UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Guilherme Claudino e Silva

Estudo de ferramentas computacionais para simulação e visualização de aplicações de drones

São Carlos

2021

Guilherme Claudino e Silva

Estudo de ferramentas computacionais para simulação e visualização de aplicações de drones

Monografia apresentada ao Curso de Engenharia Aeronáutica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Aeronáutico.

Orientador: Prof. Dr. Glauco Augusto de Paula Carin

São Carlos

2021

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Claudino e Silva, Guilherme
C615e Estudo de ferramentas computacionais para simulação e visualização de aplicações de drones / Guilherme Claudino e Silva; orientador Glauco Augusto de Paula Caurin. São Carlos, 2021.

Monografia (Graduação em Engenharia Aeronáutica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2021.

1. Airsim. 2. ROS. 3. Drones. 4. Vants. 5. Simulação. 6. GUI. I. Título.

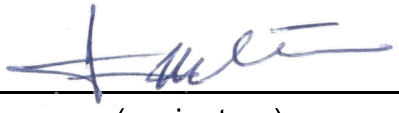
Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Candidato: Guilherme Claudino e Silva
Título do TCC: Estudo de ferramentas computacionais para simulação e visualização de aplicações de drones
Data de defesa: 20/08/2021

Comissão Julgadora	**Resultado**
Prof. Titular Fernando Martini Catalano	9,5
Instituição: EESC - SAA	
Prof. Dr. Jorge Henrique Bidinotto	9,5
Instituição: EESC - SAA	

Presidente da Banca: Prof. Titular Fernando Martini Catalano



(assinatura)

AGRADECIMENTOS

Acredito que os agradecimentos de um Trabalho de Conclusão de Curso devem se estender para além das pessoas que contribuíram ativamente para o projeto. O TCC marca o fim de toda uma etapa que, no meu caso, representa mais de 25% da minha vida e boa parte das minhas memórias. Assim, gostaria de dedicar algumas palavras de gratidão também àquelas pessoas que me marcaram ao longo desse período.

Primeiramente, agradeço minha família, em especial minha mãe Angela e minha irmã Aline que sempre estiveram ao meu lado, me apoiando em todas minhas decisões ao longo do curso. Agradeço ainda à Renata, minha companheira, que foi um suporte muito importante em diversos momentos da minha caminhada e sem dúvida contribuiu ativamente para que eu superasse os diversos desafios que enfrentei até aqui.

Na sequência, gostaria de agradecer a Universidade de São Paulo e a Escola de Engenharia de São Carlos por todas as portas que me abriu, todos os ensinamentos e pessoas maravilhosas que pude conhecer. Sem dúvida, sem essa estrutura, sem os incentivos que recebi e sem o auxílio de pessoas tão marcantes quanto o técnico José Claudio, a secretária Gisele Lavadeci e servidores como o Bruno Sevciuc e João Bettoni que sempre estiveram a disposição quando precisei.

Agradeço também todos os professores com quem convivi ao longo do curso, em especial aos Professores Glauco Augusto, Ricardo Angélico, Fernando Catalano, Hernan Muñoz e Jorge Bidinotto por todos os ensinamentos dentro e fora de sala de aula. Acredito que essa lista poderia ser bem mais longa, mas começo a ter pouco espaço para tal.

Também dedico um agradecimento especial à École Centrale de Lille, universidade na qual pude, não somente conviver com pessoas que me ajudaram tanto como Monique Bukowski, Armand Toguyeni e Veronique Dzwiniel e foi uma grande fonte de aprendizado, contribuindo para que eu me formasse não só como engenheiro mas como pessoa.

Ademais, agradeço meus tutores de estágio, Susanna Cortes-Borgmeyer, Cyril Dietz, Marc Tamm, Phillip Laval e Rodrigo Andrade com os quais pude conviver e aprender de maneira mais direta os deveres e atribuições de um engenheiro.

Por fim, mas não menos importante, agradeço os amigos que me apoiaram nessa caminhada, dentro e fora da sala de aula. Vocês foram, sem dúvida alguma, parte fundamental dessa caminhada.

RESUMO

CLAUDINO E SILVA, G. **Estudo de ferramentas computacionais para simulação e visualização de aplicações de drones**. 2021. 89p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2021.

Na indústria aeronáutica, alguns dos principais de exemplos de aplicação das tecnologias de VANTS (Veículos Aéreos Autônomos). Esses veículos, também conhecidos como drones, são um termo de crescente interesse em diversos países. O presente trabalho visa avaliar diferentes ferramentas computacionais para simulação de Drones e suas missões, buscando avaliar comparativamente prós e contras de algumas das principais soluções existentes como ROS e GAZEBO e Airsim. Avaliadas as ferramentas, o projeto visa construir uma interface gráfica para dialogar com o Airsim, o software de simulação escolhido, propondo rotinas de validação da ferramenta através de perfis de missão pré-definidas com análises comparativas de trajetórias e geração de imagens e animações.

Palavras-chave: Airsim. ROS. Drones. Vants. Simulação. GUI.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface - Interface de programação de aplicações
CNES	Centre National d'Études Spatiales - Centro Nacional de Estudos Espaciais
CNI	Confederação Nacional da Indústria
GUI	Graphical User Interface - Interface Gráfica do Usuário
IBGE	Instituto Brasileiro de Geografia e Estatística
OSRF	Open Source Robotics Foundation - Fundação de Robótica de Código Livre
PIB	Produto Interno Bruto
PSF	Python Software Foundation - Fundação de Software Python
ROS	Robot Operating System - Sistema Operacional de Robôs
SO	Sistema Operacional
UAVs	Unmanned Aerial Vehicles - Veículos Aéreos Autônomos
VANTs	Veículos Aéreos Autônomos
WSL	Windows Subsystem Linux - Subsistema Linux para Windows

LISTA DE SÍMBOLOS

c_r	- Comprimento do Retângulo
l_r	- Largura do Retângulo
R	- Raio de circunferência
$r_{\vec{AB}}$	- Vetor Posição de um ponto A em relação a um ponto B
$\cos(\alpha)$	- Cosseno de um ângulo qualquer
$\sin(\alpha)$	- Seno de um ângulo qualquer
k	- Ponto atual da discretização
n	- Número de lados em um polígono
θ	- Ângulo da trajetória circular
$(x, y, z)_0$	- Variáveis de posição ao início de um movimento
$(x, y, z)_c$	- Variáveis de posição do centro de uma circunferência

SUMÁRIO

1 INTRODUÇÃO

Atualmente, a indústria é sem dúvida, um elemento central de toda e qualquer sociedade. No Brasil, mesmo com o PIB em queda e a indústria desacelerada, como apontam os dados mais recentes do IBGE, indicando uma retração de 4.1% para o ano de 2020 (G1, 2021), é inviável imaginar o país sem uma base industrial. Soma-se à isso, o elevado contingente de trabalhadores empregados no setor: 9,7 milhões de pessoas, segundo informações da CNI (CNI, 2021). Ademais, de acordo com dados do Banco Mundial, extraídos de sua plataforma Databank, nos últimos 20 anos, a participação do segundo setor no PIB nacional, se manteve acima de 17% durante todo o período. Ainda segundo a plataforma Databank, esse valor é similar à de países considerados referência mundial na indústria, como a França e o Reino Unido com, respectivamente, 17,1% e 17,4% de participação em 2019. No entanto, países como a Coreia do Sul, que apresentou um crescimento de seu Produto Interno próximo de 29,6% entre 2011 e 2019, conta com uma participação industrial mais expressiva, se aproximando dos 33% (MUNDIAL, 2021).

Assim sendo, o desenvolvimento industrial do Brasil é crucial para manter o funcionamento do país - garantindo empregos, renda e poder de compra a milhões de brasileiros - e recuperar sua economia fragilizada - permitindo uma retomada do PIB e uma menor dependência de produtos importados, trabalhando também no alívio da balança comercial. Tal necessidade pode muito bem se aproveitar das tecnologias propostas pela Quarta Revolução Industrial, também chamada de Indústria 4.0, para alavancar uma nova onda de industrialização e produção tecnológica "tupiniquim". O termo surgiu na Alemanha em 2011 (SILVEIRA, 2017) e descreve as atuais mudanças que tem ocorrido na indústria ao redor do mundo, podendo ser resumido como sendo um vocábulo chave para novas tecnologias e conceitos para organização da cadeia de valor, contando com fábricas e serviços inteligentes que só foram possíveis de serem atingidos com as recentes evoluções digitais (HERMANN; PENTEK; OTTO, 2016).

Neste sentido, a Indústria 4.0 pode ser dividida em 9 pilares principais, os chamados *Big Data* e *Analytics*, a utilização de Robôs Autônomos, as Ferramentas de Simulação Computacional, a Internet das Coisas, a Segurança Digital, as Tecnologias de Nuvem, a Integração de Sistemas, a Realidade Aumentada e a Manufatura Aditiva (ERBOZ, 2017). Esses pilares, muitas vezes se interconectam em projetos de engenharia como, por exemplo, em sensores inteligentes que utilizam ferramentas em nuvem para analisar dados e fornecer informações úteis para gestores que passariam muitas vezes despercebidas. No que diz respeito às tecnologias aeronáuticas que fazem parte dessa revolução, os VANTs, também conhecidos pelo termo Drones, são uma das tecnologias de maior destaque atualmente.

No Brasil, dados do Google Trends, mostram que o interesse por pesquisas ligadas

ao termo cresceu consideravelmente na última década, com destaques para estados como Mato Grosso e Roraima com pontuação máxima de interesse (GOOGLE, 2021b), como mostra a Figura 1. Esses veículos autônomos, podem atuar em soluções que interligam diferentes pilares da Indústria 4.0 e podem atuar promovendo soluções para todos os setores da economia. Desta maneira, estudar o tema e desenvolver tecnologias relacionadas se mostra não só uma possibilidade de evolução para o segundo setor brasileiro, como também representa uma demanda nacional. Um exemplo de aplicação prática do uso de drones por empresas brasileiras, é o caso da EDP que em junho de 2021 teve autorizada a utilização de veículos aéreos autônomos para o monitoramento de suas redes elétricas (EDP, 2021).

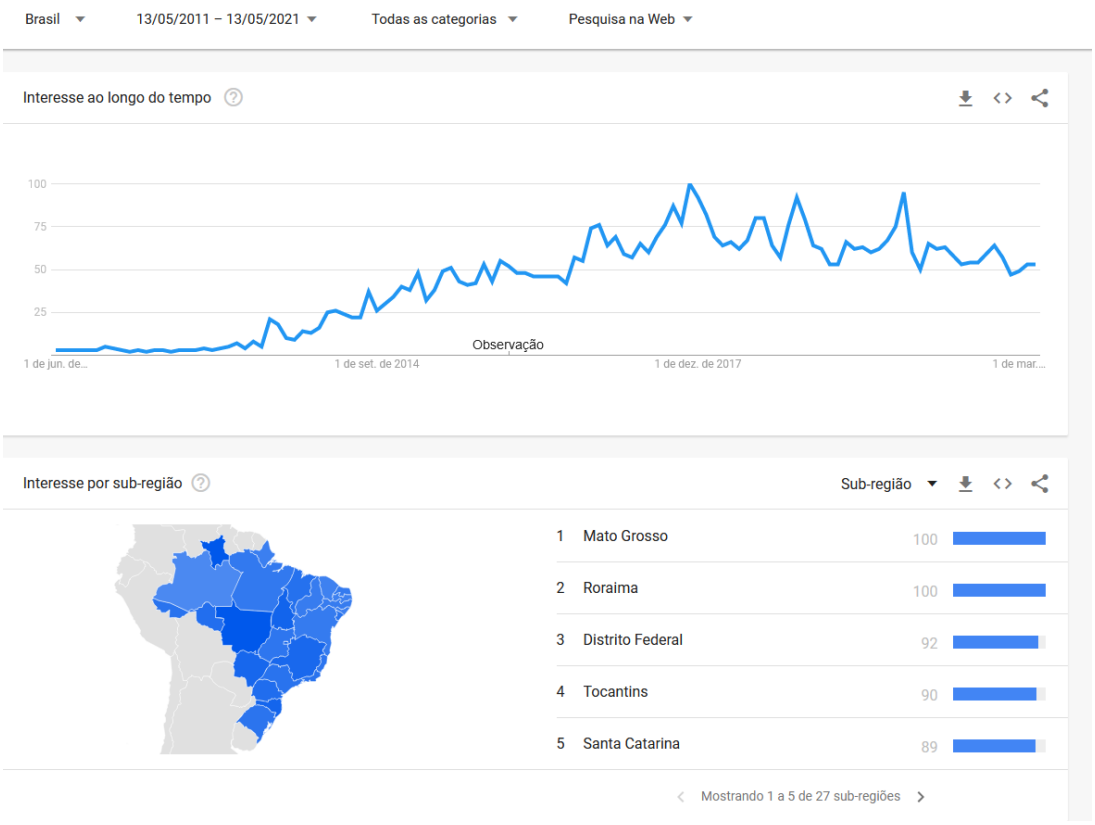


Figura 1 – Resultado de uma pesquisa sobre Drones no Google Trends, feita em maio de 2021. Nota-se aqui um gráfico onde o interesse de pesquisa do termo apresentou uma grande crescente na última década e mais abaixo, um mapa do Brasil com o interesse por estados. Retirado de (GOOGLE, 2021b).

1.1 Objetivos

O presente trabalho visa avaliar diferentes métodos de simulação computacional para Veículos Aéreos Autônomos. Após a realização de uma análise de pontos positivos e negativos de diferentes métodos e plataformas, uma rotina computacional foi desenvolvida visando fornecer as bases para simular missões de monitoramento aéreo de rodovias através de drones que poderá, futuramente, ser adaptada para sistemas mais complexos,

permitindo inclusive teste de ferramentas como câmeras e outros sensores diversos que poderão interagir com o ambiente simulacional. Essa rotina permitiu a criação de uma interface gráfica de usuário de forma a facilitar as interações de usuários com diferentes níveis de conhecimento para com a plataforma. Simular VANT's se mostra cada vez mais uma necessidade em uma sociedade onde as missões para essas aeronaves se tornam cada vez mais complexas e universais, atuando desde pequenos galpões terrestres, até em missões interplanetárias.

1.2 Estrutura do Trabalho

O trabalho inicia-se com uma revisão bibliográfica de ferramentas e meios já existem de programação para robótica móvel, buscando compreender em que situações uma ferramenta seria preferível à outra, definindo assim a ferramenta de trabalho de base para o estudo, sendo apresentado na seção 2. Na sequência, são abordados os métodos utilizados para desenvolver as rotinas computacionais usadas para a validação do ambiente de simulação na seção e as principais escolhas de projeto 3, enquanto os resultados obtidos e os testes realizados serão abordados e discutidos na seção 4. Por fim, uma conclusão do trabalho é feita, abordando também possíveis vias de desenvolvimento futuras.

2 REVISÃO BIBLIOGRÁFICA

Visando compreender melhor as ferramentas e possibilidades existentes, uma revisão bibliográfica das ferramentas de simulação existentes, visando compreender vantagens e desvantagens de cada uma das soluções. Nessa seção, também serão discutidas algumas das linguagens de programação que podem ser utilizadas em conjunto com cada um dos programas e suas principais aplicações e funcionalidades. Para a escolha final da ferramenta, uma pesquisa de opinião foi realizada com diversos estudantes e engenheiros para complementar a decisão. A primeira parte, trata das principais soluções existentes no mercado para simular e visualizar o comportamento dos veículos aéreos. A segunda, avalia ferramentas e linguagens de programação diversas que podem ser utilizadas de maneira complementar aos softwares, visando fornecer comandos e permitir a interação usuário máquina. Por fim, a escolha final da ferramenta é abordada.

2.1 Softwares de Interesse

O levantamento bibliográfico deste trabalho inicia-se através de uma avaliação de diferentes ferramentas de simulação para robôs e VANTs. As ferramentas foram avaliadas de diferentes pontos de vista, considerando trabalhos existentes e os materiais disponíveis no site de cada uma destas.

2.1.1 ROS

Iniciando a análise das ferramentas destaca-se a existência do ROS (Sistema Operacional de Robôs). ROS conta com uma série de ferramentas para auxiliar na programação e simulação de robôs com aplicações diversas, desde pequenos aspiradores de pó automatizados até grandes drones. Esse Sistema Operacional (SO) é utilizado em pesquisas e trabalhos de diversas entidades como a ETH Zurique a EPFL e a Universidade Técnica de Darmstadt. (OSRF, 2021)

O ROS conta ainda com uma série de ferramentas e bibliotecas que incluem funcionalidades como envio de mensagens, estimativas de posição e ferramentas de visualização de imagens. ROS é uma das referências na programação e simulação de robôs e constantemente renova seus programas e funcionalidades. Atualmente, 3 versões de ROS estão disponíveis de maneira simultânea, conforme é possível ver na figura 2 (OSRF, 2021):

- ROS Melodic, com suporte previsto até 2023 e compatível com o Ubuntu 18.04;
- ROS Noetic, disponibilizado em 2020, com suporte previsto até 2025 e compatível com o Ubuntu 20.04;

- ROS Foxy, distribuição de ROS compatível com outros SO's que não linux é feita utilizando ROS2, versão do sistema operacional que é bem menos utilizada.

Além dessas distribuições, existiram outras como ROS Kinetic e Indigo que foram recentemente descontinuadas e eram idealizadas preferencialmente para os SO's Ubuntu 16.04 e 14.04 respectivamente, embora seja alegado que possam funcionar com algumas limitações em outros sistemas. Destaca-se também que, usualmente, as versões de distribuição de ROS contam também com a possibilidade de instalação do GAZEBO, um software de visualização gráfica das simulações.

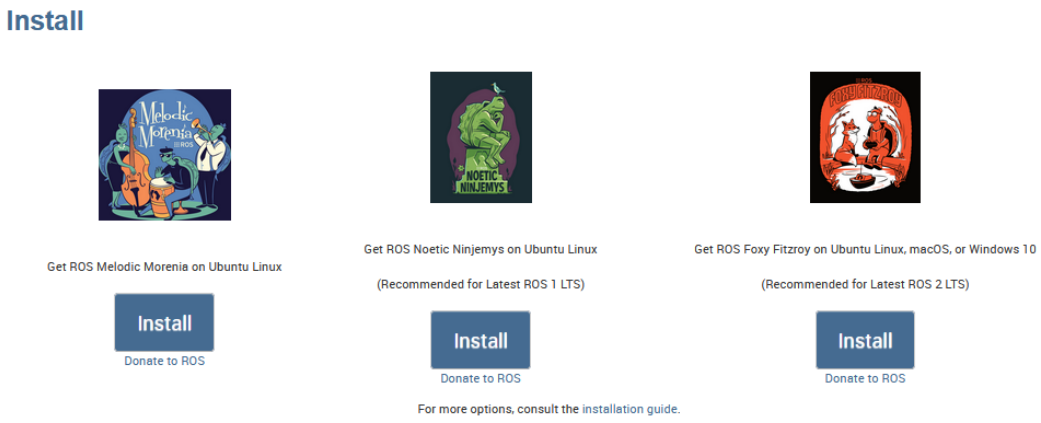


Figura 2 – Versões de ROS atualmente disponíveis para Download. (OSRF, 2021)

2.1.1.1 GAZEBO

GAZEBO é um software de simulação voltado para aplicações de robótica. Essa ferramenta foi desenvolvida para permitir uma visualização mais completa do que está sendo programado (GAZEBO, 2021). Entre suas principais funcionalidades destacam-se:

- Existência de modelos prévios de robôs e objetos;
- Vasta gama de sensores disponíveis para implementação;
- Grande biblioteca de modelos disponíveis providenciados e liberados ao público como é o caso do quadrirrotor Hector (MEYER et al., 2012).

Essas bibliotecas de modelos já existentes permite uma implementação mais rápida das ferramentas e códigos necessários para implementar uma rotina de validação de ferramentas para robôs existentes, compatível com SO's Linux. No caso do Hector, desenvolvido pela Universidade Técnica de Damrstadt, apresenta um modelo de aeronave com diversas bibliotecas já validadas experimentalmente (MEYER et al., 2012). No entanto, embora o Hector seja uma prova de todo o potencial desse conjunto de ferramentas, ele também é um exemplo de um dos seus maiores problemas. Desenvolvido em 2012, o modelo foi concebido para versões do software que foram descontinuadas: as distribuições Kinetic e Indigo.

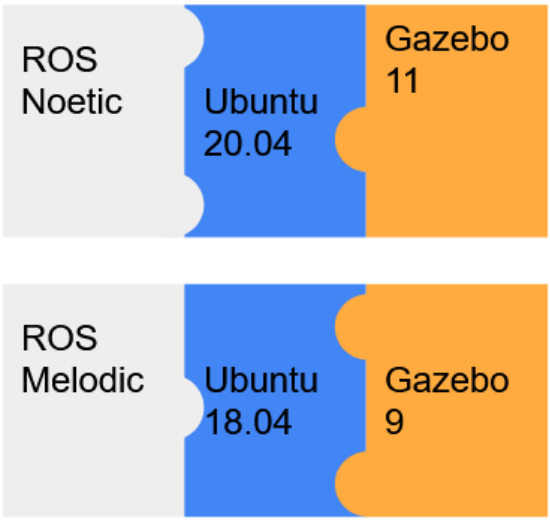


Figura 3 – Esquema simplificado de como funciona a compatibilidade entre as versões de ROS, Gazebo e Ubuntu.

No entanto, para o bom funcionamento do ROS e suas dependências, ele precisa trabalhar quase como um quebra-cabeça, ligando a versão correta do software com a distribuição do Ubuntu e do GAZEBO adequadas como ilustrado na figura 3. Embora as distribuições sejam constantemente atualizadas, cada versão de GAZEBO conta com bibliotecas próprias o que faz com que para que um projeto construído para versões anteriores do ROS funcionar com uma nova distribuição do GAZEBO, boa parte das bibliotecas precisam ser manualmente atualizadas. Na mesma direção, instalar modelos de ROS em uma versão do Ubuntu diferente da mais adequada pode trazer problemas de compatibilidade. Assim, mesmo que existam modelos de base para simulação, caso os blocos não estejam muito bem alinhados, erros podem aparecer e atrapalhar o processo de simulação.

Esta dificuldade foi experimentada ao longo do desenvolvimento deste projeto durante a fase de avaliação das ferramentas. Diversas tentativas de instalação e aplicação de modelos utilizando o conjunto ROS e GAZEBO foram feitas. Algumas, usando máquinas virtuais como a interface *Windows Subsystem Linux - WSL* - desenvolvida pela Microsoft. Em outras, tentativas de acesso remoto a uma máquina com uma configuração Linux pré-existente foram realizadas. Em todos os testes realizados no decorrer do projeto, colocar em prática uma solução funcional e comandável se mostrou um grande desafio e os problemas de compatibilidade resultaram em diversas falhas.

2.1.2 MATLAB/Simulink

MATLAB e Simulink (MATHWORKS, 2021) possuem uma vasta gama de aplicações de engenharia como: controle e automação, análises estatísticas, cálculos estruturais,

Figura 4 – Exemplo de interface gráfica desenvolvida usando o MATLAB App Designer. Essa interface foi desenvolvida pelo autor para auxiliar num projeto de dimensionamento e análise de sistemas propulsivos.

desenvolvimento de aplicativos e muito mais. De maneira geral, MATLAB é um ambiente de programação que utiliza sua própria linguagem. Esse ambiente conta com diversos recursos que buscam facilitar o desenvolvimento de rotinas computacionais como:

1. MATLAB Live Script: Criador de códigos e rotinas *ao vivo*. Esse sub-ambiente permite que os códigos sejam executados em bloco, permitindo um dinamismo maior na hora de executar programas complexos.

2. MATLAB App Designer: Sub-ambiente de programação que auxilia na criação de Interfaces Gráficas de Usuário (GUI's) por meio de um sistema arraste e solte. As interfaces podem então ser complementadas com código para executar rotinas distintas e trabalhar com outras funções pré-existentes. A figura 4 mostra um exemplo do que pode ser feito utilizando a interface. O App Designer conta ainda com ferramentas voltadas para aplicações aeroespaciais, com a possibilidade de inserção de elementos como altímetros e velocímetros.

3. Simulink: Ambiente do MATLAB de programação por blocos. O Simulink possui diversas ferramentas para simulações de eventos e é amplamente utilizado para o desenvolvimento de rotinas de controle. O Simulink também é conhecido por apresentar extensas bibliotecas como a *Aerospace Toolbox* e possuir uma interface de programação multi-física conhecida como Simscape, ampliando enormemente sua quantidade de aplicações.

Todas essas funcionalidades se complementam e podem ser trabalhadas em conjunto. No entanto, seu maior ponto negativo é o acesso à ferramenta. Embora, algumas formações, como é o caso da engenharia aeronáutica na USP, existam cursos e disciplinas que estimulam o aprendizado da ferramenta, pode ser uma ferramenta de difícil acesso por se tratar de uma

solução paga. De acordo com o site oficial do programa, uma licença perpétua ao programa custa mais de R$ 12000,00 (MATHWORKS, 2021) com a cotação do dólar de 10 de julho de 2021 (1 USD = 5.26 BRL) (GOOGLE, 2021a), podendo inviabilizar o desenvolvimento de uma solução nesse sentido. Esse preço pode ser ainda mais elevado se levadas em conta a inclusão de bibliotecas adicionais como *Aerospace Toolbox* (aproximadamente R$ 7900,00) e o próprio Simulink (aproximadamente R$ 18700,00).

2.1.2.1 Flight Gear

Embora as bibliotecas de programação existentes em MATLAB para aplicações aeroespaciais sejam, de maneira geral, completas, é interessante utilizar uma ferramenta externa de processamento gráfico em parceria com os códigos e rotinas de simulação para auxiliar na visualização e compreensão dos resultados. Nesse sentido, uma interessante ferramenta de apoio para o trabalho usando o MATLAB é o FlightGear.

FlightGear, é um simulador de voo de código aberto que aceita contribuições de diferentes desenvolvedores ao redor do mundo. O programa pode ser usado para simular diferentes tipos de veículos aéreos, indo desde pequenas aeronaves, até mesmo foguetes como apresentado na figura 5. Entre as características que colocam o *software* em posição de destaque, ressalta-se:

- Existência de bibliotecas prévias em MATLAB e Simulink para a realização de simulações com FlightGear;
- Existência de modelos distintos de aeronaves de base que poderiam ser utilizadas para simulações simplificadas;
- Utilizado por projetos que envolvem diferentes atores aeronáuticos como o CNES, como pode ser visto na figura 5;
- Desenvolvimento contínuo da ferramenta, contando com diversas atualizações para sua modernização, tendo sua versão mais recente lançada em dezembro de 2020.

Figura 5 – Exemplo de Simulação Conduzida em Simulink visualizada pelo Software FlightGear de um modelo de Foguete Representativo do Projeto Perseus. Retirado do acervo pessoal do autor.

2.1.3 AirSim

Na sequência, destaca-se o *Airsim*, uma ferramenta de código aberto desenvolvida pela Microsoft em mais uma de suas investidas em fornecer ambientes de trabalho de código aberto, como foi feito com o WSL. O software, inspirado em ferramentas como o próprio Gazebo e trabalhos como o Hector segue um design modular, contando com modelos pré-definidos de veículos, condições físicas já estabelecidas, e uma larga gama de sensores já inclusos, como diferentes tipos de câmeras e barômetros para dar mais realismo à simulação (SHAH et al., 2017). Falando da simulação de drones em específico, o ambiente conta com um modelo padrão de aeronave, um quadricóptero cuja representação simplificada está presente na figura 6.

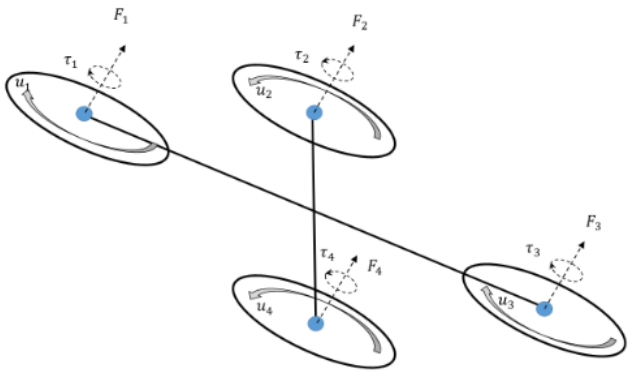


Figura 6 – Modelo simplificado do quadricóptero de base do *Airsim*. Retirado de (SHAH et al., 2017).

Os modelos físicos por trás da simulação são descritos em (SHAH et al., 2017), onde é possível verificar, por exemplo, as equações de forças e momentos de cada um dos rotores, as rotinas de cálculo que apoiam sensores como os acelerômetros, o modelo de arrasto utilizado e o modelo de gravidade. Além disso, as condições de pressão e temperatura da atmosfera se baseiam em modelos padrões de atmosfera. Como a ferramenta ainda é nova, diversas novas funcionalidades vem sendo adicionadas, como as bibliotecas para classificação de imagens que foram adicionadas em junho de 2021 nos dados do programa. (MICROSOFT, 2021)

Diferentemente das soluções anteriores, o *Airsim* conta com a possibilidade de interfacear a parte de programação com as soluções visuais de motores de desenvolvimento de jogos como a *Unreal Engine*, de maneira operacional, e a *Unity*, ainda em fase de implementação. A principal vantagem de contar com motores gratuitos de desenvolvimento de jogos é que existe uma indústria crescente de mídias visuais que está sempre investindo em novas ferramentas para trazer realismo aos seus produtos, de maneira participativa por diversos usuários que podem enviar modelos de partes e ambientes. Desta forma, é de se esperar, não somente encontrar uma vasta gama de objetos que possam ser incorporados, como simulações graficamente mais completas, podendo acrescentar veículos, pedestres e

diversos outros itens de interesse. Na figura 7, é possível ver um exemplo de uma simulação de um acidente aéreo gerada utilizando o ambiente da *Unreal* (ATTACHE, 2021). Já na figura 8, é possível ver o drone de base do *Airsim* em um ambiente padrão de simulação chamado *Blocks*. Para que o motor de jogos funcione corretamente, uma aplicação do *Visual Studio* é aberta e a rotina principal é executada. A partir daí, o usuário do software pode controlar, seja um carro, seja um drone. Embora existam modelos padrão para ambos os veículos, também é possível que o usuário envie seus próprios modelos tal qual ocorre com o ROS e o Gazebo. (MICROSOFT, 2021)

Figura 7 – Exemplo de Simulação de Acidente em Voo realizada com o suporte da Unreal Engine. Retirado de (ATTACHE, 2021).

Figura 8 – Drone de base voando em um ambiente de simulação com o modelo *Blocks*, padrão do Airsim.

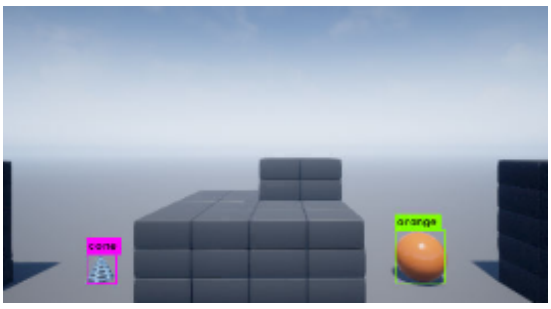
Ainda segundo a documentação do *Airsim* (MICROSOFT, 2021), para o controle dos veículos, algumas opções estão disponíveis e já implementadas, como:

1. Controle manual usando teclado (somente carros);
2. Controle usando controle remoto e/ou controle de videogame (carros e drones via software de acesso livre QGroundControl);

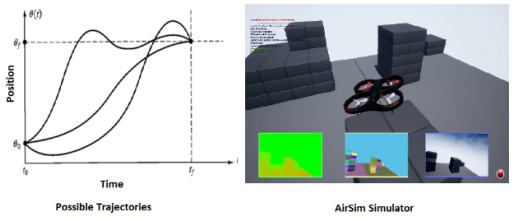
3. API's em Python e C++ para controle dos veículos.

Ademais, o software está disponível para utilização em múltiplos SO's incluindo Windows, o que pode facilitar o acesso à ferramenta. Destaca-se também, a crescente utilização do programa para a validação de diversas pesquisas e atividades de competição, mostrando o crescente interesse da comunidade acadêmica no *Airsim*. Entre estas atividades, menciona-se:

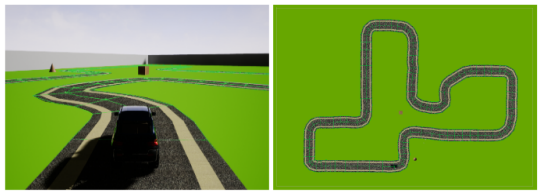
- Atividades de pesquisa de classificação de imagens com o drone em movimento, como pode ser visto na figura 9a (AGGARWAL et al., 2018);
- Estudos de geração de trajetórias, tal qual apresentado na figura 9b (BHUSHAN, 2019);
- Pesquisas com a utilização de redes neurais para aprendizado no ambiente simulado, conforme imagem da figura 9c (MERTENS, 2018);
- Simulações de múltiplas aeronaves em situações diversas como corridas, conforme consta na imagem 9d (MADAAN et al., 2020).



(a) Exemplo de simulação no *Airsim* para criação de programas de classificação de imagens para drones. Retirado de (AGGARWAL et al., 2018).



(b) Exemplo de simulação no *Airsim* para estudos diversos de geração de trajetórias. Retirado de (BHUSHAN, 2019).



(c) Exemplo de simulação no *Airsim* com o estudo de redes neurais. Retirado de (MERTENS, 2018).

(d) Simulação de múltiplos drones no contexto de uma corrida. Retirado de (MADAAN et al., 2020).

Figura 9 – Exemplo de pesquisas desenvolvidas com Airsim nos últimos anos.

Todas essas razões tornam o *Airsim* um candidato cada vez mais interessante para utilização em simulações de drones.

2.1.4 Outras Soluções

Entre as outras soluções de destaque para a programação de drones, destaca-se a plataforma RoboMaker desenvolvida pela *Amazon Web Services* - AWS. Essa solução de computação em nuvem pode permitir que usuários realizem trabalhos complexos de simulação de robôs sem a necessidade de possuir uma máquina com elevado poder de processamento de dados. A figura 10, mostra um exemplo de um ambiente gerado usando a ferramenta *WorldForge* para simulações com RoboMaker Destaca-se ainda, a integração da solução desenvolvida pela AWS pode contar com o suporte de outras ferramentas da companhia, incluindo soluções para:

- Aprendizado de Máquinas;
- Gerenciamento de Banco de Dados;
- Internet das Coisas;
- e outros...

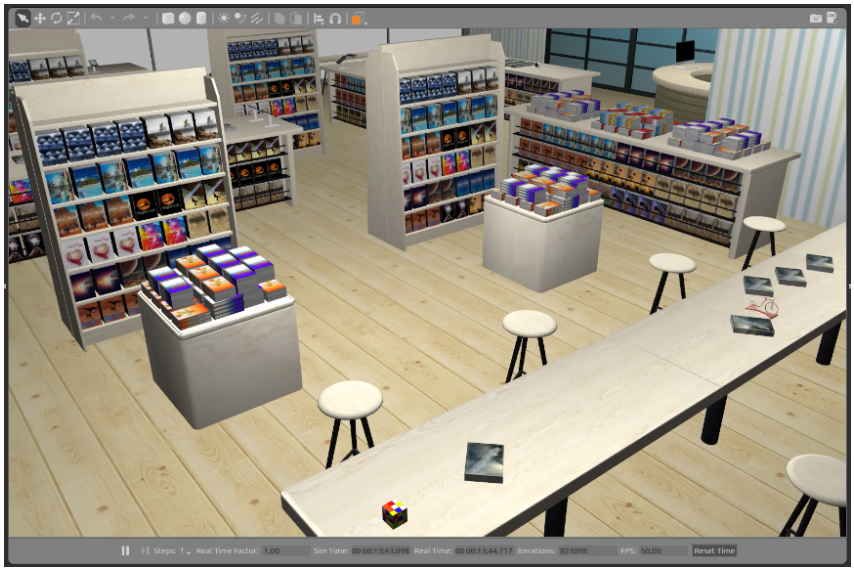


Figura 10 – Exemplo de modelo padrão de ambiente para o AWS RoboMaker. Retirado de (AWS, 2021).

Destaca-se, porém, que as ferramentas da AWS podem acabar se tornando muito custosas para o desenvolvimento de simulações, visto que, não somente os preços são com base na cotação do dólar como também o uso de diferentes funcionalidades pode acarretar custos adicionais em todas as aplicações.

2.2 Ferramentas de Programação

Buscando descrever as ferramentas que complementam os softwares de simulação de drones, essa seção visa descrever as ferramentas e linguagens de programação que podem ser utilizadas para criar as rotinas de diálogo com os softwares.

2.2.1 C/C#/C++...

Iniciando a discussão sobre possíveis linguagens de programação a serem utilizadas no desenvolvimento do trabalho, temos C, uma linguagem de programação orientada objeto. Desenvolvida no início dos anos 70, C é uma das linguagens mais difundidas no mundo da programação. A linguagem surgiu a partir do trabalho de pesquisadores da *Bell Labs* e começou a se espalhar mais fortemente na década de 80 em virtude do sucesso de alguns experimentos de portabilidade (RITCHIE, 1993). C é considerada uma linguagem capaz de fornecer as ferramentas essenciais para todos os programadores e otimizada. Atualmente, a linguagem C recebeu dois sucessores C# e C++ que compartilham muitas de suas características. C++, por exemplo, é uma linguagem muito utilizada em controladores como ocorre com os produtos da *Microchip* (MICROCHIP, 2021), além de possuir compatibilidade com Interfaces de programação de Aplicações (APIs) do *Airsim* e com ROS.

C se tornou a linguagem de programação de diversos cursos de engenharia, mas alguns pesquisadores buscam a anos alternativas que sejam mais adaptadas para que os estudantes possam ter um contato mais apropriado com a primeira linguagem de programação (WIRTH; KOKVESI, 2006), (FANGOHR, 2004) por ser considerada de sintaxe complexa e por possuir menos bibliotecas disponíveis que outras soluções como MATLAB e Python.

2.2.2 MATLAB

Como esperado, MATLAB é a ferramenta de programação desenvolvida pela MATHWORKS (MATHWORKS, 2021) para uso dentro do seu software, com diversas funcionalidades adicionais como conversão de código MATLAB para outras linguagens e possibilidade de chamar rotinas de MATLAB por outros ambientes. Essa ferramenta é considerada uma linguagem de script (FANGOHR, 2004), na qual o código é interpretado durante sua execução. Com o crescimento da utilização da ferramenta na indústria, diversas instituições tem pensado e aplicado a linguagem como ferramenta de ensino à programação para engenharia (WIRTH; KOKVESI, 2006), como é o caso da própria Universidade de São Paulo e o curso de Engenharia Aeronáutica. O artigo de Wirth e Kokvesi destaca ainda propriedades fundamentais da linguagem como:

- Sintaxe facilitada auxiliando a aprendizagem e permitindo que os programadores se foquem mais em solucionar problemas do que na escritura do código e compilação,

fato também ressaltado pelo artigo de Fangohr;
- Sistema claro de reportar erros, auxiliando na correção de problemas;
- Existência de um grande ambiente de apoio para programação.

Ainda segundo os autores, essas vantagens superam problemas como a falta de otimização comparado a linguagens mais clássicas como C. Destaca-se, porém, que a utilização da linguagem enfrenta os mesmos problemas de acesso que o software em si, devido ao alto custo de utilização e aquisição de licenças, embora constantes bibliotecas estejam sendo desenvolvidas e novas atualizações da interface sejam feitas de forma rotineira.

2.2.3 Python

Python é uma linguagem de código aberto gerida pela Python Software Foundation (PSF, 2021). Essa linguagem é descrita em seu site como sendo de fácil aprendizado e intuitiva. Python é uma linguagem considerada por alguns autores (FANGOHR, 2004) como ainda mais fácil e intuitiva que MATLAB para utilizar como primeira forma de aprendizado de programação, tendo vantagens como:

- Acesso grátis por se tratar de uma linguagem de código aberto;
- Existência de compiladores para diversos sistemas operacionais;
- Existência de bibliotecas de programação que acrescentam funcionalidades de desempenho e visualização de dados.

O autor destaca ainda que a linguagem passou por um grande salto de utilização sendo referência em trabalhos ligados à NASA e ao Google, por exemplo. Ademais, foi realizado um estudo comparativo do uso de Python, C e MATLAB em sala de aula e os autores destacam que Python foi uma excelente linguagem para estudantes de engenharia e ciências (FANGOHR, 2004). Ainda sobre a linguagem, a fundação destaca em sua documentação (PSF, 2021) uma série de bibliotecas de interesse como:

1. Tkinter - Biblioteca para a criação de interfaces gráficas de usuário;

2. Threading - Biblioteca para a execução e criação de fios de execução distintos;

3. Email - Biblioteca que pode gerenciar mensagens de e-mail, auxiliando os códigos à se conectarem à internet.

Todas essas bibliotecas complementam e muito o potencial de utilização da linguagem para diversas aplicações. Destaca-se ainda que além das bibliotecas básicas, existem diversas bibliotecas disponibilizadas gratuitamente por usuários diversos em repositórios online com o Pypi (PYPI, 2021).

2.3 Escolha da Ferramenta Final

A partir do levantamento bibliográfico presente no capítulo 2, foi possível se traçar um panorama global das principais ferramentas utilizadas atualmente no ambiente de programação robótica, sobretudo aquelas ligadas à operação de veículos aéreos. Embora algumas soluções como utilizar a dupla ROS/GAZEBO ou ainda o trio MATLAB/Simulink/FlightGear se mostrem promissores para a solução de problemas reais de engenharia, é preciso se conhecer de maneira aprofundada seus pontos positivos e negativos antes de realizar a escolha definitiva.

2.3.1 Pesquisa de Opinião

No ambiente de trabalho de engenharia, a interdisciplinaridade sempre foi muito importante. Num projeto de um veículo aéreo, diversas áreas se interconectam para fazer o produto final. Aerodinâmica, estruturas, programação, eletrônica e gerenciamentos de projeto são só alguns exemplos de onde o engenheiro pode atuar num projeto aeronáutico. Ao mesmo tempo, para cumprir sua missão, o engenheiro precisa dialogar com diferentes pessoas envolvidas no tema que não necessariamente terão a mesma formação e assim, é importante não somente saber se comunicar como também entender as necessidades de cada um dos atores envolvidos na cadeia de desenvolvimento. Assim, uma pesquisa de opinião foi conduzida visando entender um pouco melhor a formação de engenheiros de diferentes escolas do Brasil, seu interesse pelo tema e sua familiaridade com diferentes ferramentas computacionais.

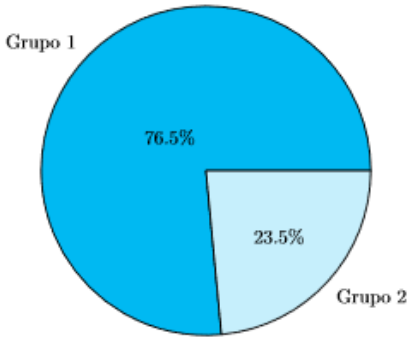
A pesquisa contou com a participação de 51 pessoas de 7 universidades distintas. Cada uma das pessoas teve que responder sobre alguns temas como:

- Interesse por drones;
- Linguagens de programação e sistemas operacionais mais usados;
- Linguagens de programação inicialmente estudadas.

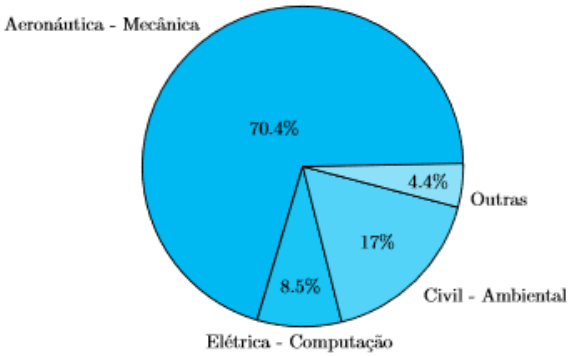
As figuras 11, 12, 13 e 14 resumem algumas das respostas apresentadas ao levantamento de informações para perguntas de múltipla escolha. Além dessas perguntas, alguns campos de opinião foram deixados aberto para quaisquer comentários adicionais pudessem ser usados para complementar este trabalho.

Assim, como é possível ver na Figura 11, mais de três quartos dos participantes eram alunos de graduação (grupo 1), enquanto 23.5% eram graduados ou pós-graduandos (grupo 2). Nota-se também uma predominância de percursos de formação ligados às engenharias civil, aeronáutica e mecânica. Por sua vez, a figura 12 traz uma informação interessante sobre veículos aéreos autônomos. Entre os participantes da enquete, embora quatro quintes possua interesse por temas ligados à drones, menos de 12% dessas pessoas acabam trabalhando com o tema, seja de maneira direta (projeto e afins), de maneira

indireta (consultoria) ou por passatempo. Esse comportamento por parte dos entrevistados chama a atenção visto que pode indicar a existência de possíveis entraves para acesso a esse tipo de tecnologia.

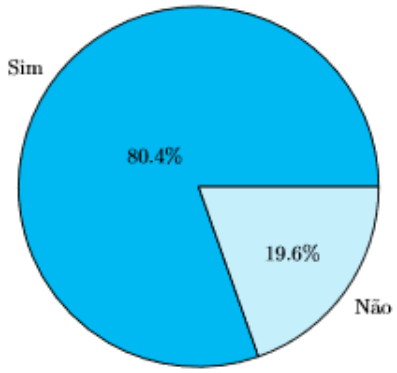


(a) Levantamento de nível de formação da pesquisa de opinião.

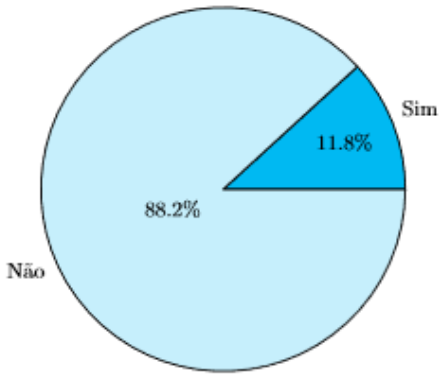


(b) Levantamento de nível de formação da pesquisa de opinião.

Figura 11 – Levantamento de perfil dos participantes do estudo.



(a) Você tem interesse por temáticas ligadas à drones?

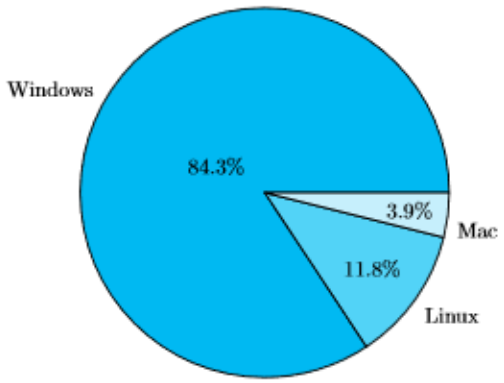


(b) Você trabalha com drones, de maneira direta ou indireta?

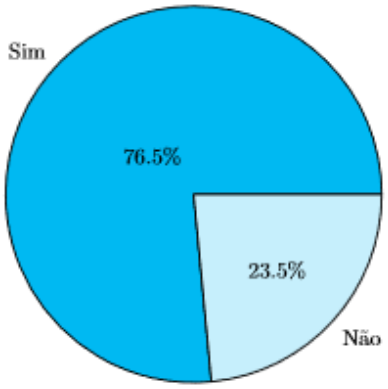
Figura 12 – Levantamento da relação dos participantes com o mundo dos drones.

Na sequência, o gráfico da figura 13a mostra que a ampla maioria dos entrevistados consideram o Windows como seu SO mais familiar. Ao mesmo tempo, como é mostrado na figura 13b, mais de três quartos dos entrevistados tiveram que em algum momento da vida, trabalhar com mais de um sistema operacional rodando na mesma máquina. De maneira complementar, as figuras 14a e 14b mostram que a maioria dos entrevistados precisou realizar tal configuração para conseguir acesso a sistemas Linux sendo que 76.9% de todas as configurações foram por alguma obrigatoriedade ligada ao sistema. Isso mostra que muito embora os entrevistados demonstrem habilidade para trabalhar com múltiplos

sistemas, a maior parte só o fez por necessidades muito específicas. Além disso, alguns participantes relataram dificuldades do processo de configuração de máquinas virtuais e *dual boot*, o que pode atrapalhar na implementação dessas ferramentas.

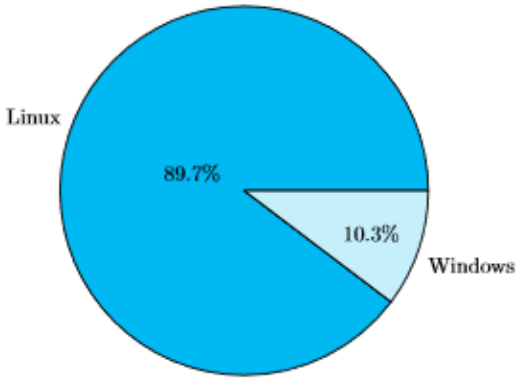


(a) Qual o SO com o qual você tem mais familiaridade?

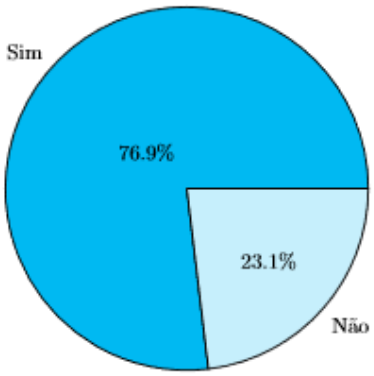


(b) Você já teve que trabalhar com máquinas virtuais/*dual boot*?

Figura 13 – Levantamento da relação dos participantes sobre sistemas operacionais e uso de máquinas virtuais/*dual boot*



(a) Seu interesse por máquinas virtuais/*dual boot* foi para conseguir acesso à qual SO?

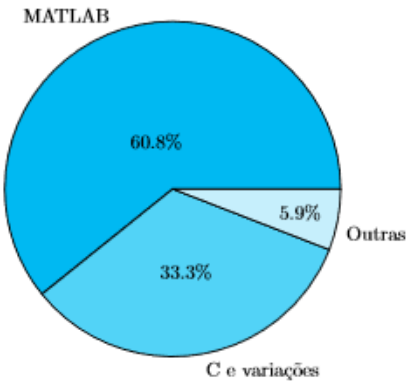


(b) Você precisou fazer isso pela necessidade de um programa? Por exemplo, código que só roda em Linux.

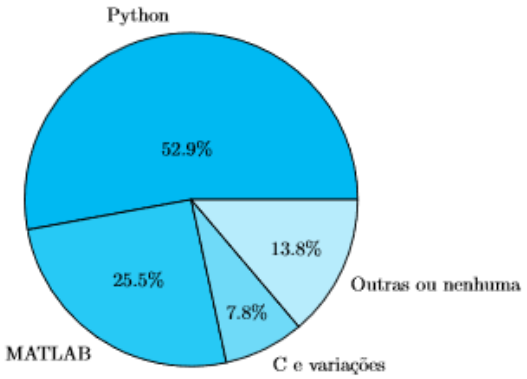
Figura 14 – Levantamento da relação dos participantes com máquinas virtuais e configurações de *dual boot*.

Por fim, os dois gráficos da figura 15 mostram também outra característica muito peculiar dos entrevistados. Como esperado, MATLAB e C (e variações) foram as principais ferramentas de programação que os participantes tiveram contato. C, costuma ser a linguagem de programação de diversos cursos de iniciação às ciências da computação, enquanto MATLAB é a primeira forma de programação apresentada no curso de Engenharia Aeronáutica da USP. No entanto, Python e outras linguagens como Java, foram aprendidas

pelos participantes e passaram a ocupar o lugar de preferência. Desta forma, embora a figura 15a mostre que somente 5.9% das pessoas tiveram seu contato inicial com ferramentas que não fossem C (e variações) e MATLAB, a figura 15b mostra que somente 33.3% dos participantes as tem como ferramentas preferidas. Assim, é possível inferir que, embora, uma parte considerável dos estudantes de engenharia tenham seu primeiro contato com a programação somente durante a universidade, suas atividades os levam a ter contato com uma gama muito maior de linguagens, permitindo que encontrem aquelas mais adaptadas a seus interesses e atividades.



(a) Qual a primeira ferramenta de programação que você teve contato com?



(b) Qual sua ferramenta preferida atualmente?

Figura 15 – Qual sua ferramenta de programação preferida?

2.3.2 Decisão Final

Finalizada a etapa de levantamento bibliográfico, as tabelas 1 e 2 foram confeccionadas e listam alguns dos principais pontos positivos e negativos encontrados para cada uma das ferramentas e linguagens analisadas.

Tabela 1 – Pontos positivos e negativos dos diferentes softwares base

Software Principal	Prós	Contras
ROS/Gazebo	- Existência de Modelos de Base - Comunidade Grande e Participativa - Gratuito - Crescente Interesse em Pesquisa	- Difícil compatibilidade de versões e SO's - Ao invés de as versões serem atualizadas elas são descontinuadas e novas versões são lançadas tornando os códigos obsoletos - Problemas de retrocompatibilidade
MATLAB/Simulink	- Excelente Poder Computacional - Desenvolvimento constante de novas ferramentas de suporte - Capacidade de integrar diversas funcionalidades para uma simulação completa - Capacidade de criar GUI's	- Elevado preço para utilização do software - Software não necessariamente conhecido por um grande número de usuários - Necessidade de procurar aplicações de terceiros para uma boa visualização gráfica
Airsim	- Capacidade de utilizar todo o potencial gráfico da Unreal Engine - Compatibilidade com Windows - Possibilidade de utilizar controles remotos para teste - API's existentes para Python e C - Constante desenvolvimento de novas funcionalidades	- Menor liberdade para criação de modelos de robôs - Necessidade de maior poder computacional para utilizar a Unreal Engine - Diversas funcionalidades ainda estão em fase de desenvolvimento

Tabela 2 – Pontos positivos e negativos das diferentes ferramentas de programação

Ferramentas de Programação	Prós	Contras
MATLAB	- Primeira Ferramenta vista no curso de Engenharia Aeronáutica da USP - Excelente gama de bibliotecas existentes - Possibilidade de diálogo direto com o Simulink	Os mesmos da utilização do software, somados à perda de interesse por parte de diversos alunos, como indicou a pesquisa de opinião.
C/C#/C++...	- Primeira Ferramenta vista em diversos outros cursos de engenharia - Utilização gratuita - Rápido processamento - Existência de diversos compiladores e tutoriais - Extensa comunidade online	- Complexidade maior para escritura dos códigos - Necessidade de controlar a gestão de memória - Menor quantidade de bibliotecas disponíveis - Também apresentou perda de interesse na pesquisa
Python	- Excelente gama de bibliotecas existentes - Utilização gratuita - Linguagem preferida pela maior parte dos participantes da pesquisa - Linguagem de sintaxe amigável - Constante expansão de suas aplicações - Extensa comunidade online	- Ainda não é a primeira linguagem de contato nos cursos de engenharia - Não pode utilizar ponteiros

Assim, após análise das diferentes possibilidades existentes e com base nos resultados da pesquisa de opinião e nos levantamentos de informações acima realizados, optou-se por realizar a implementação computacional utilizando o conjunto *Airsim*, *Unreal* e *Python*. Esse triplete foi escolhido como o mais adequado para o trabalho por apresentar características como:

- Compatibilidade com múltiplos sistemas operacionais, incluindo Windows;
- Constante desenvolvimento de novas ferramentas o que pode auxiliar em desenvolvimento de simulações futuras e seu uso crescente em competições;
- Possibilidade de contar com a *Unreal Engine* para a visualização de cenários extremamente realistas, podendo incluir pessoas e afins;
- Existência de API's em Python para trabalhar com o simulador permitindo uma ligação com uma linguagem de programação conhecida por diversos estudantes e engenheiros;
- Com exceção de alguns ambientes e ferramentas que possam complementar o ambiente da *Uneral*, todas as ferramentas são gratuitas;
- Possibilidade de contar com as diversas bibliotecas de Python já existentes para

análise de dados, geração de gráficos e criação de interfaces gráficas.

Estas características contribuíram para o excelente desempenho do conjunto nas matrizes de decisão apresentadas nas tabelas 3 e 4. A matriz apresenta os pesos definidos para cada um dos principais tópicos avaliados com notas variando entre 1 até 3. O conjunto escolhido conta com a maior média ponderada pelos pesos adotados. Abaixo, um breve resumo dos critérios adotados:

1. Custos: diz respeito à facilidade de acesso à solução. No caso do Airsim, embora o software seja de acesso livre, a Unreal Engine acaba por exigir uma capacidade computacional mais elevada e assim, é necessário ter uma máquina mais potente;

2. Compatibilidade: trata da possibilidade de utilizar sistemas operacionais e versões distintas do software;

3. Facilidade de Implementação: avalia a facilidade de instalar o software e realizar simulações com a ferramenta;

4. Capacidade computacional necessária/Processamento: critérios que buscam compreender qualitativamente quantos recursos são necessários para executar a solução;

5. Bibliotecas existentes: diz respeito à existência de conjuntos pré-prontos de códigos para a realização de funções diversas;

6. Complexidade: trata de possíveis entraves para a implementação de novos códigos e rotinas;

7. Tradição: aborda a quanto tempo a ferramenta é utilizada para resolver problemas de engenharia ligados ao tema;

8. Continuabilidade: principal parâmetro ligado à pesquisa de opinião, aborda as preferências de usuários para a continuabilidade da solução para o futuro.

Tabela 3 – Matriz de Decisão para os Softwares de Simulação

		Software de simulação		
Critério	Peso	ROS	Airsim	MATLAB
Custos	4	3	2	1
Compatibilidade	3	1	3	2
Facilidade de Implementação	3	1	3	2
Capacidade computacional necessária	3	3	1	2
Tradição	2	3	1	2
Continuabilidade	2	1	3	2
Média Ponderada		2,1	2,2	1,8

Tabela 4 – Matriz de Decisão para as Ferramentas de Programação

		Ferramenta de Programação		
Critério	Peso	C/C#/C++...	Python	MATLAB
Custos	4	3	3	1
Bibliotecas Existentes	3	1	2	3
Processamento	3	3	2	1
Complexidade	3	1	3	2
Tradição	2	3	1	2
Continuabilidade	2	1	3	2
Média Ponderada		2,1	2,4	1,8

Desta forma, convencionou-se a criação de um projeto inicial como o apresentado na figura 16, onde uma interface gráfica desenvolvida com *Tkinter* interage com o ambiente da *Unreal* gerido pelo *Airsim*.

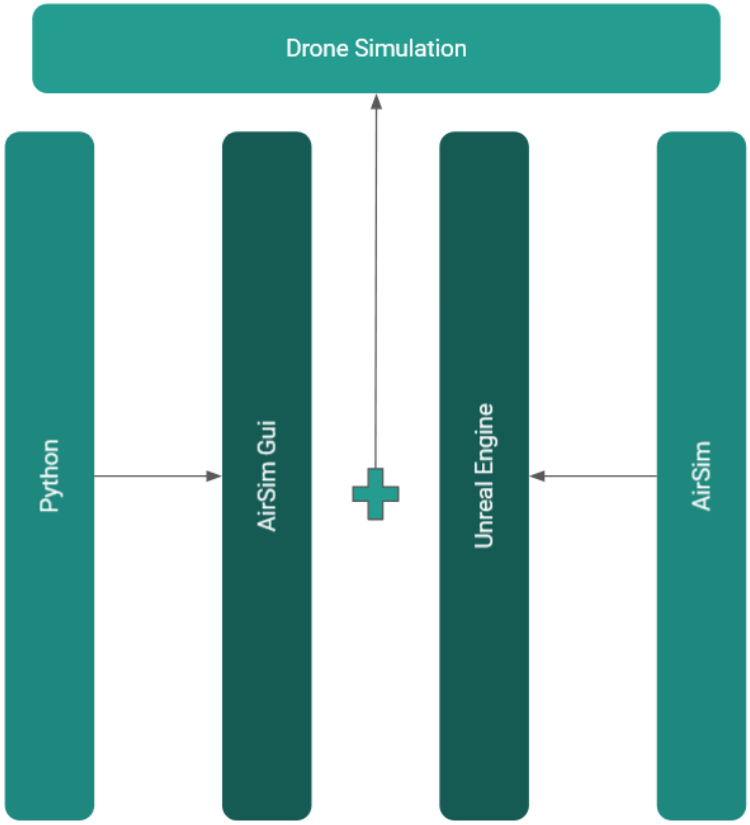


Figura 16 – Esquema simplificado dos componentes inicialmente idealizados para a simulação computacional de drones escolhida.

3 IMPLEMENTAÇÃO

Definida a ferramenta e a arquitetura de base desejada para o sistema, foi possível passar para a parte de implementação

3.1 Familiarização com a Ferramenta

A familiarização com a ferramenta *Airsim* ocorreu por duas frentes principais. A primeira, uma interação direta com o software e seu ambiente. Para tal, o *Airsim* oferece duas possibilidades principais:

1. Interação direta via comandos de teclado;

2. Interação via controle remoto.

A interação direta via comandos de teclado funciona somente com o ambiente de simulação configurado para a utilização de veículos terrestres. Com isso, é possível realizar alguns pequenos comandos indicando para onde o veículo deve ir em terra, permitindo um pouco melhor o conhecimento de como a ferramenta se comporta. Já a interação via controle remoto é proporcionada por meio do software de acesso livre *QGroundControl*. Essa possibilidade de utilização de controles externos é extremamente interessante, pois poderá permitir no futuro a criação de uma rotina computacional mista que permita tanto o controle direto da aeronave via controle remoto, quanto o controle via interface gráfica, tudo isso, com rotinas computacionais sendo executadas para auxiliar na missão final do VANT.

A segunda possibilidade de familiarização é a utilização de algumas rotinas de teste pré-desenvolvidas para verificar na prática o que algumas funções e comandos fazem. Algumas rotinas como a *HelloDrone.py* ensinam o utilizador à se conectar via Python ao ambiente de simulação enquanto outras mostram como adicionar efeitos ambientais como vento. Essas rotinas foram ligeiramente adaptadas para a realização de alguns testes de simulação e depois as adaptações foram incluídas direta ou indiretamente na solução final.

Por fim, o ambiente da *Unreal Engine* foi levemente modificado adicionando modelos gratuitos de veículos para verificar como o ambiente de criação de mundo se comporta e quais os procedimentos necessários para se criar um mundo mais parecido com a realidade.

A figura 17 mostra que alguns carros foram adicionados ao ambiente, enquanto a figura 18 mostra uma condição na qual a aeronave padrão do *Airsim* sobrevoa alguns desses automóveis.

Figura 17 – Exemplo de modificação do ambiente *Blocks* na *Unreal Engine* com a adição de carros.

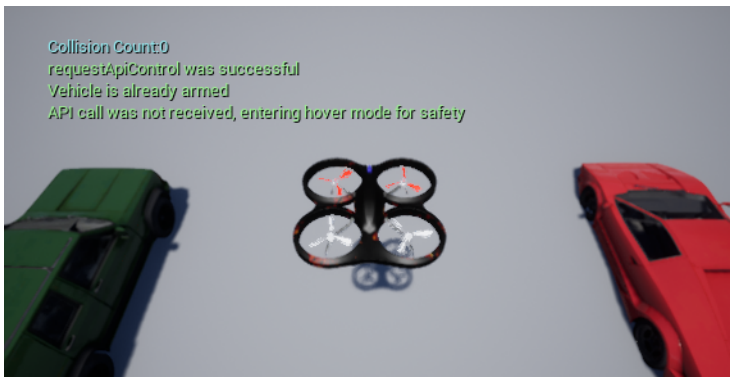


Figura 18 – Drone sobrevoando alguns carros no ambiente final de simulação.

3.2 Criação do Projeto

Após as etapas de familiarização inicial com a ferramenta, o projeto da interface gráfica pode ser realizado. Para tal, duas etapas foram necessárias.

1. A definição das missões que o usuário poderia executar com a plataforma;

2. A criação das rotinas computacionais de apoio.

Essas etapas são apresentadas em mais detalhes a seguir. Destaca-se aqui, somente uma alteração com relação às definições do *Airsim*. Embora os eixos coordenados como definidos na *Unreal Engine* sigam o padrão da figura 19, com os eixos x e y formando um plano paralelo ao chão e o eixo z vertical para cima, a simulação em si, trata o eixo z como negativo. Assim, na implementação do código, a interface gráfica recebe os parâmetros de z conforme a identificação da *Unreal Engine* e faz a conversão de sinal para a simulação. O procedimento inverso ocorre quando a GUI recebe um valor do *Airsim* e precisa disponibilizá-lo ao usuário.

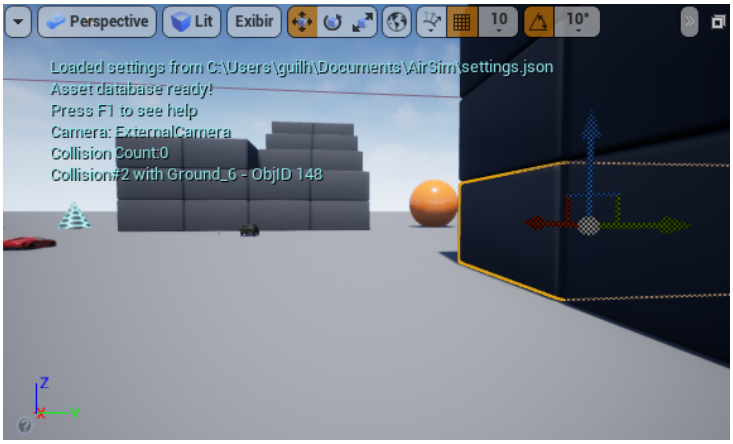


Figura 19 – Definição de Eixos na *Unreal Engine*.

3.2.1 Definição da Missão

Como a principal motivação do código à ser desenvolvido é permitir a criação de uma ferramenta de suporte para que os usuários interajam com o *Airsim*, foi necessário assumir algumas convenções. A primeira, seria que em condições normais de utilização, os movimentos do drone seriam divididos em três grandes blocos. Um primeiro, com movimentações entre pontos distintos que poderia considerar variações de posição nos 3 eixos coordenados, i.e., criação de retas entre dois pontos. Essas retas poderiam ser unidirecionais, nas quais se parte de um ponto inicial para um final, ou bi-direcionais nas quais o ponto inicial e o final são os mesmos, mas existe uma posição para a qual a aeronave se desloca antes de retornar ao início. Já o segundo, aborda movimentações seguindo formas geométricas pré-determinadas. Esse modelo de movimentação foi escolhido, por se assemelhar mais a um padrão de possíveis missões reais de monitoramento, onde a aeronave deve fazer uma varredura de uma região específica e possivelmente buscará manter sua altitude para a obtenção e análise de dados distintos por meio das câmeras e sensores instalados. Por fim, o último bloco aborda os movimentos de pouso e decolagem.

Destaca-se que o *Airsim* conta com API's pré-existentes para a realização de movimentos como pouso e decolagem e apresenta rotinas como *moveToPositionAsync* que permitem a realização de um movimento indo de um ponto de coordenadas conhecidas para outro. Já para os movimentos restantes, foi necessário criar algumas convenções dado que seria necessário combinar algumas das API's originais com códigos diversos. Assim, por opção de design, todos os comandos que envolvem trajetórias que divergem da combinação mais simples, i.e., movimentação entre dois pontos, ocorrerão no "Plano XY", muito embora, em virtude da complexidade inerente aos graus de liberdade do sistema, é de se esperar pequenas variações no plano vertical durante a manobra.

Desta forma, foram configurados comandos para a realização de movimentos em trajetórias retangulares e circulares em um plano definido, além das operações já mencionadas acima. A figura 20 apresenta uma representação do que foi convencionado

como sendo um movimento retangular. A posição inicial da aeronave é adotada como um ponto de partida situado em um dos vértices do retângulo. Do repouso, a aeronave parte em sentido anti-horário percorrendo cada um dos lados do retângulo com dimensão definida pelo usuário. A equação 3.1 ilustra o caminho percorrido pela aeronave passando pelos quatro vértices (numerados de 1 até 4) e retornando à posição inicial (x_0, y_0).

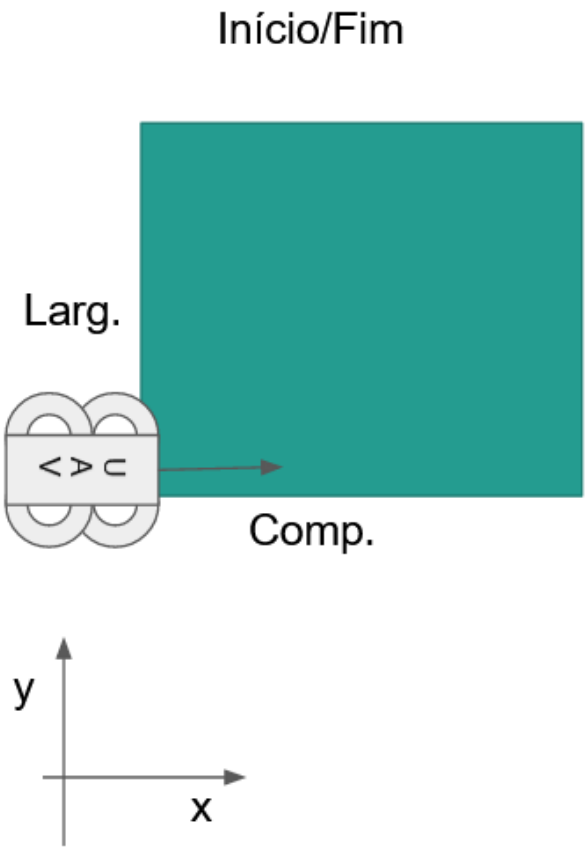


Figura 20 – Esquema simplificado de como um voo retangular foi definido no programa.

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_1 & y_1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 \\ x_0 + c_r & y_0 \\ x_0 + c_r & y_0 + l_r \\ x_0 & y_0 + l_r \\ x_0 & y_0 \end{bmatrix} \tag{3.1}$$

Para controlar a sequência de manobras necessárias para a varredura, foi utilizado um simples controle de tempo com base numa estimativa de quanto o movimento duraria caso a velocidade de movimento do drone seja realmente a desejada, como pode ser visto

na equação 3.2.

$$\begin{bmatrix} t_l \\ t_c \end{bmatrix} = \begin{bmatrix} l_r/v \\ c_r/v \end{bmatrix} \tag{3.2}$$

Por sua vez, o movimento circular pode ser representado pelo esquema da figura 21. Esse movimento, também realizado no sentido anti-horário é definido principalmente pelo raio da circunferência. Novamente, adotando a posição inicial do veículo como sendo o binômio (x_0, y_0), pode-se definir as coordenadas do centro da circunferência por meio da equação 3.3.

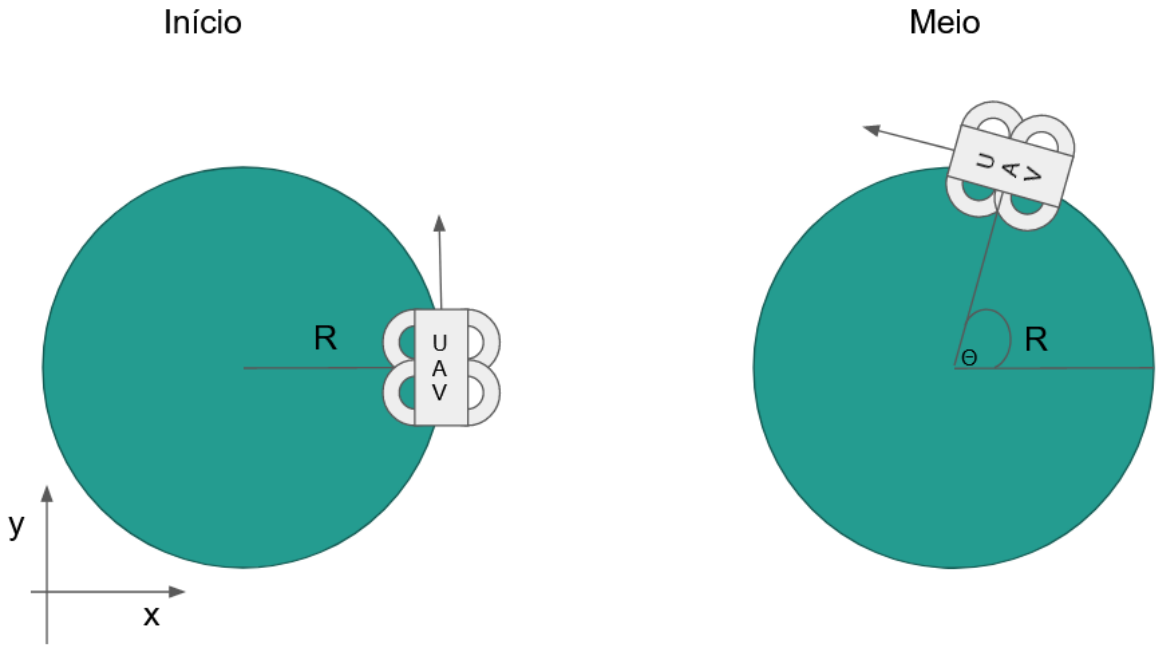


Figura 21 – Esquema simplificado de como um voo circular foi definido no programa.

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x_0 - R \\ y_0 \end{bmatrix} \tag{3.3}$$

No entanto, embora seja relativamente simples definir geometricamente um círculo, optou-se por discretizar o círculo, transformando-o num polígono com uma quantidade de lados determinada pelo utilizador. Quanto mais lados o utilizador escolherá, mais parecida a trajetória final será de uma circunferência. Para obter a velocidade angular e o período total, recuperar o ângulo atual e a posição do drone no polígono, utilizou-se as equações 3.4, 3.5, 3.6 e 3.7.

$$\omega = \frac{v}{R} \tag{3.4}$$

$$T = \frac{\omega}{2\pi R} \tag{3.5}$$

$$\theta = \omega \frac{T}{n} k \tag{3.6}$$

$$\begin{bmatrix} x_\theta \\ y_\theta \end{bmatrix} = \begin{bmatrix} x_c + Rcos(\theta) \\ y_c + Rsin(\theta) \end{bmatrix} \tag{3.7}$$

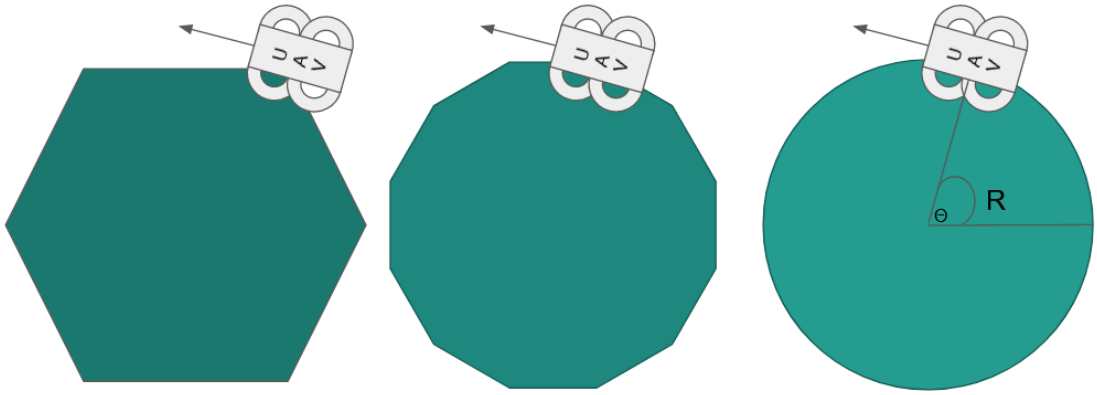


Figura 22 – Ilustração de diferentes níveis de discretização para um círculo.

3.2.2 Rotinas Principais

De maneira geral, o código construído para a interface gráfica e sua interação com o ambiente simulacional do Airsim pode ser dividido em algumas partes principais:

1. A criação da GUI propriamente dita, com todos seus itens e ações de retorno;
2. A criação de Threads para gerir ações em tempo real paralelamente ao comando da GUI;
3. Funções diversas que complementam o código e permitem ações diversas como a criação de animações.

3.2.2.1 Interface Gráfica

Desta forma, com o intuito de tornar o projeto mais amigável à qualquer usuário, uma interface gráfica em Python foi desenvolvida com o auxílio da biblioteca *Tkinter*. A principal ideia por trás da criação de uma interface gráfica é a possibilidade de difusão da ferramenta para uma comunidade mais ampla. Com um ambiente de programação mais amigável, seria possível que pessoas com diferentes níveis de conhecimento e funções dentro do projeto interajam com a ferramenta e extraiam resultados coerentes com aquilo

que lhes é de interesse. A figura 23 mostra como foi pensada a arquitetura inicial para a Interface Gráfica enquanto a figura 24 mostra uma das versões de desenvolvimento da interface.

Essa arquitetura trabalha com 4 seções principais:

- Seção de inicialização: responsável por conectar a interface ao ambiente de simulação do *Airsim*;
- Seção solo/voo: responsável por permitir o acesso às rotinas de pouso e decolagem;
- Seção de gerenciamento dos movimentos: responsável por permitir e gerir os diferentes tipos de missão. Essa parte da interface também permite a visualização da trajetória realizada e da desejada;
- Seção auxiliar: responsável por permitir ao usuário realizar tarefas adicionais como a geração de animações da trajetória.

Os códigos utilizados para a criação da interface estão presentes de maneira mais detalhada no Apêndice A.

Figura 23 – Ideia de base para o design da interface gráfica via *Tkinter*.

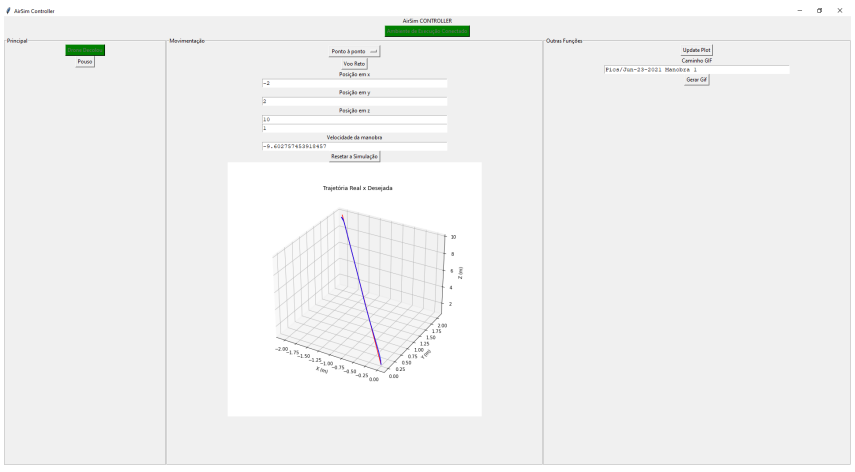


Figura 24 – Versão de desenvolvimento da interface gráfica de usuário feita com o auxílio da biblioteca *Tkinter*.

Para garantir o funcionamento da solução, a interface gráfica contou com a inclusão de rotinas de retorno que são ativadas com diferentes tipos de interação como pressionar botões ou alterar o tipo de movimentação via uma lista suspensa. Essas sub-rotinas interagem de maneira direta ou indireta com outras duas partes da ferramenta: as *Threads* e as Outras Rotinas. Ambas são apresentadas em mais detalhes a seguir.

3.2.2.2 Threads

Visando melhorar o desempenho do programa e permitir uma interação em tempo real entre o ambiente simulacional e o usuário do programa, foi necessário se desenvolver um sistema de fios de execução, ou *threads*. Esse sistema se mostrou também crucial para permitir o funcionamento da interface gráfica por algumas limitações da biblioteca utilizada. De maneira geral, o *Tkinter* cria uma operação que é executada continuamente enquanto a interface se mantém aberta. Isso ocorre para permitir que todas as ações executadas por um usuário na aplicação sejam tratadas e as respectivas funções de retorno sejam acionadas. Assim, a execução de uma pausa em decorrência direta de uma ação do usuário pararia toda e qualquer sub-rotina auxiliar dado que todas as rotinas estariam contidas no mesmo processo, fazendo que, por exemplo, a aeronave deixe de atualizar sua posição ao se mover.

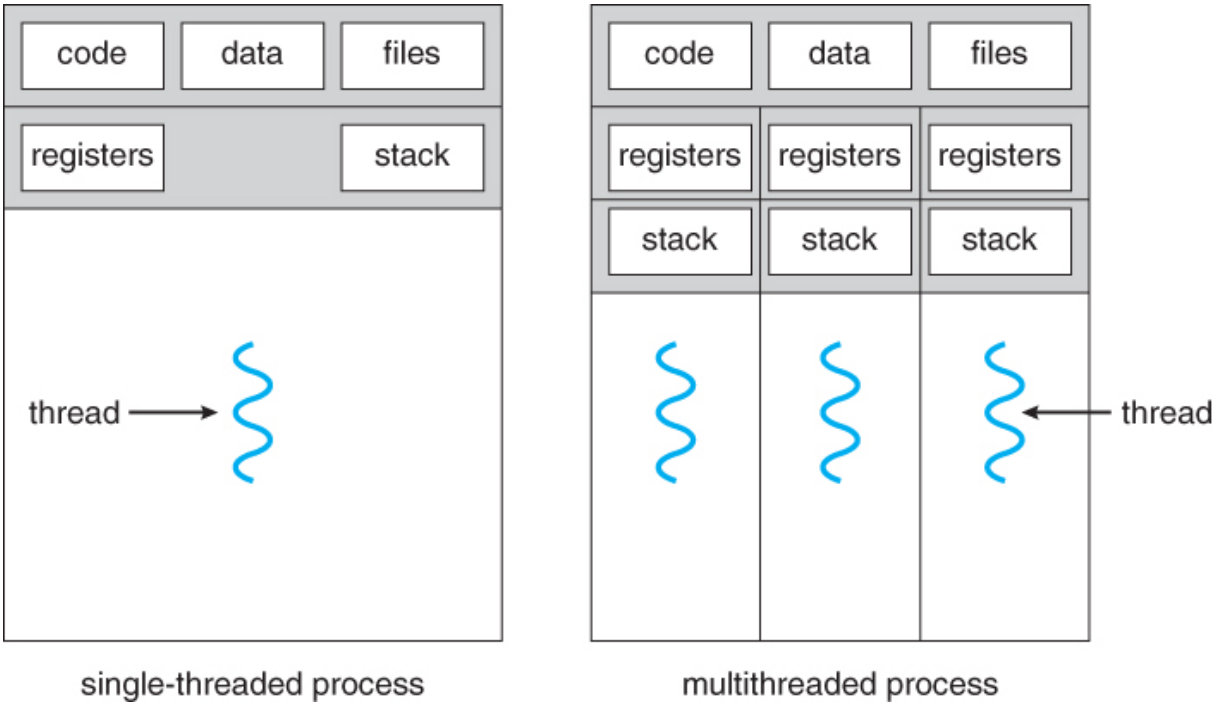


Figura 25 – Exemplificação de processos com um único fio de execução (*thread*) e com múltiplos. Retirado de (BELL, 2006).

Para contornar esse problema, uma possibilidade é a criação de fios de execução. Cada *thread*, representa uma unidade de utilização de CPU e um único processo pode ter diferentes fios (BELL, 2006), como exemplificado na figura 25. Com isso, é possível

executar rotinas de maneira independente da rotina principal de atualização da interface gráfica. Para a construção dos fios, a biblioteca *threading* do Python foi utilizada. Com ela, foi possível criar três sequências de execução em paralelo à rotina principal. Para garantir o bom funcionamento do esquema, as *threads* ocorreriam em repetições constantes.

Entretanto, esses fios de execução não poderiam ser diretamente acionados pela GUI. Caso isso ocorresse, toda a execução poderia ser subitamente encerrada. Desta forma, optou-se por utilizar diferentes *flags* ou bandeiras. Essas bandeiras, indicam qual o estado atual do programa e dependendo de seus valores, as *threads* executam ações distintas. Assim, para que o usuário possa interagir com os múltiplos fios sendo executados simultaneamente, basta que a interface altere o valor das *flags*.

Assim, as sequências se caracterizam por:

1. Sequência *movManager*: responsável por escolher o perfil de movimento que o drone deve realizar ao pedido do usuário e executar a missão. Nessa rotina, o tempo para passar de um trecho à outro é controlado com base na velocidade de voo desejada;

2. Sequência *posUpdater*: responsável por manter atualizada informações de movimentação da aeronave, como posição e velocidade. Essa rotina também é responsável por captar as imagens da câmera e gravar dados de voo em um arquivo de texto;

3. Sequência *posSaver*: responsável por tratar as informações de movimentação para serem disponibilizadas no gráfico e transformar as imagens obtidas em arquivos trabalháveis. Aqui, as posições são salvas enquanto a velocidade for maior que um limite de $5 * 10^{-3}$ m/s.

O código por trás de cada um dos fios de execução está disponível no Apêndice B.

3.2.2.3 Outras Rotinas

Além dessas rotinas principais, outras funções auxiliares foram necessárias e podem ser acessadas tanto pela janela principal, quanto pelos fios de execução. Essas rotinas executam funções diversas como:

- Permitir o uso ou retirar a permissão para a utilização de determinados botões, limitando as ações do usuário conforme a configuração atual do drone;
- Comandar um determinado perfil de missão;
- Gerar animações;
- et. al.

As principais funções auxiliares estão presentes no Apêndice C para conferência das rotinas.

3.2.3 Arquitetura resultante

Definidos os perfis de missão que a aeronave poderia executar e realizado o trabalho de programação da GUI, dos fios de execução e das funções auxiliares, foi possível se chegar à arquitetura apresentada na figura 26. A figura detalha os 4 fios existentes, seus subcomponentes e suas principais funcionalidades. Por fim, a figura 27 mostra a GUI em sua forma final.

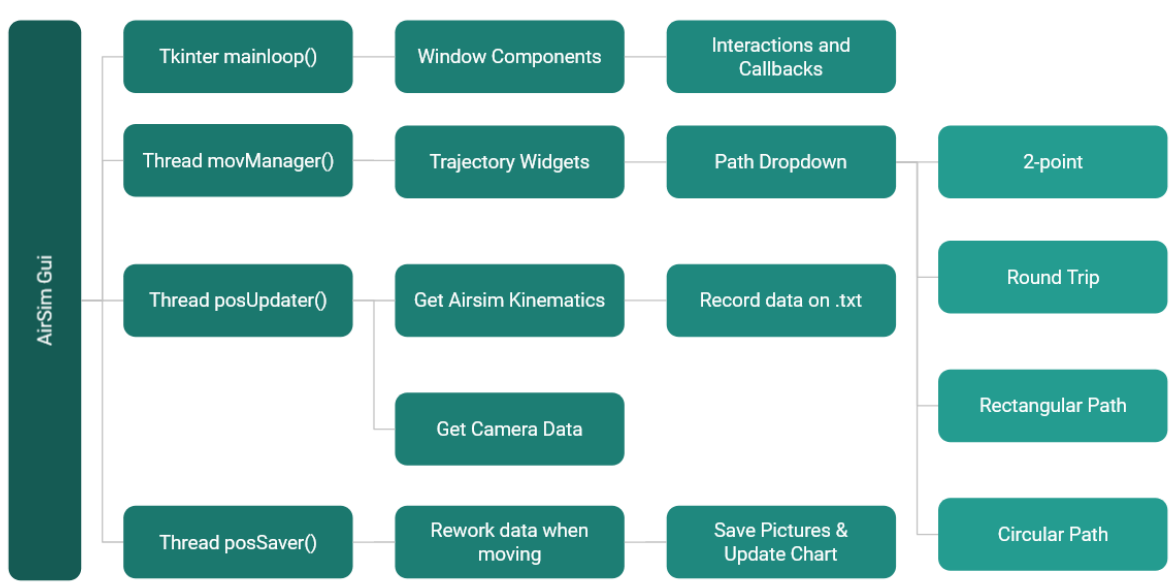


Figura 26 – Esquema simplificado da arquitetura atual da solução computacional para simulação com AirSim.

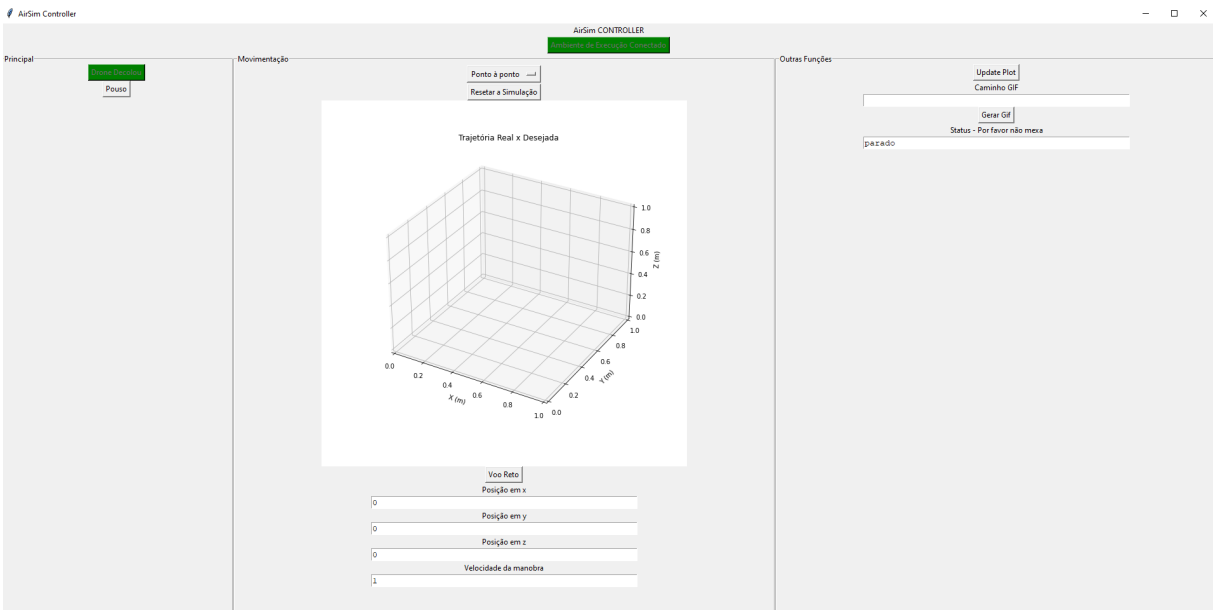


Figura 27 – GUI em seu estado final obtido no projeto.

4 RESULTADOS E DISCUSSÃO

Após a criação das rotinas computacionais necessárias para a simulação do Drone usando o Airsim, foi possível obter uma solução gráfica amigável e poderosa, capaz de realizar ações de diferentes níveis de complexidade com o software de simulação. Essa seção visa então abordar a execução simulações distintas, a trajetória executada e a obtenção de imagens utilizando o software.

4.1 Simulações

Para avaliar a capacidade da interface gráfica de dialogar com o *Airsim* e também a capacidade do software de lidar com os comandos de missão desejados, 6 perfis de validação foram definidos. Esses perfis estão presentes na Tabela 5.

Tabela 5 – Lista de missões enviadas via GUI para o simulador.

Missão	Detalhes
Ponto a ponto	- Origem em [0,0,0] - Final em [0,0,50] - Velocidade 25 m/s
Ponto a ponto	- Origem em [0,0,0] - Final em [0,4,10] - Velocidade 2 m/s
Ponto a ponto	- Origem em [0,0,0] - Final em [-2,2,10] - Velocidade 5 m/s
Circular	- Raio de 20m - Início em [0,0,50] - Velocidade 4 m/s - Discretização com 10 pontos
Retangular	- Comprimento 30m - Largura 20m - Início em [0,0,50] - Velocidade 10 m/s
Retangular	- Comprimento 30m - Largura 20m - Início em [0,0,50] - Velocidade 2 m/s

Com isso, foi possível utilizar a interface para gerar os comandos de movimentação e os resultados obtidos são discutidos na Seção 4.2.

4.2 Discussão

Os roteiros presentes na seção 4.1 deste relatório foram testados por meio da interface gráfica foram geradas imagens de câmera ao longo da movimentação da aeronave e também foram atualizados os gráficos. Destaca-se aqui que embora todas as trajetórias tenham tirado fotos, elas não serão todas apresentadas, pois os gráficos de movimento acabam apresentando mais informações do que estas imagens. Além disso, essa análise de resultados objetiva construir uma avaliação mais qualitativa da ferramenta e da GUI produzida, focando o estudo na capacidade da GUI de enviar comandos e estes serem reproduzidos pelo *Airsim*. Desta forma, não se discutirão os erros entre a posição desejada e a trajetória real, busca-se, agora, somente que a simulação responda aos comandos de missão enviados, mesmo que a trajetória não seja perfeitamente reta num voo do tipo *ponto à ponto*, retangular ou circular visto que outros fatores como clima, vento e altitude podem influenciar na simulação e não foram alvos do controle da rotina desenvolvida. Outro ponto que pode afetar as trajetórias é a forma de controle de passagem de uma posição à outra, como apresentada no capítulo 3.

A figura 28, apresenta as curvas obtidas para a primeira trajetória da tabela 5. Nela, é possível ver que no caso de uma movimentação vertical, as trajetórias real e desejada se confundem e se sobrepõem mesmo com uma velocidade elevada.

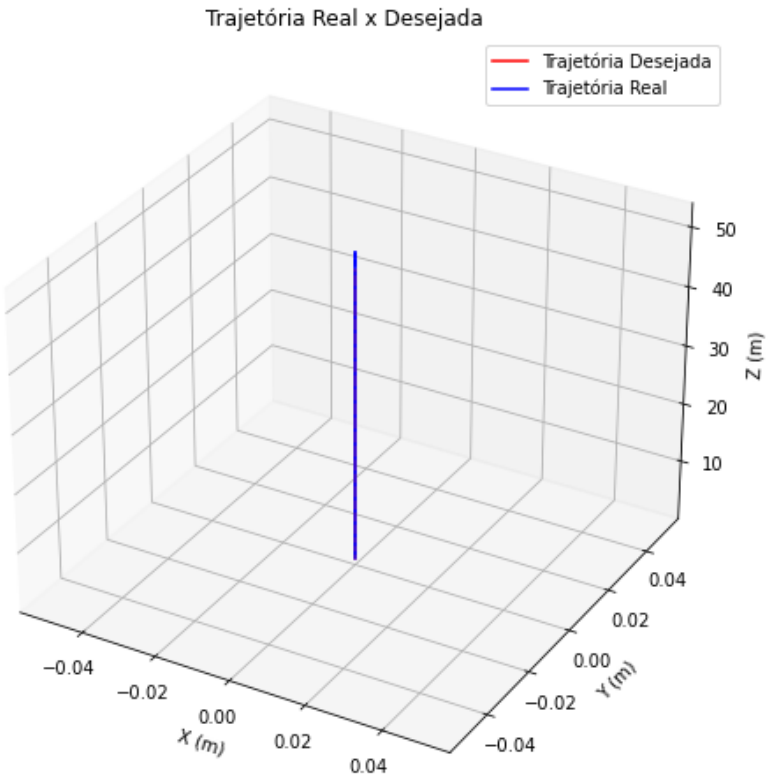


Figura 28 – Resultado para um voo partindo da posição inicial [0,0,0] indo até o ponto o [0,0,50].

Destaca-se ainda, que o gráfico não aparentou mostrar oscilações nos eixos x e y em decorrência desse movimento o que leva a indicar uma boa precisão da ferramenta para a realização de movimentos unidirecionais no eixo z. A figura 29, por sua vez, mostra que ao se adicionar movimentos no plano xy, pequenas oscilações (ordem de 10^{-7} são adicionadas, mesmo que o plano da aeronave seja se deslocar sob um único eixo. Como a trajetória enviada não possui etapas adicionais, esse comportamento oscilatório deve ser oriundo dos métodos e equações por trás do gerenciamento da física do *Airsim*. No entanto, como a ordem de grandeza é muito pequena comparada ao deslocamento, essas oscilações podem ser consideradas desprezíveis.

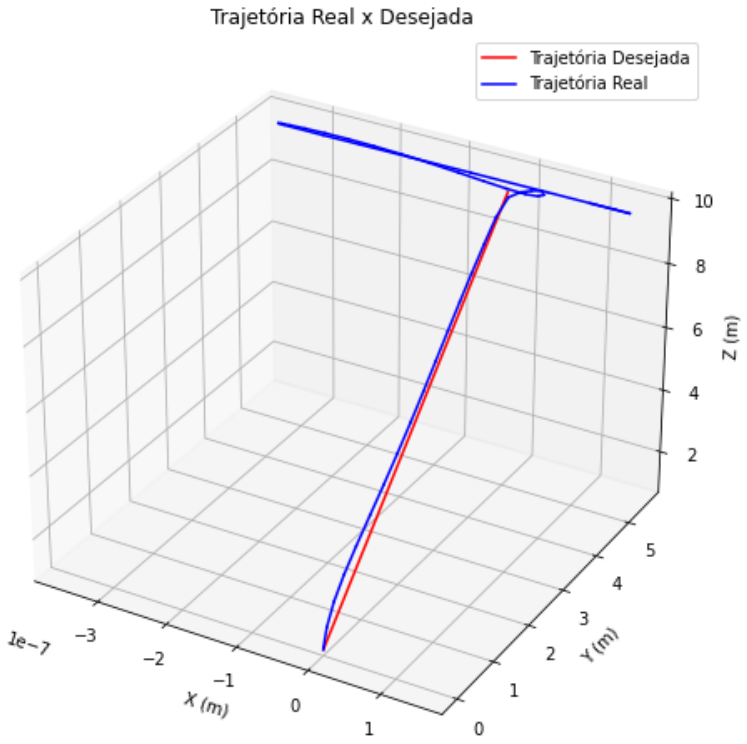


Figura 29 – Resultado para um voo partindo da posição inicial [0,0,0] para o ponto final [0,4,10].

Já no que diz respeito à figura 30, é possível verificar que para um comando de movimentação com componentes de deslocamento nos 3 eixos coordenados, as oscilações também se fazem presentes. É possível verificar visualmente no gráfico 2 zonas principais de diferenciação do código: o começo, onde a curva azul se distancia da curva desejada; e o final, onde as curvas voltam a se diferenciar após uma reaproximação no centro da trajetória desejada com as curvas novamente se confundindo, embora sejam esperadas algumas pequenas variações no meio, como foi visto no teste anterior.

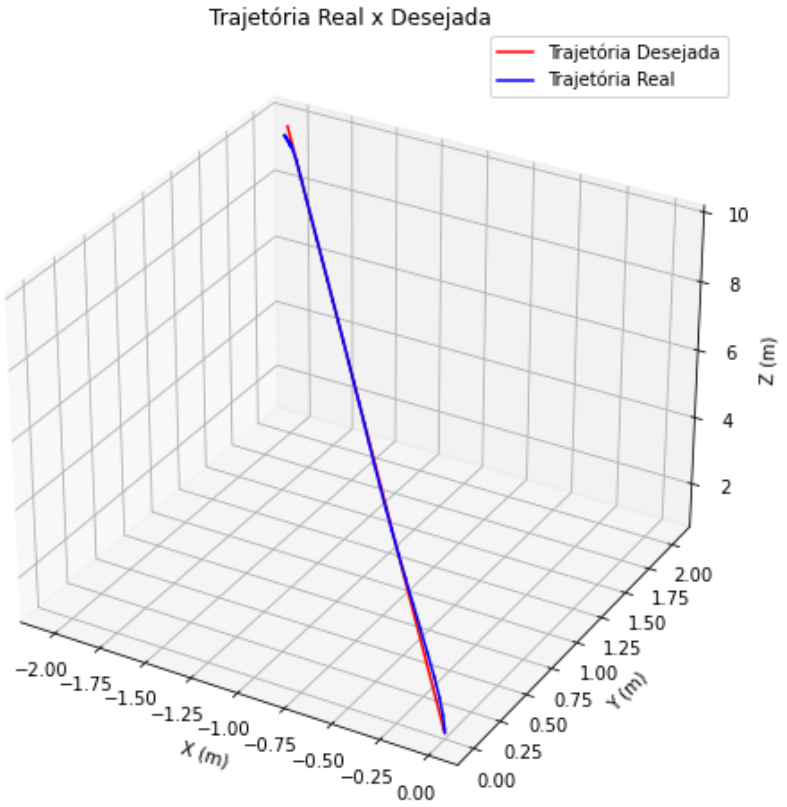
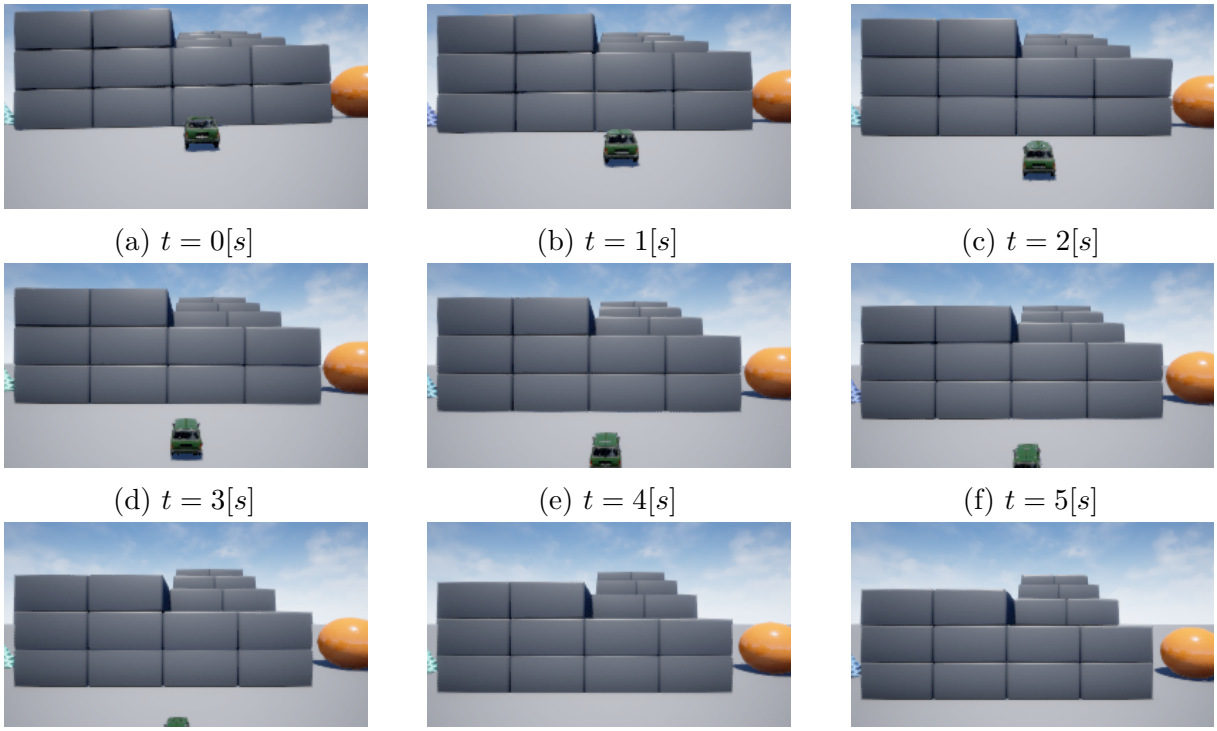


Figura 30 – Resultado para um voo partindo da posição inicial [0,0,0] para um vetor com inclinação de aproximadamente 8º com a vertical. O ponto final é o [-2,2,10].

Além disso, as sub-figuras 31a até 31i da figura 31, mostram como as imagens da câmera se comportaram ao longo da trajetória. Com elas, é possível ver, com o auxílio dos objetos disponíveis no ambiente da *Unreal Engine* que a aeronave vai gradualmente, subindo e se deslocando no plano, trazendo os blocos do fundo para a parte frontal da câmera, bem como aumentando a área de exposição da esfera laranja e fazendo com que o carro suma de vista da aeronave. Isso mostra que, caso um código de classificação de imagens seja implementado no *Airsim*, é de se esperar que a aeronave consiga alterar seu percurso ou enviar informações personalizadas em tempo real sobre a condição do ambiente ao qual ela está sobrevoando visando validar métodos e rotinas. Destaca-se ainda que estas imagens de câmera também foram geradas pelas outras missões, mas nem todas serão apresentadas neste documento por apresentar resultados menos informativos que os gráficos de comparação de trajetória.

Na sequência, a figura 32, conta com os resultados de uma trajetória *circular* enviada pela interface gráfica. Essa missão, realizada com os parâmetros da tabela 5. Embora esta missão apresente uma discretização na qual é possível perceber visualmente os limites do decágono enviado ao software, mostrou que o drone aproximou-se de uma trajetória mais homogênea.



(a) $t = 0[s]$ (b) $t = 1[s]$ (c) $t = 2[s]$

(d) $t = 3[s]$ (e) $t = 4[s]$ (f) $t = 5[s]$

(g) $t = 6[s]$ (h) $t = 7[s]$ (i) $t = 8[s]$

Figura 31 – Algumas imagens capturadas com o drone em movimento durante o caminho da trajetória descrita pela figura 30.

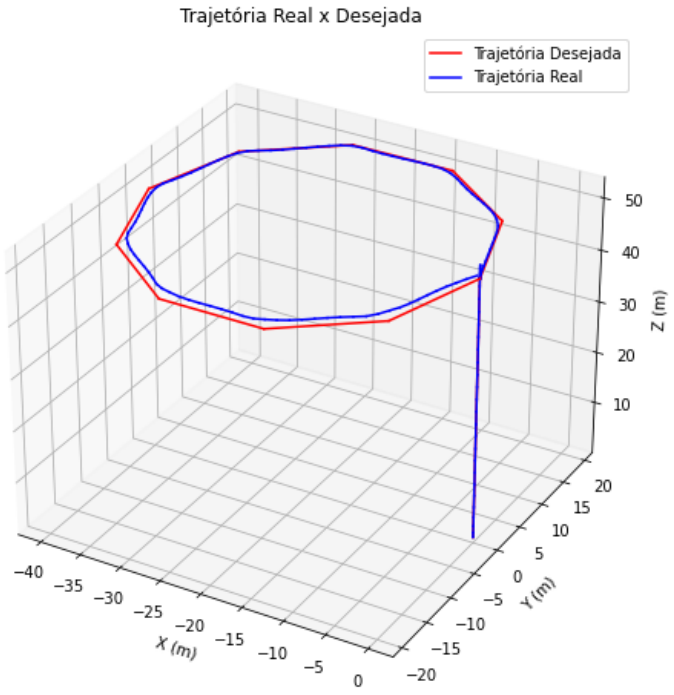


Figura 32 – Resultado de uma missão circular iniciada no ponto [0,0,50] para uma circunferência com 40 m de diâmetro.

Ademais, é importante mencionar que conforme o drone foi percorrendo a trajetória, o movimento executado foi se distanciando cada vez mais do inicialmente previsto. Isso está provavelmente ligado com o sistema de controle da *thread* movManager que utiliza o tempo calculado entre dois lados para enviar o próximo comando. Considerando a velocidade da manobra e o tamanho da circunferência, é possível inferir que a aeronave esteja iniciando a manobra seguinte sem de fato ter conseguido chegar à posição final do trecho e conforme avança em sua missão, esses erros vão se somando até ficarem cada vez mais perceptíveis.

Esse mesmo problema é encontrado nas trajetórias retangulares obtidas e mostradas nas figuras 33 e 34. Embora, ambas as trajetórias constituam os mesmos retângulos partindo do ponto inicial [0, 0, 50], o primeiro movimento é feito com uma velocidade desejada maior. Na condição de maior velocidade, o retângulo obtido apresenta uma série de deformações que descaracterizam o movimento. É possível perceber no trecho inicial do movimento que o veículo inicia seu voo com um certo distanciamento do lado do retângulo ideal e quando ele está se aproximando de sua trajetória já inicia a próxima etapa. Nos outros lados do retângulo o efeito se repete fazendo com que a aeronave se afaste cada vez mais. É notável aqui a influência do tempo de espera entre os comandos de manobras, mostrando que a simples relação $t = \frac{v}{\Delta S}$ não se mostra a mais adequada.

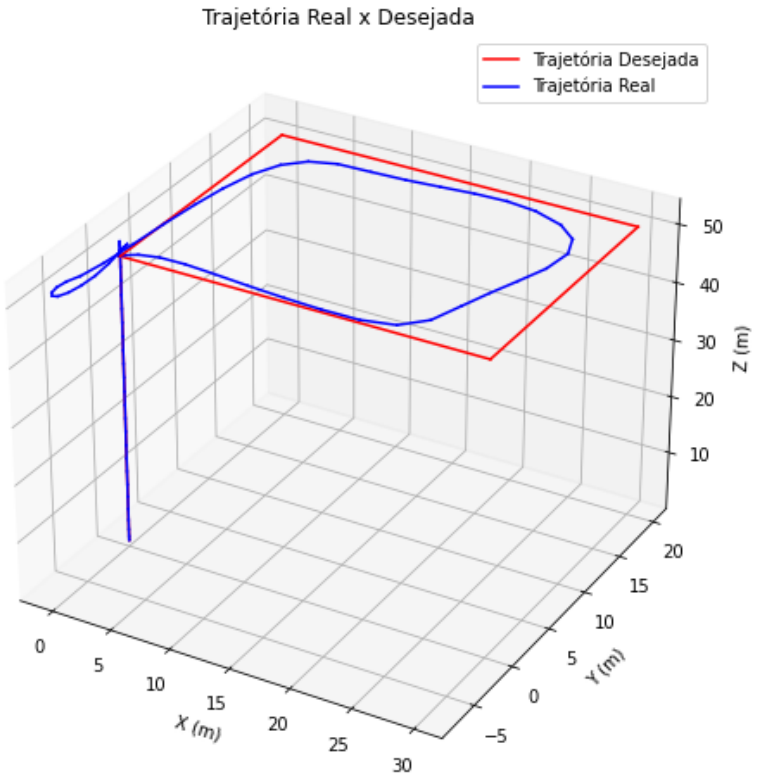


Figura 33 – Resultado para um voo partindo da posição inicial [0,0,50] para um retângulo com 30 m de comprimento e 20 m de largura. Essa trajetória foi obtida usando uma velocidade de 10 m/s.

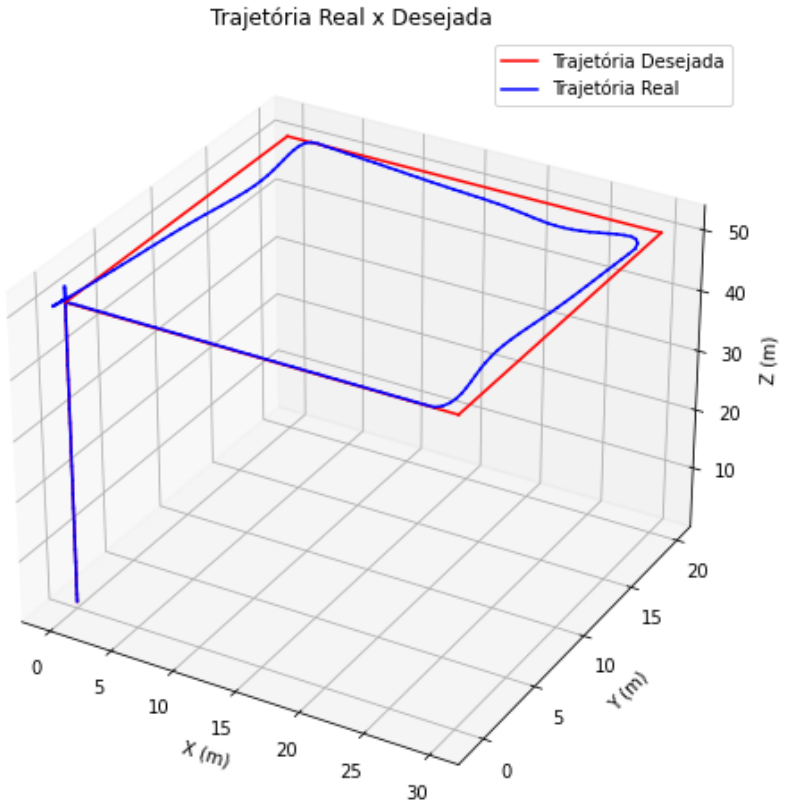


Figura 34 – Resultado para um voo partindo da posição inicial [0,0,50] para um retângulo com 30 m de comprimento e 20 m de largura. Essa trajetória foi obtida usando uma velocidade de 2 m/s.

Ainda sobre a trajetória da figura 33, é possível ver que no final, a aeronave passa do ponto inicial, errando por quase 5 metros no eixo y até iniciar uma manobra de retorno. Já na figura 34, é possível ver que a redução de velocidade permitiu ao drone seguir a trajetória de maneira mais realista, se afastando do trajeto inicial somente no fim do primeiro lado do retângulo, chegando a voltar a estar em contato na terceira parte da trajetória. Neste trecho, a aeronave também demonstrou um menor desvio no final, voltando à posição inicial de maneira mais direta, mesmo que o controle por tempo de manobra também tenha mostrado os seus problemas.

Assim, foi possível obter com a interface gráfica e o *Airsim*, uma série de trajetórias que se aproximaram dos perfis de missão desejados e passados pelo usuário para o software, mostrando que a aplicação desenvolvida se encontra funcional com capacidade de geração de gráficos, registros de posicionamento e também captura de imagens e animações. Destaca-se, porém, que algumas modificações ainda poderiam ser realizadas para melhorar a reprodução da trajetória desejada e incrementar as funções disponíveis.

5 CONCLUSÃO

Com este trabalho foi possível fazer uma avaliação de diferentes ferramentas computacionais para a criação de simulações com veículos aéreos. Esse levantamento, auxiliou na compreensão de que nem sempre, as ferramentas mais tradicionais são as mais adequadas, é necessário se compreender os objetivos que se buscam atingir e qual público irá estar em contato com o produto final. Nesse sentido, *Airsim* e *Python* se mostraram os meios ideais para a realização deste projeto de conclusão de curso.

Utilizando estes recursos, foi possível construir uma Interface Gráfica de Usuário capaz de permitir à usuários com diferentes níveis de conhecimento uma interação com a ferramenta e controlar quais missões a aeronave irá realizar e extrair dados da ferramenta, gerando imagens e animações. A ferramenta serviu também para mostrar que o *Airsim* em conjunto com a *Unreal Engine* possui enorme potencial para aplicações mais complexas, mesmo que as trajetórias divirjam um pouco daquilo que foi enviado, o que é um comportamento normal, considerando a complexidade do sistema. Este trabalho poderá servir de base para desenvolvimentos futuros e algumas sugestões de vias de melhoria são expressas aqui nessa conclusão.

Assim, alguns pontos de interesse foram levantados para que a solução aqui desenvolvida seja complementada em trabalhos futuros de graduação ou pós-graduação.

1. Criação de rotinas ligadas à classificação de imagens. Considerando que a ferramenta já pode realizar alguns tratamentos básicos de imagem, salvando figuras e criando animações, uma atualização futura que contemple além dessas ferramentas, uma análise em tempo real da imagem para dizer, por exemplo, quantos carros estão presentes na tela ou criar rotas complexas utilizando aprendizado de máquina para seguir padrões definidos. Além disso, pode ser interessante trabalhar com novos sensores.

2. Avaliação das diferenças de posição obtidas entre o desejado e o simulado. Considerando que as trajetórias podem ter oscilações e variações com aquilo que era desejado, seria muito importante implementar meios de validar a obtenção do perfil de missão desejado por uma análise de erros. Também seria interessante rever os critérios de definição de fim de etapa para tornar a simulação ainda mais coerente com o perfil de interesse.

3. Atualização da interface gráfica e das funções para permitir uma gama ainda maior de comandos por parte do operador. A melhoria da parte computacional também

pode ser feita visando otimizar os tempos de execução do código e a capacidade de processamento necessárias.

4. Implementação de melhorias no ambiente da Unreal Engine. O ambiente da *Unreal* possui um grande potencial de personalização. Novos carros podem ser incluídos, assim como casas, ruas e semáforos. Além disso, cada um desses elementos pode ter acionamentos e movimentos independentes, i.e., os carros podem se movimentar nas ruas e os semáforos podem alterar sua sinalização. Assim, uma interessante via de melhoria seria a criação de mapas mais fidedignos às aplicações de drone que se deseje avaliar. Também pode ser interessante testar o comportamento do sistema com variações climáticas e ciclos de dia e noite.

5. Aprimorar a quantidade de rotinas de "usabilidade". É importante prevenir possíveis erros de usuário como, por exemplo, enviar uma posição inatingível ou utilizar entradas incorretas nos campos. Essas alterações poderão permitir uma usabilidade com menos erros.

REFERÊNCIAS

AGGARWAL, S. et al. Smart drone. 2018. Atividade Acadêmica da Universidade da Califórnia São Diego.

ATTACHE, A. **Plane Crash Simulation in Unreal Engine 5**. 2021. Acesso: 12/06/2021. Disponível em: <https://www.youtube.com/watch?app=desktop&v=h_DZRBMq81w&feature=youtu.be>.

AWS. **Robomaker**. 2021. Acesso: 12/06/2021. Disponível em: <https://aws.amazon.com/pt/robomaker>.

BELL, J. **University Illinois Chicago- Course Notes Operating Systems - Threads**. 2006. Acesso: 24/05/2021. Disponível em: <https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html>.

BHUSHAN, N. Uav: Trajectory generation andsimulation. 2019. Thesis Substitute Project - Universidade do Texas Arlington.

CNI. **Emprego: indústria foi o setor que mais abriu vagas formais em 2020**. 2021. Acesso: 24/05/2021. Disponível em: <https://noticias.portaldaindustria.com.br/noticias/economia/emprego-industria-foi-o-setor-que-mais-abriu-vagas-formais-em-2020/>.

EDP. **EDP é a primeira empresa do setor elétrico certificada pela ANAC para monitoramento de redes com uso de drones**. 2021. Acesso: 14/06/2021. Disponível em: <https://bit.ly/2TavglE>.

ERBOZ, G. How to define industry 4.0: The main pillars of industry 4.0. **Managerial Trends Managerial trends in the development of enterprises in globalization era**, 2017. P. 761-767.

FANGOHR, H. **A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering.** 2004. Publicado em: In: Bubak M., van Albada G.D., Sloot P.M.A., Dongarra J. (eds) Computational Science - ICCS 2004. ICCS 2004. Lecture Notes in Computer Science, vol 3039. Springer, Berlin, Heidelberg.

G1. **PIB do Brasil despenca 4,1% em 2020**. 2021. Acesso: 24/05/2021. Disponível em: <https://g1.globo.com/economia/noticia/2021/03/03/pib-do-brasil-despenca-41percent-em-2020.ghtml>.

GAZEBO. **GAZEBOSIM**. 2021. Acesso: 27/05/2021. Disponível em: <https://www.gazebosim.org/>.

GOOGLE. **Google Finance**. 2021. Acesso: 13/05/2021. Disponível em: <https://www.google.com/intl/pt-BR/googlefinance>.

_____. **Google Trends**. 2021. Acesso: 13/05/2021. Disponível em: <https://trends.google.com.br/trends/>.

HERMANN, M.; PENTEK, T.; OTTO, B. Design principles for industrie 4.0 scenarios. **2016 49th Hawaii international conference on system sciences (HICSS). IEEE**, 2016. P. 3928-3937.

MADAAN, R. et al. Airsim drone racing lab. **Proceedings of Machine Learning Research**, 2020. NeurIPS 2019 Competition and Demonstration Track.

MATHWORKS. **MATLAB**. 2021. Acesso:30/06/2021. Disponível em: <https://www.mathworks.com/products/matlab/>.

MERTENS, J. Generating data to train a deep neuralnetwork end-to-end within a simulated environment. 2018. Master thesis at Department of Mathematics and ComputerscienceIntelligent Systems and Robotic Labs - Universidade Livre de Berlim.

MEYER, J. et al. Comprehensive simulation of quadrotor uavsusing ros and gazebo. **Conference: Proceedings of the Third international conference on Simulation, Modeling, and Programming for Autonomous Robots**, 2012.

MICROCHIP. **MPlab**. 2021. Acesso: 27/05/2021. Disponível em: <https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-x-ide>.

MICROSOFT. **Airsim Documentation**. 2021. Acesso: 31/05/2021. Disponível em: <https://microsoft.github.io/AirSim/index.html>.

MUNDIAL, B. **Databank**. 2021. Acesso: 24/05/2021. Disponível em: <https://databank.worldbank.org/home>.

OSRF. **ROS**. 2021. Acesso: 27/05/2021. Disponível em: <https://www.ros.org/>.

PSF. **Python**. 2021. Acesso: 20/05/2021. Disponível em: <https://www.python.org/>.

PYPI. **Python Package Index**. 2021. Acesso: 27/05/2021. Disponível em: <https://www.pypi.org/>.

RITCHIE, D. M. The development of the c language. **Second History of Programming Languages conference**, 1993.

SHAH, S. et al. Airsim: High-fidelity visual and physicalsimulation for autonomous vehicles. **Field and Service Robotics conference**, 2017.

SILVEIRA, C. B. **Indústria 4.0: O que é, e como ela vai impactar o mundo.** 2017. Acesso: 24/05/2021. Disponível em: <https://www.citisystems.com.br/industria-4-0/>.

WIRTH, M.; KOKVESI, P. Matlab as an introductory programming language. **Computer Applications in Engineering Education**, 2006.

Appendices

APÊNDICE A – CÓDIGO GERADOR DA INTERFACE GRÁFICA

Essa seção trata da construção da interface gráfica em si, utilizando a biblioteca *Tkinter*. Abaixo, o código detalhado e comentado para a geração da interface final.

```
# Informações genéricas
window = tk.Tk()
window.geometry("1920x1080")
window.protocol("WM_DELETE_WINDOW", closeWindow)
window.title('AirSim Controller')

# Criando o arquivo de log
day = datetime.datetime.now()
day_hour = day.strftime("%b-%d-%Y %H_%M_%S")
try:
    os.mkdir('Arquivos Txt')
except Exception:
    pass
f = open('Arquivos Txt/'+ day_hour + '.txt', "x")

# Definição de threads
thread = threading.Thread(target=posUpdater, name="Position Updater")
threadchart = threading.Thread(target=posSaver, name="Position Saver")
threadmov = threading.Thread(target=movManager, name="Gerente de Movimentação")

# Interface inicial - conexão da aeronave
greeting = tk.Label(text="AirSim CONTROLLER")
greeting.pack()
b0 = tk.Button(window, text ="Conectar a aeronave", command = connect)
b0.pack()
to = 0

# Criação do primeiro subgrupo - pouso e decolagem
labelframe = tk.LabelFrame(window, text="Principal")
b1 = tk.Button(labelframe, text="Decolagem", command = takeoff)
b1.pack()
b10 = tk.Button(labelframe, text="Pouso", command = landing)
b10.pack()
```

```
# Criação do espaço de movimentação na Gui
labelframe2 = tk.LabelFrame(window, text="Movimentação")

tkvar = tk.StringVar(window)
tkvar2 = tk.StringVar(window)

# Criação da lista de movimentos
mov_types = { 'Ponto à ponto', 'Bate e Volta','Voo Retangular','Voo em Círculo'}
tkvar.set('Ponto à ponto') # set the default option
tkvar.trace('w', chooseMov)
popupMenu = tk.OptionMenu(labelframe2, tkvar, *mov_types)
popupMenu.pack()

# Botão de Reset
b3 = tk.Button(labelframe2, text="Resetar a Simulação", command = reset)
b3.pack()

# Espaço para o gráfico de movimentação
fig = plt.Figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
ax.grid(True)
ax.set(title = "Trajetória Real x Desejada",
        xlabel = "X (m)",
  ylabel = "Y (m)",
  zlabel = "Z (m)")
plt.show()
canvas = FigureCanvasTkAgg(fig, master=labelframe2)
canvas.draw()
canvas.get_tk_widget().pack()

# Criação do botão de voo
b2 = tk.Button(labelframe2, text="Voo Reto", command = fly)
b2.pack()

# Caixas de texto para as manobras ponto à ponto e bate e volta
pxt = tk.Text(labelframe2, height = 1, width = 52)
lpx = tk.Label(labelframe2, text = "Posição em x")
```

```
pyt = tk.Text(labelframe2, height = 1, width = 52)
pxt.insert('end-1c','0')
lpy = tk.Label(labelframe2, text = "Posição em y")
pzt = tk.Text(labelframe2, height = 1, width = 52)
pyt.insert('end-1c','0')
lpz = tk.Label(labelframe2, text = "Posição em z")
tt = tk.Text(labelframe2, height = 1, width = 52)
pzt.insert('end-1c','0')
pvt = tk.Label(labelframe2, text = "Velocidade da manobra")
tt.insert('end-1c','1')
lpx.pack()
pxt.pack()
lpy.pack()
pyt.pack()
lpz.pack()
pzt.pack()
pvt.pack()
tt.pack()

# Caixas de texto para a manobra retangular
llr = tk.Label(labelframe2, text = "Largura do Retângulo")
tlr = tk.Text(labelframe2, height = 1, width = 52)
tlr.insert('end-1c','0')
lcr = tk.Label(labelframe2, text = "Comprimento do Retângulo")
tcr = tk.Text(labelframe2, height = 1, width = 52)
tcr.insert('end-1c','0')

# Caixas de texto para a manobra circular
tcr.insert('end-1c','1')
lr = tk.Label(labelframe2, text = "Raio do Círculo")
tr = tk.Text(labelframe2, height = 1, width = 52)
tr.insert('end-1c','0')


# Criação do espaço para funções auxiliares
labelframe3 = tk.LabelFrame(window, text="Outras Funções")

# Botão de atualizar gráfico
b4 = tk.Button(labelframe3, text="Update Plot", command = updateChart)
```

```
b4.pack()

# Criação da interface para geração de animação
gt = tk.Text(labelframe3, height = 1, width = 52)
#gt.config(state='disabled')
lg = tk.Label(labelframe3, text = "Caminho GIF")
lg.pack()
gt.pack()
b4 = tk.Button(labelframe3, text="Gerar Gif", command = gifSaver)
b4.pack()

# Interface para flag do movManager
stat = tk.Text(labelframe3, height = 1, width = 52)
lstat = tk.Label(labelframe3, text = "Status - Por favor não mexa")
lstat.pack()
stat.pack()
stat.insert('end-1c','parado')
stat.config(state = 'disabled')

# Para salvar posições
pxt2 = tk.Text(labelframe2, height = 1, width = 52)
lpx2 = tk.Label(labelframe2, text = "Última Posição em x")
pyt2 = tk.Text(labelframe2, height = 1, width = 52)
pxt2.insert('end-1c','0')
lpy2 = tk.Label(labelframe2, text = "Última Posição em y")
pzt2 = tk.Text(labelframe2, height = 1, width = 52)
pyt2.insert('end-1c','0')
lpz2 = tk.Label(labelframe2, text = "Última Posição em z")
pzt2.insert('end-1c','0')

# Criação do loop do tkinter
window.mainloop()
```

APÊNDICE B – CÓDIGO DAS THREADS

Visando facilitar a compreensão das rotinas computacionais utilizadas, essa seção traz os códigos desenvolvidos para a criação das *threads* mencionadas no capítulo 3. Os códigos das *threads* são aqui apresentados em ordem alfabética do nome escolhido e não pela ordem de início por uma mera convenção organizacional.

```
"""

Thread movManager - Coordena qual tipo de comando de movimentação
será passado à aeronave. Trabalha com dados de posição gerados pela posUpdater
e por meio de uma "flag" chamada "currStat", sabe quando a aeronave deve
voar e qual movimento executar

"""

def movManager():
    global posplot # Variável global para a criação de gráficos
    while(True):
        # Definição da possição e status atuais
        currStat = stat.get("1.0",'end-1c')
        currPos = pos2
        posplot = currPos

        # Checagem do status
        if currStat == 'voando':

            # Verificação de qual tipo de movimentação foi escolhida
            currMov = tkvar.get()
            if currMov == 'Ponto à ponto':
                flyToPosition()
            elif currMov == 'Bate e Volta':
                flyBateVolta(currPos)
            elif currMov == 'Voo Retangular':
                flySquare(currPos)
            else:
                flyCircle(currPos)
            # Atualização do status após execução
```

```
            stat.config(state = 'normal')
            stat.delete(1.0,'end-1c')
            stat.insert('end-1c','parado')
            stat.config(state = 'disabled')
        time.sleep(0.1)

        # Encerramento
        if kill3:
            break


"""

Thread posUpdater - Roda paralelamente ao código principal. Serve para retirar
informações importantes do programa em tempo real como posição, velocidade
e também serve para criar a variável que trabalha com a câmera

"""

def posUpdater():
    # Definição de variáveis globais que serão acessadas
    # externamente
     global pos
     global pos2
     global pos3
     global vel
     global responses
     while(True):
          # getKinematics --> Obtém informações do movimento
         pos = client.simGetGroundTruthKinematics().position
         pos2 = pos
         vel = client.simGetGroundTruthKinematics().linear_velocity
         # Salva as informações no log
         f.write('Posição [X, Y, Z]: [' + str(pos.x_val) + ', ' + str(pos.y_val) \
         + ', ' + str(pos.z_val) + '] \n' )
          # getImages --> Captura imagens da camera
         responses = client.simGetImages([
        airsim.ImageRequest("1", airsim.ImageType.Scene, False, False) #PNG
])
         time.sleep(0.1)
```

```
            # Encerramento
            if kill:
                break


"""

Thread posSaver - Roda paralelamente ao código principal. Trabalha com as in-
formações da posUpdater

"""

def posSaver():
    # Definição de variáveis globais
    global responses

    # Definição de flags e variáveis de apoio
    cap = 0
    pos_real = np.empty((0,3), float)
    flag0 = 0
    flag1 = 0
    lastpos = []

    # Salva foto
    takePicture(flag1)

    while(True):
        # Atualização da posição atual
        posplot = pos2
        lastpos = [posplot.x_val, posplot.y_val, -posplot.z_val]
        while(True):
            posplot = pos
            velplot = vel
            velcalc = np.sqrt(vel.x_val**2 + vel.y_val**2 + vel.z_val**2)

            # Verificação se a velocidade está abaixo de um determinado
            # limite. Caso esteja acima, assume-se movimento

            if velcalc <= 5*10**-3:
```

```
            time.sleep(0.1)
            if flag0 == 1:
                flag0 = 0
                flag1 = flag1 + 1
        else:
            append = [posplot.x_val, posplot.y_val, -posplot.z_val]
            print(append)

            # Plot e atualização da última posição
            ax.plot([lastpos[0], append[0]],[lastpos[1], append[1]],  \
            [lastpos[2], append[2]], 'b', label = 'Posição Real')
            lastpos = [posplot.x_val, posplot.y_val, -posplot.z_val]
            print('Velocidade Calculada')
            print(velcalc)
            time.sleep(0.1)

            # Verificação do tempo para tirar novas fotos
            if cap % 10 == 0:
                takePicture(flag1)
                flag0 = 1
                #  updateChart()
                print('Tirando Foto')
            # Atualização do contador de foto para que ocorra uma vez por
            # segundo apenas.
        cap = cap+1
            # takePicture()

    if kill2:
     break
```

APÊNDICE C – CÓDIGO DAS FUNÇÕES PRINCIPAIS

Complementando as informações dos Apêndices A e B, essa seção traz os códigos desenvolvidos para a criação das *Funções Principais* mencionadas no capítulo 3. Novamente, os códigos estão organizados em ordem alfabética por simples convenção organizacional.

```
"""

Rotina build - Serve para atualizar os status dos botões e os widgets
presentes na GUI após a escolha do tipo de movimento. pack() faz os widgets
aparecerem enquanto pack_forget() os faz sumir.

"""

def build(mov):
    if mov == 'Ponto à ponto':
        b2['text'] = 'Ponto à ponto'
        llr.pack_forget()
        tlr.pack_forget()
        lcr.pack_forget()
        tcr.pack_forget()
        pvt.pack_forget()
        tt.pack_forget()
        lr.pack_forget()
        tr.pack_forget()
        lpx.pack()
        pxt.pack()
        lpy.pack()
        pyt.pack()
        lpz.pack()
        pzt.pack()
        pvt.pack()
        tt.pack()

    elif mov == 'Bate e Volta':
        b2['text'] = 'Bate e Volta'
        llr.pack_forget()
        tlr.pack_forget()
```

```
        lcr.pack_forget()
        tcr.pack_forget()
        pvt.pack_forget()
        tt.pack_forget()
        lr.pack_forget()
        tr.pack_forget()
        lpx.pack()
        pxt.pack()
        lpy.pack()
        pyt.pack()
        lpz.pack()
        pzt.pack()
        pvt.pack()
        tt.pack()

    elif mov == 'Voo Retangular':
        b2['text'] = 'Voo Retangular'
        lpx.pack_forget()
        pxt.pack_forget()
        lpy.pack_forget()
        pyt.pack_forget()
        lpz.pack_forget()
        pzt.pack_forget()
        pvt.pack_forget()
        tt.pack_forget()
        lr.pack_forget()
        tr.pack_forget()
        llr.pack()
        tlr.pack()
        lcr.pack()
        tcr.pack()
        pvt.pack()
        tt.pack()

    else:
        b2['text'] = 'Voo em Circulo'
        lpx.pack_forget()
        pxt.pack_forget()
        lpy.pack_forget()
```

```
        pyt.pack_forget()
        lpz.pack_forget()
        pzt.pack_forget()
        pvt.pack_forget()
        tt.pack_forget()
        llr.pack_forget()
        tlr.pack_forget()
        lcr.pack_forget()
        tcr.pack_forget()
        lr.pack()
        tr.pack()
        pvt.pack()
        tt.pack()


"""

Rotina chooseMov - Apenas uma interface para chamar a função build


"""

def chooseMov(*args):
    print("Entrando no ChooseMov")
    print(tkvar.get())
    # Verifica a variável e chama a rotina build
    build(tkvar.get())

"""

Rotina cleanChart - Serve para apagar o gráfico ao pousar.

"""

def cleanChart():
    ax.clear()
    print('Gráfico Limpo')

"""
```

```
Rotina closeWindow - Define procedimentos para o fechamento da janela
da GUI. Os comandos kill = true servem para encerrar as Threads em
execução paralela.

"""

def closeWindow():
    #    if messagebox.askokcancel("Sair", "Você realmente deseja sair?"):
    try:
        # reset()
        global kill
        kill = True
        global kill2
        kill2 = True
        global kill3
        kill3 = True
        window.destroy()
    except:
        window.destroy()

"""

Rotina connect - Cria algumas variáveis globais caso não tenha
sido feito ainda. A rotina é responsável por se conectar ao cliente
do Airsim, permitir o controle deste pela API. Também é possível obter
algumas informações da aeronave como informações de GPS. Aqui se
inicia a Thread posUpdater

"""

def connect():
    try:
        global client
        global Running
        global kill
        global kill2
        global kill3
    except:
```

```
    pass

kill = False
kill2 = False
kill3 = False
Running = True

# Conexão
client = airsim.MultirotorClient()
client.confirmConnection()

# Controle da API
client.enableApiControl(True)
client.armDisarm(True)

# Levantamento de informações e print
state = client.getMultirotorState()
s = pprint.pformat(state)
print("state: %s" % s)

imu_data = client.getImuData()
s = pprint.pformat(imu_data)
print("imu_data: %s" % s)

barometer_data = client.getBarometerData()
s = pprint.pformat(barometer_data)
print("barometer_data: %s" % s)

magnetometer_data = client.getMagnetometerData()
s = pprint.pformat(magnetometer_data)
print("magnetometer_data: %s" % s)

gps_data = client.getGpsData()
s = pprint.pformat(gps_data)
print("gps_data: %s" % s)

# Alteração em tempo real do Layout da GUI
b0['text'] = 'Ambiente de Execução Conectado'
b0['bg'] = 'green'
```

```
    turnButton(b0)
    turnButton(b10)
    labelframe.pack(side = tk.LEFT, fill="both", expand="yes")
    try: # Tentativa de execução da thread posUpdater
        thread.start()
    except Exception:
        pass
    print('Thread Iniciada')
    time.sleep(2)

"""

Rotina fly - Serve para atualizar a flag da thread movManager.
A função também pode iniciar a thread caso seja sua primeira execução.

"""

def fly():
    # Atualização da flag
    stat.config(state = 'normal')
    stat.delete(1.0,'end-1c')
    stat.insert('end-1c','voando')
    stat.config(state = 'disabled')

    try: # Tentativa de execução da thread movManager
        threadmov.start()
    except Exception:
        pass

"""

Rotina flyBateVolta - Define as funções necessárias para realizar um voo
do tipo bate e volta, indo de uma posição inicial, até uma posição
intermediária e retornando.

"""


def flyBateVolta(position_current):
```

```
    # Define posições iniciais
    init_pos_bv = position_current
    init_x = init_pos_bv.x_val
    init_y = init_pos_bv.y_val
    init_z = init_pos_bv.z_val

    # Definição do ponto alvo
    mid_x = float(pxt.get("1.0",'end-1c'))
    mid_y = float(pyt.get("1.0",'end-1c'))
    mid_z = float(pzt.get("1.0",'end-1c'))
    positionmod = np.sqrt((mid_x-init_x)**2+(mid_y-init_y)**2+(mid_z-init_z)**2)
    vmod = float(tt.get("1.0",'end-1c'))
    t = positionmod/vmod

    # Voo até o alvo
    flyToPosition()
    time.sleep(0.01)
    time.sleep(t)

    # Atualização do ponto alvo como sendo a origem do movimento
    pxt.delete(1.0,'end-1c')
    pxt.insert('end-1c', str(init_x))
    pyt.delete(1.0,'end-1c')
    pyt.insert('end-1c',str(init_y))
    pzt.delete(1.0,'end-1c')
    pzt.insert('end-1c',str(-init_z))

    # Voo até a origem
    flyToPosition()
    time.sleep(t)

"""

Rotina flyCircle - Define as rotinas para a realização de um voo circular

"""

def flyCircle(starting_position, discret=10):
    vc = float(tt.get("1.0",'end-1c'))
```

```
    radius = float(tr.get('1.0','end-1c'))
    omega_c = vc/radius
    tc = 2*np.pi/omega_c
    tc_control = tc/discret
    xc = starting_position.x_val-radius
    yc = starting_position.y_val
    zc = starting_position.z_val
    xold = starting_position.x_val
    yold = yc
    zold = -zc
    for i in range(1, discret+1):
        t_use = tc_control*i
        angle = omega_c*t_use
        startPlot = [xold, yold, zold]
        x_now = xc + np.cos(angle)*radius
        y_now = yc + np.sin(angle)*radius
        client.moveToPositionAsync(x_now, y_now, zc, vc)
        endPoint = [x_now, y_now, zold]
        ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
        [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
        time.sleep(tc_control)
        xold = x_now
        yold = y_now
        print(tc_control)
        print("Voo circular")


"""

Rotina flyToPosition - Define as relações para que a aeronave possa voar de um
ponto à outro

"""

def flyToPosition():
    print("Andando em Frente...")
    labelframe2.pack(fill="both", expand="yes")
    init_pos = pos
    print(init_pos.x_val)
```

```
    print(init_pos.y_val)
    print(init_pos.z_val)
    trajectoryDesired(tkvar.get())


    try:
        client.moveToPositionAsync(float(pxt.get("1.0",'end-1c')),\
                                   float(pyt.get("1.0",'end-1c')),\
                                       -float(pzt.get("1.0",'end-1c')),\
                                           float(tt.get("1.0",'end-1c')))#.join()
        #takePicture()
        try:
            threadchart.start()
        except:
            pass
    except:
        tk.messagebox.showerror('Impossível chegar até a posição','Parece que \
                                você esqueceu que as posições precisam ser \
                                valores numéricos. Por favor corrija antes \
                                    de executar o código novamente.')

"""

Rotina flySquare - Define as relações para o voo retangular

"""


def flySquare(position_current):
    print("Voo retangular")
    comp = float(tcr.get("1.0",'end-1c'))
    larg = float(tlr.get("1.0",'end-1c'))
    curr_speed = float(tt.get("1.0",'end-1c'))
    init_pos_r = position_current
    init_xr = init_pos_r.x_val
    init_yr = init_pos_r.y_val
    init_zr = init_pos_r.z_val
    t1 = comp/curr_speed
    t3 = t1
```

```
    t2 = larg/curr_speed
    t4 = t2
    print(init_zr)
    client.moveToPositionAsync(init_xr+comp, init_yr, init_zr, curr_speed)
    startPlot = [init_xr, init_yr, -init_zr]
    endPoint = [init_xr + comp, init_yr, -init_zr]
    ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
    [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
    #time.sleep(0.01)
    time.sleep(t1)
    client.moveToPositionAsync(init_xr+comp, init_yr+larg, init_zr, curr_speed)
    startPlot = [init_xr + comp, init_yr, -init_zr]
    endPoint = [init_xr + comp, init_yr + larg, -init_zr]
    ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
    [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
    #time.sleep(0.01)
    time.sleep(t2)
    client.moveToPositionAsync(init_xr, init_yr+larg, init_zr, curr_speed)
    startPlot = [init_xr+comp, init_yr+ larg, -init_zr]
    endPoint = [init_xr, init_yr + larg, -init_zr]
    ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
    [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
    #time.sleep(0.01)
    time.sleep(t3)
    client.moveToPositionAsync(init_xr, init_yr, init_zr, curr_speed)
    startPlot = [init_xr, init_yr+ larg, -init_zr]
    endPoint = [init_xr, init_yr, -init_zr]
    ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
    [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
    #time.sleep(0.01)
    time.sleep(t4)



"""

Rotina gifSaver - Objetivo: Criar um gif com as imagens capturadas em
```

```
takePictures

"""

def gifSaver():
    png_dir = os.path.dirname(os.path.abspath(__file__)) + "/" + /
    gt.get("1.0",'end-1c')
    images = []
    for file_name in sorted(os.listdir(png_dir)):
        if file_name.endswith('.png'):
            file_path = os.path.join(png_dir, file_name)
            images.append(imageio.imread(file_path))
    imageio.mimsave(png_dir + '/video.gif', images)


"""

Rotina Landing - Função para pousar a aeronave e atualizar os botões.

"""

def landing():
    print("Landing...")
    client.landAsync()
    b1['text'] = 'Drone Pousou - Decole Novamente'
    b1['bg'] = 'red'
    turnButton(b10)
    turnButton(b1)
    cleanChart()
    #labelframe2.pack(side = tk.LEFT, fill="both", expand="yes")
    #labelframe3.pack(side = tk.LEFT, fill="both", expand="yes")



"""

Rotina reset - visa retornar o drone à sua posição inicial fazendo com que
o drone pouse.
```

```
"""

def reset():
    client.reset()
    b1['text'] = 'Drone de volta ao estado inicial'
    b1['bg'] = 'red'
    b0['text'] = 'Ambiente Desconectado'
    b0['bg'] = 'red'
    turnButton(b0)
    turnButton(b1)
    turnButton(b10)
    labelframe.pack_forget()
    labelframe2.pack_forget()
    cleanChart()
    try:
        client.landAsync()
    except:
        pass


"""

Rotina takeoff - Objetivo: Fazer com que o drone decole, liberar os widgets
escondidos do tkinter e atualizar o comportamento de botões

"""

def takeoff():
    print("Taking off...")
    client.takeoffAsync()
    #client.moveToPositionAsync(0, 0, -10, 2).join()
    b1['text'] = 'Drone Decolou'
    b1['bg'] = 'green'
    turnButton(b1)
    turnButton(b10)
    labelframe2.pack(side = tk.LEFT, fill="both", expand="yes")
    labelframe3.pack(side = tk.LEFT, fill="both", expand="yes")
```

```
"""

Rotina takePicture - Objetivo: Receber o arquivo da camera do Airsim, criar as
pastas corretas e salvar o arquivo em formato PNG

"""


def takePicture(flag):
#scene vision image in uncompressed RGBA array
    response = responses[0]
    folderNumb = str(flag)
    #print(response)
    pictime = datetime.datetime.now()
    pichour = pictime.strftime('%H_%M_%S')
    picname = pictime.strftime('%b-%d-%Y')
    try:
        os.mkdir('Pics')
    except Exception:
        pass
    try:
        os.mkdir('Pics/' + picname + ' Manobra ' + folderNumb)
    except Exception:
        pass
    gt.delete(1.0,'end-1c')
    gt.insert('end-1c','Pics/' + picname + ' Manobra ' + folderNumb)
    try:
        img1d = np.frombuffer(response.image_data_uint8, dtype=np.uint8)
        img_rgb = img1d.reshape(response.height, response.width, 3) array H X W X 3
        cv2.imwrite('Pics/' + picname + ' Manobra ' + folderNumb + '/' /
        + pichour + '.png', img_rgb)
        print('Imagem Feita em' + pichour)
    except Exception:
        return


"""

Rotina tracjetoryDesired - auxilia a geração de gráficos caso a trajetória seja
```

```
bate e volta ou ponto à ponto.

"""


def trajectoryDesired(mov):
    # plt.clf()
    if mov == 'Ponto à ponto' or mov == 'Bate e Volta':
        startPoint = pos
        startPlot = [float(startPoint.x_val), float(startPoint.y_val), /
        float(-startPoint.z_val)]
        endPoint = [float(pxt.get("1.0",'end-1c')), /
        float(pyt.get("1.0",'end-1c')), float(pzt.get("1.0",'end-1c'))]
        ax.plot([startPlot[0],endPoint[0]],[startPlot[1],endPoint[1]],/
        [startPlot[2],endPoint[2]], 'r', label = 'Posição Desejada')
    #else:
        #tk.messagebox.showinfo('Nenhuma trajetória encontrada', 'Nenhuma /
        trajetória encontrada')
    #updateChart()


"""

Rotina turnButton - serve para definir se um botão é editável ou não

"""


def turnButton(button_name):
    if button_name['state'] == 'normal':
        button_name['state'] = 'disabled'
    else:
        button_name['state'] = 'normal'


"""

Rotina updateChart - pode ser chamada por outras rotinas para atualizar
o gráfico após a movimentação.
```

```
"""

def updateChart():
    canvas.draw()
    ax.legend(['Trajetória Desejada', 'Trajetória Real'])
```