzy logic**. Advance Trends in Soft …, 2014.

BOUABDALLAH, S.; MURRIERI, P.; SIEGWART, R. **Design and control of an indoor micro quadrotor**. IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004, v. 5, n. April, p. 4393–4398, 2004.

BRESCIANI, T. Modelling **, Identification and Control of a Quadrotor Helicopter.** English, v. 4, n. October, p. 213, 2008.

BRISTEAU, P. *et al.* **The role of propeller aerodynamics in the model of a quadrotor** UAV. European Control Conference Proceedings, n. November, p. 683–688, 2009.

CASTIBLANCO, C.; RODRIGUEZ, J. **Air drones for explosive landmines detection**. **… :** First Iberian Robotics …, 2014.

Clearpath tutorial. 2012. Disponível em: http://files.clearpathrobotics.com/ROS%20Toolkit%20Example.pdf. Acessado em: 13 abril 2017.

COCCHIONI, F. *et al.* **Visual Based Landing for an Unmanned Quadrotor**. Journal of Intelligent and Robotic Systems: Theory and Applications, 2015.

COUSINS, S. *et al.* **Sharing software with ROS**. IEEE Robotics and Automation Magazine, v. 17, n. 2, p. 12–14, 2010.

DAVIS, E.; NIZETTE, B. E.; YU, C. **Development of a Low Cost Quadrotor Platform for Swarm Experiments**. Proceedings of the 32nd Chinese Control Conference, n. November, p. 7072–7077, 2013.

DEMARCO, K.; WEST, M. E.; COLLINS, T. R. **An implementation of ROS on the Yellowfin autonomous underwater vehicle (AUV).** Oceans 2011, p. 1–7, 2011.

DRAGANFLYER. **Draganflyer X6**. 2013. Disponível em: http://www.draganfly.com/uavhelicopter/draganflyer-x6/. Acessado em: 24 abr 2016.

DUNKLEY, O. *et al.* **Visual-Inertial Navigation for a Camera-Equipped 25 g Nano-Quadrotor**. IROS2014 Aerial Open Source Robotics Workshop, p. 4–5, 2014.

ERLE-COPTER DRONE. Disponível em: http://erlerobotics.com/blog/product/erle-copter-diy-kit/. Acessado em: 16 fev 2016.

ERMACORA, G. *et al.* **A Cloud Based Service for Management and Planning of Autonomous UAV Missions in Smart City Scenarios**. Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014, Rome, Italy, May 5-6, 2014, Revised Selected Papers, v. 8906, p. 20, 2014.

FURTADO, V. H. *et al.* **Aspectos de segurança na integração de veículos aéreos não tripulaos (VANT) no espaço aéreo brasileiro**. VII Sitraer Simpósio de Transportes Aéreos, v. 7, p. 506–517, 2008.

GAZEBO. Disponível em: www.gazebosim.org. Acesso em: 02 de maio de 2016.

GEOMETRY MSGS. Disponível em: http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html. Acesso em: 02 de maio de 2016.

GIL, A. C. **Como elaborar projetos de pesquisa**. 5. ed. São Paulo: Atlas, 2008.

GRABE, V. *et al.* **The TeleKyb *framework* for a modular and extendible ROS-based quadrotor control**. 2013 European Conference on Mobile Robots, ECMR 2013 - Conference Proceedings, p. 19–25, 2013.

HAMAMOTO, M. *et al.* **A fundamental study of wing actuation for a 6-in-wingspan flapping microaerial vehicle**. IEEE Transactions on Robotics, v. 26, n. 2, p. 244–255, 2010.

HALVORSEN, Hans-Petter. **Introduction to LabVIEW**. 07 de setembro de 2016. Disponível em: home.hit.no/~hansha/documents/labview/training/Introduction%20to%20LabVIEW/Introduction%20to%20LabVIEW.pdf. Acesso em: 13 de abril de 2017.

HECTOR QUADROTOR DESCRIPTION. 2016. Disponível em: http://wiki.ros.org/hector_quadrotor_description. Acesso em: 02 de maio de 2016.

HECTOR QUADROTOR GAZEBO. Disponível em: (http://wiki.ros.org/hector_quadrotor_gazebo. Acesso em: 02 de maio de 2016.

HECTOR QUADROTOR GAZEBO PLUGINS. Disponível em: (http://wiki.ros.org/hector_quadrotor_gazebo_plugins. Acesso em: 02 de maio de 2016.

HECTOR QUADROTOR TELEOP. Disponível em: http://wiki.ros.org/hector_quadrotor_teleop. Acesso em: 02 de maio de 2016.

Hector_quadrotor Tutorial. 2014. Disponível em: http://wiki.ros.org/hector_quadrotor/Tutorials/Quadrotor%20outdoor%20flight%20demo. Acesso em: 13 de abril de 2017.

HOLD-GEOFFROY, Y. *et al.* **Ros4mat: A Matlab programming interface for remote operations of ROS-based robotic devices in an educational context**. Proceedings - 2013 International Conference on Computer and Robot Vision, CRV 2013, p. 242–248, 2013.

JUNG, C. R.; OSÓRIO, F. S.; ROBERTO, C. **Computação Embarcada : Projeto e Implementação de Veículos Autônomos Inteligentes**. XXV Congresso da Sociedade Brasileira de Computação, p. 1358–1406, 2005.

KOVAL, M.; MANSLEY, C.; LITTMAN, M. **Autonomous quadrotor control with reinforcement learning**. Disponível em: <http://mkoval.org/projects/quadrotor/files/quadrotor-rl.pdf>.

LEVI, N. *et al.* **The DARPA virtual robotics challenge experience**. 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2013, 2013.

LI, R. *et al.* **ROS based multi-sensor navigation of intelligent wheelchair**. Proceedings - 2013 4th International Conference on Emerging Security Technologies, EST 2013, p. 83–88, 2013.

LINDSEY, Q.; MELLINGER, D.; KUMAR, V. **Construction of Cubic Structures with Quadrotor Teams**. Mechanical Engineering, 2011.

MARTINEZ, A.; FERNÁNDEZ, E. **Learning ROS for Robotics Programming**.Packt Publishing 2013.

MAS, I. *et al.* **Visual target-tracking using a formation of unmanned aerial vehicles**. ASME - Intenational Design Engineering Technical Conferences & Computers and Information in Engineering Conference, n. AUGUST, 2015.

MELO, SALLES E ALMEIDA, 2010. **Implementação De Uma Aeronave Miniatura Semiautônoma Com Quatro Propulsores Como Plataforma De Desenvolvimento**. XVIII Congresso Brasileiro de Automática - CBA, p. 1805 – 1810, 2010.

MESTER, G. Cloud Robotics Model. **Interdisciplinary Description of Complex Systems**, v. 13, n. 1, p. 1–8, 2015.

MEYER, J. *et al.* **Comprehensive simulation of quadrotor UAVs using ROS and Gazebo**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), v. 7628 LNAI, p. 400–411, 2012.

MICRODRONES. Disponível em: http://www.microdrones.com. Acessado em 27 mai 2015.

MONTUFAR, D.; MUNOZ, F. **Multi-UAV testbed for aerial manipulation applications**. Unmanned Aircraft …, 2014.

MyRIO. 2014. Disponível em: http://www.ni.com/white-paper/52093/en/. Acessado em 13 abril 2017.

myRIO Publisher. 2015. Disponível em: http://forums.ni.com/t5/ROS-for-LabVIEW-TM-Software/Publisher-Video-Tutorial/gpm-p/3523166. Acessado em 13 abril 2017.

myRIO Subscriber. 2015. Disponível em: http://forums.ni.com/t5/ROS-for-LabVIEW-TM-Software/ROS-for-LabVIEW-Tutorial-pdf/gpm-p/3514159. Acessado em 13 abril 2017.

OGATA, K. **Engenharia de Controle Moderno**. Ed. Prentice Hall. 2000.

O'KANE, Jason M. **A Gentle Introduction to ROS**. 2014.

OLIVARES-MENDEZ, M. **See-and-avoid quadcopter using fuzzy control optimized by cross-entropy**. … Systems (FUZZ-IEEE …), 2012.

OLIVARES-MENDEZ, M. A.; KANNAN, S.; HOLGER VOOS. V-REP & ROS **Testbed for Design, Test, and Tuning of a Quadrotor Vision Based Fuzzy Control System for Autonomous Landing**. IMAV 2014: International Micro Air Vehicle Conference and Competition 2014, Delft, The Netherlands, August 12-15, 2014, p. 172–179, 2014.

OSA, Y.; UCHIKADO, S.; TANAKA, K. **Study on Roll Angle Control for Hovering Tilt Rotor Aircraft on One Side Thrust Reduction**. n. Cacs, p. 26–29, 2014.

PASTOR-MORENO, D. **Optical flow localisation and appearance mapping (OFLAAM) for long-term navigation**. … Aircraft Systems (ICUAS) …, 2015.

PERSPECTIVES AERIALS, **Perspectives Aerials**. 2013. Disponível em: http://www.perspectiveaerials.com/. Acessado em 27 abr 2016.

PESTANA, J.; SANCHEZ-LOPEZ, J. **A vision based aerial robot solution for the iarc.** By the technical university of Madrid. 2014.

KROO, ILAN, and FRITZ PRINZ. **The Mesicopter: A Meso-Scale Flight Vehicle NIAC Phase II Technical Proposal**. Technical report, Stanford University, 2001.

QUADROTOR BASE XACRO. Disponível em: (https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor/blob/HEAD/hector_quadrotor_description/urdf/quadrotor_base.urdf.xacro. Acesso em: 02 de maio de 2016

QUADROTOR HOKUYO. 2016. Disponível em: https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor/blob/HEAD/hector_quadrotor_description/urdf/quadrotor_hokuyo_utm30lx.urdf.xacro. Acesso em: 02 de maio de 2016.

QUADROTOR XACRO. Disponível em: https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor/blob/HEAD/hector_quadrotor_description/urdf/quadrotor.urdf.xacro. Acesso em: 02 de maio de 2016.

R C TOYS. Disponível em: http://www.rctoys.com. Acessado em: 24 jan 2016.

ROS, **About ROS. 2015**. Disponível em: http://www.ros.org/about-ros/. Última visita: 09 jan 2016.

ROS DISTRIBUTIONS, 2017. Disponível em: http://wiki.ros.org/Distributions. Acessado em: 13 abril 2017.

ROS for LabVIEW Software. 2016. Disponível em: https://github.com/tuftsBaxter/ROS-for-LabVIEW-Software. Acessado em: 13 abril 2017.

SILVA, R. A. A., **Modelagem, simulação, controle e análise de um quad-rotor na decolagem e pouso.** Monografia (graduação) – Engenharia de Controlo e Automação, Instituto Federal de Educação, Ciências e Tecnologia de São Paulo - IFSP, 2014.

SILVERLIT. Disponível em: http://www.silverlit.com. Acessado em: 10 jan 2016.

SPICA, R. *et al.* **An Open-Source Ready-to-Use Hardware / Software Architecture for Quadrotor UAVs**. IEEE/RSJ International Conference on Intelligent Robots and Systems, 2013.

SPICA, R. *et al.* **Interfacing Matlab/Simulink with V - REP for an Easy Development of Sensor - Based Control Algorithms for Robotic Platforms.** Workshop on MATLAB/Simulink for Robotics Education and Research, IEEE International Conference on Robotics and Automation (ICRA 2014), p. 3–4, 2014.

TUM_SIMULATOR. **tum_simulator.** Disponível em: http://wiki.ros.org/tum_simulator. Acesso em: 30 mar 2017.

WIKI ROS. Disponível em: http://wiki.ros.org/hector_quadrotor. Acesso em: 02 de maio de 2016.

ZAMAN, S.; SLANY, W.; STEINBAUER, G. **ROS-based mapping, localization and autonomous navigation using a Pioneer 3-DX robot and their relevant issues**. Saudi International Electronics, Communications and Photonics Conference 2011, SIECPC 2011, p. 0–4, 2011.

ZHANG, M. *et al.* **A High Fidelity Simulator for a Quadrotor UAV using ROS and Gazebo**. IECON2015-Yokohama November 9-12, 2015: p. 2846–2851, 2015.