UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Matheus Kraemer Bastos do Canto

Rotas adaptativas para inspeção autônoma de subestação por meio do uso de VANT

Florianópolis
2019

Matheus Kraemer Bastos do Canto

Rotas adaptativas para inspeção autônoma de subestação por meio do uso de VANT

Relatório submetido à Universidade Federal de Santa Catarina como requisito para a aprovação na disciplina **DAS 5511: Projeto de Fim de Curso** do curso de Graduação em Engenharia de Controle e Automação.
Orientador(a): Prof. Ubirajara Franco Moreno

Florianópolis
2019

Matheus Kraemer Bastos do Canto

Rotas adaptativas para inspeção autônoma de subestação por meio do uso de VANT

Esta monografia foi julgada no contexto da disciplina DAS5511: Projeto de Fim de Curso e aprovada na sua forma final pelo Curso de Engenharia de Controle e Automação.

Florianópolis, 30 de julho de 2019

Banca Examinadora:

Alexandre Marcondes
Orientador na Empresa
Fundação CERTI

Prof. Ubirajara Franco Moreno
Orientador no Curso
Universidade Federal de Santa Catarina

Prof. Max Hering de Queiroz
Avaliador
Universidade Federal de Santa Catarina

Iuri Cuneo Ferreira
Debatedor
Universidade Federal de Santa Catarina

Ariel Dutra Farias
Debatedor
Universidade Federal de Santa Catarina

Esse trabalho é dedicado aos meus pais Miriam e Jefferson, que com o tempero de sua individualidade me ajudaram a chegar até aqui.

AGRADECIMENTOS

Ao meu pai Jefferson e minha mãe Miriam que cada qual, a sua maneira, me ajudaram no ingresso e apoiaram minha permanência nesta instituição.

Ao meu companheiro de quarto e primo Júlio por me aguentar e me manter constantemente em contato com a diferença.

Agradeço enormemente ao meu amigo Thiago, pelo seu soberbo senso crítico e disposição na revisão desse texto, e junto aos amigos Diogo e Maycon também pelo companheirismo nos melhores e piores momentos desta graduação, que definitivamente foi essencial para a conclusão dessa etapa da minha vida.

Ao meu amigo Kevin, pelos interessantes grupos de trabalhos e reflexões que tivemos aqui e lá fora.

A Fundação CERTI, instituição que serviu como meu berço profissional, pela estrutura, apoio e companheirismo imprescindíveis de seus colaboradores para a conclusão desse trabalho.

A toda a equipe de Metrologia e Instrumentação & Testes da Fundação CERTI, em especial meu coordenador Alexandre Marcondes, pelos incentivos, paciência, diálogos e discussões que tivemos ao longo desta caminhada.

A todas as mãos que sustentam a universidade pública e contribuem para o futuro do Brasil.

A todos que contribuíram direta ou indiretamente com este trabalho meu eterno agradecimento.

RESUMO

A solução, em formato de sistema, foi desenvolvida na Fundação CERTI como parte do módulo experimental da automação do processo de aquisição de imagens termográficas de equipamentos de subestação de um projeto de P&D, contratado pela concessionária de energia COPEL. O projeto propõe um sistema de geração e execução de rotas de inspeção adaptativas livres de colisão, passando por pontos definidos pelo usuário. As rotas apresentam comportamento adaptativo pois ao longo da inspeção podem ter seu curso alterado por uma condição externa de tratamento de imagem e não são fixas entre os pontos de inspeção, se adaptam caso o mapa de obstáculos seja alterado. Para definição dos pontos de inspeção o operador utiliza um ambiente virtual, com o modelo de colisão dos obstáculos e de um VANT genérico. Neste ambiente, o VANT é controlado manualmente pelo operador através de um controle USB genérico. Para representação do modelo de colisão, o trabalho apresenta um método de conversão de modelos 3D para *Octotree*, um padrão de dados utilizado por algoritmos de *motion planning.* Para geração de rotas entre os pontos coletados o sistema propõe o uso de algoritmos de *motion planning* disponíveis em bibliotecas gratuitas distribuídas para ROS. Para execução das rotas no VANT, um aplicativo para sistema *Android* foi desenvolvido para controlar o processo de tratamento e execução dos pontos recebidos pelo sistema gerador de rotas. Para a troca de mensagens e execução de serviços entre os principais módulos do sistema, o *framework* ROS foi utilizado. Os resultados do sistema foram satisfatórios quanto ao mecanismo de geração e controle das rotas, como comprovam testes realizados com o sistema como evitar obstáculos, controle de posição e orientação, conversão entre coordenadas cartesianas e de GPS e teste de execução de rotas no simulador do próprio VANT DJI Inspire 1.

Palavras-chave: VANT, Inspeção, Rotas, ROS, *Octotree*

ABSTRACT

The solution, in system format, was developed at CERTI Foundation as part of the experimental module of automation of the process of acquisition of thermographic images of substation equipment of a R&D project, hired by the energy company COPEL. The project proposes a system of generation and execution of adaptive inspection routes free of collision, passing through points defined by the operator. The routes present an adaptive behavior because along the inspection they can have their course altered by an image processing external condition and are not fixed between the inspection points, they adapt if the obstacle map is altered. In this environment, the UAV is manually controlled by the operator through a generic USB control. To represent the collision model, the work presents a method for converting 3D models to Octotree, a data standard used by motion planning algorithms. To generate routes between the collected points, the system uses sample-based motion planning algorithms available in free libraries distributed for ROS. To execute the routes in the UAV, an application for Android system was developed to control the process of treatment and execution of the points received by the route generator system. For the exchange of messages and execution of services between the main modules of the system, the ROS framework was used. The results of the system were satisfactory in terms of the mechanism for generating and controlling routes, as shown by tests performed with the system to avoid obstacles, position and orientation control, conversion between Cartesian and GPS coordinates and route execution test in the simulator of the VANT DJI Inspire 1 itself.

Key-words: UAV. Inspection. Adaptive Route

LISTA DE ILUSTRACÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BSD - Berkeley *Software Distribution*

CERTI - Centros de Referência em Tecnologias Inovadoras

COPEL - Companhia Paranaense de Energia

IBGE - Instituto Brasileiro de Geografia e Estatística

ROS - *Robot Operating System*

SDK - *Software Development Kit*

SLAM - *Simultaneos Localization and Mapping*

UFSC - Universidade Federal de Santa Catarina

VANT - Veículo Aéreo Não Tripulado

SUMÁRIO

1. INTRODUÇÃO

De acordo com o Censo Demográfico 2011 o setor elétrico brasileiro é responsável pela geração, transmissão e distribuição de energia elétrica para mais de 57 milhões de lares brasileiros (1). A estrutura do setor reflete da continentalidade inerente ao território nacional e seu desenvolvimento e avanço são historicamente entrelaçados ao progresso socioeconômico brasileiro. O setor é composto por agentes que gerenciam as três áreas chave: geração, transmissão e distribuição.

Os agentes de geração produzem a energia que é transportada para regiões próximas aos centros urbanos pelos agentes de transmissão e finalmente entregues ao consumidor final pelos agentes de distribuição (2). Este intercâmbio de energia é possível devido à extensa malha de conexão, Sistema Integrado Nacional (SIN), com mais de 100 mil quilômetros de comprimento (2) (3).

No ano de 2017, apesar da queda de 1,2 %, o consumo de energia elétrica no Brasil alcançou 467 TWh, colocando o Brasil entre os 10 maiores consumidores do mundo (4). De acordo com o estudo e projeções realizadas pela Empresa de Pesquisa Energética (EPE) para 2027 o consumo de energia no Brasil está estimado para 744 TWh (5).

Considerando a importância da energia elétrica no desenvolvimento do país e um aumento previsto nos anos que seguem, o funcionamento contínuo e integridade são, portanto, fatores de preocupação estatal que na prática toma forma através de ações regulatórias de agências tal como a Agência Nacional de Energia Elétrica (ANEEL). Para assegurar a qualidade da distribuição de energia pode-se destacar o uso pela ANEEL de dois indicadores o DEC (Duração Equivalente de Interrupção por Unidade Consumidora), e o FEC (Frequência Equivalente de Interrupção por Unidade Consumidora), que medem, respectivamente, a duração e frequência de interrupções de energia (6).

Como parte integrante do sistema de distribuição, as concessionárias são responsáveis pela redução da tensão das linhas de transmissão. Essa conversão ocorre em subestações que são instalações que agrupam o conjunto de equipamentos necessários à execução de sua função. A manutenção e contínuo melhoramento são de responsabilidade das concessionárias e as faltas e prejuízos gerados ao consumidor decorrente do seu mau funcionamento, são monitorados e fiscalizados pela ANEEL através de análise de indicadores.

O Brasil conta com mais de 63 concessionárias de energia elétrica (6), e dentre elas a Companhia de Eletricidade Paranaense (COPEL) resolve por investir em um projeto de pesquisa de caráter experimental para o aumento da confiabilidade dos planos de manutenção preditiva dos equipamentos que compõe suas subestações. Substituindo o processo manual de coleta de imagens termográficas de equipamentos para posterior análise de danos, por um sistema de geração e execução de rotas de inspeção adaptativas guiadas por um Veículo Aéreo Não Tripulado (VANT), a COPEL espera obter uma melhoria nos seus indicadores DEC e FEC, redução de gastos com deslocamento de equipes de inspeção, melhoria no planejamento dos planos de manutenção e aumento da segurança dos colaboradores em operações de campo em subestação.

1.1 Contextualização e motivação

Este Projeto de Fim de Curso (PFC) está contextualizado no Projeto PD-02866-0015/2019 de título Monitoramento Inteligente de Falhas em Equipamentos com Uso de Termografia e VANTs, que está em andamento na Fundação Centros de Referência em Tecnologias Inovadoras (CERTI), sob a demanda da Companhia de Eletricidade Paranaense (COPEL). O projeto entre CERTI e COPEL tem por objetivo realizar o monitoramento de falhas em equipamentos elétricos em subestações de energia por meio de termografia e algoritmos inteligentes de processamento de imagens.

O desenvolvimento e resultados apresentados neste PFC compreendem as demandas de um dos objetivos específicos do projeto supracitado. Esse objetivo específico é o:

- Desenvolvimento e teste o do controle adaptativo e autônomo para uso de VANT associado a câmeras térmicas no monitoramento de equipamentos elétricos.

Assim, este PFC contribui com o desenvolvimento de um sistema experimental capaz de gerar e controlar a execução de rotas adaptativas, livres de colisão, em um VANT de inspeção de forma autônoma.

1.2 Justificativa

A implementação desta solução permite a o monitoramento de baixo custo de equipamentos de uma subestação e a possiblidade de modificação e avaliação de novas rotas de inspeção caso novos equipamentos ou requisitos sejam alterados/adicionados pelo cliente.

Em alternativa à inspeção manual, onde o operador está na subestação controlando o VANT através de um controle remoto, a solução proposta também visa aumentar a segurança diminuindo a exposição do trabalhador aos riscos, como por exemplo, proximidade a linhas energizadas.

Devido a falhas em equipamentos a COPEL gastou cerca de R$ 85 milhões em multas devido à análise de indicadores como DEC e FEC. Com os planos de manutenção melhorados pela repetitividade dos resultados obtidos por análises de fotos tiradas durante a execução das rotas adaptativas os indicadores DEC e FEC tendem diminuir.

O contratante possui cerca de 183 subestações e o planejamento de manutenção demanda profissionais dedicados a gerar fotos dos equipamentos em cada uma delas. O custo para cada hora de trabalho é de R$ 83 e a duração em média de uma inspeção, somada ao deslocamento, é cerca de 4 horas. Por motivos de segurança dois profissionais são necessários.

Considerando os valores citados, as 8 horas (2 profissionais) para cada uma das 183 subestações impacta em um custo total de R$ 121.500,00. A redução de tempo de inspeção esperada após a implementação do sistema autônomo é de pelo mínimo 50 %, incorrendo, em uma economia de R$ 60.750,00.

1.3 Trabalhos relacionados

O uso de VANTs para inspeção no setor elétrico ainda é pouco difundido, porém seu horizonte de aplicação é promissor para área de geração, transmissão e distribuição de energia, alcançando, segundo estudos da PwC, um mercado de até

US$ 9.46 bilhões (7). Refletindo esta tendência, projetos com o uso de VANT principalmente para verificação de linhas de transmissão, usinas eólicas e painéis solares lideram as primeiras tentativas de aproveitamento do uso de VANTS.

Segundo Rangel, Kienitz e Brandao (8) os métodos tradicionais de inspeção de linhas de transmissão consistem no sobrevoo da linha com o uso de um helicóptero ou da patrulha com veículos terrestres no qual uma equipe acompanha a linha de transmissão do solo, segundo o estudo ambos os métodos apresentam problemas. O uso de helicóptero aproxima o operador da linha viva e sua performance depende principalmente das condições climáticas e de terreno onde as linhas se encontram, expondo os operadores a um grande risco. A patrulha terrestre é limitada, pois grande parte das linhas de transmissão ficam em regiões de difícil acesso.

O trabalho de Rangel, Kienitz e Brandao consiste na utilização de um VANT equipado com câmeras de vídeo e dispositivos de telemetria e controle para o monitoramento de linhas de transmissão de alta tensão. A operação do VANT é realizada em uma base de comando equipada com laptop, dispositivos receptores de frames de vídeo, telemetria, *joystick*, dispositivos de visualização e softwares específicos. O *software* de navegação desenvolvido, atualiza a interface em tempo real com informações de voo, latitude, longitude e imagens de câmera. Baseado nas informações coletadas o *software* permite a criação de rotas em três dimensões posteriormente usadas para inspeção das linhas. A interface do sistema é realizada na tela de um *notebook* ou utilizando um capacete equipado com monitores LCD.

Ainda sobre inspeção de linhas de transmissão Deng, Wang, Huang, Tan e Liu (9) exploram as vantagens no aumento de eficiência no uso de múltiplos VANTs e um modelo de comunicação multiplataforma para inspeção de linhas de transmissão. Os vários VANTS servem de fonte de imagens de curta e longa distância da linha que são transmitidas em *realtime* para uma estação de controle para o sistema de navegação e para um escritório onde são realizadas a análise das imagens.

Para a operação de inspeção é proposto a união das capacidades de diferentes tipos de VANTs como os de asa fixa que entregam tempos longos de voo porém com imagens não tão detalhas quanto a VANTs multirotores, que obtém imagens mais detalhadas devido a flexibilidade de controle que permite pairar próximo

a pontos de interesse, mas com duração de bateria limitada à curtos intervalos de tempo.

É proposto também um novo modelo de comunicação entre os diferentes tipos de VANT e a estação de comando, aumentando a capacidade de comunicação a longa distância, permitindo a inspeção de linhas de transmissão mais longas. Segundo levantamento, o limite para controle e comunicação com VANT multirotores é limitado a linha de visão, que na prática está entre 100 m e 1 km enquanto os de asa fixa costumam não permitir a comunicação em *realtime*, neste último a configuração da missão costuma ser possível apenas antes do voo. A proposta utiliza um VANT que permanece conectado a alimentação e instalado em um ponto intermediário da rota de inspeção fornecendo um *link* de comunicação para todos os outros VANTs durante a inspeção. Esta técnica é utilizada quando os VANTs fogem a linha de visão, o que pode ser frequente em terrenos irregulares.

O trabalho ainda propõe uma solução para transmissão das imagens de inspeção para os escritórios especialistas. Em contrapartida a prática anterior de *download* das imagens em cartões de memória, transporte e finalmente análise em escritórios, localizados a centenas ou milhares de quilômetros do local de inspeção, as imagens de alta qualidade são transmitidas via satélite, sendo possível o envio simultâneo de 4 canais de vídeos de alta definição.

Frente a crescente implementação de plantas fotovoltaicas ao redor do mundo Oliveira, Aghaei, Madukanya, Nascimento e Rüther avaliam a aplicabilidade comercial do uso simultâneo de técnicas de inspeção de painéis fotovoltaicos por meio de termografia infravermelha (IRT) e o uso de VANT (aIRT) para cobrir grandes áreas referentes aos parques fotovoltaicos (10).

Seu estudo expande o atual método de coleta de imagens realizado em câmeras infravermelhas com auxílio de sistemas de elevação. Este método consome grandes intervalos de tempo e é intensivo em relação ao trabalho exigido para a coleta de imagens, portanto costuma ser praticado em setores do parque selecionados de forma aleatória. No caso citado, o estudo foi motivado por um evento chamado tsunami meteorológica que comprometeu o funcionamento de células fotovoltaicas de um parque inteiro, a coleta de dados manual tornaria o processo de verificação e posterior manutenção extremamente laborioso.

Na solução foram utilizados uma câmera termográfica acoplada a um VANT DJI Phantom 3 Professional. O VANT é controlado remotamente e transmite as

imagens para estação de comando via uma antena Cloverleaf de 5.8 GHz para posterior análise por meio de técnicas de termografia.

1.4 Objetivo geral

Desenvolver um sistema para controle de rotas autônomas e adaptativas para VANT.

1.5 Objetivos específicos

- Desenho da arquitetura conceitual do sistema.
- Mapeamento das alternativas de implementação dos módulos do sistema.
- Implementação de um ambiente virtual com VANT simulado na estação de comando.
- Implementação do módulo de *motion planning* no ambiente virtual na estação de comando.
- Implementação do módulo de coleta de pontos na estação de comando
- Implementação do módulo de geração e execução de trajetória e controle adaptativo na estação de comando.
- Analisar o funcionamento da execução de trajetórias e controle adaptativo do VANT em ambiente de simulação na estação de comando.
- Implementação da comunicação entre VANT e estação do comando.
- Implementação do ambiente de simulação no VANT.
- Analisar o funcionamento do controle adaptativo no ambiente de simulação no VANT.
- Analisar o funcionamento do controle adaptativo no VANT em aplicação real.
- Análise sobre melhorias em VANT utilizado.

1.6 A Empresa

Como instituição de Ciência, Tecnologia e Inovação, a Fundação Centros de Referência em Tecnologias Inovadoras – Fundação CERTI nasceu direcionada para a pesquisa tecnológica aplicada, num contexto em que o Brasil demandava saltos de qualidade e desenvolvimento de know how próprio e inovador especialmente no campo da informática e das tecnologias de ponta, incluindo particularmente a automação industrial (11).

A CERTI, em seu histórico de 35 anos de existência, atua em prol do desenvolvimento econômico, sustentabilidade, inclusão social e educação, promovendo a inserção estratégica de tecnologias voltadas para a melhoria da qualidade de vida da sociedade brasileira. Para tanto, atua como entidade realizadora de ações de Inovação, desenvolvendo produtos, processos industriais, *software*, soluções de serviços e novos modelos de negócios.

A abrangência dos trabalhos compreende desde planos de negócio de novos produtos e tecnologias ao processo de lançamento no mercado, passando por prototipagem, teste, certificação, fabricação de lotes piloto, desenvolvimento de fornecedores e demais etapas da criação de soluções completas e qualificadas no mercado, atuando em diversos setores, entre eles: Energia Sustentável, Economia Verde, Tecnologia da Informação e Comunicação, Convergência Digital e Desenvolvimento de produtos, Metrologia e Automação.

Atualmente a CERTI é composta por diversos centros que operam diferente áreas de competência, tais centros são conhecidos como Centros de Referência. São eles:

- CDM - Centro de Convergência Digital e Mecatrônica
- CMI - Centro de Metrologia e Instrumentação
- CPC - Centro de Produção Cooperada
- CEI - Centro de Empreendedorismo Inovador
- CELTA - Centro Empresarial para Laboração de Tecnologias Avançadas
- CEV - Centro de Economia Verde
- CES - Centro de Energia Sustentável

O Centro de Metrologia e Instrumentação (CMI), onde este trabalho foi desenvolvido, é composto por dois conjuntos laboratoriais referentes as áreas de metrologia e instrumentação. Este trabalho foi desenvolvido no laboratório de instrumentação do CMI, que desempenha atividades como:

- Desenvolvimento de sistemas de medição e bancadas de teste: desenvolvimento de produtos voltados para o controle de qualidade e de suporte a engenharia de produtos.
- Serviços de planejamento e execução de testes especiais: com foco em placas eletrônicas é realizado o planejamento, execução e análise de testes funcionais para produtos mecatrônicos.
- Desenvolvimento de soluções para monitoramento de ambientes: integração de sensores de variáveis ambientais como: velocidade do vento, temperatura, umidade do ar, pressão atmosférica, radiação solar e nível para instalação de sistemas de monitoramento em regiões remotas.

1.7 Abordagem metodológica

A metodologia de desenvolvimento a ser utilizada para este projeto segue modelo utilizado pela CERTI, denominado Sistema de Desenvolvimento de Soluções (SDS). Esse modelo é aplicável de forma abrangente a projetos de desenvolvimento de sistemas e produtos.

O SDS divide um projeto em quatro possíveis fases, apresentadas a seguir:

- Fase 0. Mapeamento e elaboração do conceito básico da solução: esta etapa tem por objetivo levantar requisitos e mapear soluções e cenários de uso, para desenvolvimento de um conceito de produto e definição da arquitetura a ser desenvolvida.

- Fase 1. Desenvolvimento da solução conceitual: esta etapa tem por objetivo caracterizar a arquitetura detalhada da solução escolhida, agregando informações

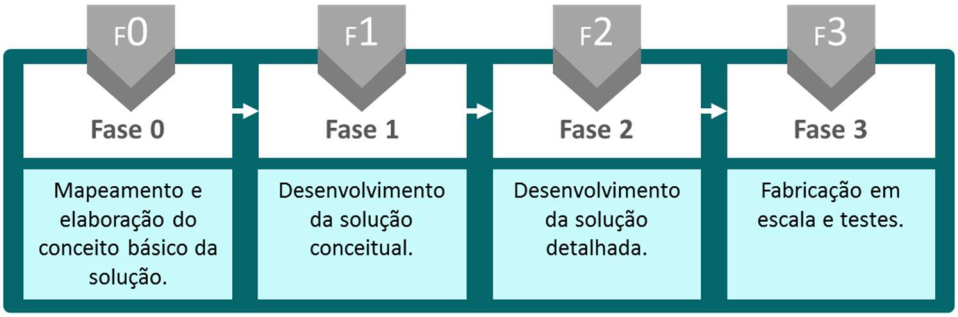
detalhadas sobre o conceito e modelo, potenciais provedores, mercado e demais fatores envolvidos. O objetivo é desdobrar o conceito de solução até o nível que permita ser desenvolvido um produto ou solução.

- Fase 2. Desenvolvimento da solução detalhada: esta etapa tem por objetivo a elaboração do projeto de engenharia detalhado da solução, incluindo *hardware*, *software*, aspectos de usabilidade, bem como materiais e processo básico para sua fabricação. Nesta etapa são realizadas atividades de *design* e construção de produto.

- Fase 3. Fabricação em escala e testes: esta etapa tem por objetivo a fabricação em escala do produto.

O projeto proposto realizará de forma, inteira ou parcial, somente as fases pertinentes ao objeto desta proposta.

Figura 1 - Fluxograma da metodologia



Fonte: Arquivo pessoal

1.8 Estrutura do documento

Este trabalho está divido em 6 capítulos, dos quais o conteúdo é descrito a seguir. Este capítulo de introdução apresenta o assunto, objetivos, justificativas, um estudo sobre o desenvolvimento de trabalhos relacionados e descreve a empresa e o departamento onde este trabalho foi realizado.

No capítulo 2 são descritos os conceitos e embasamento teórico sobre os tópicos necessários para a compreensão das tecnologias utilizadas ao longo do desenvolvimento deste trabalho.

No capítulo 3 são apresentadas as arquiteturas conceitual e de tecnologias acompanhas das justificativas sobre a decisão de uso.

No capítulo 4 são exibidos e apresentados os diagramas de desenvolvimento de *software*, requisitos de *software*, funções dos principais módulos do sistema, os códigos fonte dos módulos desenvolvidos e as interfaces de comando do operador.

O capítulo 5 é composto por uma análise dos resultados da execução de testes sobre o sistema gerenciador de rotas.

No capítulo 6 se conclui sobre os resultados alcançados, se analisa os limites da aplicação e se propõe a continuidade da pesquisa em forma de desenvolvimento de novos trabalhos na área.

2 FUNDAMENTACÃO TEÓRICA

Neste capítulo são descritos os conceitos necessários para a compressão do conteúdo envolvido nas atividades de desenvolvimento do sistema.

2.1 ROS

O ROS (*Robot Operating System*) é um sistema operacional para aplicações em robótica, que reúne ferramentas, bibliotecas e convenções que têm por objetivo dar suporte ao desenvolvimento de aplicações de alta complexidade e robustez. O propósito do seu desenvolvimento é baseado no desenvolvimento colaborativo entre laboratórios ou desenvolvedores entusiastas, no qual a *expertise* dos colaboradores é disseminada em uma aplicação de nível mundial e de alta confiança (12).

A seguir serão descritos os conceitos básicos de funcionalidades do ROS utilizados ao longo do desenvolvimento do trabalho.

2.1.1 Nodos

Nodos são programas que realizam operações modulares. Em aplicações ROS, na qual a modularidade é característica, é intrínseco encontrar muitos nodos trabalhando em conjunto compreendendo a solução final. Cada nodo opera funções que dependem de informações e dados provindos de diferentes programas, para isso comunicam-se entre si através de serviços, tópicos ou *actionlibs*. Por exemplo, em uma aplicação de navegação de um robô móvel é possível que um nodo seja responsável pelo controle das rodas, outro pelo controle de posição, outro pela leitura do sensor laser, outro pela detecção de obstáculos, outro pelo sistema de localização, e outro pela interface de usuário (13).

Até o presente momento nodos podem ser escritos em Java, C++ e Python. Com esta funcionalidade fabricantes de sensores e atudores, ou desenvolvedores de algoritmos podem integrar a funcionalidade de seus desenvolvimentos utilizando as API´s (*Application Programming Interface*) fornecidas nas linguagens acima mencionadas.

2.1.2 Pacotes

Para ser distribuído um nodo precisa pertencer a um pacote. Um pacote é um conjunto de nodos, bibliotecas, *softwares* proprietários, arquivos de configurações, tipos de dados, tabelas e tudo que é essencial para o funcionamento de uma determinada função. A ideia é de que o que estiver reunido e denominado sob um determinado pacote realize funcionalidades similares, que exista uma hierarquia de suporte para uma funcionalidade superior entre seus integrantes (14).

A estrutura de um pacote é padronizada, e é necessário que durante o desenvolvimento de uma aplicação se siga com atenção a estrutura determinada pelo *framework* ROS para que o código seja compilado utilizando a ferramenta *catkin_make*. Exemplos de pacotes são:

- *Moveit*: pacote utilizado para *path planning* de braços robóticos e robôs móveis.
- *Depthimage_to_scan*: pacote utilizado para conversão de nuvem de pontos em tipo de dados *scan*.
- *Navigation*: pacote utilizado para navegação especializada em robótica móvel.
- *Geonav_transform*: pacote utilizado na conversão de coordenadas geográficas para coordenadas cartesianas.

A comunidade ROS encoraja o desenvolvimento e modificação dos códigos fornecidos em pacotes. Essa prática intensifica o uso, publicação e revisão de *bugs* e correções. Alguns desses pacotes são extensivamente testados e com robustez comprovada para aplicações em robótica.

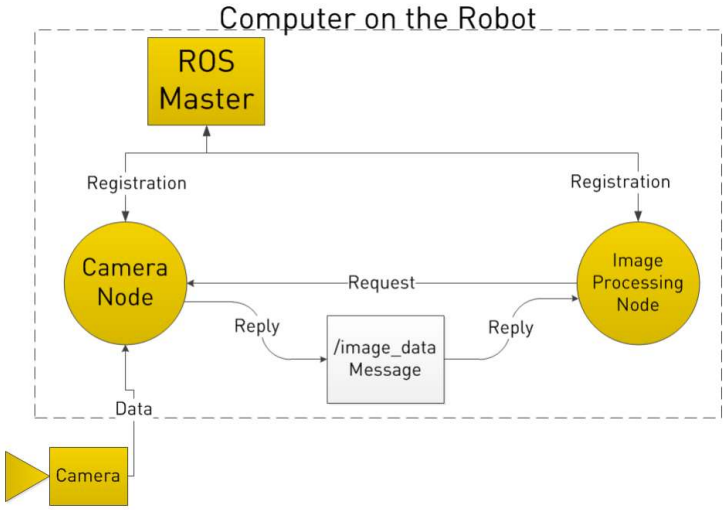
2.1.3 ROS *master*

O ROS *master* é o programa *core* que realiza a comunicação entre os diferentes nodos que permeiam a solução. ROS *master* pode ser pensando como um comunicador. Todos os nodos que rodam na camada ROS registram-se no *master*

para poder operar, no ato de registro os nodos armazenam em uma tabela, localizada no master, suas intenções de leitura e escrita (15).

O *master* atua como um comunicar/distribuidor que no instante de recebimento de um dado o distribui para quem o está requisitando. Como exemplo, pode-se citar a conversão de dados do tipo nuvem de pontos para o tipo *scan*. Na Figura 2 os nodos *Camera Node* e *Image Processing Node* realizam o registro de intenções de escrever e ler a informação contida em um espaço de memória chamado */image_data*. O nodo *Camera Node* fornece os dados brutos da câmera nesse espaço de memória enquanto o nodo *Image Processing Node* aplica um algoritmo para realizar a conversão de dados para o tipo *scan*. O tipo de dados *scan* pode ser utilizado para aplicações de mapeamento e detecção de obstáculos. A Figura 2 exibe os registros dos dois nodos com o ROS *master*.

Figura 2 - Registro com ROS Master



Fonte: https://robohub.org/ros-101-intro-to-the-robot-operating-system/

2.1.4 Tópicos

Tópico é a forma mais simples de troca de mensagem disponível em ROS, é conhecida como comunicação 1-para-muitos. Funcionam como *buses* virtuais que desacoplam o produtor do consumidor. Os nodos ao registrarem suas intenções de escrita (*publishers*) e leitura (*subscribers*), denominam um nome dado ao espaço de memória onde deverão ser publicadas determinadas informações. Pode-se entender esse espaço de memória com um tópico (16).

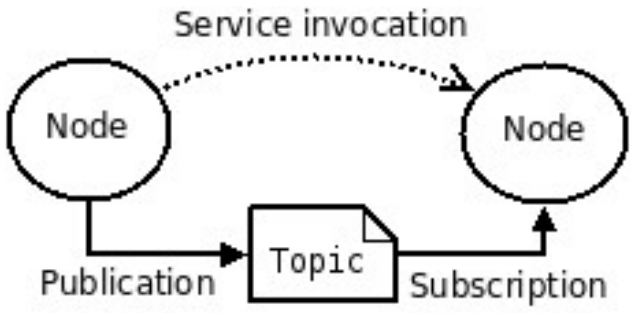
O ROS *master* gerencia as publicações, enviando os dados aos nodos que a solicitam quando um outro nodo fornece o dado. Os tópicos são um mecanismo chave de comunicação em ROS, pois promovem a modularidade entre os nodos. Utilizando um tópico o nodo que consume um dado não tem como saber “quem” está enviando dados para a realização da função, assim como o nodo que publica o dado não conhece onde está sendo realizada a aplicação de seus dados. O formato de mensagens utilizado nos tópicos é definido em arquivos de tipos nos pacotes.

2.1.5 Serviços

Serviço é a forma de comunicação 1-para-1 mais simples disponível no ROS. Em alternativa à dinâmica dos tópicos, os serviços entregam a funcionalidade de pergunta e resposta. Essa propriedade é utilizada quando necessária a confirmação de execução de algum procedimento. O formato de mensagens utilizado nos serviços é definido em arquivos de tipos nos pacotes (17).

Na sua inicialização, um nodo registra no *master* quais são os serviços que fornece, caso forneça algum. Os serviços dependem da aplicação a qual o nodo pertence. Por exemplo, em pacotes de SLAM *(Simultaneos Localization and Mapping)* é comum haver um nodo fornecendo um serviço de reconfiguração do mapa. Esse serviço, quando chamado, reinicializa os parâmetros da aplicação promovendo uma nova localização do robô em um mapa de obstáculos. Na Figura 3 é exibido dois nodos trocando mensagens por serviços e tópicos.

Figura 3 - Serviços e Tópicos



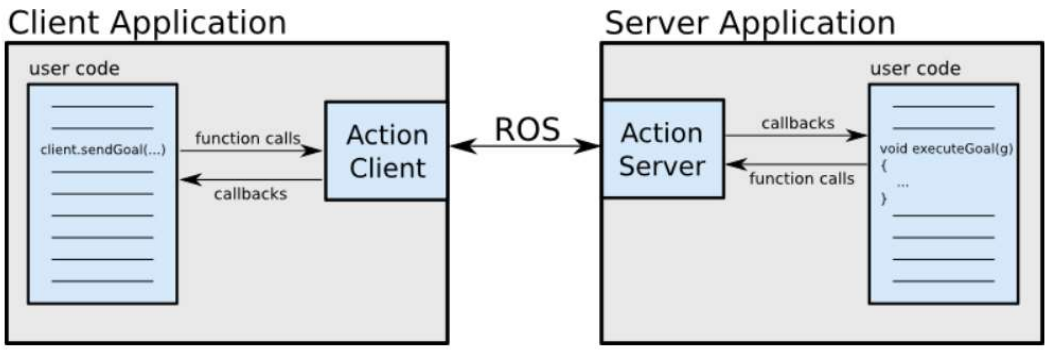
Fonte: http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)Concepts.html

2.1.6 Actionlibs

Actionlib é a forma de serviço mais complexa utilizada em ROS. Possui as mesmas caraterísticas que motivam a necessidade do uso de serviços, porém entregam um acompanhamento de estado necessário para aplicações críticas. Seu funcionamento é no formato cliente-servidor.

O *actionlib* pode ser compreendido como uma interface. Nessa interface um *actionlib client* pode realizar uma requisição (*goal*) ao *actionlib server* como uma posição para um robô móvel. Durante a execução da operação com dados de entrada fornecidos pelo cliente, o servidor mantém o cliente atualizado utilizando mensagens (*feedback*), no caso do robô móvel esta mensagem poderia ser a atual posição do robô ou se há um obstáculo ou não. Ao término da execução um resultado é entregue através de mensagem (*result*), este resultado inclui informações referentes ao sucesso da requisição do cliente, no caso acima poderia ser se o robô alcançou a posição ou foi impossibilitado devido a alguma condição de obstáculo instransponível (18). A Figura 4, a seguir apresenta a interface entre cliente e servidor usando *actionlib*.

Figura 4 - Interface de Actionlib



Fonte: http://wiki.ros.org/actionlib

Neste sentido, *actionlibs* são comumente fornecidos em aplicações que envolvem movimento de partes físicas de robôs, pois promovem a possiblidade de parada caso alguma condição perigosa seja detectada.

2.1.7 Outros módulos ou funcionalidades

A seguir são descritos de maneira sucinta algumas funcionalidades de algumas nomenclaturas utilizadas neste trabalho:

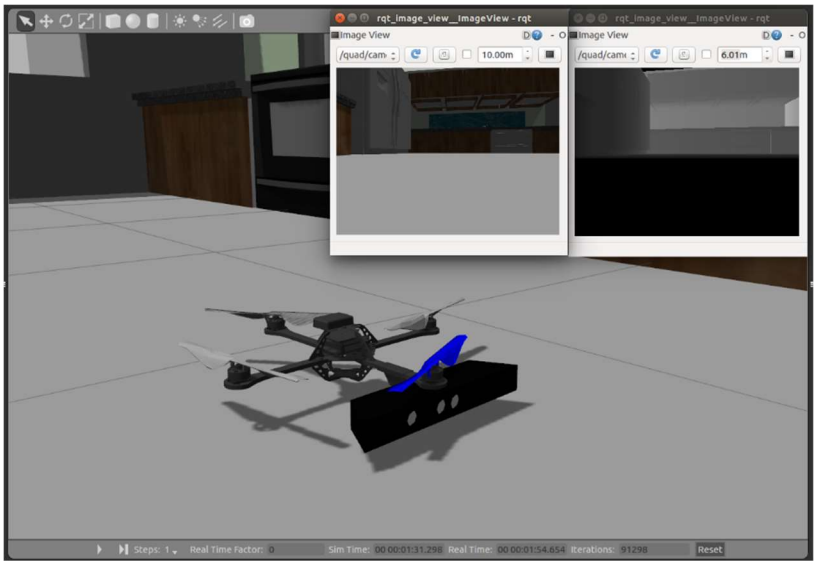
- *Launch file:* arquivo XML que serve para iniciar nodos e configurar parâmetros no ROS.
- Inserção de pacotes: Neste trabalho existem 2 tipos de pacotes, aqueles instalados a partir do comando "*apt-get"* do Ubuntu e outros pacotes compilados do código fonte.

2.2 Gazebo

Gazebo é o *software* de simulação que acompanha o ROS, servindo como suporte à criação de um ambiente de teste com simulação de física, sensores, atuadores e robôs. O Gazebo é uma ferramenta de simulação especializada para robótica. Através do seu uso é possível realizar a avaliação de algoritmos de *path planning*, navegação, execução de rotas, treino de algoritmos de inteligência artificial e testes de múltiplos robôs em ambientes *indoor* e *outdoor* (19).

Através de *plugins* é possível gerar dados simulados como coordenadas de GPS (*Global Positioning System*), iGPS (*indoor* GPS), *feedbacks* de *encoders*, variação térmica de objetos, entradas digitais dentre outros. Alguns *plugins* estão disponíveis em pacotes ROS. A Figura 5 apresenta um VANT no ambiente Gazebo, uma câmera Kinect está acoplada ao VANT.

Figura 5 - Simulação no Gazebo



Fonte: https://www.wilselby.com/research/ros-integration/model-dynamics-sensors/

Seu funcionamento é realizado através de dois programas principiais: um responsável por realizar os cálculos referentes à simulação física e outro por exibir a simulação ao usuário. O funcionamento da simulação é separado da visualização, podendo o usuário optar por não mostrar a simulação caso necessário economizar processamento.

Arquivos de objetos 3D podem ser importados no Gazebo desde que possuam o formato COLLADA. Essa funcionalidade é bastante proveitosa, do ponto de vista que os robôs podem ser testados em seus ambientes de produção virtuais antes mesmo de serem comprados ou construídos.

2.3 Sampled-based Motion planning

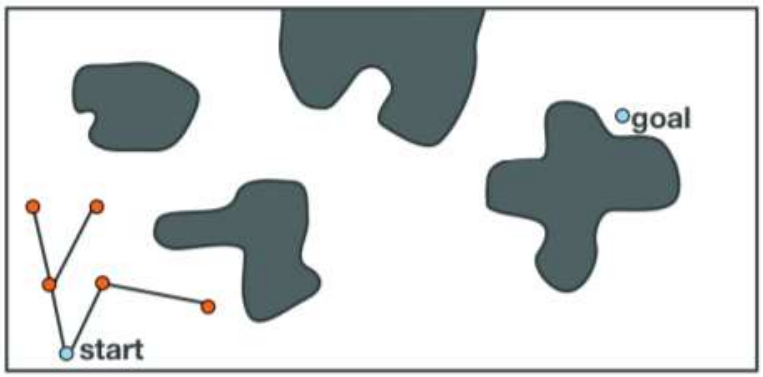
Para exemplificar os conceitos sobre *sampled-based motion planning* consideremos um problema clássico no qual um objeto 3D precisa realizar um deslocamento de um ponto A para um ponto B ao mesmo tempo que existem obstáculos em sua vizinhança que devem ser evitados. O movimento do objeto possui 6 graus de liberdade, 3 para movimentos nos planos coordenados (x, y, z) e 3 para representar a rotação em torno dos eixos (*roll*, *pitch*, *yaw*). Para o entendimento da técnica de *motion planning* os seguintes conceitos são fundamentais:

- *Workspace*: espaço físico onde o objeto pode operar.
- *State space*: representa todas as possíveis configurações para o objeto dentro do *workspace*, um único ponto no espaço representa um estado.
- *Free State Space*: parcela do *state space* que representa estados livres de obstáculos.
- *Path*: conjunto de estados que representam uma solução para o problema de deslocamento de um ponto inicial para um ponto final obedecendo a restrições.

Ao contrário dos algoritmos tradicionais, as técnicas de *sampled-based motion planning* não realizam uma análise contínua de todo o *state space.* Algoritmos de análise contínua, quando envolve elevado número de graus de liberdade, tende a consumir muito tempo de processamento devido ao número de possíveis soluções e complexidade referentes ao tamanho do *workspace* (20).

O *Sampled-based Motion planning* pode ser implementado de formas diferentes, neste trabalho serão abordados os algoritmos baseados em expansão de árvore de estados. Esses algoritmos operam a partir de um estado inicial dado (posição inicial do objeto), realizam a expansão de uma árvore de estados e, para cada novo estado uma verificação de restrições é realizada. Caso o caminho entre o estado antigo e novo obedeça às restrições, o estado é retido, caso contrário é descartado. Na Figura 6 pode-se observar o estado inicial, final e 2 ramos de expansão a partir do estado inicial.

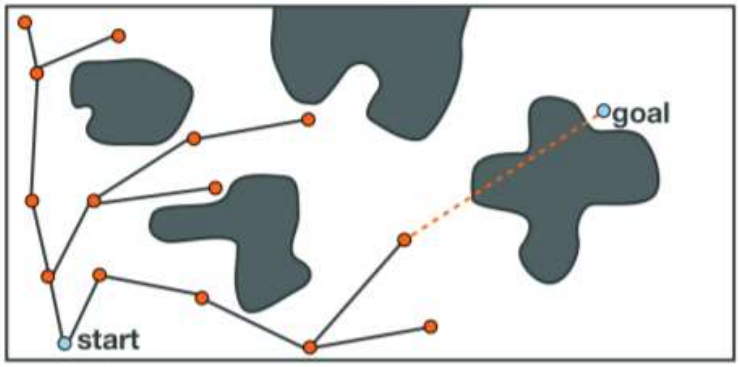
Figura 6 – Motion planning: *State space* inicial



Fonte: *Open Motion Planning Library: A primer*

Na Figura 7 pode-se observar uma condição em que o caminho definido entre o estado gerado e o anterior não obedece às restrições de colisão. Neste caso, o algoritmo descarta este estado e realiza uma nova tentativa de geração de estados.

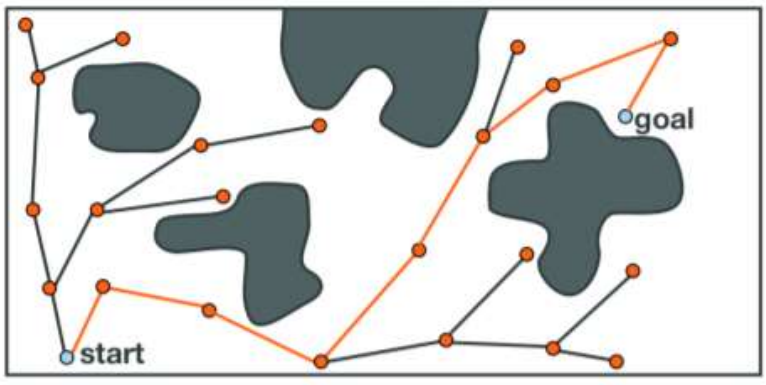
Figura 7 - Motion planning: estado descartado



Fonte: *Open Motion Planning Library: A primer*

Na Figura 8 pode-se observar que um caminho encontrado até o estado desejado. Observa-se também que ramificações sem resultado são geradas, porém esta técnica de *motion planning* requer menos memória que as tradicionais por não precisar de uma representação explicita do espaço de estados no decorrer de suas interações (20).

Figura 8 - Motion planning: goal state alcançado



Fonte: *Open Motion Planning Library: A primer*

A forma como a árvore é expandida e quantas ramificações possui depende do algoritmo utilizado, normalmente o algoritmo possui o nome da heurística utilizada na determinação de novos estados (20).

O pacote *MoveIt* disponível para ROS possui a biblioteca OMPL (*Open Motion Planning Library*), que possui a implementação de diferentes algoritmos que implementam a técnica de *sampled-based motion planning* (21).

2.4 Octomap framework

2.4.1 Octomap

Octomap é um *framework* que apresenta uma proposta para manter as vantagens dos métodos tradicionais incluindo a representação volumétrica do espaço como diferencial. O *framework* está disponível através de uma biblioteca em C++ e é acessível diretamente através do ROS via um pacote. (22).

É comum as aplicações na área de robótica utilizarem representações volumétricas do espaço onde executam suas funções. Um robô móvel autônomo quando executando função de navegação, precisa conhecer o ambiente onde a tarefa é executada, assim como um braço manipulador quando executando movimentos em sua célula de trabalho. Muitas das aplicações em robótica requerem três aspectos em relação às características de uma representação 3D:

- Representação probabilística: as medições utilizadas para criação de um modelo 3D são oriundas de sensores, e estes possuem uma incerteza associada aos seus resultados. Um modelo 3D robusto deve levar em consideração as incertezas de múltiplas medições ou até mesmo de mais de uma fonte de dados para estimar a real localização dos objetos na cena.
- Áreas não mapeadas: a representação de áreas não mapeadas é importante para que o robô não utilize áreas desconhecidas para estabelecer rotas, caso contrário uma rota em ambiente desconhecido poderia levar à colisão.
- Eficiência: a velocidade de acesso e quantidade de memória ocupada pelos mapas deve ser reduzida ao máximo para que o tempo de acesso

seja reduzido e que a aplicação não fique restrita por requerer no robô uma elevada capacidade de armazenamento.

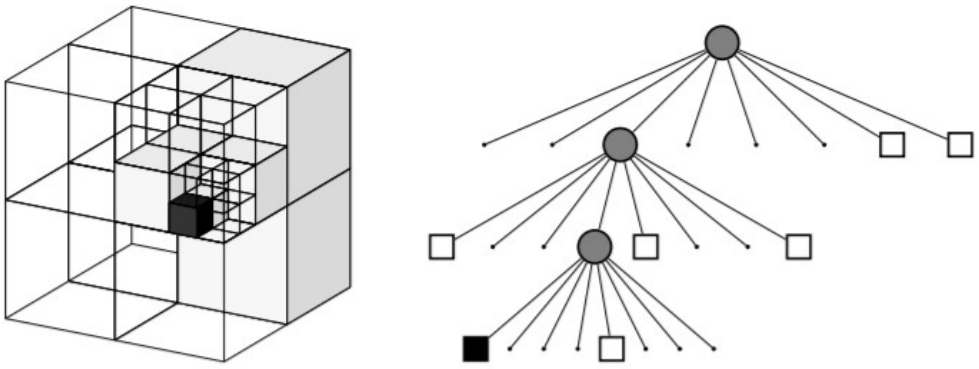
Alguns dos métodos de mapeamento 3D costumam apresentam problemas em algum desses aspectos, modelos criados diretamente a partir de nuvem de pontos ocupam bastante espaço de memória e em sua maioria não permitem representar áreas não descobertas.

2.4.2 Representação do *Octomap*

A representação de um *Octomap* é baseada em *Octotree*. *Octotree* é uma estrutura hierárquica do tipo árvore que representa subdivisões de um espaço 3D. Cada nó da árvore representa o espaço contido em um volume cúbico, os nodos são conhecidos como *voxel*.

A expansão da árvore acontece a partir da expansão do primeiro *voxel* em 8 subdivisões de igual volume. Os nodos gerados são novamente subdivididos em 8 partes e este processo acontece recursivamente até que um *voxel* mínimo seja alcançado. O tamanho desse *voxel* mínimo representa a resolução do *Octomap*. Na Figura 9 pode-se ver um exemplo de *Octotree* com a indicação sobre estado ocupado (volume em preto) (22).

Figura 9 - Expansão de uma Octotree

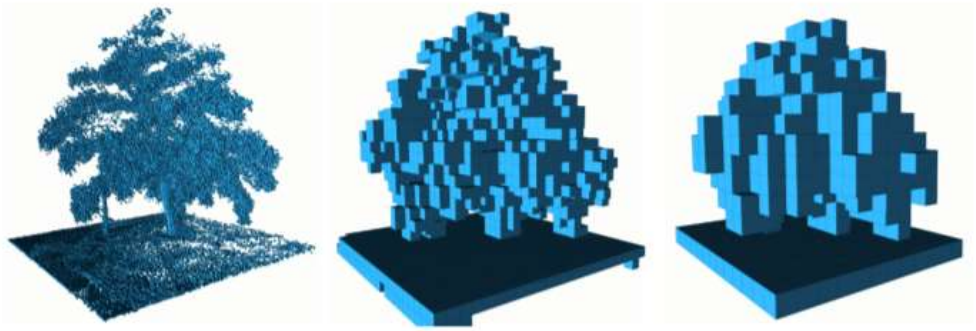


Fonte: *Octomap*: An Efficient Probabilistic 3D Mapping Framework Based on Octrees

Limitando a profundidade da expansão da árvore é possível obter representações de um objeto em diferentes resoluções. Um exemplo de Octotree é

apresentado na Figura 10, na qual é possível visualizar o mesmo objeto sendo representado em diferentes resoluções (0,08 m, 0,64 m e 1,28 m respectivamente). A resolução representa o comprimento da aresta que compõe os *voxels*.

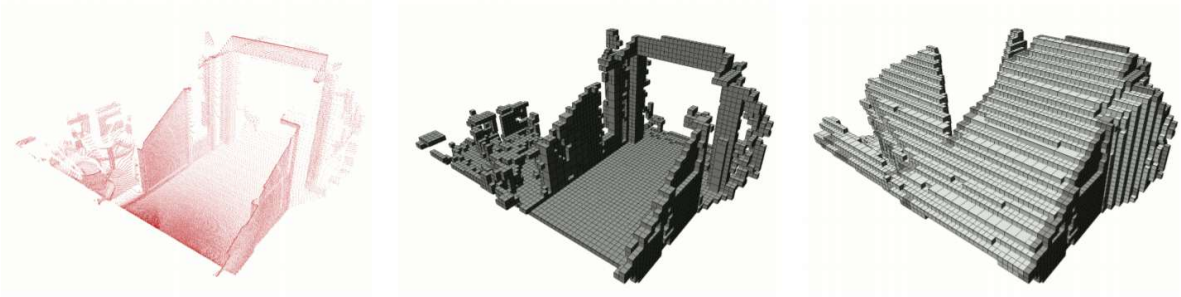
Figura 10 - Representação de resoluções diferentes de um *Octomap*



Fonte: *Octomap*: An Efficient Probabilistic 3D Mapping Framework Based on Octrees

O *Octotree* apresenta representação das áreas ocupadas e livres. Na Figura 11 é possível observar uma representação de uma escadaria. Os dados brutos são de uma câmera que fornece dados de nuvem de pontos.

Figura 11 - Áreas ocupadas e livres de um *Octomap*



Fonte: *Octomap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees*

Ainda na Figura 11 podemos observar os dados brutos (à esquerda), a área ocupada (centro) e a área livre e ocupada (em preto e à direita).

2.5 Sistema de referência global e projeções de mapa

A seguir serão detalhados aspectos ligados a forma de representar a posição de um objeto sobre a superfície da Terra e como representar essa posição em um mapa 2D, conivente para visualização e criação de rotas.

2.5.1 Sistema de referência geométrico

Para aplicações que requerem a geolocalização é comum a utilização das coordenadas fornecidas pelo sistema GPS ou o GLONASS (*Globalnaya navigatsionnaya sputnikovaya sistema)*. Este trabalho escolheu o sistema GPS que é mantido pelo governo americano. O sistema GPS utiliza uma série de padrões e parâmetros para representar a posição de um objeto sobre a superfície da Terra. A essa série de características, regras e parâmetros dá-se o nome de sistema de coordenadas geográficas.

Um sistema de coordenadas geográficas utiliza a superfície de uma esfera tridimensional para estabelecer localização sobre a superfície da Terra. Além disso, inclui em suas definições unidade angular de medida, um meridiano primário, e um ponto de referência baseado em um esferoide (23). Um ponto é referenciado utilizando ângulos chamados de latitude e longitude, que são medidos a partir do centro da Terra até o ponto de interesse.

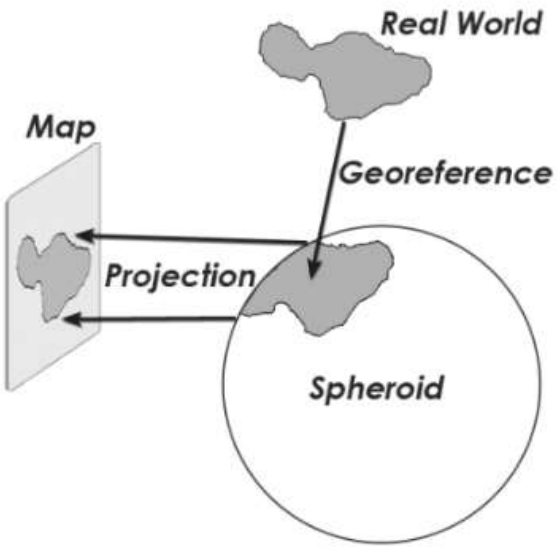
O sistema de referência geográfica utilizado pelo GPS chama-se WGS84 (*World Global System* 1984), foi desenvolvido pelo departamento de defesa dos EUA (24) e possui as seguintes características (25):

- Faixa de latitude: -90 ° a 90 °.
- Faixa de longitude: -180 ° a 180 °.
- Meridiano primário: Greenwich.
- Coordenação centrais: intercessão com Equador (0,0).

2.5.2 Projeção de mapas

Para representar de forma mais conveniente a superfície da Terra, objetivando avaliar distâncias, são utilizadas técnicas de projeção para representação em duas dimensões (26). Na Figura 12 pode-se ver a representação de uma projeção genérica.

Figura 12 - Mapa projetado



Fonte: What are Map Projections?

Como é impossível representar perfeitamente uma superfície 3D em uma 2D (26) sempre haverá distorções, porém existem diversas técnicas de projeção, cada qual com suas vantagens e desvantagens. Exemplos desta projeção são a Universal Transversa de Mercator (UTM), projeção de Mercator e projeção de Peters. Nesse trabalho para realizar conversões entre coordenadas geográficas e cartesianas é utilizada uma biblioteca disponível para ROS de nome *geonav_transform.* Essa biblioteca realiza conversões entre os sistemas UTM e WGS84.

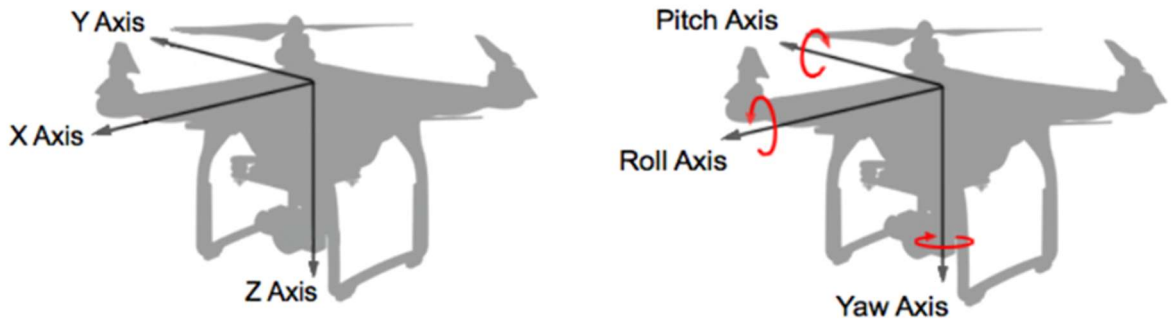
2.6 Dinâmica de voo

Para controle de um VANT, são necessários conceitos sobre o funcionamento desses equipamentos para entender seu funcionamento manual ou autônomo (27). Esses conceitos são referentes à família de VANT oferecidos pela DJI.

2.6.1 Sistema de coordenadas do corpo

Sistema de coordenadas do corpo é um sistema centrado no centro de massa do equipamento. O sentido positivo do eixo X é direcionado para frente do equipamento, o eixo Y a direita e Z para baixo. A rotação é descrita sob os mesmos eixos utilizando a regra da mão direita. Rotação em X, Y e Z são representadas respectivamente por *roll*, *pitch* e *yaw*. A Figura 13 traz a representação da translação e rotação.

Figura 13 - Translação e rotação com DJI



Fonte: DJI *Flight Control*

2.6.2 Sistema de coordenada global

Uma convenção popular em aplicações com VANT é a convenção NED (*North-East-Down*). Essa convenção alinha as direções X, Y e Z com as direções norte, leste e para baixo. A convenção com Z para baixo pode parecer pouco intuitivo, porém o sistema fica consistente uma vez que segue a regra da mão direita.

2.6.3 Orientação e voo

Para controlar o VANT são utilizadas as combinações ângulos *roll*, *pitch*, *yaw*.

- *Pitch*: O ângulo de *pitch* determina se o VANT está deslocando para frente ou para trás. Para frente os motores de trás giram mais rapidamente que os da frente, enquanto para trás os motores da frente giram mais rapidamente.
- *Roll*: O ângulo de *roll* determina como o VANT gira em torno do eixo X. Para inclinar a direita os motores da esquerda giram mais rapidamente,

enquanto para girar a esquerda os motores da direita giram mais rapidamente.

- *Yaw*: O ângulo *yaw* determina como o VANT gira em torno do eixo Z. Em aplicações com VANT metade dos motores giram em um sentido e a outra metade em outro sentido. Para girar no sentido horário os motores que assim estão configurados giram mais rapidamente, enquanto que para girar no sentido anti-horário os motores neste sentido giram mais rapidamente.

3 ARQUITETURA E TECNOLOGIAS

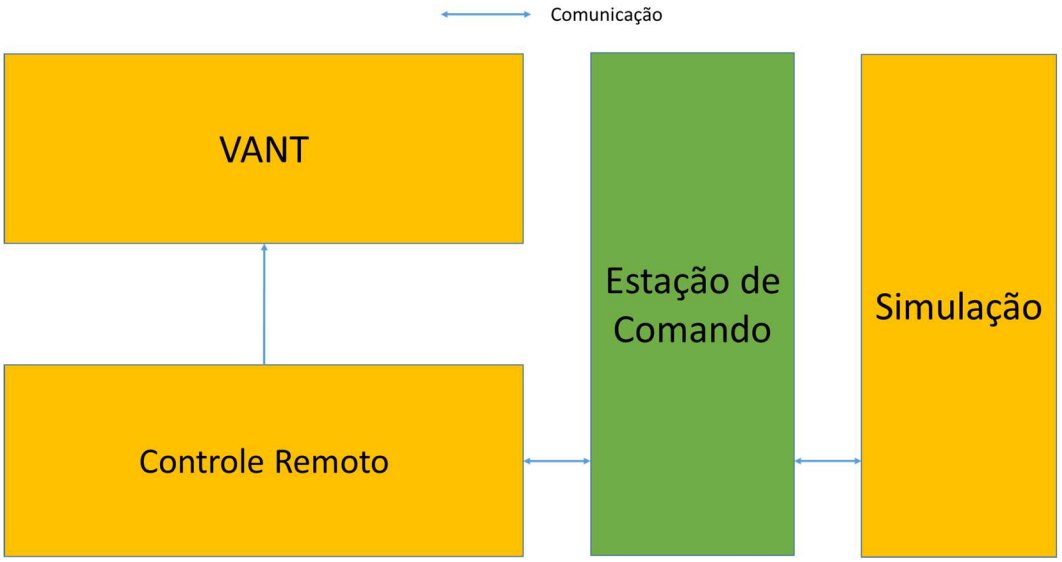
Este capítulo contém as descrições da arquitetura conceitual e das tecnologias escolhidas para atender as demandas do sistema. Na arquitetura conceitual, são apresentadas as funções básicas de cada módulo da solução, isto é, aquilo que cada componente deve desempenhar para que o sistema funcione de forma integrada. Em seguida são descritas as tecnologias utilizadas para atender as premissas básicas da arquitetura conceitual, assim como as suas justificativas de uso.

3.1 Arquitetura conceitual

De forma simples o sistema deve permitir o usuário navegar com o VANT em ambiente virtual com modelos de sólidos que representem o que se deseja inspecionar, e selecionar pontos de interesse e gerar um documento com os pontos coletados. O sistema deve ser capaz de a partir de pontos de interesse coletados gerar e enviar rotas livres de colisão para o VANT em um ambiente de simulação. O caminho que as rotas podem tomar pode variar dependendo de uma indicação externa ao sistema. A inspeção completa pode ser dividida em trechos que compreendem a trajetória entre os pontos de interesse. Ao término da execução de cada trecho as rotas são enviadas ao VANT virtual, essas são enviadas para o VANT real que às executa em paralelo.

A arquitetura conceitual tem por objetivo representar módulos que atendam o descrito no parágrafo anterior de forma simples e direta. Na Figura 14 podemos observar uma proposta para a arquitetura conceitual.

Figura 14 - Arquitetura conceitual



Fonte: Arquivo pessoal

A compressão do modelo conceitual pode ser realizada em duas etapas. Na primeira o módulo estação de comando fornece as interfaces para o operador definir os pontos de inspeção no ambiente de simulação através do controle USB. No ambiente de simulação é possível selecionar os pontos e definir quais compõe rotas adaptativas, podem haver diversos níveis de rotas adaptativas em uma inspeção. Uma vez definido os pontos no ambiente de simulação o operador gera um arquivo com as informações dos pontos coletados. Nesse trabalho foi utilizado um modelo genérico de subestação, mas na prática a estação deveria ser modelada e inserida no sistema para obter o ambiente de simulação.

Na segunda etapa o operador inicia a rotina de execução de rotas na estação de comando passando como entrada um arquivo de pontos de interesse. A estação de comando executa o primeiro trecho da inspeção no ambiente de simulação, sendo que no final da execução do trecho os pontos do respectivo trecho são enviados ao módulo controle remoto, enquanto a estação de comando aguarda o retorno. No módulo controle remoto ocorrem verificações quanto aos pontos recebidos e a criação da missão que o VANT real deverá realizar. Após realizar verificações dos pontos a serem carregados o módulo controle remoto envia os pontos para o VANT real e executa o trecho da rota.

Após a execução no VANT real, a estação de comando recebe a confirmação de execução e realiza a execução no ambiente de simulação do próximo trecho e repete as operações até o último trecho. Caso o próximo ponto de destino seja um ponto adaptativo a estação de comando espera um resultado externo ao sistema para decidir se o VANT deverá navegar na rota adaptativa ou prosseguir na mesma rota.

Essa arquitetura contém os sistemas necessários à execução do planejamento e execução de rotas adaptativas. Os principais módulos são descritos abaixo:

3.1.1 VANT

O VANT é o veículo aéreo que será utilizado. Para tanto deve incluir em suas funcionalidades:

- Controlador de voo: deve incluir funções de alto nível para controle através manual utilizando um controlador de rádio.
- GPS: deve ser possível obter a posição geográfica do VANT durante voo.
- Geração e controle alto nível de missões: possibilidade de receber um conjunto de coordenadas geográficas (latitude, longitude e altura) que devem ser executas em sequência.
- Receber comandos da estação de comando: deve receber as mensagens de comando da estação fixa. Deve ser capaz de executar os procedimentos referentes às mensagens. Essas mensagens são mensagens para criação, *upload* e execução de rotas.
- Simular o controlador de voo: um simulador de voo deve ser configurável para que testes de execução de rotas possam ser realizados.
- A comunicação com a estação de comando: esta comunicação deve ser estabelecida preferencialmente utilizando algum protocolo de comunicação com garantia de recebimento de mensagens.

3.1.2 Controlador de rádio

O VANT deve vir acompanhado de um controlador de rádio ou sua integração com um controle deve ser possível. A principais funções a serem desempenhadas pelo controlador de rádio são:

- Controle manual: através deste, deve ser possível controlar a translação e a orientação do VANT.
- *Takeoff*: através do controle deve ser possível enviar o comando para que o VANT estabeleça posição padrão de *takeoff*, ficando a alguma distância do solo. Essa distância varia entre fabricantes, sendo possível sua alteração em algumas ocasiões.
- *Land*: botão de ação rápida no controle, que tem a função de pousar o VANT independentemente da posição onde estiver. Esse comando deve colocar o VANT em solo.
- Comunicação com VANT: os comandos enviados ao VANT devem ser realizados via um protocolo de comunicação proprietário. Caso contrário deve se seguir as instruções do fabricante para integração.

3.1.3 Estação de comando

A estação de comando é o principal módulo da solução. A estação deve manter a aplicação de geração de rotas funcional. Para isso deve possuir capacidade de *hardware* e *software* suficientes para implementar suas funcionalidades. Suas principais funções são:

- Visualização: a estação de comando deve apresentar a visualização do estado atual do VANT e exibir o mapa 3D em que se está navegando com o VANT.
- *Motion Planning:* a estação deve possuir um *software* planejador de rotas que, baseado em um mapa de colisão, estabeleça rotas que

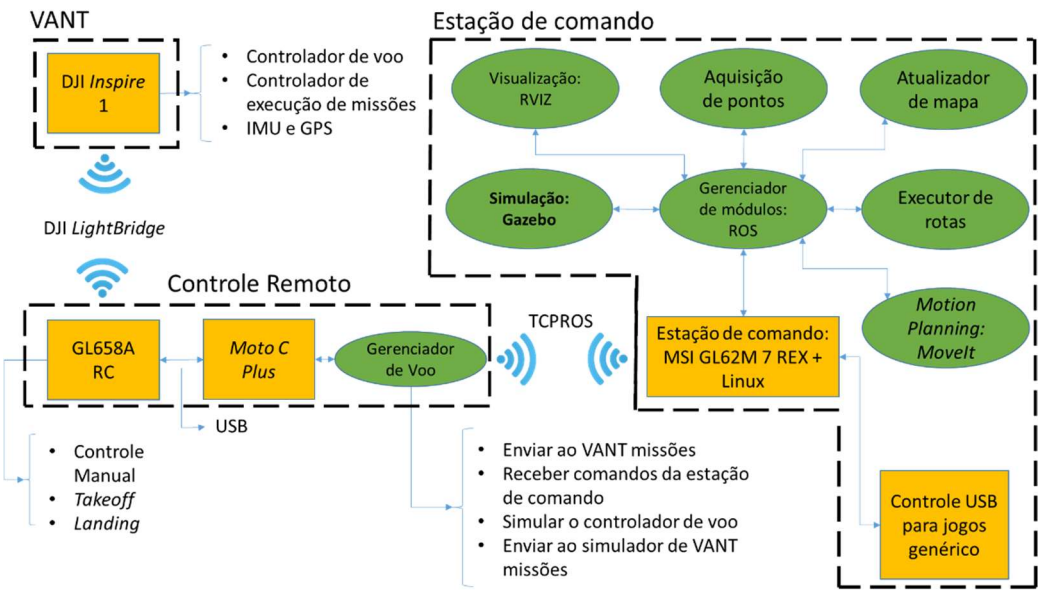
evitem a colisão com as áreas de alta probabilidade de colisão do mapa.

- Geração de rotas adaptativas: a estação deve possuir um *software* que fornece a geração de rotas, com possibilidade de realizar variações de rotas mediante recebimento de variáveis externas.
- Executor de rotas: a estação deve possuir um *software* capaz de controlar o envio de pontos para o gerenciador de voos. Pela natureza do controle adaptativa, a estação determina, sob resultado de outros processos, qual rota deverá enviar ao gerenciador.
- Simulação: a estação deve possuir um *software* capaz de simular o VANT em seu ambiente de operação para que se possa realizar testes e avalições de rotas geradas.
- Gerenciador de módulos: a estação deve implementar um gerenciador de módulos que integra os diferentes módulos que compõe a estação.
- Controle de orientação e translação do VANT: deve ser possível deslocar o VANT dentro do ambiente virtual alterando a posição cartesiana e a orientação em volta do eixo Z (*yaw*).
- Coleta de pontos: deve ser possível selecionar a posição atual como ponto de interesse para posterior cálculo de rotas.
- Armazenar pontos: deve ser possível salvar os pontos em um arquivo.
- Modo autônomo: deve ser possível mudar o VANT para o modo autônomo.
- Modo manual: deve ser possível mudar o VANT para o modo manual

3.2 Arquitetura de tecnologias

A Figura 15 apresenta uma proposta para a arquitetura de tecnologias e como se comunicam entre si para atender as demandas da arquitetura conceitual proposto anteriormente.

Figura 15 - Arquitetura de tecnologias



Fonte: Arquivo pessoal

Os principais módulos são descritos abaixo:

3.2.1 DJI Inspire 1, GL658A e LightBridge

Como opção à função de VANT e controlador de rádio, este trabalho foi desenvolvido com o uso do VANT DJI *Inspire 1* e o controlador de rádio que vêm de fábrica com o multicoptero, o GL658A.

A opção pelo DJI *Inspire 1* e o controlador de rádio GL658A é justificada principalmente pela sua imediata disponibilidade e por atender aos requisitos descritos na secção 3.1.1 e 3.1.2

O protocolo LightBridge é um protocolo proprietário da DJI e permite, além da comunicação de longo alcance (5 km), o *download* simultâneo de imagens da câmera integrada ao VANT (28). A obrigatoriedade de seu uso atende aos requisitos necessários para comunicação e facilita o desenvolvimento, uma vez que não é necessário desenvolver um protocolo de comunicação próprio para controlar o VANT.

3.2.2 ROS

A escolha do ROS como gerenciador de módulos do sistema é principalmente justificada pela disponibilidade de pacotes, ambiente de desenvolvimento colaborativo e pelo tipo de licença de uso.

Aliado à variedade de pacotes disponíveis, mais de 3000 (29), o desenvolvimento de ferramentas para ROS conta com uma comunidade bastante ativa. A comunidade conta com mais de 10000 usuários distribuídos em *sites* de fórum e suporte e, além disso, a *wiki* do ROS recebe cerca de 30 atualizações diárias (29).

Para fins comerciais o ROS é interessante, pois sua licença de uso é do tipo BSD (*Berkeley Software Distribution*), que é bastante permissiva para o reuso de código em aplicações comerciais de código fechado (29).

3.2.3 Estação de comando MSI FL6258M 7REX + Linux

Como *hardware,* a estação de comando é implementada em um *notebook* MSI GL62M 7RE com sistema operacional Ubuntu 16.04 instalado. A escolha pelo computador é justificada pelo poder de processamento (Intel i7 H *series*), placa gráfica dedicada (GTX 1050Ti), capacidade de armazenamento (1 TB) e pela disponibilidade imediata. A escolha pelo sistema operacional Linux realizado devido a recomendações dos desenvolvedores quanto a confiabilidade oriunda de testes realizados em plataforma Linux (30).

3.2.4 Moto C Plus, Gerenciador de Voo e comunicação com Estação de Comando

Para acessar as funções de mais baixo nível do DJI, é necessário utilizar o DJI *Mobile* SDK (*Software Development Kit*) que fornece duas bibliotecas, uma para *Android* e outra para *iOS* (31). Devido à imediata disponibilidade do telefone Moto C *Plus,* esse aparelho foi usado como *hardware* do gerenciador de voo.

A comunicação entre o Moto C Plus e o DJI é estabelecida unicamente através de conexão USB (32). E devido ao sistema no aparelho ser o *Android Nougat 7.0* a biblioteca adotada para o *software* do gerenciador de voo foi a disponível para *Android*.

Para comunicação entre o gerenciador de voo e a estação de comando, foi utilizada uma biblioteca para Java que implementa a comunicação com ROS em aplicações distribuídas. Essa biblioteca foi utilizada porque na estação de comando o

sistema gerenciador de pacotes é o ROS e é desejável aproveitar os padrões de comunicação garantidos pelo *framework* para evitar o desenvolvimento de um protocolo próprio de comunicação apenas para troca de mensagens.

Para realizar a comunicação, essa biblioteca utiliza o TCPROS. O TCPROS é uma camada de transporte para mensagens e serviços que estabelece comunicações padrão TCP *socket* para realizar o transporte das mensagens (33)

3.2.5 Aquisição de pontos, atualizador de mapa, executor de rotas e controle USB

Os módulos de *software* que desempenham as funções de aquisição de pontos, atualizador de mapa e executor de rotas foram implementados em Python devido à disponibilidade de API para desenvolvimento para ROS. A escolha por Python se deve maior disponibilidade de funções de alto nível e pela facilidade de desenvolvimento e teste a opção em C++.

A escolha pelo controle USB genérico é justificada pela disponibilidade imediata e pelo controle disponível atender às demandas de botões para realizar as funções de coleta, armazenamento de arqueios, troca de modos e os controles analógicos para realização de controle de translação e orientação do VANT virtual.

3.2.6 RViz, Gazebo e *MoveIt*

A escolha para visualização do ambiente 3D e do VANT foi o visualizador RViz. Este módulo acompanha a instalação da versão completa do ROS e possui confiável integração para visualização de variáveis, sensores e câmeras. A escolha pelo Gazebo é devido a sua integração com ROS, vir acompanhado da instalação da versão completa e por possuir os plugins para simulação de GPS, colisão e aceitar modelos STL e COLLADA conforme descritos em 2.2.

O uso do pacote *MoveIt* é justificado pelo fato do mesmo possuir integração com a biblioteca OMPL, que conforme descrito em 2.3, apresenta técnicas com custo reduzido de processamento para estabelecer uma rota livre de colisões. O uso do *MoveIt* também é justificado por aceitar mapas de colisões originados com o protocolo *Octomap* descrito em 2.4. O uso do *Octomap* é justificado pelo seu compacto

armazenamento, cálculo probabilístico de localização e por ser possível gerar um mapa a partir de um modelo 3D (34).

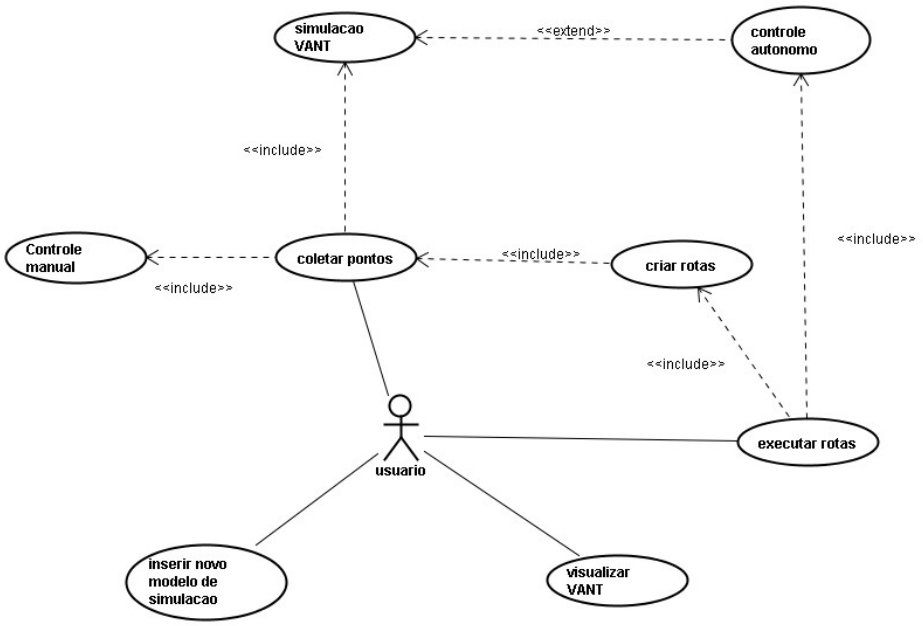
4 DESENVOLVIMENTO

Este capítulo aborda o desenvolvimento referente ao sistema proposto neste trabalho. A descrição da implementação está dividida em 4 seções, cada qual descrevendo os resultados de cada parte da metodologia utilizada para a execução do plano de desenvolvimento.

4.1 Casos de uso

Como forma de captação de requisitos utilizou-se um diagrama de casos de uso para análise das principais funções que o sistema deve desempenhar. Na Figura 2, observa-se no diagrama de casos de uso as principais funcionalidades que o sistema apresenta ao usuário.

Figura 16 - Diagrama de caso de uso



Fonte: Arquivo pessoal

A seguir a descrição dos casos de uso principais:

- Visualizar VANT: o usuário deseja visualizar o estado atual do VANT e do ambiente em que se encontra.
- Inserir novo modelo de simulação: o usuário deseja inserir um novo modelo 3D que representa para o sistema o ambiente que o VANT irá navegar.
- Coletar pontos: o usuário navega com o VANT no ambiente virtual, através do caso de uso “controle manual” e “simulação VANT”, e seleciona pontos para gerar rotas.
- Executar rotas: o usuário seleciona um arquivo originado a partir do caso de uso “coleta de pontos”. Com esse arquivo rotas são criadas, através do caso de uso “criar rotas”, e enviadas para o controlador de rotas que controla a execução das trajetórias no VANT que opera de forma autônoma, caso de uso “controle autônomo”.
- Controle autônomo: o caso de uso “executar rotas” requer a utilização do caso de uso “controle autônomo” do VANT.

4.2 Requisitos

Para levantamento de requisitos, além da modelagem do caso de uso, utilizou-se dados coletados de algumas reuniões de acompanhamento. Devido ao estágio inicial do projeto, e pelo fato de estar associado a um módulo experimental da solução, requisitos originados pelo cliente não puderam em grande parte compor a Tabela 1.

Tabela 1 - Requisitos de software

F1. Coletar posições	
Descrição: O sistema deve possuir um módulo de coleta de pontos	
Requisitos não-funcionais	
Identificação	Descrição
F1.1	A coleta de pontos deve ser realizada em ambiente simulado
F1.2	O deslocamento do VANT para coleta deve ser realizado através de controle manual
F1.3	O controle utilizado para coleta de pontos deve ser um controle USB genérico com no mínimo 3 botões de funções disponíveis

F1.4	O módulo de coleta de pontos deve possuir interface com ROS
F1.5	O módulo de coleta de pontos deve gerar um arquivo com posição e orientação dos pontos coletados
F1.6	O módulo de coleta de pontos deve implementar a função de seleção de ponto, salvar arquivo e zerar pontos
F1.7	O módulo de coleta de pontos deve gerar arquivos com nomes sequenciais
F2. Criar rotas de inspeção	
Descrição: O usuário do sistema deve ser capaz de criar rotas de inspeção	
Requisitos não-funcionais	
Identificação	Descrição
F2.1	A rota deve ter pelo menos 2 pontos
F2.2	A rota não deve colidir com obstáculos
F2.3	A rota deve manter distância mínima de 0,2 metro de qualquer objeto
F2.4	A rota deve ser criada com base em um mapa 3D inserido pelo usuário
F2.5	A rota deve ser criada com base em lista de posições e orientações coletadas
F2.6	A rota deve ser gerada por um algoritmo de *sampled-motion planning* em no máximo 1 segundo
F2.7	O algoritmo de *sampled-motion planning* deve possuir parâmetro configurável para distância máxima
F2.8	O algoritmo de *sampled-motion planning deve* possuir parâmetro configurável área de interesse
F2.9	O algoritmo de *sampled-motion planning* deve gerar pontos o mais próximo possível
F2.10	O algoritmo de *sampled-motion* planning deve usar um *Octomap* como fonte de dados de colisão
F2.11	A inserção de novos *Octomap* deve ser feita de forma manual
F2.12	O *Octomap* pode ser atualizados durante execução
F2.13	A rota deve ser criada utilizando lógica de nível
F2.14	A rota deve ser composta por elementos com dados de posição, orientação e nível
F2.15	Nível é um número positivo inteiro de 1 à infinito
F3. Executar de rotas de inspeção	
Descrição: O usuário do sistema deve ser capaz de executar rotas de inspeção	
Requisitos não-funcionais	
Identificação	Descrição

F3.1	A execução das rotas quando em aplicações com VANT real deve ser realizada em ambiente simulado e real ao mesmo tempo
F3.2	A execução das rotas pode ser realizada somente em ambiente simulado
F3.3	A execução das rotas deve ser gerenciada pelos módulo gerenciador de voo e módulo executor de rotas
F3.4	O módulo gerenciador de voo e executor de rotas devem possuir interface com ROS
F3.5	O módulo gerenciador de voo e executor de rotas devem comunicar-se somente via ROS (TCPROS)
F3.6	O módulo gerenciador de voo e executor de rotas devem ser implementados em sistemas terrestres
F3.7	O módulo gerenciador de voo deve possuir conexão com a internet
F3.8	O módulo gerenciador de voo deve possuir uma interface de criações de missões
F3.9	O módulo gerenciador de voo deve possuir uma interface de execução de missões
F3.10	O módulo gerenciador de voo deve possuir uma interface de leitura de posição e orientação
F3.11	O módulo executor de rotas deve executar rotas com base em arquivos de rota
F3.12	O módulo executor de rotas deve converter os pontos cartesianos da rota para coordenadas geométricas WGS84 (latitude, longitude e altura)
F3.13	O módulo executor de rotas deve filtrar as rotas para que tenham pelo menos 0,5 m de distância entre um ponto e outro
F3.14	O módulo executor de rotas deve aguardar recebimento de sinal de rota concluída vindo do módulo gerenciador para continuar a enviar segmentos da rota global
F3.15	O módulo executor de rotas deve executar as rotas por lógica de nível
F3.16	O módulo executor de rotas deve executar em sequência rotas de mesmo nível
F3.17	O módulo executor de rotas deve aguardar sinal de decisão quando há alteração de nível de rotas
F4. Simular ambiente com VANT	
Descrição: O sistema deve ser capaz de gerar um ambiente virtualizado com VANT inserido	
Requisitos não-funcionais	
Identificação	Descrição
F4.1	A simulação do ambiente com VANT deve permitir a aplicação de forças nos objetos

F4.2	A simulação do ambiente com VANT deve permitir a inserção de modelos STL e COLLADA
F4.3	A simulação do ambiente com VANT deve incluir um VANT com câmera RGB e *point cloud*
F4.4	A simulação do ambiente com VANT deve prover simulação física.
F4.5	A simulação do ambiente com VANT deve prover interface de controle de translação e rotação
F4.6	A simulação do ambiente deve ser mantida por um módulo atualizador de mapa
F4.7	O módulo atualizador de mapa deve manter uma cena com um *Octomap* e parâmetros de colisão
F5. Simular controlador de voo do VANT	
Descrição: O sistema deve ser capaz de gerar uma simulação de um controlador de voo	
Requisitos não-funcionais	
Identificação	Descrição
F5.1	A simulação do controlador de voo deve permitir interface de controle de translação e rotação
F5.2	A simulação do controlador de voo deve permitir leitura de posição e orientação e envio
F6. *Motion Planning* do VANT	
Descrição: O sistema deve ser realizar motion planning com VANT virtualizado	
Requisitos não-funcionais	
Identificação	Descrição
F6.1	A visualização do VANT deve permitir visão de primeira pessoa no VANT
F6.2	A visualização do VANT deve permitir visão do *Octomap*
F6.3	A visualização do VANT deve permitir realizar *sampled-motion planning* entre ponto inicial e ponto de interesse
F6.4	A visualização do VANT deve permitir escolha de diferentes tipos de algoritmos de *sampled-motion planning*
F6.5	A visualização do VANT deve permitir a visualização da rota planejada

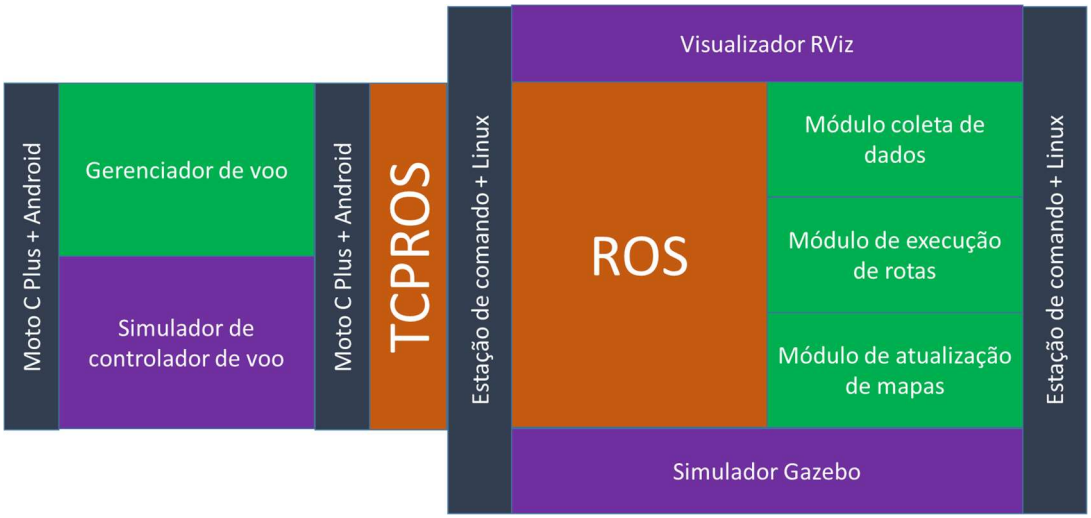
Fonte: Arquivo pessoal

Os requisitos aqui apresentados foram definidos como essenciais para o estabelecimento de um sistema de navegação. Podemos destacar por exemplo o requisito de distância mínima de obstáculos.

4.3 Arquitetura do *software*

Segundo os requisitos descritos na seção 4.2, uma arquitetura de *software* foi estabelecida a fim de atender as demandas apresentadas nessa seção. A Figura 17 ilustra a arquitetura de *software*:

Figura 17 - Arquitetura de software



Fonte: Arquivo pessoal

Os retângulos com texto na vertical limitam são os sistemas físicos que limitam os módulos descritos nos retângulos com textos na horizontal. A comunicação entre os dois sistemas físicos é realizada via TCPROS. Os sistemas físicos implementam:

- Celular Moto C Plus: Sistema operacional Android com módulos de gerenciador de voo e simulador de controlador de voo da DJI.
- Estação de comando: Sistema operacional Linux Ubuntu 16.04, com ROS, módulo de coleta de pontos, módulo de execução de rotas, módulo de atualização de mapas, simulador Gazebo e visualizador RViz.

A arquitetura foi definida pensando no desenvolvimento de um sistema que possui uma espécie de cópia digital, na qual em um primeiro momento simulações podem ser realizadas para se obter as melhores rotas e parâmetros para posteriormente executar as rotas no VANT real de forma sincronizada com sua

virtualização. Dessa forma é possível o acompanhamento em tempo real do estado do VANT em um ambiente virtualizado.

4.4 Implementação

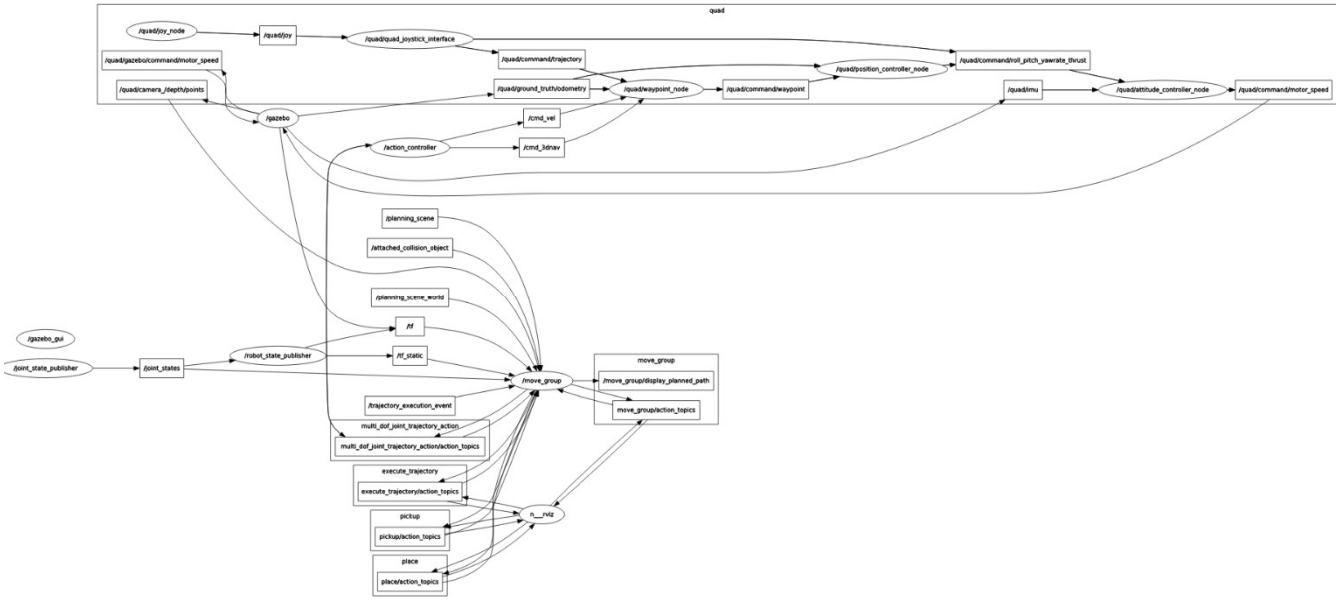
A implementação do sistema foi realizada sob os módulos especificados na arquitetura. Nas seções que seguem serão descritas as estratégias e as tecnologias utilizadas na implementação dos módulos.

4.4.1 Visualização e simulação do VANT

Para a implementação da simulação e visualização foi utilizado um pacote para ROS com integração com simulação em Gazebo, que orginalmente fora desenvolvido para execução de mapeamento com base em *Octomap* e posterior navegação em ambiente virtual utilizando um VANT genérico com uma câmera Kinect. Esse modelo foi desenvolvido por Will Selby e atende às especificações descritas na seção de requisitos 4.2 (35).

Posteriormente, como será descrito, houveram modificações no pacote original, como por exemplo, o modelo original foi substituído por uma subestação. Na Figura 18 pode-se observar os nodos (losangos), tópicos e serviços (retângulos) que constituem o pacote utilizado.

Figura 18 - Visualização e simulação do VANT: nodos, tópicos e serviços



Fonte: Arquivo pessoal

A compressão das interfaces de alguns dos nodos é essencial para o entendimento da implementação do trabalho. A seguir serão descritos os principais módulos e suas interfaces.

4.4.1.1 move_group

O nodo move_group é pertencente ao pacote *MoveIt*. É o principal nodo deste pacote, sendo a central de planejamento de rotas do sistema desenvolvido. Nesse nodo são realizadas as requisições de planejamento de rotas, que são respondidas em tópicos de resultado.

- Subscrições
  - *planning_scene*: tópico no qual dados de percepção do ambiente são mantidos. Contém dados brutos que definem o *Octomap*, localização do *Octomap* e distância mínima das juntas do VANT ao mapa. Atualização desse tópico é realizada por nodos de percepção.
  - *joint_states*: tópico no qual os valores das juntas que o MoveIt controla deve ser publicado por outros nodos.
- Serviços

- *move_group/action_topics*: através desse serviço é possível solicitar o planejamento de rota entre um ponto e outro. Por meio de protocolo específico, é possível selecionar o algorítmico utilizado e a área permitida para determinação da rota.
- *multi_dof_joint_trajectory_action/action_topics*: através desse serviço é possível realizar a execução de rotas no VANT simulado. Este serviço precisa de valores de entrada, que são as rotas produzidas com o serviço anterior.

4.4.1.2 *action_controller*

Este nodo realiza o controle da execução dos resultados do nodo *move_group*. Este nodo adquire todas as posições de uma rota planejada e controla a sua execução. A interface é descrita abaixo:

- Publicações
  - *cmd_3dnav*: este tópico possui uma missão completa publicada por este controlador. Posteriormente este tópico será subscrito pelo controlador de missões que executará cada ponto da missão.
- Subscrições
  - *multi_dof_joint_trajectory_action/action_topics*: Dentre os action topics fornecidos pelo nodo *MoveIt* está *multi_dof_joint_trajectory_action/result* que ao término de um planejamento de rota bem-sucedido publica as rotas de inspeção. O *action_controller* redireciona as trajetórias para o tópico cmd_3dnav.

4.4.1.3 n_rviz

Este nodo implementa a interface gráfica de usuário. O RViz permite uma flexibilidade elevada em relação aos dados que podem ser visualizados, é possível selecionar tópicos de acordo com o teste que está sendo realizado. Os principais tópicos são o *planning_scene,* que contém a definição do *Octomap,* e os tópicos de câmera, que permitem a visualização em primeira pessoa do ambiente virtual.

Os principais serviços vistos no nodo n_rviz são o planejamento de rota (*plan*) e execução de rota (*execute*). Após o planejamento é possível observar uma linha de sucessão de estados que representa a trajetória proposta pelo algoritmo escolhido.

4.4.1.4 gazebo

O nodo gazebo fornece a simulação do VANT e o modelo do ambiente. Publica e subscreve em tópicos informações de sensores para que o controle do VANT seja possível. A seguir as principais publicações e subscrições do nodo gazebo:

- Publicações
  - *quad/camera_/depth/points*: este tópico possui os dados de *point cloud* gerados pelo sensor Kinect simulado. Estes dados se desejável servem para atualizacao do *Octomap* durante a inspeção.
  - *quad/ground_truth/odometry*: este tópico possui informações de iGPS (*indoor GPS*) para alimentar o controlador de posição do VANT.
  - */tf*: este tópico possui a transformada entre o sistema global e a base do VANT. Este dado é necessário para que o nodo move_group conheça a posição atual do VANT.
- Subscrições
  - *quad/command/motor_speed*: este tópico possui as 4 velocidades que são configuradas aos motores do VANT simulado no gazebo. As velocidades são resultado do controlador de orientação.
  - *Quad/imu*: este tópico possui os dados referentes às acelerações do VANT simulado nos três eixos coordenados. O

controlador de orientação é alimentado com os dados deste tópico.

4.4.1.5 quad/joy_node

Este nodo realiza a interface entre a porta USB da estação de comando e o *framework* ROS. É realizado a transferência dos comandos mecânicos gerados no controle *USB* para um tópico na camada ROS. Sua ação é dada orientada a eventos do usuário no controle, somente publica novas mensagens no ROS quando o usuário pressiona o controle. Abaixo a descrição da principal publicação deste nodo:

- Publicações
  - *quad_joy:* este tópico contém valores de entrada dos comandos do controle USB.

4.4.1.6 quad/quad_joystick_interface

Este nodo realiza a interpretação dos botões pressionados transferindo-os para entrada de controle do VANT. Suas funções são a troca entre manual e autônomo, publicar as acelerações e impulsos sobre eixos coordenados, no caso de manual. Suas principais publicações são descritas abaixo:

- Publicações
  - *quad/command/trajectory*: neste tópico são publicados os comandos de mudanças de estado. Os possíveis estados de controle do VANT são manual e automático.
  - *quad/command/roll_pitch_yawrate_thrust*: neste tópico são publicados os valores de aceleração nos eixos coordenados quando o modo de controle do VANT é manual. É realizada validação e quando necessário, arredondamentos para valores máximos de aceleração sobre os eixos coordenados caso ultrapassem os limites.

4.4.1.7 quad/waypoint_node

Este nodo é o controlador de missão do VANT. Este controlador checa se novas posições são publicadas no tópico *cmd_3dnav* e dispara eventos de controle de cada posição da missão. Realiza o controle de missão do VANT. Sua interface é descrita abaixo:

- Publicações
  - *quad/command/waypoint*: este tópico possui dados de um ponto da missão.
- Subscrições
  - */cmd_3dnav*: este tópico possui os *setpoints* da missão publicados pelo *action_controller*.
  - *quad/command/trajectory*: este tópico possui dados relacionados ao modo como o VANT deve operar, manual ou autônomo.
  - *quad/ground_thruth*: este tópico possui dados sobre a localização do VANT no ambiente simulado. Este dado é utilizado pelo controlador para checar se o ponto da missão foi alcançado.

4.4.1.8 *quad/position_controller_node*

Este nodo é o controlador de posição do VANT. Sua interface é descrita abaixo:

- Publicações
  - *quad/command/roll_pitch_yawrate_thrust*: quando em modo manual este tópico receberá dados deste controlador. Os dados são a aceleração nos eixos coordenados e o impulso.
- Subscrições
  - *quad/command/waypoint:* este tópico possui o *setpoint* que deve ser controlado no VANT.

- *quad/ground_truth/odometry*: este tópico possui os dados de localização do VANT no ambiente simulado.

4.4.1.9 quad/atitude_controller_node

Este nodo é o controlador de orientação do VANT. Este controlador controla os ângulos *roll*, *pitch* e *yaw* através da modificação das velocidades dos motores no VANT simulado. É utilizado em ambos os modos de funcionamento, manual ou automático. Sua interface é descrita abaixo:

- Publicações
  - *quad/command/motor_speed*: neste tópico são publicadas as velocidades dos motores. A lógica de como a posição e orientação do VANT pode ser modificada através da alteração da velocidade é descrita na seção 2.6.
- Subscrições
  - *quad/imu*: este tópico possui dados de aceleração do VANT medidos pelos sensores simulados do Gazebo. É utilizado pelo controlador para controlar os ângulos nos eixos.
  - *quad/command/roll_pitch_yawrate_thrust:* este tópico possui as *setpoints de* angulo que devem ser controladas pelo nodo.

4.4.2 Instalação do ROS

O ROS possui diferentes distribuições. O objetivo de estabelecimento de versões do ROS é garantir a validação e confiabilidade de uma coleção de pacotes desenvolvidos para uma determinada versão após correções da comunidade. Uma nova versão do ROS é lançada sempre que uma nova versão do Ubuntu é lançada (36).

O critério para decisão de qual versão instalar foi tomado com base no pacote utilizado para simulação do ambiente virtual e do VANT descrito na seção 4.4.1. Essa decisão foi tomada com prioridade a este pacote, pois este apresenta muitos sub

pacotes que são essenciais para o funcionamento da lógica de *motion planning* e de movimentação do VANT virtual. O pacote foi originalmente implementado utilizando a versão Kinetic Kame do ROS, portanto esta foi a distribuição escolhida.

A instalação da versão completa ROS Kinetic é bem genérica e dispensável de descrição de detalhes. Foi realizada conforme instruções do site do ROS.org (37).

4.4.3 Inserção do modelo de simulação e mapa de colisão.

Esta seção trata da inserção do modelo 3D pelo usuário no ambiente de simulação e na conversão do modelo 3D para um mapa de colisão no formato *Octomap*. Com base no mapa de colisão, os nodos do *MoveIt* realizam o planejamento com base nas informações de áreas livres e ocupadas que o mapa de colisão possui.

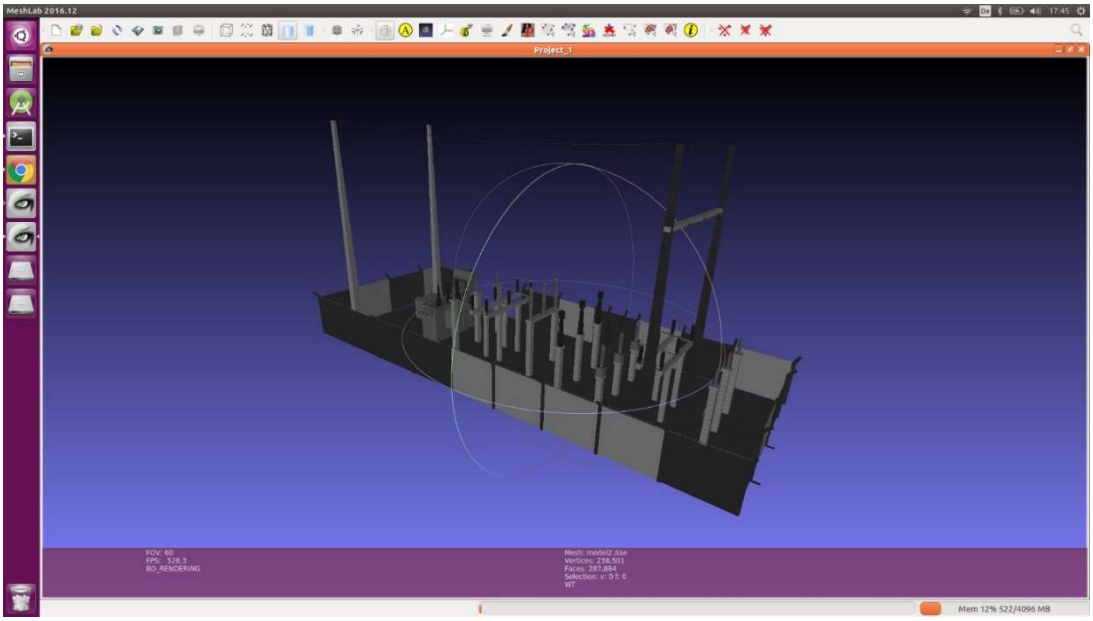
Para o trabalho em desenvolvimento optou-se por não automatizar o processo de conversão de modelo 3D para o mapa de obstáculos. Em aplicações reais a conversão do modelo 3D do ambiente poderia servir como um primeiro mapa do ambiente, porém devido a dinâmica e mudança nos ambientes é recomendada a atualização constante através sensores de *point cloud*.

Para inicializar o modelo de simulação no ambiente virtual, não foi realizada nenhuma transformada ou ajuste para referencias reais. A seguir serão descritos os procedimentos utilizados na conversão de um modelo 3D de subestação para o arquivo que representa o *Octomap* deste modelo.

4.4.3.1 Encontrar modelo

O modelo de subestação foi encontrado no site https://3dwarehouse.sketchup.com, porém em um caso prático, é necessário realizar a modelagem da subestação em uma ferramenta específica. Nesse site, modelos 3D gratuitos podem ser encontrados. Na busca pelo modelo optou-se por um modelo que contivesse isoladores, transformadores e cabos. O formato para inserção no Gazebo (simulação) é COLLADA, portanto optou-se por um modelo nesse formato. O modelo é representado no *software* MeshLab (http://www.meshlab.net/) na Figura 19.

Figura 19 - Modelo 3D de simulação

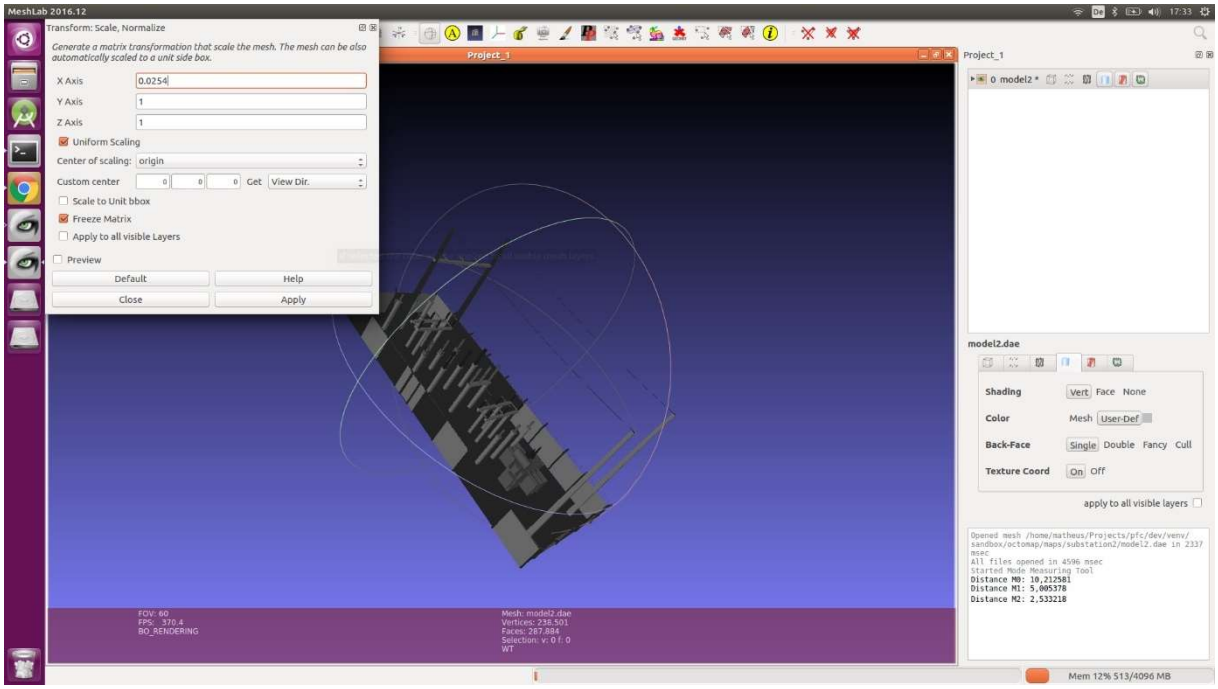


Fonte: arquivo pessoal

Para realizar a conversão do modelo 3D para *Octomap* é necessário que o modelo esteja no formato STL segundo a ferramenta utilizada. O modelo escolhido está em no formato COLLADA, portanto foi necessário converte-lo para STL. A conversão foi realizada no *software* MeshLab.

A maioria dos modelos no formato COLLADA disponíveis no https://3dwarehouse.sketchup.com estão originalmente configurados com unidade de polegadas. O *software* MeshLab espera que os arquivos estejam com unidade configurada em metros, portanto é necessário realizar um ajuste de escala nesse caso. Para realizar modificação todas as dimensões foram escaladas usando o fator de multiplicação de 0,0254. Na Figura 20 o processo de alteração de escala é exibido.

Figura 20 - Alteração de escala no Modelo 3D



Fonte: Arquivo pessoal

Uma vez obtido o mapa na escala correta, o assistente de exportação é utilizado para converter o modelo para o formato STL.

4.4.3.2 Converter para formato *Octomap*

O procedimento para converter um modelo 3D STL para um formato *Octomap* com árvore hierárquica de *voxels* consiste em dois processos: transformação do modelo 3D em um *grid* de *voxels* e posterior transformação *grid* em uma árvore hierárquica.

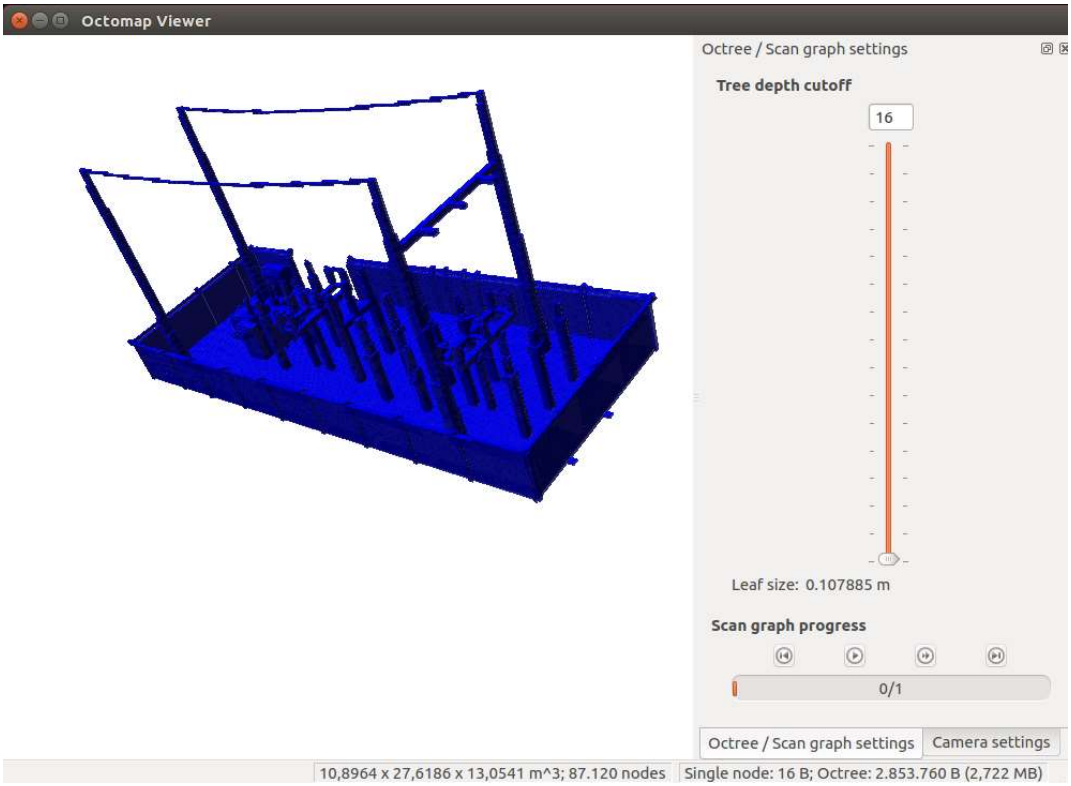
Para transformar o modelo 3D em um grid de *voxels* foi utilizado o programa binvox. O binvox cria um *grid b*inário de *voxels* (34). O binvox é encontrado no site do desenvolvedor (http://www.patrickmin.com/binvox/). O resultado será um arquivo no formato binvox.

Em seguida esse *grid* de *voxels* é transformado em uma árvore hierárquica de *voxels* através do programa binvox2bt. Esse programa vem como parte do pacote *Octomap* para ROS. A árvore gerada representa as regiões livres e ocupadas, que são usadas como entrada para algoritmos de *motion planning* como os contidos no

pacote *MoveIt*. A entrada é formada por arquivos no formato *binvox* e a saída por arquivos no formato *bt.*

O arquivo no formato *bt* pode ser visualizado através do programa *octovis* (http://wiki.ros.org/octovis). Na Figura 21 o resultado da conversão do modelo original para o *Octomap*.

Figura 21 - *Octomap* construído



Fonte: Arquivo Pessoal

A informação de *leaf size* na Figura 21 é o tamanho das arestas do menor *voxel*, portanto pode ser entendido como a resolução do mapa obtido.

4.4.3.3 Carregar modelo 3D e mapa de colisão

Para utilizar o modelo 3D na aplicação é necessário colocar os arquivos em uma pasta específica para que o Gazebo identifique o modelo. Para tornar o mapa de colisão identificável pelo *MoveIt* o arquivo .bt pode ser colocado em qualquer diretório.

Esse diretório deve ser lembrado, pois será necessário informar o caminho do mapa de colisão para que o módulo atualizador de mapa alimente o *MoveIt.*

Para tornar o modelo 3D executável pelo Gazebo é necessário colocar o modelo 3D no diretório ~/.gazebo/models. A pasta pode ter qualquer nome e deve conter os seguintes arquivos:

- model.sdf
- model.config
- model.dae: arquivo COLLADA do modelo 3D

O arquivo SDF (*Simulation Description Format*) contém a descrição de visual físico e de colisão do modelo dos arquivos de malha utilizados e a posição inicial do modelo. A Figura 22 mostra que o modelo é posicionado em z = -15 m. Isso se deve a um erro de modelagem no arquivo original, que possui *offsets* em X, Y, e Z.

Figura 22 - Código modelo 3D arquivo SDF

```
<?xmlversion="1.0"?>
   <sdf version="1.0">
     <model name="Substation">
       <static>true</static>
       <link name="link">
         <pose>0 0 -15.5 0 0 0</pose>
        <collision name="wall1">
           <geometry>
             <mesh>
               <uri>model://substation/model.dae</uri>
             </mesh>
           </geometry>
         </collision>
         <visual name="visual1">
           <geometry>
             <mesh>
               <uri>model://substation/model.dae</uri>
             </mesh>
           </geometry>
         </visual>
       </link>
     </model>
   </sdf>
```

Fonte: arquivo pessoal

O arquivo *model.config* como mostra a Figura 23 contém a definição de nome, versão do modelo, nome do arquivo SDF.

Figura 23 - Código modelo 3D arquivo config

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


    <model>
      <name>Substation</name>
      <version>1.0</version>
      <license></license>
      <sdf version="1.0">model.sdf</sdf>
      <author>
        <name>Matheus</name>
        <email>mkc@certi.org.br</email>
      </author>


      <description>
        A substation.
      </description>
    </model>
```

Fonte: Arquivo pessoal

4.4.4 Módulo de atualização de mapas

O módulo de atualização de mapas realiza a leitura do arquivo *Octomap* do modelo 3D convertido na seção 4.4.3, realiza a modificação de parâmetros do mapa, e atualiza, com uma determinada frequência, um tópico chamado *planning_scene*. Esse tópico inclui mapa, parâmetros e a transformada entre a base do VANT simulado e o sistema de referencial global do ambiente simulado.

O tópico *planning_scene* é utilizado pelo *MoveIt* como fonte de informações sobre o mapa de colisão e pelo RViz para visualização dos planejamentos e execuções de rotas. Principais publicações e subscrições:

- Publicações

- *planning_scene*

- Subscrições:
  - *tf*
  - *octomap_full*

A seguir as partes do código referentes às suas principais funções serão detalhadas:

4.4.4.1 *Imports* e inicialização

Na Figura 24 pode-se observar o código referente aos *imports* do módulo.

Figura 24 - Módulo atualizador de mapa: Imports

```
1. #!/usr/bin/env python
2. import rospy
3. import roslib
4. from octomap_msgs.msg import Octomap
5. from moveit_msgs.msg import PlanningSceneWorld, PlanningScene, LinkPadding
6. from geometry_msgs.msg import Transform, TransformStamped
7. import tf
8. mapMsg = PlanningScene()
9. flag = None
10. octomap = None
11. listener = tf.TransformListener()
12. r = rospy.Rate(0.25)
```

Fonte: Arquivo pessoal

As explicações sobre o código da Figura 24 seguem abaixo:

- Linhas 2 e 3: referências à biblioteca que se comunica com o ROS. Através dessa biblioteca é que se torna possível ler, escrever e utilizar tópicos e serviços disponíveis no ROS.
- Entre linhas 4 e 6: estes comandos referenciam os tipos das mensagens utilizadas pelo módulo atualizador de mapas. Para publicar ou subscrever em algum tópico é necessário estabelecer um tipo de mensagem no mecanismo de comunicação, esse tipo deve ser o mesmo tipo publicado no caso de uma subscrição.

- Linha 7: este comando referência uma biblioteca utilizada para ler transformadas entre *frames* no ROS.
- Linha 8: inicialização da varável que receberá os dados de *planning_scene*.
- Linha 9: *flag* utilizada no módulo para saber quando os dados do *Octomap server* foram recebidos.
- Linha 10: variável utilizada no módulo para armazenar os dados do arquivo *Octomap* recebidos do *octomap_server.*
- Linha 11: inicialização do objeto que recolhe valores de transformadas entre *frames* que são fornecidas na camada ROS. Esse objeto servirá para ler a transformada entre o frame do VANT simulado e o sistema de referência do mundo simulado.
- Linha 12: Inicialização de um objeto temporizador do ROS. Neste caso quando acionado trava a execução do programa por 4 segundos, pois está configurado para 0,25 Hz.
- Linha 14: este comando utiliza a biblioteca do ROS para Python para inicializar um nodo chamado.

Na Figura 25 pode-se observar a inicialização do nodo.

Figura 25 - Módulo atualizador de mapa: inicialização

```
42. rospy.init_node("atualizador_mapa")
43. rospy.Subscriber("octomap_full", Octomap, cb, queue_size=1)
44. pub = rospy.Publisher('/planning_scene', PlanningScene, queue_size=1)
```

Fonte: Arquivo Pessoal

As explicações sobre o código da Figura 25 seguem abaixo:

- Linha 42: este comando criar um nodo chamado atualizador_mapa.
- Linha 43: este comando inicializa a subscrição no nodo *octomap_full*. Este tópico é publicado pelo nodo *octomap_server*. O nodo *octomap_server* lê o arquivo de *Octomap* e publica com o tipo de dados

Octomap, portando esse subscritor é do tipo *Octomap.* O comando especifica a função *cb* como função de *callback* ao receber a mensagem. Detalhes na seção de função de suporte.

- Linha 44: este comando cria um objeto pelo qual se realiza publicações com o tipo de mensagem *PlanningScene*. As atualizações que o nodo atualizador de mapa realizada posição do VANT e mapa são concatenas e publicadas através deste objeto.

4.4.4.2 Funções de suporte

A função detalhada nesta seção é a função *cb* que recebe os valores do *Octomap* via tópico. A implementação da função de suporte *cb* é mostrada na Figura 26:

Figura 26 - Módulo atualizador de mapa: função cb

```
15. def cb(msg):
16.     global mapMsg
17.     global flag
18.     global octomap
19.
20.     octomap = msg
21.
22.     psw = PlanningSceneWorld()
23.     psw.octomap.header.stamp = rospy.Time.now()
24.     psw.octomap.header.frame_id = "world"
25.     psw.octomap.octomap = msg
26.     psw.octomap.octomap.header.frame_id = "world"
27.     psw.octomap.origin.position.z = -15.5
28.     psw.octomap.origin.position.y = -10.0
29.
30.     print psw.octomap.octomap.resolution
31.
32.     psw.octomap.origin.orientation.w = 1
33.
34.     test = PlanningScene()
35.
36.     test.world = psw
37.
38.     mapMsg = test
39.
40.     flag = 1
```

Fonte: Arquivo pessoal

Os detalhes são detalhados abaixo:

- Entre as linhas 16 e 19: este comando referência para variáveis de globais do código.
- Linha 20: este comando salva os dados lidos na variável *octomap.*
- Entre 22 e 32: nestas linhas a variável do tipo *PlanningSceneWorld* psw é inicializada preenchida com informações de tempo, frame de referência, *octomap* e a posição do *octomap* em relação ao frame de referência. Vale notar aqui que há uma correção na posição do modelo, pois o modelo escolhido continha um erro de dimensionamento com offset em Z e Y, portanto aqui esse erro é corrigido.
- Entre linhas 34 e 40: compreendido nestas linhas está a inicialização da vaiável mapMsg do tipo *PlanningScene*. Essa variável recebe em seu atributo *world* a variável psw. Ao final do procedimento a variável *flag* é configurada para 1 indicando ao *software* o recebimento do mapa de custo.

4.4.4.3 Main

A primeira parte da implementação da função de suporte *main* é mostrada na Figura 27:

Figura 27 - Módulo atualizador de mapa: main parte 1

```
listener.waitForTransform('/base_link', '/world', rospy.Time(), rospy.Duration(4))

 while(not rospy.is_shutdown()):

     if (flag is not None):

         trans = [0,0,0]
         rot = [0,0,0,0]
         try:
             (trans, rot) = listener.lookupTransform('/world', '/base_link', rospy.Time(0))
             #print (trans, rot)
         except:
             pass

         link_padding = list()

         links = ["base_link_inertia"]

         for link in links:
             link_new = LinkPadding()
             link_new.link_name = link
             link_new.padding = 0.2
             link_padding.append(link_new)

         mapMsg.link_padding = link_padding

         mapMsg.robot_state.joint_state.header.stamp = rospy.Time.now()
         mapMsg.robot_state.joint_state.header.frame_id = "world"
         mapMsg.robot_state.joint_state.name = ["quad/ground_truth/odometry_sensorgt_joint", "quad/imu_joint",
                                                "quad/imugt_joint", "rotor_0_joint",
                                                "rotor_1_joint", "rotor_2_joint", "rotor_3_joint"]
         mapMsg.robot_state.joint_state.position = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
         mapMsg.robot_state.multi_dof_joint_state.header.frame_id = "world"
         mapMsg.robot_state.multi_dof_joint_state.joint_names = ["virtual_joint"]

         mapMsg.robot_model_name = "quad"
```

Fonte: arquivo pessoal

Os detalhes são exibidos abaixo:

- Linha 46: este comando trava o programa até o recebimento da primeira transformada entre a base do VANT e o frame do sistema simulado.
- Linha 48: este comando mantem o *while loop* executando até que o ROS *master* seja interrompido.
- Linha 50: esta condição espera que que a *flag* de recebimento do mapa de custo esteja ativa, para então executar os procedimentos necessários para publicar a *PlanningScene.*
- Linha 55: Este comando realiza uma tentativa de ler a transformada entre a base do VANT e o frame de coordenadas do sistema simulado para salvar nas variáveis trans e rot os valores de translação e rotação respectivamente.
- Entre 60 e 70: Nestas linhas é criado uma área de segurança no elo *base_link_intertia*. Esse elo representa a base do VANT, e serve para assegurar que ele não vá se aproximar de obstáculos. É criado um enchimento de 0,2 m em torno do elo. A informação do enchimento de elo é inserida na variável *mapMsg.*
- Entre 72 e 79: Nestas linhas as características das juntas são definidas e inseridas na variável *mapMsg*. São configurados os nomes e a posição de cada junta. O sistema de controle de juntas não utiliza as informações de juntas por este meio, portanto os valores podem ser zerados. Esses valores são por motivos de visualização.
- Linha 81: Aqui o nome do modelo do VANT que está na cena é configurado e armazenado na variável *mapMsg*.

A segunda parte do código de implementação da função *main* é exibido na Figura 28

Figura 28 - Módulo atualizador de mapa: main parte 2

```
98.             transform = Transform()
99.             transform.translation.x = trans[0]
100.            transform.translation.y = trans[1]
101.            transform.translation.z = trans[2]
102.
103.            transform.rotation.x = rot[0]
104.            transform.rotation.y = rot[1]
105.            transform.rotation.z = rot[2]
106.            transform.rotation.w = rot[3]
107.
108.            mapMsg.robot_state.multi_dof_joint_state.transforms = [transform]
109.
110.            pub.publish(mapMsg)
111.
112.        else:
113.            pass
114.        r.sleep()
```

Fonte: Arquivo pessoal

Os detalhes abaixo:

- Entre 98 e 108: Nestas linhas é criada a variável *transform* e nesta armazenadas as informações de translação e rotação do VANT em relação ao eixo coordenado do ambiente simulado. Por fim a variável *transform* é inserida na propriedade *robot_state* da variável mapMsg.
- Linha 110: A variável *mapMsg* é publicada no tópico *PlanningScene* pelo publicador *pub*.
- Entre 112 e 114: Estas linhas são percorridas caso o mapa de colisão ainda não tenha sido recebido pelo nodo atualizador de mapas. Caso percorre o tempo de 4 segundos é esperado.

4.4.5 Módulo de coleta pontos

O módulo de coleta de pontos gerencia a aquisição de pontos para criação de arquivos de rota. Esse módulo realiza a leitura dos acionamentos do controle e toma ações de registro de pontos e armazenamento de arquivos.

A operação de registro de pontos é realizada através do posicionamento do VANT no ponto desejado no ambiente virtual, através do controle em modo manual, e

posterior comando para registro da posição. Antes da posição ser registrada a validade daquela posição é verificada, se está em contato com algum obstáculo o módulo não salva a posição. O usuário pode desfazer os pontos adquiridos, através de comando, caso queira recomeçar a adquirir os pontos. O comando de registro de ponto armazena a posição (X, Y, Z) e a orientação na forma de *quaternion*.

Após a aquisição dos pontos o operador realiza um comando no controle para gerar o arquivo com pontos coletados, ao realizar este comando os pontos atuais são armazenados em um arquivo .txt e o *array* interno é zerado. Os arquivos gerados por este módulo são utilizados pelo módulo executor de rotas para gerar trajetórias que são enviadas de forma controlada para o VANT.

Principais subscrições e serviços utilizados:

- Serviços
    - *get_planning_scene*
    - *check_state_validity*
- Subscrições
    - *quad/joy*

4.4.5.1 Mapa do controle

A seguir será descrito o mapa do controle utilizado. Entende-se por mapa do controle como a descrição dos acionamentos mecânicos em termos de funções no módulo de aquisição de pontos. Os acionamentos mecânicos, conforme explicado em 4.4.1.5, são armazenados no tópico *quad/joy*.

Neste tópico há um *array* para os botões e um *array* para os eixos. Os valores de eixo são referentes ao acionamento nos controles analógicos do controle. Os acionamentos de botões possuem valores binários de 0 e 1, enquanto os eixos variam continuamente entre -1 e 1. Esse módulo utiliza apenas os valores referentes aos botões.

A posição no *array* está relacionada ao botão acionado e depende do *hardware* do controle que está conectado ao computador. No desenvolvimento deste trabalho o controle utilizado foi um controle USB genérico para jogos que possui o mapeamento indicado na Figura 29.

Figura 29 - Mapa do controle

Fonte: arquivo pessoal

As funções referentes ao módulo de coleta de pontos são:

- Coletar pontos.
- Armazenar pontos coletados.
- Aumentar nível.
- Diminuir nível.
- Zerar pontos.
- Troca de modos.
- Controles analógicos para movimentação do VANT.

Na Figura 29 também está apresentado os acionamentos que o nodo *quad/quad_joystick_interface* gerencia, a descrição desse item pode ser encontrada na seção 4.4.1.6. As funções geridas pelo nodo são:

4.4.5.2 Lógica de nível

A lógica de coleta de pontos é realizada de tal forma que cada ponto possui um valor de nível associado ao ser armazenado. O arquivo de pontos pode ser

entendimento como um *array* em que cada posição é um tipo de dado que contém o valor de posição, orientação e nível daquele ponto.

Ao inicializar o módulo o valor de nível é configurado para 1. Para aumentar e diminuir o valor do nível as setas superior e inferior do controle podem ser usadas respectivamente. O nível é utilizado pelo módulo executor de rotas para detectar rotas alternativas e essa lógica será detalhada na seção do módulo executor de rotas.

4.4.5.3 Imports e inicialização

A seguir na Figura 30 a o código de imports do módulo de coleta de pontos

Figura 30 - Módulo coletor de pontos: imports

```
1.  #!/usr/bin/env python
2.  import rospy
3.  from sensor_msgs.msg import Joy
4.  from moveit_msgs.msg import PlanningSceneComponents
5.  from moveit_msgs.srv import GetPlanningScene, GetStateValidityRequest, GetStateVa
    lidity
6.  from os import listdir
7.  from os.path import isfile, join
8.  import pickle
9.
10. path = "/home/matheus/Projects/pfc/dev/ros_kinetic/src/ROS_quadrotor_simulator/qu
    ad_collect/scripts/lists"
11.
12. nivel = 0
13.
14. r = rospy.Rate(1)
15.
16. positions = list(list())
```

Fonte: Arquivo pessoal

Detalhes sobre a implementação abaixo:

- Linha 2: *import* da biblioteca de ROS para Python
- Entre 3 e 5: *import* de tipos de mensagem utilizados no módulo de coleta de pontos.
- Linhas 6 e 7: *import* de funções de sistema para leitura de arquivos em diretórios.

- Linha 8: *import* da biblioteca *pickle* utilizada para converter tipos de dados para texto. Esta operação é necessária para salvar os pontos em formato texto.
- Linha 10: esta linha inicializa uma variável *path* e insere o caminho padrão para salvar os arquivos de pontos.
- Linha 12: Esta linha inicializa a variável nível com o valor 0. O valor zero significa para o usuário o valor de nível 1.
- Linha 14: Esta linha inicializa um objeto temporizador. Este objeto é utilizado na função *main* para que o módulo não consuma recursos computacionais de forma exaustiva. É configurado para travar execuções a cada 1 segundo.
- Linha 16: este comando inicializa o *array* temporário onde são armazenados os pontos de coleta para posterior armazenamento em arquivo.

A seguir na Figura 31 a o código de inicialização do módulo de coleta de pontos:

Figura 31 - Módulo coletor de pontos: inicialização

```
79. rospy.init_node("pickpoint")
80.
81. rospy.Subscriber("quad/joy", Joy, pointCallback, queue_size=1)
82.
83. getValidity = rospy.ServiceProxy("/check_state_validity", GetStateValidity)
84. getPlanningScene = rospy.ServiceProxy("/get_planning_scene", GetPlanningScene)
85. components = PlanningSceneComponents()
86. components.components = 2
```

Fonte: Arquivo pessoal

Detalhes sobre a implementação abaixo:

- Linha 79: este comando inicializa o nodo *pickpoint.*
- Linha 81: este comando subscreve no tópico quad/joy e especifica a função *pointCallback* para executar ao receber uma mensagem neste tópico.

- Linha 83: este comando inicializa o objeto *getValidity* como um cliente do serviço *check_state_validity* oferecido pelo nodo *move_group*. O objetivo deste serviço é verificar a validade de uma posição do VANT.
- Linha 84: este comando inicializa o objeto *getPlanningScene* como um cliente do serviço *get_planning_scene* oferecido pelo nodo *move_group*. O objetivo desse serviço é obter o estado atual do VANT em relação à posição e orientação.
- Linhas 85 e 86: estes comandos inicializam a variável *component* que possui um valor inteiro 2. Esta variável é utilizada como parâmetro de entrada para chamar o serviço *getPlanningScene*. O valor inteiro 2 significa que o cliente está interessado em saber o estado do VANT.

4.4.5.4 Funções de suporte

A seguir na Figura 32 a primeira parte do código da função *pickpoint.*

Figura 32- Módulo coletor de pontos: primeira parte *pointCallback*

```
19. def pointCallback(msg):
20.     global nivel
21.
22.     # seta esquerda
23.     if (msg.axes[4] == 1):
24.         pass
25.
26.     # dir direita
27.     if (msg.axes[4] == -1):
28.         pass
29.
30.         # seta cima
31.     if (msg.axes[5] == 1):
32.         nivel += 1
33.         rospy.logwarn("Nivel: " + str(nivel + 1))
34.
35.     # dir baixo
36.     if (msg.axes[5] == -1):
37.         if nivel == 0:
38.             pass
39.         else:
40.             nivel += -1
41.             rospy.logwarn("Nivel: " + str(nivel + 1))
42.
43.     # select R2
44.     if (msg.buttons[7] == 1):
45.         del positions[:]
46.         rospy.logwarn("Posicoes resetadas")
47.
```

Fonte: Arquivo pessoal

Detalhes sobre a implementação abaixo:

- Linha 20: este comando referência a variável nível como uma variável global.
- Entre 22 e 28: Estes comandos servem apenas para detectar os acionamentos das setas esquerda e direita, contudo não possuem nenhuma ação
- Entre 30 e 42: Este trecho se refere ao aumento e diminuição do valor da variável nível por meio do acionamento das setas para cima e para baixo respectivamente. O limite inferior da variável nível é configurado para 0

- Entre 43 e 47: Este trecho trata da lógica apara deletar os pontos adicionados ao array temporário. Pode ser usado pelo usuário quando cometido algum erro de percurso.

A seguir na Figura 33 a segunda parte do código da função *pickpoint.*

Figura 33- Módulo coletor de pontos: segunda parte *pointCallback*

```
48.     #select button
49.     if (msg.buttons[8] == 1):
50.
51.         robot_state = getPlanningScene(components).scene.robot_state
52.
53.         state = GetStateValidityRequest()
54.         state.robot_state = robot_state
55.         state.group_name = "Quad_base"
56.
57.         validityResponse = getValidity(state)
58.
59.         if (validityResponse.valid):
60.             positions.append([robot_state, nivel])
61.             rospy.logwarn("Posicao de nivel " + str(nivel+1))
62.             rospy.logwarn(robot_state.multi_dof_joint_state.transforms[0])
63.         else:
64.             rospy.logwarn("Posicao nao valida!")
65.
66.     #start button
67.     if (msg.buttons[9] == 1):
68.         if len(positions) == 0:
69.             rospy.logwarn("Nenhuma posicao para salvar")
70.         else:
71.             file_number = len([f for f in listdir(path) if isfile(join(path, f))]
    )
72.             with open(path + "/list" + str(file_number) + ".txt", "w") as f:
73.                 pickle.dump(positions, f, 1)
74.
75.             rospy.logwarn("takepoint: Arquivo salvo!")
76.             del positions[:]
```

Fonte: Arquivo pessoal

Detalhes sobre a implementação abaixo:

- Entre 49 e 64: Este trecho trata da lógica para salvar um ponto no *array* temporário ao pressionar a tecla *select* no controle. Primeiro ocorre

uma requisição ao *move_group* por meio de uma chamada pelo cliente *getPlanningScene*. A resposta é o estado do VANT, que é verificada através e uma chamada pelo cliente *getValidity*. Caso o estado seja valido este é armazenado junto com o valor do nível atual no *array positions*, caso contrário a posição é descartada e um uma mensagem é enviada ao usuário.

- Entre 66 e 76: Este trecho trata da lógica para salvar um arquivo com as informações contidas no *array positions*. Caso não haja nenhuma posição, uma mensagem é enviada ao usuário alertando sobre a falta de pontos. Caso haja pontos um número referente ao sequenciamento de arquivo é detectado do diretório padrão e incrementado de 1 para gerar um novo nome de arquivo com o prefixo “list”. O arquivo é salvo utilizado o método *dump* da biblioteca *pickle* e uma mensagem de sucesso é enviada ao usuário.

4.4.5.5 Main

A seguir na Figura 34 o código e implementação da função *main.*

Figura 34 - Módulo coletor de pontos: *main*

```
88. rospy.wait_for_service('/get_planning_scene')
89.
90. while (not rospy.is_shutdown()):
91.
92.     r.sleep()
```

Fonte: Arquivo pessoal

Detalhes sobre a implementação abaixo:

- Linha 88: Antes de entrar na função main este comando mantem o programa bloqueado até que o serviço *get_planning_scene* esteja disponível.
- Linha 90: este comando mantem o loop *while* em operação até que o ROS master seja interrompido.

- Linha 92: este comando mantem uma taxa de execução de 1 segundo no loop *while.*

4.4.6 Módulo executor de rotas

O módulo executor de rotas gerencia a execução de rotas no VANT virtual e ao mesmo tempo envia as rotas para o gerenciador de voo que cria as missões e envia para o VANT real. O módulo executor é um programa que necessita de um parâmetro de entrada que é o arquivo de pontos, gerados a partir do módulo coletor de pontos. A execução das rotas é finita, isto é, uma vez que a rota foi executada o programa para, ao contrário do módulo coletor de pontos que é executado interruptamente. As principais publicações, subscrições e serviços utilizados por este módulo são:

- Serviços
  - *move_group*
  - *get_planning_scene*
- Subscrições
  - *dji/status*
  - *move_group/result*
  - *image/status*
- *Publicações*
  - *dji/command*
  - *dji/waypoint*

4.4.6.1 Execução das rotas no ambiente virtual

O módulo executor de rotas, ao iniciar, realiza a leitura do arquivo de pontos e insere os pontos em um *array*, em seguida insere na posição inicial, deste *array*, a posição atual do VANT, para que este possa sair da posição atual e ir ao primeiro ponto da rota sem colidir com obstáculos. De forma similar ao módulo coletor de pontos, a posição atual é obtida através do uso do objeto de serviço *getPlanningScene.*

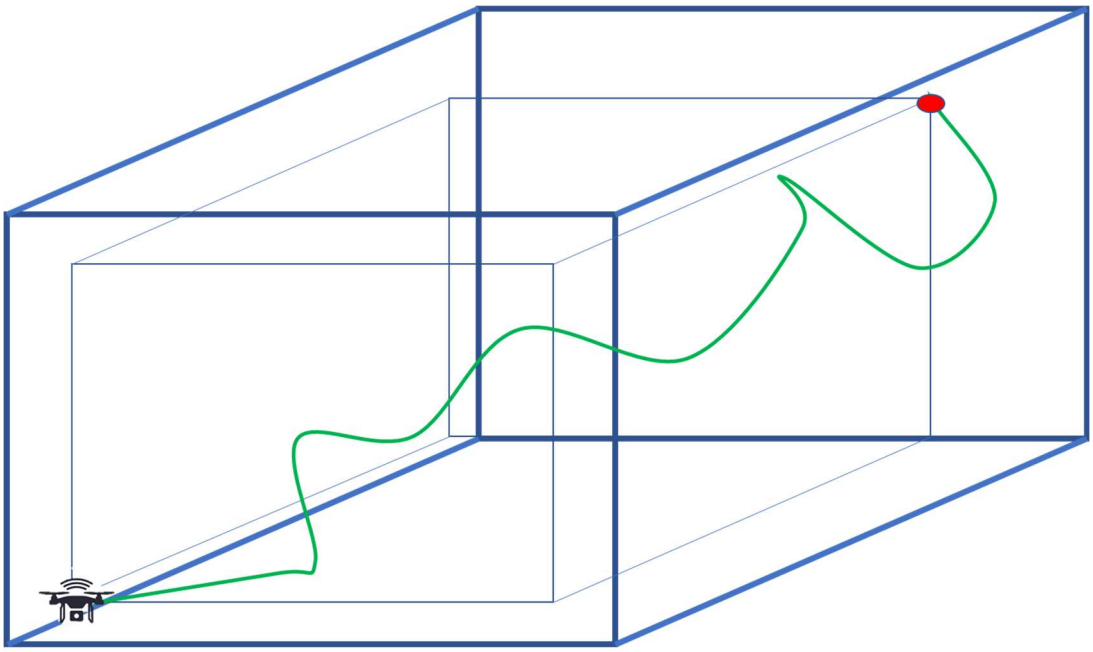
O módulo executor de rotas percorre o *array* de posições e realiza requisições ao nodo *move_group* através da interface *actionlib move_group.* Cada requisição contém o ponto onde o VANT está, e onde se deseja chegar. Ao realizar a requisição o módulo executor de rotas fica bloqueado esperando o resultado da execução.

O módulo executor de rotas ao enviar uma requisição de *sampled-motion-planning* define o tipo de algoritmo utilizado para o cálculo de rotas, neste caso o algoritmo escolhido foi o *RRTConnect* disponível na biblioteca OMPL.

O *RRTConnect* realiza duas procuras em árvore, uma partir do estado desejado e outra a partir do estado inicial (38). Os caminhos explorados não colidem com obstáculos, portanto a interseção dos caminhos representa um caminho livre de obstáculos entre os dois pontos. Optou-se por este algoritmo por causa de sua velocidade de execução, é rápido pois não possui otimização, e durante as execuções de rotas, devido a limitações de bateria, é importante que se economize a maior quantidade de tempo possível.

Contudo o *RRTConnect* pode resultar na execução de rotas ineficientes em que o VANT percorre caminhos muito longos para alcançar a posição desejada. Este problema foi reduzido ao inserir também na requisição, uma limitação do espaço permitido de planejamento. O espaço de planejamento é definido dinamicamente para cada par de pontos como um retângulo no qual extremos de um dos pares de arestas opostas são os pontos de desejado e de partida. Para dar flexibilidade de planejamento a este retângulo são adicionados valores de acréscimo em as direções de 5 m. Na Figura 35 um exemplo do volume de planejamento, o retângulo interno é o volume entre o ponto desejado e o inicial, enquanto que o externo é o volume inflado, que permite uma flexibilidade maior de planejamento.

Figura 35 - Volume de planejamento



Fonte: Arquivo pessoal

Ao receber a requisição, o servidor da interface *actionlib* executa o *sampled-motion-planning*, conforme descrito na seção 2.3, a fim de encontrar um caminho livre de obstáculos. Ao término do cálculo da rota, salva todas as posições calculadas no tópico *cmd_3dnav*. A partir da escrita de posições neste tópico o controlador de missões, descrito na seção 4.4.1.7, realiza a execução sequencial dos pontos da missão no VANT virtual.

O nodo *move_group* monitora através das transformadas entre a base do VANT e a origem do eixo coordenado do ambiente virtual, quando a requisição é concluída. Este monitoramento de transformadas é realizado através da consulta ao tópico *tf.* Ao alcançar a posição desejada, o *move_group* escreve os resultados no tópico *move_group/result* e libera o objeto de serviço no módulo executor de rotas.

Ao término da execução da requisição, o módulo executor de rotas envia as posições escritas no tópico *move_group/result* para o gerenciador de voo e fica aguardando o VANT chegar na posição desejada, porém agora no mundo real.

Após o recebimento da confirmação de chegada na posição no mundo real, através de comunicação com o gerenciador de voo, o módulo executor de rotas executa a próxima rota se houver.

4.4.6.2 Execução das rotas no ambiente real

A execução em ambiente real começa quando os pontos da rota são escritos no tópico *move_group/result*. O módulo executor de rotas possui um subscritor neste tópico e ao receber a mensagem de rota, prepara os pontos para serem enviados ao gerenciador de voo através do tópico *dji/waypoint*.

Ao receber a mensagem de rota o módulo executor envia um comando para o gerenciador de rotas solicitando que ocorra uma limpeza dos pontos antigos no gerenciador através do tópico *dji/command*.

Ao receber a confirmação da limpeza de pontos, o módulo executor filtra a mensagem recebida em *move_group/result* de modo que a distância mínima entre os pontos seja de 0,8 m. Esta limitação é oriunda do fabricante do VANT em uso, o DJI Inspire 1.

Apesar de não haver explicações no manual do fabricante ou do desenvolvedor, verificou-se na prática que missões com pontos com distâncias menores que 0,5 m não podem ser criadas. Essa limitação se confirmou com a leitura de fóruns de desenvolvimento, porém relacionados a outros modelos. Sabendo das distorções geradas de conversão entre coordenadas cartesianas e geográficas, a distância limite foi configurada para 0,8 m para garantir que a conversão não resulte em uma distância menor que 0,5 m.

O filtro de distância poderia resultar em colisões caso as distâncias entre os pontos originais forem reduzidas, porém essa condição pode ser evitada por dois meios. A distância de mínima de objetos, configurada nesse trabalho para 0,2 m, na configuração do *PlanningScene* faz com que o algoritmo procure caminhos com distância mínima de 0,2 m a qualquer fonte de obstáculo, o aumento deste parâmetro implica na geração de rotas mais seguras. E a proximidade dos pontos pode ser configurada para que seja a maior possível durante as requisições de planejamento de rota, com mais pontos próximos a chance de percorrer um longo caminho pelo corte do filtro é reduzida.

Após o filtro de distâncias, cada ponto da rota é convertido para coordenadas geométricas com o uso da biblioteca *geonav_transform* (39). Esta biblioteca realiza a conversão entre o sistema UTM e o sistema WGS84 (latitude e longitude). A conversão é obtida utilizando um ponto de referência em latitude e longitude que depende de onde se deseja navegar o VANT. O ideal é alinhar a origem o sistema de coordenadas local (2D) com a coordenada geométrica de referência.

A conversão ocorre primeiro convertendo o ponto de referência para o sistema UTM, o valor obtido é acrescido, em suas componentes X e Y, dos valores X e Y do ponto em 2D que se deseja converter. Em seguida as coordenadas em UTM são convertidas novamente para o sistema WGS84. A altura é mantida e enviada junto as coordenadas de latitude e longitude.

A implementação segue as especificações da DJI que indica que o ângulo em torno do eixo Z no VANT é definido em graus, e o valor 0 º está apontando para o norte, variando positivamente no sentido horário e dentro da faixa de -180 ° a 180 ° (40). A orientação contida nos pontos recebidos de rota está no formato de *quaternion*. Para converter o quaternion para ângulos de Euler (roll, pitch e yaw) foi utilizada a função *euler_from_quaternion* presente na classe *transformation* da biblioteca *tf* para ROS. Após a conversão somente o ângulo *yaw* (em torno do eixo Z) é convertido de radiano para graus e armazenado junto ao pacote de envio.

Após a conversão de coordenadas, estas são enviadas ao gerenciador de voo por meio do tópico *dji/command*. Após a conclusão do envio de todos os pontos, um comando para configuração de missão é enviado ao gerenciador, que ao receber configura a rota e responde através do tópico *dji/status/* se a configuração foi finalizada com sucesso. Configuração é um procedimento que verifica se a rota é válida.

Após a configuração, o módulo executor envia um comando para realizar o upload da rota no VANT, o gerenciador de voo realiza o upload e responde se a operação foi concluída com sucesso. Após o término do upload, o módulo gerenciador envia o comando para executar a missão, o gerenciador de voo começa a executar a missão e envia uma resposta que a execução está em andamento.

Ao alcançar a posição final o módulo gerenciador envia uma resposta ao módulo executor informando que o ponto final da rota foi alcançado.

4.4.6.3 Rotas adaptativas

As rotas adaptativas têm este nome devido a sua característica de se adaptar ao caminho guiado por uma indicação externa e por não possuir trajetórias fixas entre os pontos. As trajetórias podem ser diferentes para uma mesma inspeção, esta característica é bastante benéfica pois na ocorrência de uma alteração no mapa de obstáculos o sistema ainda é capaz de realizar rotas livre de colisão.

Ao percorrer o *array* de pontos originados do arquivo de entrada, o módulo executor espera pela confirmação de conclusão de rota, conforme descrito na seção anterior. Além desta verificação, se houver uma troca de nível entre um ponto e outro na execução, o módulo executor irá esperar por um segundo sinal. Este sinal é recebido pelo tópico *imagem/status*.

Cada ponto no *array* possui um valor de nível associado (ver seção 4.4.5.2 pra detalhes de como o nível é armazenado) e este valor de nível serve como guia de execução das rotas adaptativas. Cada número de nível representa uma rota completa e a alteração deste valor para um nível maior, significa que o módulo executor espera por um resultado de tratamento de imagem, o nível 1 é considerado a rota principal. A ideia é que se o tratamento de imagem resultar em um resultado insatisfatório o VANT executa uma adaptativa com objetivo de melhorar a análise de imagem do módulo.

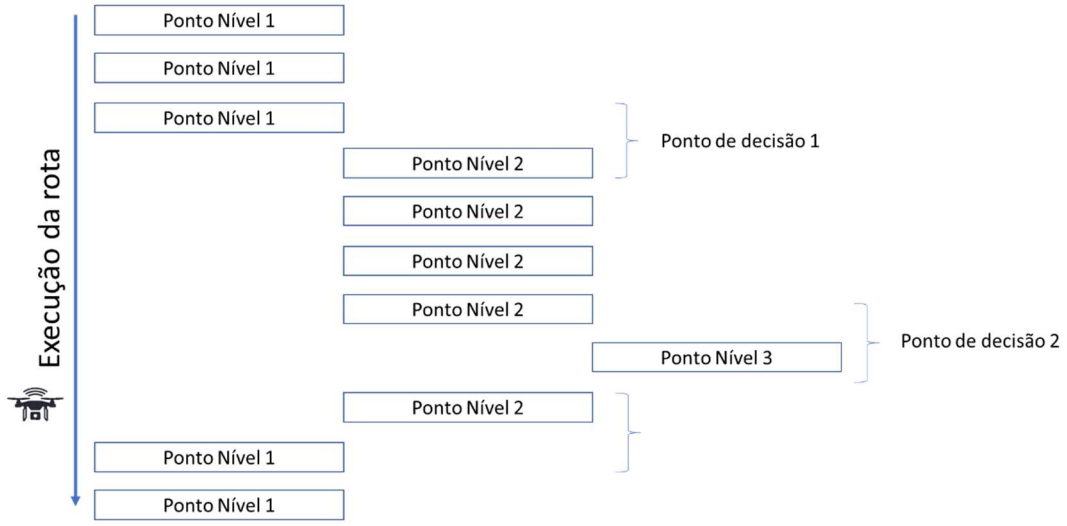
Este trabalho não é sobre o algoritmo de tratamento de imagem, porém o tópico, onde o módulo que futuramente realize essa função deve escrever, é especificado como *imagem/status*. O valor 1 significa que o VANT não deve realizar a rota, enquanto que -1 significa que o VANT deve realizar a rota alternativa.

Caso haja uma troca de nível e o módulo executor receba o comando para ignorar a rota adaptativa, todos os pontos com o valor de nível diferente do atual são ignorados até que o próximo ponto seja do mesmo nível que o atual.

Em uma troca de nível que o módulo executor de rotas recebe o comando para entrar na rota adaptativa o módulo executa em sequência todos os planeamento e execuções entre pontos do nível novo até que encontre um nível diferente. Se o nível for menor a execução continua sem interrupções do tratamento de imagem, caso o nível seja maior o módulo executor de rotas aguarda pelo resultado do tratamento de imagem.

As rotas adaptativas ocorrem no formato cascada, pode haver n níveis no arquivo e pode haver n trocas de um mesmo nível para outro. A ideia da execução das rotas adaptativas é exibida na Figura 36.

Figura 36 – Módulo executor de rotas: diagrama de níveis



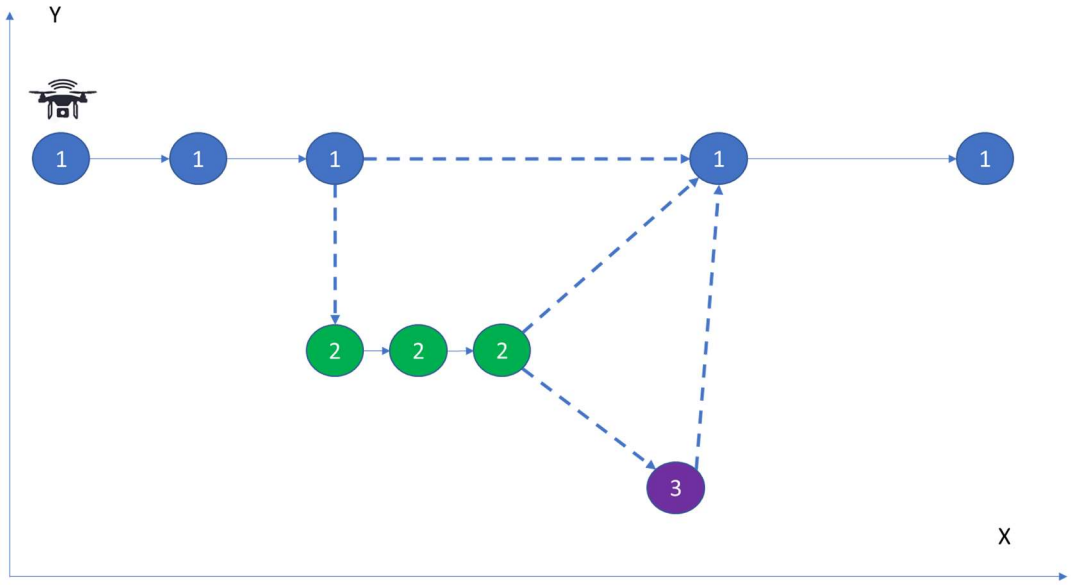
Fonte: Arquivo pessoal

A Figura 36 mostra que ao longo da rota existem dois pontos de decisão que dependem do tratamento de imagem. No primeiro ponto de decisão caso o resultado seja percorrer a rota adaptativa, todos os pontos de nível 2 são executados em sequência até a nova interrupção no ponto de decisão 2, caso contrário os pontos de nível 2 e 3 são ignorados até a ocorrência do ponto de nível 1 novamente.

O ponto de decisão 2 acontece quando no ponto de decisão 1 a opção foi executar a rota adaptativa. Ao chegar no ponto de decisão 2, caso o comando seja percorrer a rota adaptativa, o VANT se movimenta somente para uma posição do nível 3 e retorna sem interrupções para executar a última posição do nível 2 e em seguida voltar para o nível 1 sem interrupções. Caso o comando seja ignorar a rota, o VANT ignora o nível 3 e executa a última posição do nível 2, em seguida voltando sem interrupções para o nível 1.

A Figura 37 mostra em um plano, onde o Z é fixo para todos os pontos, e a posição em X, Y do VANT ao longo da execução da rota descrita na Figura 36.

Figura 37 – Módulo executor de rotas: possíveis rotas em 2D



Fonte: Arquivo pessoal

Na Figura 37 as linhas em tracejado representam as possibilidades de rotas que existem nesta aplicação. As linhas sólidas representam um sequenciamento obrigatório entre as posições e os números nos círculos os níveis das posições.

4.4.6.4 Main

A Figura 38 mostra a implementação da função *main* do módulo executor de rotas

Figura 38 - Módulo executor de rotas: *main*

```
449. for i in range(1, len(dump)): #len(dump)):
450. 
451.     current = dump[i - 1]
452.     goal = dump[i]
453. 
454.     if (i==1 or goal[1] <= current[1] and goal[1] != -9999 and current[1]!=-
     9999):
455.         send_goal(current[0], goal[0])
456. 
457.     elif (current[1] == -9999):
458.         print "Ignorando ponto intermediario"
459.         pass
460. 
461.     else:
462.         print "Aguardando resultado para rota adaptativa"
463.         #flag_dji_adaptative = input("flag:")
464.         while (flag_dji_adaptative == 0):
465.             r.sleep()
466. 
467.         if (flag_dji_adaptative == 1):
468.             print "Seguinto para proximo ponto da rota"
469.             #nao realizar rota adaptativa
470.             flag_dji_adaptative = 0
471.             for pos in dump[i:]:
472.                 if pos[1] == current[1]:
473.                     goal = pos
474.                     send_goal(current[0], goal[0])
475.                     break
476.                 else:
477.                     pos[1] = -9999
478. 
479.         if (flag_dji_adaptative == -1):
480.             print "Executando rota adaptativa"
481.             #realizar rota adaptativa
482.             flag_dji_adaptative = 0
483.             send_goal(current[0], goal[0])
```

Fonte: arquivo pessoal

Detalhes da implementação:

- Entre 449 e 453: Para cada ponto presente no *array* de posições a variável *current* e *goal* são definidas com a posição de onde o VANT está e onde se deseja alcançar.
- Entre 454 e 455: Caso o ponto atual seja o primeiro, ou o próximo ponto possua um nível menor, e o ponto objetivo e atual tenham nível

diferente de -9999 o módulo executa o planejamento e execução da trajetória através do comando *send_goal*. Ao ignorar um nível, todos os níveis dos pontos ignorados para um trecho são alterados para -9999, portanto é exigido que o nível seja *diferente* de -9999 para evitar executar rotas ignoradas.

- Entre 457 e 459: caso o ponto tenha o nível configurado em -9999, uma mensagem ao usuário aparece informando que aquele ponto está sendo ignorado.
- Linha 461: Caso as condições da linha 454 não sejam atendidas, significa que a execução da rota deve aguardar um sinal de tratamento de imagem.
- Entre 464 e 465: O módulo aguarda a o tratamento de imagem responder sobre a sobre a execução da rota adaptativa ou não. O valor 0 significa que ainda não houve resposta.
- Entre 467 e 477: caso a resposta do tratamento de imagem seja para não seguir a rota adaptativa o módulo percorre o *array* de posições configurando o nível de cada posição para -9999 até que o nível atual seja encontrado novamente.
- Entre 479 e 483: caso a resposta do tratamento de imagem seja positiva em relação a percorrer a rota adaptativa, o comando *send_goal* é executado para os valores de posição atual e desejada.

4.4.6.5 Envio de pontos completo

A Figura 39 mostra parte da implementação da função de envio de pontos

Figura 39 - Módulo executor de rotas: envio de pontos

```
217.        waypoint.pose.position.x = lat
218.        waypoint.pose.position.y = lon
219.        waypoint.pose.position.z = point.transforms[0].translation.z
220.
221.        waypoint.pose.orientation.x = point.transforms[0].rotation.x
222.        waypoint.pose.orientation.y = point.transforms[0].rotation.y
223.        waypoint.pose.orientation.z = point.transforms[0].rotation.z
224.        waypoint.pose.orientation.w = round(math.degrees(rpy[2]))
225.
226.
227.        print waypoint.pose.position
228.        print "----------------------------"
229.        dji_traj_pub.publish(waypoint)
```

Fonte: arquivo pessoal

Detalhes da implementação:

- Entre 217 e 224: Inserção dos valores de latitude, longitude, altura, roll, pitch, yaw e yaw em gruas no objeto de dados que será enviado ao gerenciador de voo.
- Entre 227 e 229: envio do objeto de posição para o gerenciador de voo pelo publicador *dji_traj_pub.*

Na Figura 40 parte da implementação do procedimento para começar uma missão após o envio dos pontos

Figura 40 - Módulo executor de rotas: procedimento para carregar pontos no gerenciador de voo

```
253.        # logica de configuracao dos pontos
254.        print "Configurando..."
255.        dji_command.publish("config")
256.        while configured == 0:
257.            r.sleep()
258.
259.        configured = 0
260.
261.        sleep(1)
262.
263.        # logica de upload
264.        print "Uploading..."
265.        dji_command.publish("upload")
266.        while uploaded == 0:
267.            r.sleep()
268.
269.        uploaded = 0
270.
271.        sleep(1)
272.
273.        # logica de start mission
274.        print "Mission Start..."
275.        dji_command.publish("start")
276.        while started == 0:
277.            r.sleep()
278.
279.        started = 0
280.        sleep(1)
281.        print "Comecou!"
```

Fonte: Arquivo pessoal

Detalhes da implementação:

- Entre 253 e 262: Envio do comando "*config*" pelo publicador *dji_comand* e posterior aguardo da *flag* de resposta. Ao receber a confirmação escreve na *flag configured* o valor 0.
- Entre 263 e 271: Envio do comando "*upload*" pelo publicador *dji_comand* e posterior aguardo da *flag* de resposta. Ao receber a confirmação escreve na *flag uploaded* o valor 0.
- Entre 273 e 281: Envio do comando "*start"* pelo publicador *dji_comand* e posterior aguardo da *flag* de resposta. Ao receber a confirmação escreve na *flag started* o valor 0.

4.4.6.6 Máquina estados de recebimento de mensagens do DJI

A Figura 41 alustra a implementação a máquina de estados que trata as mensagens recebidas do gerenciador de voo.

Figura 41 - Módulo executor de rotas: callback do gerenciador de voo

```
107. def DJICallback(msg):
108.     print msg
109.     global flag_dji, configured, started, uploaded, cleared
110.
111.     if msg.data == "chegou":
112.         flag_dji = 1
113.
114.     if msg.data == "configured":
115.         configured = 1
116.
117.     if msg.data == "uploaded":
118.         uploaded = 1
119.
120.     if msg.data == "started":
121.         started = 1
122.
123.     if msg.data == "cleared":
124.         cleared = 1
125.
```

Fonte: Arquivo pessoal

Detalhes da implementação:

- Linhas 109: Este comando referência as variáveis globais que representam as *flags* que o módulo aguarda para liberar a execução do código, quando chama por funções especificas.
- Linhas 111 e 112: Ao receber a mensagem “chegou” configura a *flag flag_dji* para 1.
- Linhas 114 e 115: Ao receber a mensagem “*configured*” configura a *flag configured* para 1.
- Linhas 117 e 118: Ao receber a mensagem “*uploaded*” configura a *flag uploaded* para 1.
- Linhas 120 e 121: Ao receber a mensagem “*started*” configura a *flag started* para 1.

- Linhas 123 e 124: Ao receber a mensagem “*cleared*” configura a *flag cleared* para 1.

4.4.7 Gerenciador de voo e simulação de controlador de voo

O gerenciador de voo permite o controle manual do VANT através da simulação de *joysticks* virtuais na interface do usuário, simulação de controlador de voo e o controle autônomo via comandos gerados na camada ROS

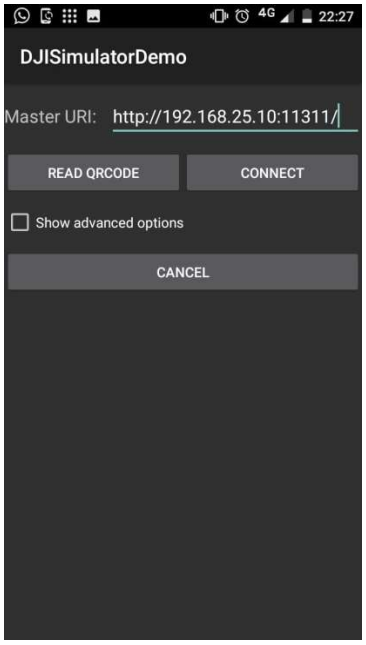
Devido a obrigatoriedade de ser implementado em um sistema Android, devido a disponibilidade da biblioteca DJI ser apenas na linguagem Java para este modelo de VANT, o gerenciador foi construído a partir de dois exemplos de implantação em linguagem Java.

Um exemplo foi retirado do site do fabricante DJI (41). O código utilizado possui as funcionalidades de comando manual e simulação do controlador de voo. O código foi então estendido de forma a atender as demandas específicas do projeto. O outro código foi retirado de exemplos disponíveis no ros.org (42), este exemplo traz as funcionalidades do ROS, como publicações e subscrições em tópicos na rede, para a linguagem Java.

Uma vez integrados e com adaptações, ajustes e códigos desenvolvidos pelo autor, a integração fornece as funcionalidades de controle manual, simulação de controle de voo e controle de missões via ROS com mensagens trocadas nos tópicos *dji/command* e *dji/status*.

Para realizar a junção dos dois módulos a classe principal do exemplo de ROS foi estendida na declaração da classe principal do módulo da DJI, onde o código específico para tratamento de mensagens trocadas via ROS foi implementado. Com esta extensão, ao executar o aplicativo, antes da classe DJI aparecer, a classe do ROS pede pelas configurações da rede ROS onde deve ser registrado o gerenciador. A interface da configuração ROS pode ser vista na Figura 42.

Figura 42 - Gerenciador de voo: interface ROS



Fonte: arquivo pessoal

Para configurar o gerenciador na rede ROS basta digitar o IP e a porta 11311 do computador onde o ROS está sendo executado no campo *MasterURI* e apertar em *connect*. Após a execução do registro na rede ROS, o gerenciador de voo inicializa a classe DJI para o controle do VANT. A interface da classe DJI pode ser visualizada na Figura 43.

Figura 43 - Gerenciador de voo: interface DJI

Fonte: arquivo pessoal

Quando a interface DJI é executada ocorre uma comparação de valores no servidor do fabricante. Esta validação é indispensável para que a biblioteca funcione e é uma exigência do fabricante. Um registro de desenvolvedor deve ser realizado junto ao site da DJI e nas informações de desenvolvimento devem ser informados os aplicativos que estão sendo programados.

Para cada aplicativo é gerada uma chave de registro que deve ser inserida no código dos aplicativos desenvolvidos para que ao serem executados, possa ser realizada a verificação com o site. Esta é a razão pela qual é necessário que o gerenciador e a estação de comando estejam em uma rede com internet.

A descrição da interface DJI pode ser observada nos botões e caixas de texto da interface exibida na Figura 43:

- *Enable Virtual Stick*: este botão ativa o modo o qual o operador pode controlar o VANT via Sticks simulados na interface.
- *Start Simulator*: este botão ativa o controlador de voo simulado, permitindo o usuário controlar o um VANT virtual, via *Virtual Sticks* ou controle autônomo.
- *Disable Virtual Stick*: este botão desativa o modo de simulação. Este botão apenas funciona se o controlador de voo estiver simulado.

- *Take off*: este botão coloca o VANT à 1 m do chão de onde estiver pousado. Este comando funciona se o VANT estiver pousado.
- *Land*: Este comando pousa o VANT em qualquer posição em que este esteja.
- *Virtual Stick:* estes são os controles virtuais que comando o VANT quando o modo de *Virtual Stick* está ativo. Se o modo *Virtual Stick* não estiver ativo o acionamento dos controles não tem efeito.

Podem ser observador os campos de texto:

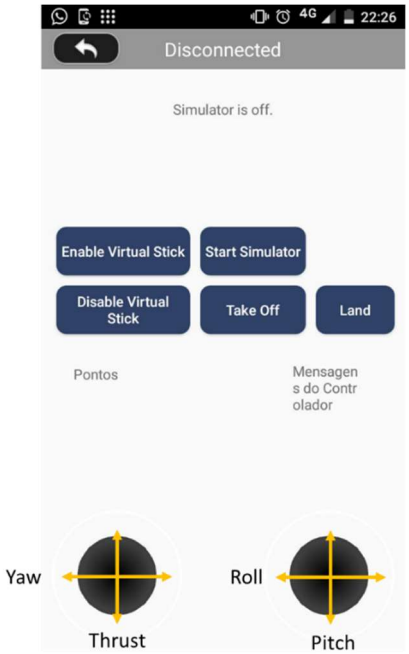
- Pontos: Esta caixa de texto mostra os valores de latitude, longitude, altura e orientação para cada ponto enviado pelo módulo executor de rotas.
- Mensagem do controlador: Esta caixa de texto mostra os valores os comandos enviados pelo módulo executor de rotas.
- Campo de texto que mostra a mensagem "*Disconnected":* esse campo apresenta o nome do VANT quando conectado ao celular.
- Campo de texto que mostra a mensagem *"Simulador is off":* Este campo, ao conectar com o VANT real, exibe a atual localização e orientação do VANT. Os valores de localização são dados em latitude, longitude e altura, os valores de orientação são exibidos me termos de roll, pitch e yaw em graus. Estas informações continuam sendo exibidas caso o VANT seja trocado pelo controlado de voo simulado.

O gerenciador possui dois modos de operação na solução o manual e o autônomo, ambos os modos podem ser executados com o VANT real ou, se ativado na interface do celular, no ambiente simulado. A seguir são descritos os modos de funcionamento do VANT no gerenciador de voo.

4.4.7.1 Modo de controle manual

O modo de controle manual é ativado quando o usuário ativa os *Virtual Sticks* para controlar o VANT. Nesta configuração o VANT se movimentará de acordo com os comandos do usuário, tanto no modo real quanto no modo simulado se o usuário ativar o controlador de voo simulado. As direções dos controles são exibidas na Figura 44.

Figura 44 - Gerenciador de voo: controle manual



Fonte: Arquivo pessoal

4.4.7.2 Modo de controle autônomo

O módulo de controle autônomo é ativado a partir de uma sequência de mensagens originadas no módulo executor de rotas e repostas dos resultados gerados pelo controlador de voo. Para executar comandos o executor de rotas deve publicar os comandos no tópico *dji/command*, e o gerenciador de voo deve publicar suas respostas no tópico *dji/status*.

O primeiro comando da sequência para enviar uma rota para o VANT é o comando "*clear*". O comando *clear* indica para o gerenciador zerar o *array* temporário de recebimento de pontos.

O segundo passo é o envio dos pontos pelo módulo executor de rotas via tópico no ROS. Cada ponto ao ser recebido no tópico *dji/waypoint,* é inserido em um

array temporário e as informações de posição e orientação do ponto são exibidas na interface principal.

O terceiro passo, após o fim do envio das posições, é o recebimento mensagem "*config*" originado no executor de rotas. O procedimento de configuração, com base no array temporário, cria uma missão com as seguintes configurações:

- Orientação: a orientação a ser seguida deve ser a orientação de cada ponto.
- Velocidade máxima: 10 m/s.
- Método e execução: o VANT é transportado de ponto para ponto seguindo uma linha reta entre eles.

Em seguida o procedimento de configuração da missão verifica se não há inconsistências na rota gerada. Um exemplo de inconsistência neste caso pode ser pontos com uma distância menor que 0,5 m na missão. Caso haja inconsistência na missão responde ao módulo executor de rotas que algum problema ocorreu, caso não haja erros responde ao módulo executor que a operação de configuração foi realizada com sucesso.

O quarto passo é, após receber a confirmação de configuração realizada com sucesso, o envio do comando "*upload*" pelo executor de rotas. O gerenciador de voo ao receber o comando realiza o *upload* da missão configurada no VANT, caso não haja erros responde que a missão foi carregada com sucesso no VANT, caso contrário envia uma mensagem indicando o erro ocorrido ao executor de rotas.

O quinto passo é o envio do comando "*start*" pelo executor de rotas. Ao receber este comando o gerenciador de voo indica o início da missão para o VANT. Caso haja erro na inicialização da missão responde ao módulo executor de rotas que algum problema ocorreu, caso não haja erros responde ao módulo executor que a missão foi inicializada com sucesso.

O sexto passo acontece quando a missão é concluída. Neste passo o gerenciador de voo envia ao módulo executor de rotas a mensagem "*chegou*". Este sexto e último passo conclui um ciclo de execução de missão. Ao receber a mensagem

de confirmação de conclusão de rotas, o executor de rotas toma as providencias descritas em 4.4.6.1 e 4.4.6.2.

4.4.7.3 Máquina de estados

A Figura 45 traz implementação da máquina de estados de recebimento de mensagens do módulo executor de rotas é exibida.

Figura 45 - Gerenciador de voo: máquina de estados

```
165. //recebe comandos
166. rosTextView.setMessageToStringCallable(new MessageCallable<String, std_msgs.Strin
     g>() {
167.     @Override
168.     public String call(std_msgs.String message) {
169.         switch (message.getData())
170.         {
171.             case "config":
172.                 configWayPointMission();
173.
174.                 break;
175.
176.             case "clear":
177.                 clearWaypointList();
178.                 rosTextView.setPublishMessage("cleared");
179.                 break;
180.
181.             case "stop":
182.                 stopWaypointMission();
183.
184.                 break;
185.
186.             case "simulaChegada":
187.                 rosTextView.setPublishMessage("chegou");
188.                 break;
189.
190.             case "upload":
191.                 uploadWayPointMission();
192.
193.                 break;
194.
195.             case "start":
196.                 startWaypointMission();
197.
198.                 break;
199.
200.
201.         }
```

Fonte: Arquivo pessoal

A seguir detalhes de implementação:

- Linha 169: este mecanismo serve para detectar qual mensagem foi recebida por este *callback* e dar-lhe o encaminhamento necessário.
- Entre 171 e 198. Dependendo da mensagem que chega ao gerenciador de voo, uma lista de comandos é executa. Além das funções já descritas o gerenciador de voo apresenta um procedimento para o comando “simulaChegada”, que escreve no tópico *dji/status* a mensagem “chegou”, simulando uma chegada do VANT na posição desejada. É bastante útil para *debug* do executor de rotas.

4.4.7.4 Recebimento dos pontos

A Figura 46 mostra implementação da lógica de recebimento de pontos do módulo executor de rotas.

Figura 46 - Gerenciador de voo: recebimento de pontos

```
212. rosTextView.setMessageToStringCallablePoint(new MessageCallable<String, geometry_
     msgs.PoseWithCovariance>() {
213.           @Override
214.           public String call(geometry_msgs.PoseWithCovariance message) {
215.
216.               //cria ponto recebido
217.               Waypoint point = new Waypoint(message.getPose().getPosition().getX
     (), message.getPose().getPosition().getY(), (float) message.getPose().getPosition
     ().getZ());
218.
219.               //configura o heading
220.               point.heading = (int) Math.round(message.getPose().getOrientation(
     ).getW());
221.
222.               //adiciona ponto na lista
223.               addWaypoint(point);
224.
225.
226.               //mostra na tela o ponto recebido
227.               String point_message = ("Latitude: "+ Double.toString(point.coordi
     nate.getLatitude()) +
228.                       " Longitude: " + Double.toString(point.coordinate.getLongi
     tude()) +
229.                       " Altitude: " + Double.toString(point.altitude) +
230.                       " Heading: " + Integer.toString(point.heading));
231.
232.               mPointView.setText(point_message);
```

Fonte: Arquivo pessoal

A seguir detalhes de implementação:

- Linha 217: É inicializado um objeto *waypoint* fornecido pela SDK DJI para Android. Neste objeto é inserida os dados de latitude, longitude e altura.
- Linha 220: O valor do ângulo em trono do eixo Z é inserido na propriedade *heading* do objeto *waypoint*.
- Linha 223: O objeto *waypoint* é inserido no *array* temporário de pontos
- Entre 227 e 230: Uma variável do tipo *string* é inicializada e os nela escritos os valores de latitude, longitude, altura e orientação.
- Linha 232: A *string* contendo as informações do ponto é publicada na interface do usuário.

4.4.8 Inicialização do sistema gerenciador de rotas

Para executar os módulos de implementação observados nesse capítulo é necessário realizar configurações na estação de trabalho, executar uma *launch file* e realizar as conexões físicas nos componentes que integram o sistema.

Esta seção descreve as configurações do ROS que devem ser realizadas na estação de trabalho, o *launch file* necessário para instanciamento dos módulos do sistema e apresenta uma lista de procedimentos para correta inicialização da solução.

4.4.8.1 Servidor ROS

Para realizar a configuração do servidor na estação de comando, é necessário inserir no arquivo *.bashrc* a o comando:

- *export* ROS_IP=”IP da máquina”:11311

No lugar de “IP da máquina” é o IP da estação de comando.

4.4.8.2 Launch file

Para a inicialização do sistema é necessário executar um *launch file* (detalhes do *launch file* ver seção 2.1.7) que inicializa os nodos necessários para a execução dos testes. Os nodos inicialização são:

- Gazebo + ambiente + VANT virtual.
- *MoveIt*.
- *pickpoint.py.*
- *quad_joystick_interface.*
- *position_controller_node.*
- *atitude_controller_node.*
- *waypoint_node_publisher.*

- *joint_state_node.*
- *robot_state_publisher.*
- *action_controller.*
- *joy_node.*
- *robot_description.*
- *launch_map.launch.*
- *Rviz.*

A Figura 47 apresenta um trecho do código do deste *launch file* onde é inicializado o *pickpoint* e o *launch_map.launch*.

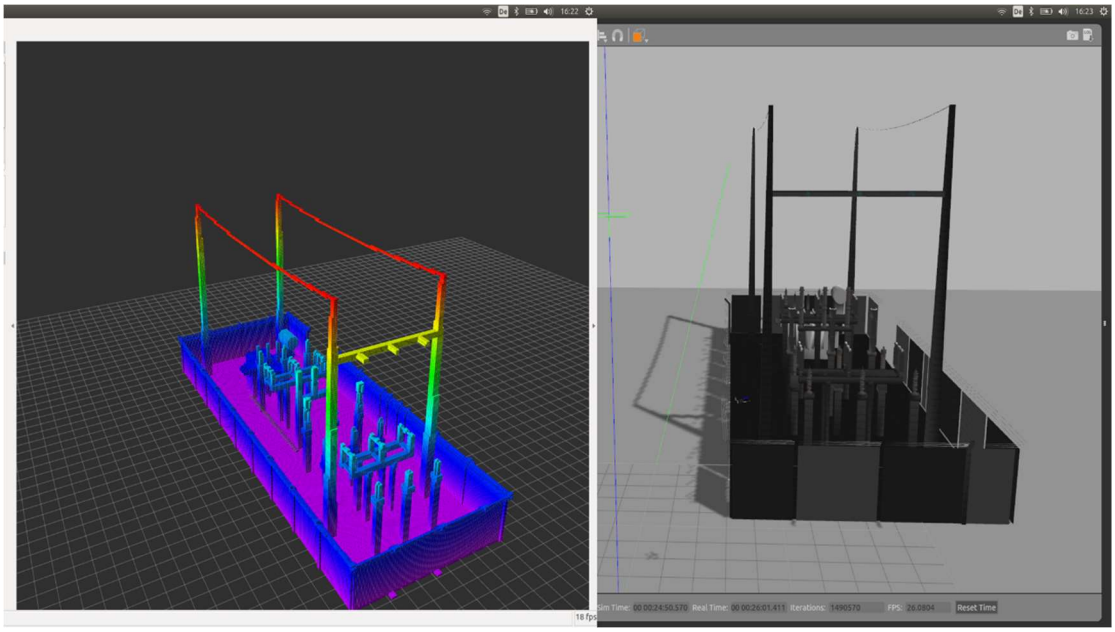
Figura 47 - Launch file

```
1. <node pkg="quad_collect" type="pickpoint.py" name="moveit_pickpoint" output="s
   creen" />
2.
3. <!-- Manage the octomap update -->
4. <include file="$(find quad_octomap)/launch/launch_map.launch"/>
```

Fonte: Arquivo pessoal

Após o término da execução do *launch file*, é exibida a tela do do Rviz e do Gazebo conforme composição na Figura 48.

Figura 48 - Subestação no RViz e Gazebo



Fonte: Arquivo pessoal

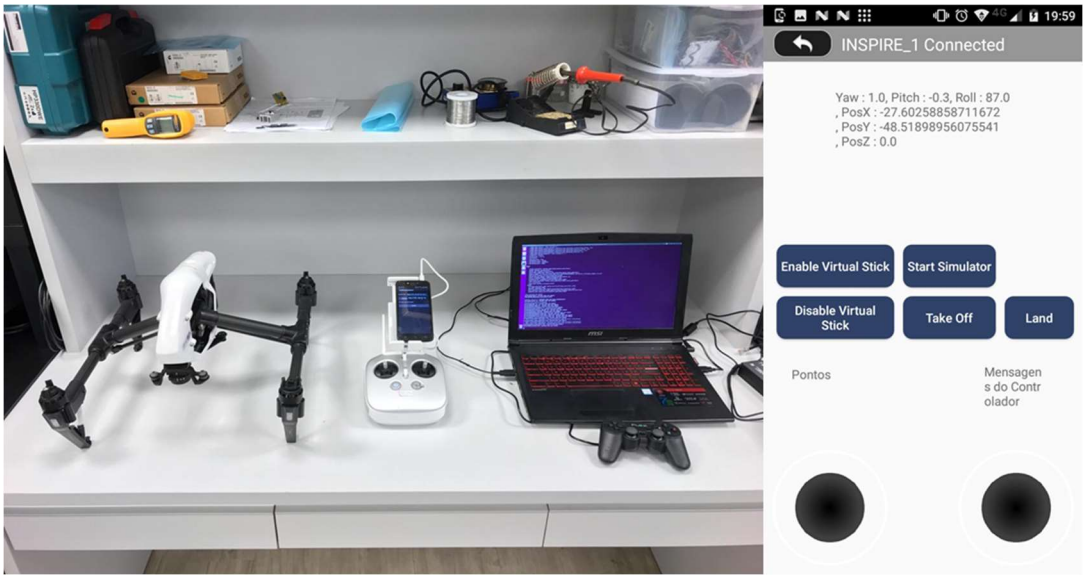
4.4.8.3 Procedimentos para inicialização

A seguir os procedimentos de conexão:

- Ligar VANT *Inspire* 1.
- Ligar controle de rádio.
- Posicionar celular no *socket* do controle de rádio e ajustar na posição.
- Conectar a ponta USB de um baco USB-Mini, no controle de rádio um cabo USB.
- Conectar o celular a uma rede Wifi com conexão à internet, ou hospedar uma rede, porém com conexão à internet.
- Conectar o controle genérico USB na estação de comando.
- Ligar a estação de comando e conectar na mesma rede Wifi. Executar o comando "*roslaunch quad_3dnav quad_3dnav*" no terminal.
- Inicializar o aplicativo *DJISimulatorDemo* (gerenciador de voo) no celular, e digitar o IP no campo MASTER URI e clicar em conectar.
- Conectar o a parte mini do cabo USB no celular. Na caixa de mensagem escolher o aplicativo *DJISimulatorDemo* e aguardar a mensagem *"Inspire 1 Connected"* na caixa de texto superior.
- Girar 4 vezes a tecla de função *Land* no controle de rádio para fazer o VANT entrar na posição de pouso.

A configuração física (esq.) e do gerenciador de voo (dir.) do teste pode ser observada na Figura 49.

Figura 49 - Configuração de teste



Fonte: Arquivo pessoal

5 RESULTADOS

Nesse capítulo são descritos os testes realizados sobre o sistema gerenciador de rotas afim de se avaliar as principais funcionalidades. As funções avaliadas são:

- Teste de desvio de obstáculos e orientação.
- Teste de rotas adaptativas.
- Teste de filtro.
- Teste de geração de pontos GPS.
- Teste de comando e resposta GPS.

Com o objetivo de facilitar a compressão e padronizar a estrutura de teste, o seguinte plano foi seguido para cada uma das avaliações:

- Definição do objetivo do teste.
- Definição do resultado esperado.
- Definição do procedimento de teste.
- Análise dos resultados.

Para todos os testes é considerado que:

- Os procedimentos descritos em 4.4.8.3 foram realizados para inicialização dos módulos do sistema.
- O alinhamento da subestação virtualizada não foi realizado com nenhuma referência real.
- Para o teste com o gerenciador de voo, além dos procedimentos de inicialização, o botão “*Start Simulator*” foi pressionado para simular o controlador de voo do VANT.
- Para testes sem o gerenciador de voo, o sistema gerenciador de rotas foi configurado para não esperar as mensagens de *feedback*.

5.1 Teste de desvio de obstáculo e orientação

Este teste tem por objetivo validar se uma rota gerada entre dois pontos evita colisão, e verificar se a orientação inicial e final de uma rota gerada entre dois pontos é próxima às da coleta de pontos.

É esperado que a rota planejada entre dois pontos não passe por nenhum objeto e que a orientação dos pontos inicial e final da rota gerada entre dois pontos seja próxima às da coleta.

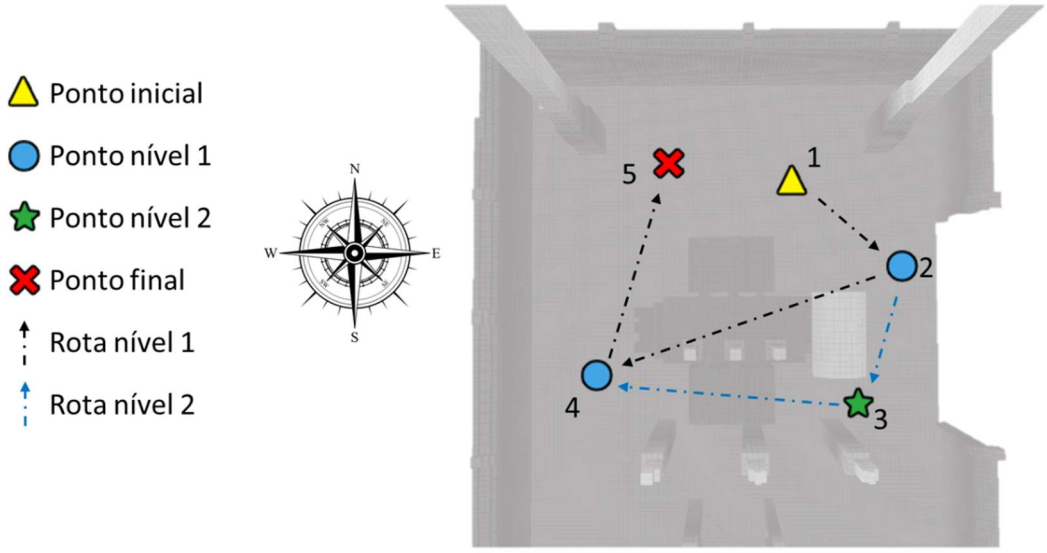
Para validar o teste uma análise visual de uma rota entre dois pontos será realizada.

5.1.1 Procedimento

Para realizar este teste é necessário realizar a execução de uma rota e analisar o percurso entre dois pontos, portanto, é necessário criar um arquivo de pontos de inspeção. O procedimento é descrito em duas partes, uma referente ao processo de coleta de pontos de inspeção e outra a execução do arquivo.

Para realizar a coleta de pontos o modo manual foi ativado pressionando a tecla “Círculo” do controle USB. Em seguida o VANT foi posicionado em 5 pontos diferentes da subestação virtual, sendo que 1 desses pontos foi configurado como rota adaptativa. Para coleta de pontos o botão *select* foi pressionado após a estabilização em cada uma das posições. Os pontos coletados são representados na Figura 50, as setas entre os pontos representam as rotas. Detalhes sobre a lógica de pontos adaptativos verificar seção 4.4.6.3.

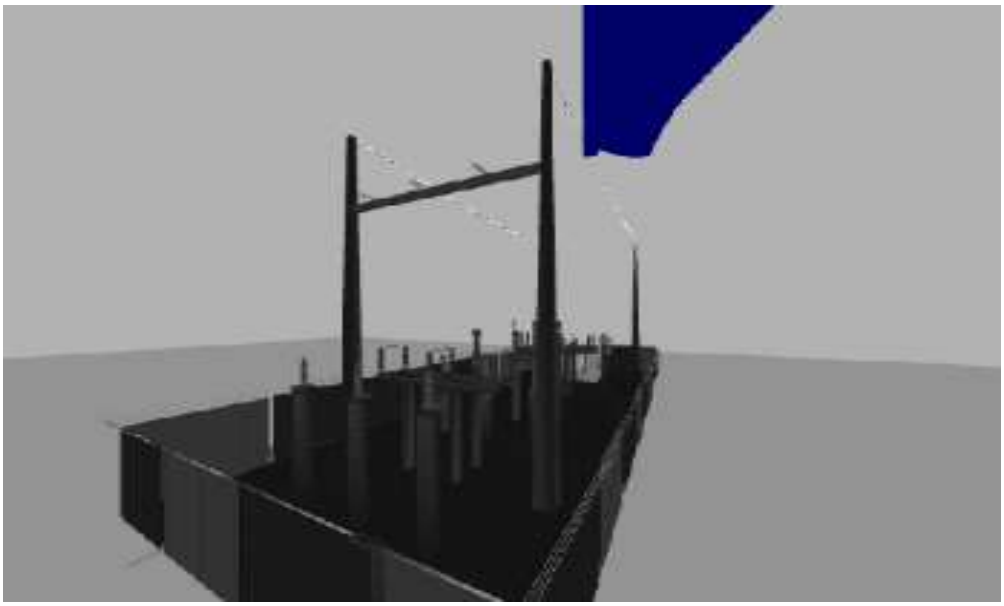
Figura 50 - Pontos coletados



Fonte: Arquivo pessoal

A Figura 51 representa a visão em primeira pessoa disponível no RViz utilizada durante o procedimento de coleta de pontos.

Figura 51 - Visão de primeira pessoa no VANT



Fonte: Arquivo pessoal

Após um fim da coleta o botão *start* foi pressionado par gerar o arquivo de pontos. A Tabela 2 apresenta as componentes dos pontos coletados.

Tabela 2 - Pontos X Y coletados

Ordem	X (m)	Y (m)	Altura (m)	*Yaw (°)*	Nível
1	9,83	29,56	2,02	180,0	1
2	11,43	27,74	2,02	-136,0	1
3	11,05	24,96	2,02	-31,0	2
4	5,13	25,67	2,02	62,0	1
5	6,27	30,12	2,02	5,0	1

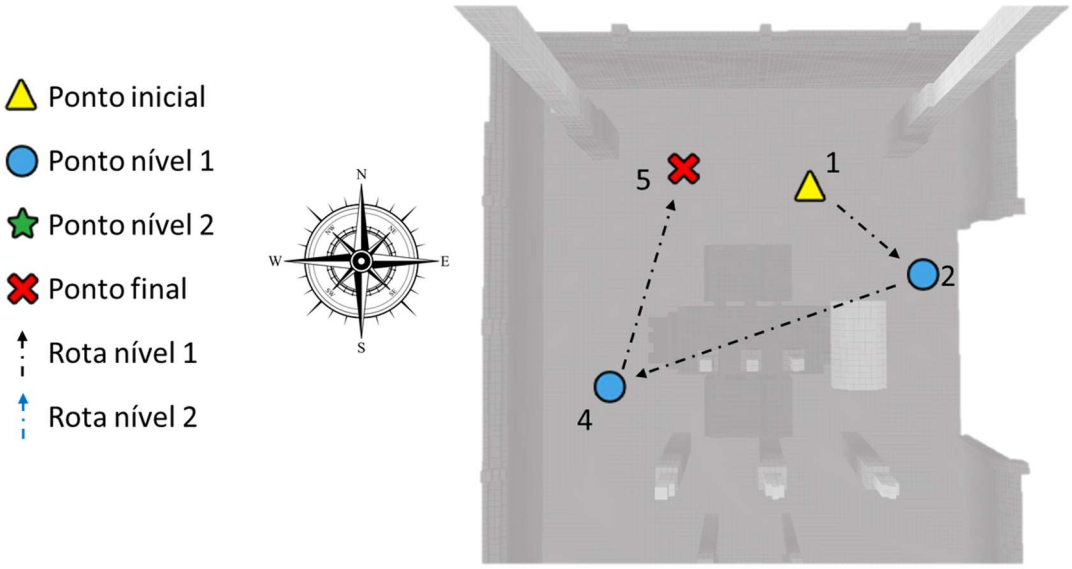
Fonte: Arquivo pessoal

Para realizar a execução da rota o modo automático foi ativado pressionando a tecla L1 do controle USB. O módulo executor de trajetória foi configurado para não executar nenhuma rota adaptativa. Em seguida o nodo *pointByPointPlanning* foi executado com o seguinte comando:

- *rosrun quad_planning pointByPointPlanning.py list17.txt*

Este nodo representa módulo executor de trajetórias, descrito em 4.4.6. O arquivo *list17.txt* foi o arquivo gerado a partir da coleta de pontos realizada no item anterior. A rota gerada é representada na Figura 52.

Figura 52 - Execução de rota parcial

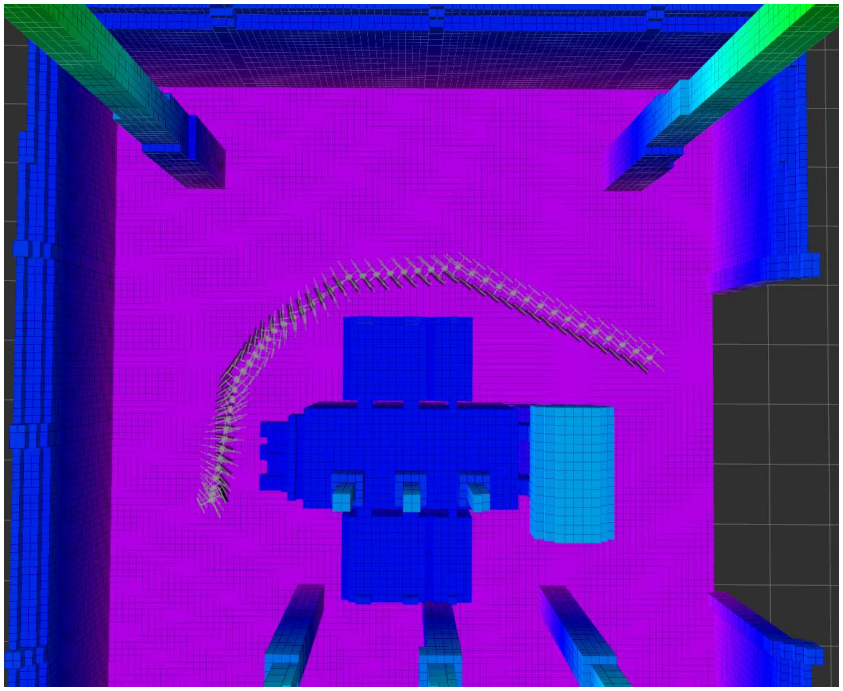


Fonte: arquivo pessoal

5.1.2 Análise

Para realizar a análise de colisão e orientação foi escolhida a rota gerada entre os pontos 2 e 4 da Figura 52. A Figura 53 exibe a rota entre 2 e 4 no momento de execução. Observa-se visualmente que o planejamento da rota foi realizado de forma a evitar a colisão com o transformador.

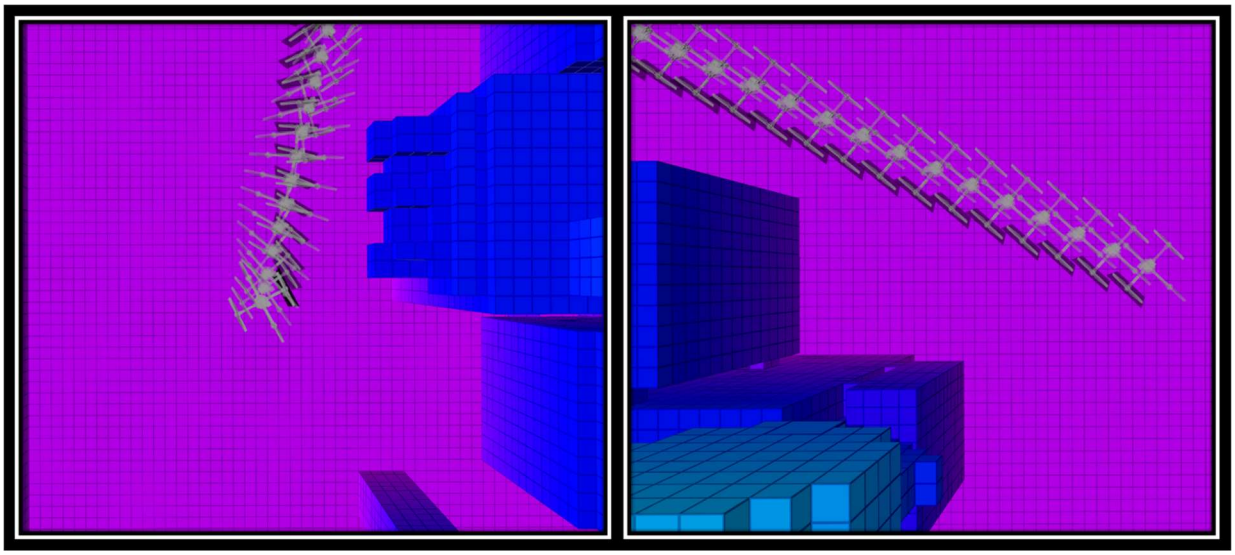
Figura 53 - Rota evitando obstáculo



Fonte: Arquivo pessoal

A Figura 54 representa a orientação no ponto final (esquerda) e a orientação no ponto inicial (direita) do trecho.

Figura 54 - Teste de orientação do VANT



Fonte: Arquivo pessoal

A implementação do módulo executor de rotas foi realizada de forma que o VANT virtual para apontar para o norte com a câmera, deve ser enviado para um ponto com orientação de 0°. Portanto na Figura 54 à direita espera-se uma leitura entre -90 ° e -180 ° e à esquerda uma leitura entre 0 ° e 90 °. Os valores obtidos para o ponto 2 e 4 na Tabela 2 atendem à essa expectativa, portanto a orientação da rota gerada é compatível com os pontos coletados.

5.2 Teste de rotas adaptativas

Este teste tem por objetivo verificar a execução completa de uma rota de inspeção, e verificar se a decisão da lógica adaptativa é respeitada ao longo da execução da rota.

É esperado que o sistema gerenciador de rotas comande o VANT entre o ponto inicial e final do arquivo de pontos, e que no ponto 2 o sistema gerenciador de rotas dependa de uma *flag* para seguir ao ponto 3, rota adaptativa, ou ao ponto 4 sem rota adaptativa.

Para validar esse comportamento uma avaliação visual da execução da rota completa será realizada.

5.2.1 Procedimento

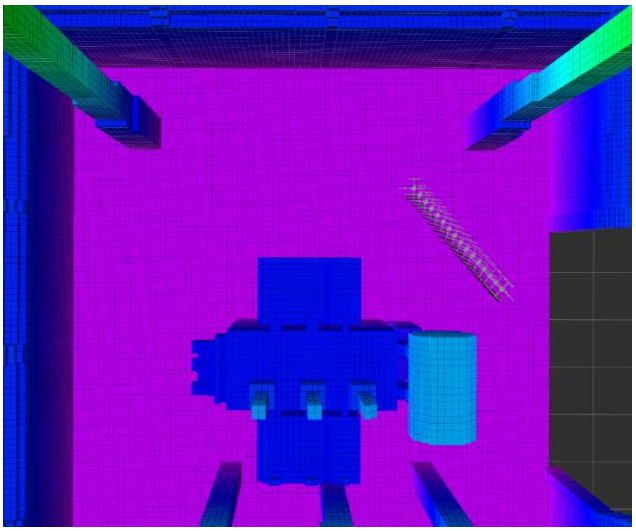
Para realizar este teste o arquivo de pontos *list17.txt* foi executado no módulo executor de rotas, desta vez com a lógica adaptativa habilitada. O arquivo foi executado duas vezes, uma realizando a rota de nível 1 e outra de nível 2 representadas na Figura 50.

Ao chegar no ponto 2, o sistema aguarda a *flag*. Para a rota de nível 1 o valor simulado foi 1, indicando que o ponto adaptativo deve ser ignorado. Para a rota de nível 2 o valor simulado foi -1, indicando que o ponto adaptativo deve ser seguido. O valor de simulação foi escrito através da interface *rqt*.

Em cada caso foi realizado o registro visual da execução dos trechos conforme descrito a seguir.

Na Figura 55 está representada o trecho entre o ponto 1 e 2.

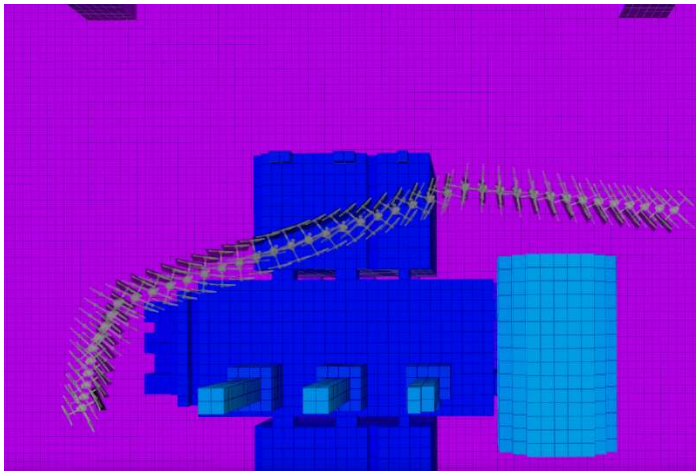
Figura 55 - Trecho entre ponto 1 e 2



Fonte: Arquivo pessoal

Na Figura 56 está representado o trecho gerado entre os pontos 2 e 4.

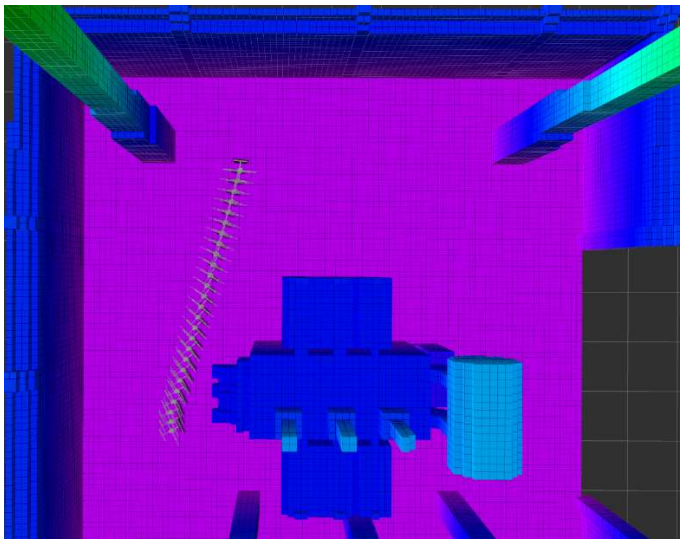
Figura 56 - Trecho entre ponto 2 e 4



Fonte: Arquivo pessoal

Na Figura 57 está representado o último trecho da rota de nível 1, entre os pontos 4 e 5.

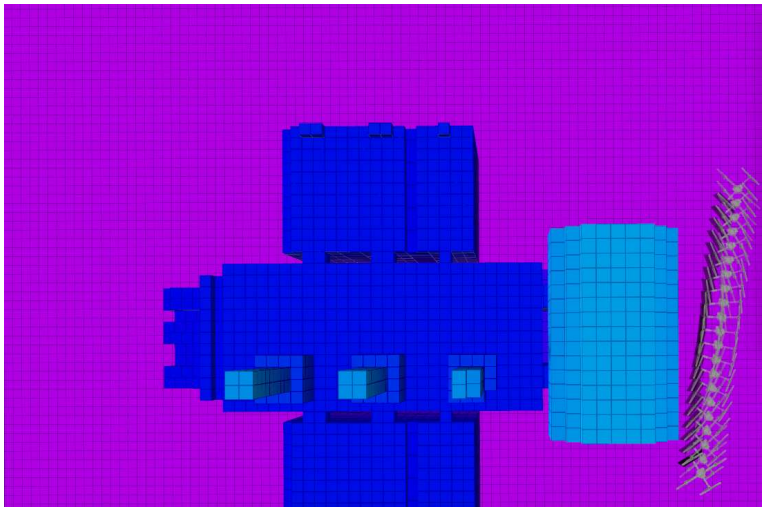
Figura 57 - Trecho entre ponto 4 e 5



Fonte: Arquivo pessoal

A rota nível 2 apresenta diferença significativa apenas nos trechos entre os pontos 2 e 4. Os trechos 1-2 e 4-5 são idênticos ao da rota 1 e não serão representados. Na Figura 58 representa o trecho gerado entre os pontos 2 e 3.

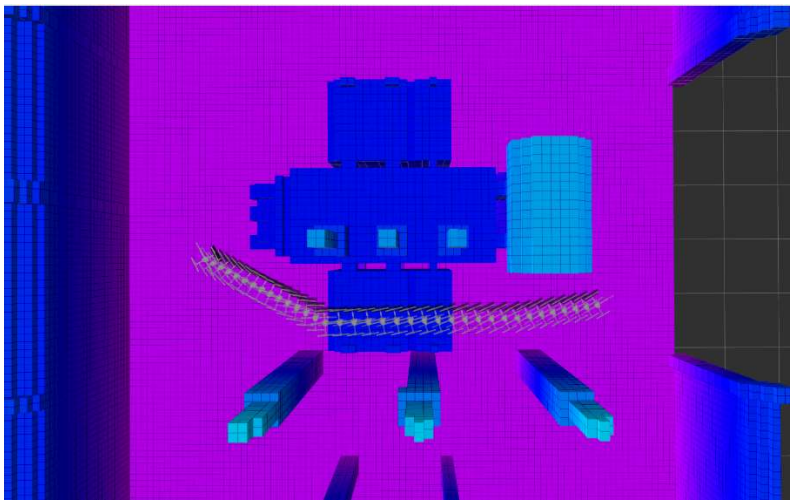
Figura 58 - Trecho entre ponto 2 e 3



Fonte: Arquivo pessoal

Na Figura 59 representa o trecho da rota 2, entre os pontos 3 e 4.

Figura 59 - Trecho entre ponto 3 e 4



Fonte: Arquivo pessoal

5.2.2 Análise

Conforme o registro visual da seção 5.2.1 as rotas de nível 1 e 2 concluíram o trajeto entre o ponto inicial e final do arquivo de pontos coletado. Ambas as rotas e ao chegar no ponto 2, a decisão por seguir para o ponto 3 ou 4 foi dependente da *flag*, o que valida a lógica de rotas adaptativas.

5.3 Teste de filtro

O VANT utilizado possui uma limitação na distância mínima entre pontos de missão, portanto se fez necessário a implementação de um filtro de distância nos pontos gerados nos trechos para execução no VANT. O filtro recebe na entrada todos os pontos gerados para um trecho e o resultado é uma lista de pontos que são distantes em sequência de no mínimo 0,8 m. O filtro, portanto, deve manter os pontos iniciais e finais da lista de pontos não filtrados para manter a posição e orientação que o operador selecionou nos pontos de interesse.

Este teste tem por objetivo validar se os pontos inicial e final não-filtrados estão inclusos no resultado do filtro, e validar se os pontos filtrados possuem distância ponto a ponto de no mínimo 0,5 m.

É esperado que os pontos inicial e final do conjunto não-filtrados sejam mantidos nos resultados do filtro, e que a distância ponto a ponto seja no mínimo 0,5 m.

Para validar o teste serão observados os pontos não-filtrados e filtrados oriundos da geração de um trecho da rota de inspeção.

5.3.1 Procedimento

Durante a execução do teste de colisão, entre os pontos 2 e 4, representados na Figura 56, os valores de pontos não-filtrados e filtrados foram armazenados.

5.3.2 Análise

Para validar as distâncias entre os pontos selecionados pelo filtro, o valor da distância euclidiana entre dois pontos consecutivos foi calculado. A Tabela 3 apresenta o valor de distância do ponto para o anterior.

Tabela 3 - Pontos filtrados

Pontos filtrados					
n	**x (m)**	**y (m)**	**z (m)**	**Orientação (º)**	**Distância n-1 (m)**

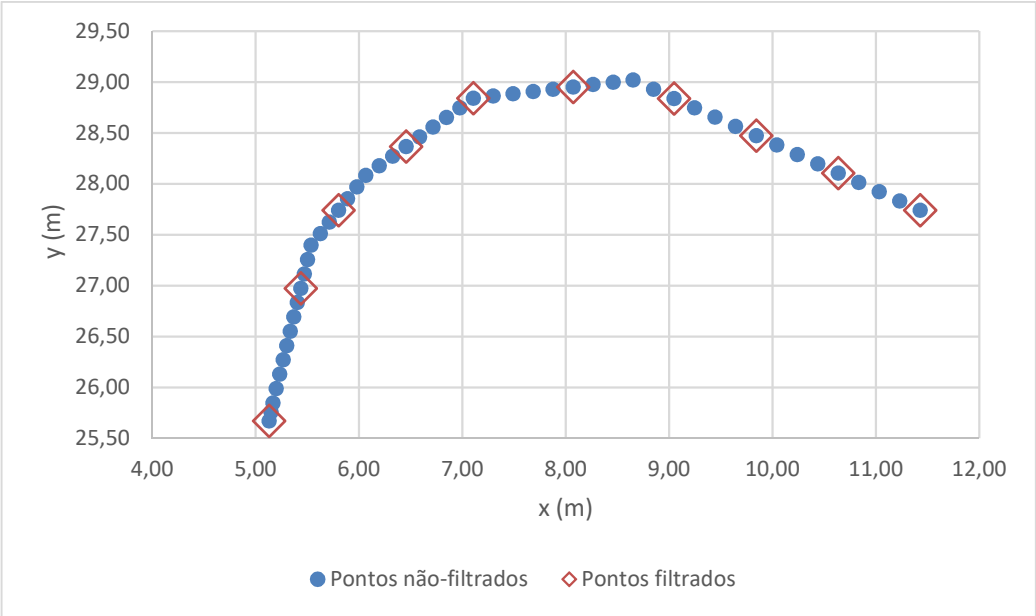
1	11,43	27,74	2,01	-136	-
2	10,63	28,11	2,05	-139	0,87
3	9,84	28,47	2,09	-141	0,87
4	9,05	28,84	2,13	-144	0,87
5	8,07	28,95	2,24	-139	0,99
6	7,10	28,84	2,38	-129	0,98
7	6,45	28,37	2,40	-104	0,80
8	5,80	27,74	2,38	-71	0,91
9	5,43	26,98	2,27	-25	0,86
10	5,13	25,67	2,02	62	1,36

Fonte: Arquivo pessoal

De acordo com a Tabela 3 conclui-se que a distância mínima de 0,5 m entre pontos foi respeitada pelo filtro. Observa-se que o último ponto possui um valor de distância maior que a média dos pontos anteriores, esse comportamento ocorre, pois, o filtro funciona em duas etapas, na primeira etapa os pontos intermediários são descartados para estabelecer pontos que sejam espaçados de no mínimo 0,8 m e em seguida é realizada a comparação entre o último ponto original e o último filtrado, caso o último ponto da rota não-filtrada não esteja no conjunto filtrado, o algoritmo força a substituição na última posição da lista dos pontos filtrados, descartando o último ponto antigo.

As coordenadas x e y dos pontos antes e após o filtro são representados na Figura 60.

Figura 60 - Resultados do filtro de posição



Fonte: Arquivo pessoal

De acordo com a Figura 60 os pontos iniciais antes e depois no filtro coincidem exatamente. O filtro funciona como um selecionador que mantém os valores de posição e orientação dos pontos escolhidos, portanto a altura e a orientação filtrada são preservadas também.

5.4 Teste de pontos GPS

Este teste tem por objetivo validar visualmente se os pontos de coleta cartesianos são corretamente convertidos para coordenadas GPS.

É esperado que as coordenadas GPS dos pontos coletados estejam em uma disposição visualmente próxima às da coleta do ambiente virtual.

As variáveis observadas para este teste são os valores das coordenadas GPS referentes aos pontos de coleta após o filtro de posições.

5.4.1 Procedimento

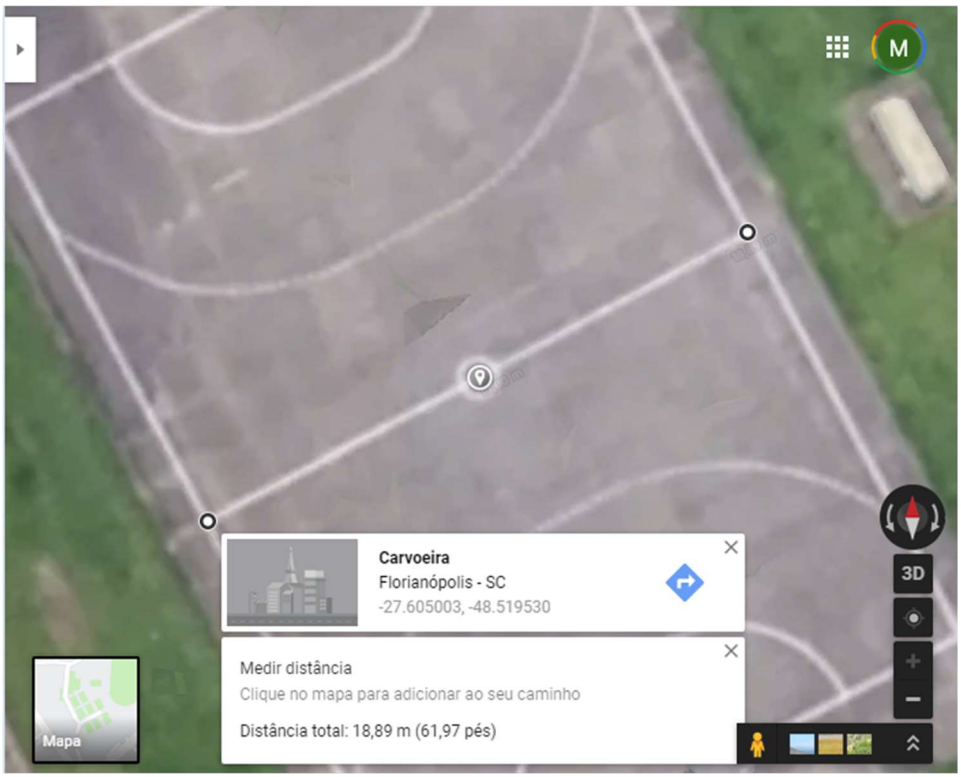
Para realizar este teste o arquivo de pontos *list17.txt* foi executado no módulo executor de rotas, desta vez com a lógica adaptativa habilitada com o valor de *flag* em 1 para forçar a rota adaptativa. O arquivo foi executado uma única vez realizando a rota de nível 2 representada na Figura 50.

O módulo executor de rotas foi configurado de modo que o ponto 1 da Figura 50 é a referência para as conversões para coordenadas GPS. Os valores das coordenadas de referência são:

- Latitude: -27,605003 °
- Longitude: -48,519530 °

Essa coordenada corresponde ao centro de uma das quadras de esporte da UFSC, representada na Figura 61:

Figura 61 - Coordenada GPS de referência



Fonte: Arquivo pessoal

Durante a execução da rota completa os todos os pontos GPS gerados foram armazenados na estação de comando.

5.4.2 Análise

Dentre os pontos coletados durante o teste, aqueles que representam somente os pontos de coleta convertidos são exibidos na Tabela 4. Os valores de altura não são convertidos, portanto são os mesmos das coordenadas cartesianas.

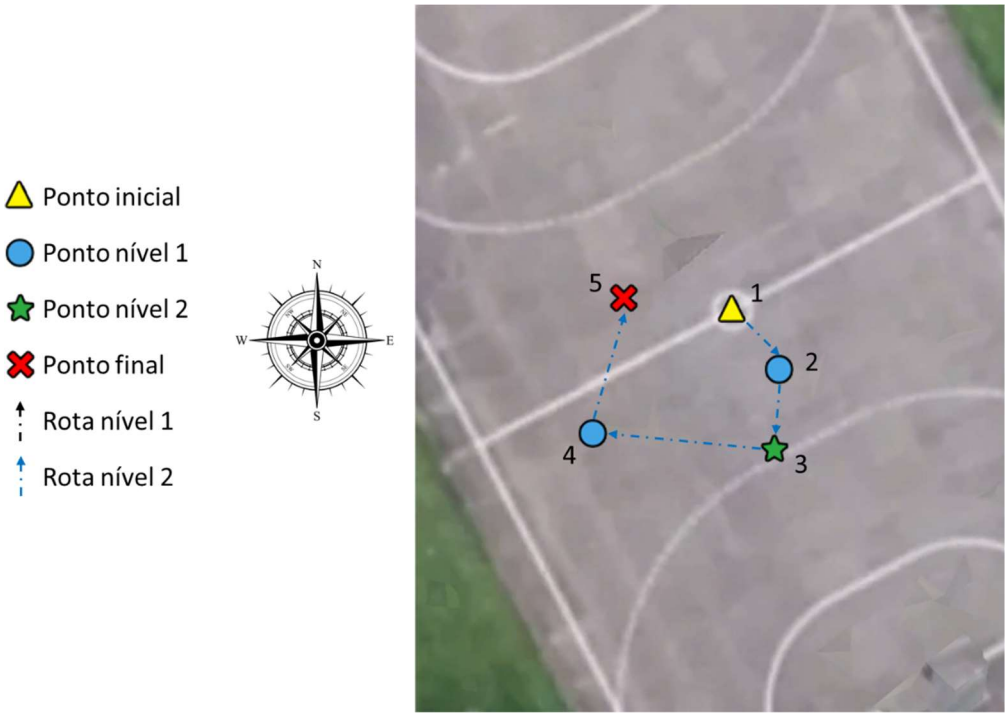
Tabela 4 - Coordenadas GPS de coleta de pontos

Ponto	Latitude (º)	Longitude (º)
1	-27,60500301	-48,51952998
2	-27,60501912	-48,51951342
3	-27,60504434	-48,51951665
4	-27,60503895	-48,51957680
5	-27,60499862	-48,51956607

Fonte: Arquivo pessoal

A Figura 62 exibe as coordenadas GPS da Tabela 4 representados no *Google Maps*.

Figura 62- Coordenadas GPS de coleta de pontos



Fonte: Arquivo pessoal

Exibido na Figura 62, o ponto 1 foi convertido para o centro da quadra conforme o esperado e o restante dos pontos seguem a mesma proporção do ambiente virtual.

Para avaliar a proporção mediu-se a distância entre os pontos geométricos utilizando o *Google Maps* e foram calculadas as distâncias euclidianas dos pontos de coleta no ambiente virtual. As diferenças entre as distâncias ficaram abaixo de 0,1 m.

5.5 Teste de comando e resposta GPS

Este teste tem por objetivo verificar as distâncias de posicionamento horizontal e vertical entre o VANT simulado no gerenciador de voo e os pontos de coleta comandados, e avaliar visualmente a execução completa da rota no VANT simulado no gerenciador de voo.

É esperado que a distância de posicionamento horizontal e vertical nos pontos de coleta comandados e recebidos sejam compatíveis com os valores declarados pelo fabricante, e que as coordenadas recebidas do VANT simulado no gerenciador de voo representem uma rota próxima à rota comandada.

As variáveis observadas para este teste são as coordenadas GPS comandadas pela estação de trabalho e as coordenadas GPS recebidas da simulação no gerenciador de voo.

5.5.1 Procedimento

Para realizar este teste o arquivo de pontos *list17.txt* foi executado no módulo executor de rotas, desta vez com a lógica adaptativa habilitada com o valor de *flag* em 1 para forçar a rota adaptativa. O arquivo foi executado uma única vez realizando a rota de nível 2 representada na Figura 50.

5.5.2 Análise

As coordenadas GPS dos pontos de coleta comandados e recebidos são representadas na Figura 63.

Figura 63 - Coordenadas GPS comando e retorno nos pontos de coleta

Fonte: Arquivo pessoal

A Tabela 5 exibe as coordenadas GPS dos pontos recebidos referentes aos pontos de coleta. A tabela também exibe e as distâncias entre as coordenadas X, Y e Z dos pontos comandados e recebidos. Para avaliar a distância entre os pontos, as coordenadas GPS foram convertidas para o sistema UTM em metros.

Tabela 5 - Coordenadas GPS recebidas e distâncias horizontal e vertical

Pontos	Recebido			Distância X para comandado (m)	Distância Y para comandado (m)	Distância Z para recebido (m)
	Latitude (º)	Longitude (º)	Altura (m)			
1	-27,60500387	-48,51952306	2,30	0,68	0,11	0,29
2	-27,60502338	-48,51950919	2,30	0,41	0,48	0,29
3	-27,60505096	-48,51951759	2,20	0,11	0,73	0,18
4	-27,60503573	-48,51958012	2,80	0,32	0,36	0,79
5	-27,60499271	-48,51956505	2,60	0,11	0,65	0,58

Fonte: Arquivo pessoal

Conforme visto na tabela a distância horizontal em X e Y está dentro da faixa declarada nas especificações do fabricante do VANT +/- 2,5 m. Já a distância vertical, Z, para os pontos 4 e 5 é superior ao declarado pelo fabricante de +/- 0,5 m. Esse comportamento pode ser motivado por alguns fatores como: parâmetros de perturbação da simulação e não correspondência direta dos parâmetros estabelecidos pelo fabricante no simulador.

A rota completa comandada e recebida são exibidas simultaneamente na Figura 64. A figura foi construída a partir da inserção das coordenadas GPS diretamente em uma ferramenta que representa as coordenadas no *Google Maps*.

Figura 64 - Coordenadas GPS comando e retorno na rota completa

Fonte: Arquivo pessoal

Visualmente pode se afirmar que a rota comandada é seguida pelo VANT simulado, apresentando um erro de posicionamento nas posições de coleta.

6 CONSIDERAÇÕES FINAIS E PERSPECTIVAS

Este trabalho trata do desenvolvimento de um sistema capaz de gerar rotas adaptativas livres de colisões baseadas em um mapa de obstáculos configurável e, além disso, capaz de controlar a sua execução em VANTs virtuais ou reais.

A concepção do sistema gerenciador de rotas começou no desenho de uma arquitetura dos módulos executor de rotas, atualizador de mapa, coletor de pontos e gerenciador de voo. Após a definição conceitual do sistema foi realizado o mapeamento de tecnologias para dar suporte às funcionalidades dos módulos. Dentre as tecnologias pode-se destacar o ROS para comunicação, o pacote de *motion planning MoveIt*, o DJI Android mobile SDK para o desenvolvimento do módulo gerenciador de voo e o Gazebo para simulação do VANT e ambiente.

Após o mapeamento das tecnologias foi configurado um ambiente de simulação na estação de trabalho com o *software* Gazebo no qual um VANT genérico e uma subestação foram inseridos. Em seguida os módulos de *motion planning*, coleta de pontos de inspeção e gerador e executor de rotas foram implementados.

A simulação do VANT foi implementada no gerenciador de voo e a comunicação entre o gerenciador de voo com a estação de trabalho foi implementada utilizando a API do ROS para Java.

Para analisar o funcionamento do sistema gerenciador de rotas, testes foram realizados no ambiente virtual da estação de trabalho a fim de se avaliar funcionalidades como evitar obstáculos, coletar pontos, executar rotas adaptativas, filtro de posições e conversão para coordenadas GPS.

Após a verificação das funcionalidades no ambiente simulado da estação de comando, testes de geração e controle de trajetórias foram realizados no VANT simulado do gerenciador de voo.

Era esperado que o VANT adotado neste trabalho não obtivesse um desempenho ótimo em relação ao controle de posição, por não ser um equipamento direcionado para desenvolvimento e aplicações em metrologia. Por exemplo, a capacidade de integração com outros dispositivos como microcontroladores genéricos não e viável *on flight*. Outros modelos como o DJI *Matrice* 210 RTK possuem SDKs alternativas, com a *Onboard* que permite a conexão direta do VANT à microcontroladores e sensores, além de oferecer um sistema de posicionamento com menor incerteza que o Inspire 1.

Os resultados do sistema gerador de rotas adaptativas foram satisfatórios quanto a geração e controle das rotas na estação de comando e no gerenciador de voo. Os resultados foram dentro do esperado quanto a evitar obstáculos, execução de rotas adaptativas, filtro de posições, conversão para coordenadas GPS e envio de rotas para o VANT simulado no gerenciador de rotas.

Quanto a geração das rotas a SDK do fabricante do VANT possui limitações em relação a flexibilidade de geração de missões. A limitação de maior impacto no desenvolvimento do sistema foi a distância mínima entre pontos que compõe uma missão de 0,5 m. Essa limitação, exige a utilização de um filtro na saída do algoritmo de *motion planning* que, se não propriamente configurado, pode acabar gerando colisões.

Outro fator que prejudica a geração de rotas está relacionado a necessidade de conversão das coordenadas cartesianas X e Y para coordenadas geográficas latitude e longitude. Essa conversão requer o uso de uma biblioteca que realiza conversões intermediárias entre diferentes sistemas coordenados, neste caso UTM e WGS84. Essas conversões estão associadas a distorções e acabam por não representar no mundo real exatamente a posição coletada no mundo virtual.

Durante a execução deste trabalho, não foi possível testar o sistema gerenciador de rotas em ambiente real devido ao tempo de projeto e isso fica como sugestão para trabalhos futuros.

Como sugestão de trabalhos e incrementos de *software* pode-se citar:

- Testar o sistema gerenciador de rotas em ambiente real;
- Implementar a atualização do mapa de obstáculo durante a execução com o uso de uma câmera;
- Implementar a atualização do VANT simulado na estação de trabalho com coordenadas reais;
- Implementar o controle de posição por meio de ângulos de orientação dos eixos espaciais, ao invés de utilizar missões com coordenadas geométricas;
- Implantação de um sistema de coordenadas local (iGPS);

- Implementar um algoritmo de *motion planning* que possui parâmetros de distância mínima entre pontos configurável;
- Implementar a automação do processo de criação de *Octomaps*;
- Implementar o controle do *Gimbal* para obtenção de ângulos diferentes na mesma posição, sem precisar deslocar o VANT.

A implementação da solucao apresentada neste trabalho tem potencial de permitir o monitoramento de baixo custo de equipamentos de uma subestação, modificação e avaliação de novas rotas de inspeção, aumento da segurança dos operadores e diminuição nos custos associados aos tempos de falta de energia relacionados às agências reguladoras. Além das potencialidades supracitadas a solução, como um sistema gerenciador de rotas, há aplicabilidade em diversas áreas. Não se restringindo somente às subestações, pode ser aplicada em qualquer ambiente que possua a necessidade da lógica de decisão das rotas adaptativas ou somente para evitar obstáculos. Como exemplos, pode-se citar a inspeção de linhas de transmissão, plataformas tanques de petróleo e monitoramento de parques de energia solar.

REFERÊNCIAS

1 BRASIL.GOV. **Energia elétrica chega a 97,8% dos domicílios brasileiros, mostra censo demográfico**, 2011. Disponivel em: <http://www.brasil.gov.br/noticias/infraestrutura/2011/11/energia-eletrica-chega-a-97-8-dos-domicilios-brasileiros-mostra-censo-demografico>. Acesso em: 29 abr. 2019.

2 ANEEL. Saiba mais sobre o setor elétrico brasileiro, 2011. Disponivel em: <http://www.aneel.gov.br/home?p_p_id=101&p_p_lifecycle=0&p_p_state=maximized&p_p_mode=view&_101_struts_action=%2Fasset_publisher%2Fview_content&_101_returnToFullPageURL=%2F&_101_assetEntryId=14476909&_101_type=content&_101_groupId=654800&_101_urlTitle=faq&>. Acesso em: 30 abr. 2019.

3 ONS. **O que é o SIN**, 2019. Disponivel em: <http://ons.org.br/paginas/sobre-o-sin/o-que-e-o-sin>. Acesso em: 30 abr. 2019.

4 EPE. **Anuário Estatístico de Energia Elétrica**. Brasília: [s.n.], 2018. 123 p.

5 EPE. **Projeção da Demanda de Energia**. Rio de Janeiro: [s.n.], 2017.

6 ANEEL. **Atlas de Energia Elétrica do Brasil**. Brasília: TDA Brasil, v. 3, 2009.

7 BAYAR, T. **How drones are playing a role in the power and utility sector**, 2018. Disponivel em: <https://www.powerengineeringint.com/articles/2018/02/how-drones-are-playing-a-role-in-the-power-and-utility-sector.html>. Acesso em: 01 maio 2019.

8 RANGEL, R. K.; KIENITZ, K. H.; BRANDÃO, M. P. **SISTEMA DE INSPEÇÃO DE LINHAS DE TRANSMISSÃO DE ENERGIA**. 3rd CTA-DLR Workshop on Data Analysis & Flight Control. São Paulo: [s.n.]. 2009.

9 DENG, C. et al. Unmanned Aerial Vehicles for Power Line Inspection: A Cooperative Way in Platforms and Communications. **Journal of Communication**, v. 9, n. 9, p. 687-692, 2017.

10 DE OLIVEIRA, A. K. V. et al. **Aerial Infrared Thermography of a Utility-Scale PV Plant After a Meteorological Tsunami in Brazil**. 7th World Conference on Photovoltaic Energy Conversion (WCPEC-7). Hawaii: [s.n.]. 2018.

11 FUNDAÇÃO CERTI. **História**. Disponivel em: <https://www.certi.org.br/pt/acerti-historico>. Acesso em: 03 maio 2019.

12 ROS.ORG. **About ROS**. Disponivel em: <https://www.ros.org/about-ros/>. Acesso em: 03 maio 2019.

13 ROS.ORG. **Nodes**, 2018. Disponivel em: <http://wiki.ros.org/Nodes>. Acesso em: 03 maio 2019.

14 ROS.ORG. **Packages**, 2019. Disponivel em: <http://wiki.ros.org/Packages>. Acesso em: 03 maio 2019.

15 CLEARPATH ROBOTICS. **General Concepts**. Disponivel em: <https://robohub.org/ros-101-intro-to-the-robot-operating-system/>. Acesso em: 03 maio 2019.

16 ROS.ORG. **Topics**, 2019. Disponivel em: <http://wiki.ros.org/Topics>. Acesso em: 03 maio 2019.

17 ROS.ORG. **Services**, 2018. Disponivel em: <http://wiki.ros.org/Services>. Acesso em: 03 maio 2019.

18 ROS.ORG. **Overview**, 2018. Disponivel em: <http://wiki.ros.org/actionlib>. Acesso em: 03 maio 2019.

19 GAZEBO.ORG. **Why Gazebo?** Disponivel em: <http://gazebosim.org/>. Acesso em: 03 maio 2019.

20 KAVRAKI LAB RICE UNIVERSITY. **Open Motion Planning Library:** A Primer. [S.l.]: [s.n.], 2019. Disponivel em: <http://ompl.kavrakilab.org/OMPL_Primer.pdf>. Acesso em: 04 maio 2019.

21 MOVEIT.ORG. **Planners Available in MoveIt**. Disponivel em: <https://moveit.ros.org/documentation/planners/>. Acesso em: 05 abr. 2019.

22 HORNUNG, A. et al. *OctoMap*: An Efficient Probabilistic 3D Mapping Framework Based on Octrees. **Autonomous Robots Journal**, v. 34, n. 3, p. 189-206, 2013.

23 ARCGIS. **What are geographic coordinate systems?**, 2013. Disponivel em: <http://resources.arcgis.com/en/help/main/10.1/index.html#//003r00000006000000>. Acesso em: 04 maio 2019.

24 JOURNAL OF INTEGRATED COASTAL ZONE MANAGEMENT. **WGS84**. Disponivel em: <http://www.aprh.pt/rgci/glossario/WGS84.html>. Acesso em: 04 maio 2019.

25 MAPTILER. **EPSG:4326**. Disponivel em: <https://epsg.io/4326>. Acesso em: 04 maio 2019.

26 GISGEOGRAPHY. **What are Map Projections? (And Why They Are Deceiving to the Human Eye)**, 2019. Disponivel em: <https://gisgeography.com/map-projections/>. Acesso em: 04 maio 2019.

27 DJI. **Flight Control**, 2016. Disponivel em: <https://developer.dji.com/mobile-sdk/documentation/introduction/flightController_concepts.html>. Acesso em: 04 maio 2019.

28 DJI. **DJI LIGHTBRIDGE**. Disponivel em: <https://www.dji.com/br/inspire-1/remote-controller#lightbridge>. Acesso em: 05 maio 2019.

29 ROS.ORG. **Is ROS For Me?** Disponivel em: <https://www.ros.org/is-ros-for-me/>. Acesso em: 05 maio 2019.

30 ROS.ORG. **Operating Systems**, 2018. Disponivel em: <http://wiki.ros.org/ROS/Introduction>. Acesso em: 05 maio 2019.

31 DJI. **Inspire Series**. Disponivel em: <https://developer.dji.com/products/#!/mobile>. Acesso em: 05 maio 2019.

32 DJI. **Connection to Application and Product**. Disponivel em: <https://developer.dji.com/mobile-sdk/documentation/introduction/mobile_sdk_introduction.html#connection-to-application-and-product>. Acesso em: 05 maio 2019.

33 ROS.ORG. **TCPROS**, 2013. Disponivel em: <http://wiki.ros.org/ROS/TCPROS>. Acesso em: 05 maio 2019.

34 PATRICKMIN. **Introduction**, 2017. Disponivel em: <http://www.patrickmin.com/binvox/>. Acesso em: 05 maio 2019.

35 WILL, S. 3D **Mapping & Navigation**. Disponivel em: <https://www.wilselby.com/research/ros-integration/3d-mapping-navigation/>. Acesso em: 08 maio 2019.

36 ROS.ORG. **What is a distribution?**, 2018. Disponivel em: <http://wiki.ros.org/Distributions>. Acesso em: 08 maio 2019.

37 ROS.ORG. **Ubuntu install of ROS Kinetic**, 2017. Disponivel em: <http://wiki.ros.org/kinetic/Installation/Ubuntu>. Acesso em: 15 maio 2019.

38 KAVRAKILAB. **Detailed Description**. Disponivel em: <https://ompl.kavrakilab.org/classompl_1_1geometric_1_1RRTConnect.html>. Acesso em: 11 maio 2019.

39 ROS.ORG. **geonav_transform**, 2017. Disponivel em: <http://wiki.ros.org/geonav_transform>. Acesso em: 11 maio 2019.

40 DJI. **property heading**. Disponivel em: <https://developer.dji.com/api-reference/android-api/Components/Missions/DJIWaypoint.html#djiwaypoint_heading_inline>. Acesso em: 11 maio 2019.

41 DJI. **DJI Simulator Tutorial**, 2019. Disponivel em: <https://developer.dji.com/mobile-sdk/documentation/android-tutorials/SimulatorDemo.html>. Acesso em: 12 maio 2019.

42 ROS.ORG. **Examples, android_tutorial_pubsub**, 2015. Disponivel em: <http://wiki.ros.org/android/Tutorials/indigo/RosActivity>. Acesso em: 12 maio 2019.