



Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Trabalho de Conclusão de Curso
Projeto Baseado em Modelo Aplicado no
Desenvolvimento do Sistema de Controle de Voo de um
Veículo Aéreo Não Tripulado

Débora Nunes Pinto de Oliveira

Campina Grande, Paraíba, Brasil

©Débora Nunes Pinto de Oliveira, 23 de outubro de 2020

Débora Nunes Pinto de Oliveira

Projeto Baseado em Modelo Aplicado no
Desenvolvimento do Sistema de Controle de Voo de um
Veículo Aéreo Não Tripulado

Trabalho de Conclusão de Curso de Bacharelado submetida à Coordenadoria de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Orientador: Prof. Antonio Marcus Nogueira Lima, Dr.

Campina Grande, Paraíba, Brasil

©Débora Nunes Pinto de Oliveira, 23 de outubro de 2020

Débora Nunes Pinto de Oliveira

Projeto Baseado em Modelo Aplicado no
Desenvolvimento do Sistema de Controle de Voo de um
Veículo Aéreo Não Tripulado

Trabalho de Conclusão de Curso de Bacharelado
submetida à Coordenadoria de Graduação em En-
genharia Elétrica da Universidade Federal de Cam-
pina Grande como parte dos requisitos necessários
para obtenção do grau de Bacharel em Ciências no
Domínio da Engenharia Elétrica.

Aprovado em ___/___/___

Prof. Antonio Marcus Nogueira Lima, Dr.

Orientador

Prof. Marcos Ricardo Alcântara Morais, Dr.

Avaliador

Campina Grande, Paraíba, Brasil

©Débora Nunes Pinto de Oliveira, 23 de outubro de 2020

Agradecimentos

Agradeço aos professores Antonio Marcus, Saulo Dornellas e Marcos Morais, pelo incentivo à pesquisa e incansável orientação e colaboração sem as quais não seria possível realizar este trabalho. Também agradeço aos colegas do laboratório Erobótica, em especial Davi, Yuri e Felipe, pelo acompanhamento, estímulo e auxílio na execução dos experimentos. Por fim, gratifico a Sarah e Augusto a acolhida no projeto.

Resumo

Esse projeto apresenta a documentação de um modelo educacional de um Veículos Aéreo Não Tripulado (VANT) comercial, cujas especificações do sistema embarcado são ocultas. Neste trabalho é proposto solucionar as abstrações dessa documentação fechada a partir da investigação dos protocolos de comunicação e da parametrização do sistema de controle de voo. Com o intuito aperfeiçoar a verossimilhança da simulação com o protótipo físico, foi proposto um novo modelo de fluxo óptico. Para uma trajetória de uma lemniscata, o erro médio entre as curvas de posição estimada pelo modelo proposto e do protótipo físico é menor do que entre esta última e a curva da planta original simulada. Dessa forma, a caracterização do sistema embarcado anteriormente fechado ampliou o espaço para otimização e customização do controle do dispositivo em estudo.

Palavras-chave: Modelo Aplicado em Desenvolvimento, Geração Automática de Código, Veículo Aéreo Não Tripulado, Sistema de Controle de Voo, Fluxo Óptico, *Hardware-in-the-loop*, *Software-in-the-loop*.

Abstract

This project presents a closed embedded system documentation of a commercial Unmanned Aerial Vehicle (UAV) educational model. The abstraction of the flight control system is surpassed through the investigation of the communication protocol of the drone. A new optical flow model is proposed to improve the similarities between the simulation model and the prototype. The standard deviation between the suggested model and the record acquired from the prototype for a lemniscate trajectory is smaller than in comparison with the original simulation model. Therefore, the optimization and customization space of the flight control system from the commercial UAV expands with the proposed model characterization.

Keywords: Model Based Design, Automatic Code Generation, Unmanned Aerial Vehicle, Flight control System, Optical Flow, hardware-in-the-loop, software-in-the-loop.

Sumário

Lista de símbolos e abreviaturas	v
Lista de Tabelas	viii
Lista de Figuras	ix
1 Introdução	1
1.1 Justificativa	1
1.2 Estudo de caso	3
1.3 Objetivo geral	6
1.4 Objetivos específicos	6
1.5 Organização do documento	6
2 Modelagem do quadricóptero	8
2.1 Dinâmica do quadricóptero	8
2.2 Modelo matemático do quadricóptero	9
2.2.1 Sistemas de coordenadas	9
2.2.2 Matriz de rotação	10
2.2.3 Matriz de transformação angular	11
2.2.4 Modelo dinâmico	13
2.2.5 Espaço de estados	18
2.3 Controle de um quadricóptero	18
2.3.1 Lei de controle	20
2.3.2 Estimador de estados	23
2.4 Características do quadricóptero em estudo	26

2.4.1	Componentes físicos do Parrot Mambo	27
2.5	Considerações finais	28
3	O ambiente de modelagem	29
3.1	Pacote de suporte a minidrones Parrot no Simulink	29
3.2	Diagrama de blocos do Parrot Mambo	29
3.2.1	Comandos de voo	33
3.2.2	Sistema de controle	34
3.2.3	Modelo de simulação	43
3.2.4	Visualização do Voo	48
3.3	Simulação e implementação	50
3.3.1	Janela de interface de voo	50
3.4	Considerações finais	52
4	A geração automática de código	53
4.1	Simulink Coder e Embedded Coder	53
4.2	Código fonte do pacote de suporte a drones Parrot	53
4.2.1	Compilação cruzada	54
4.2.2	Comunicação com o protótipo	54
4.2.3	Blocos acessórios	58
4.2.4	Análise da performance	58
4.2.5	Modelos e recursos dos diagrama de blocos	59
4.2.6	Atualização do <i>firmware</i> nativo	59
4.2.7	Código fonte em linguagem C	60
4.2.8	Configurações da geração automática de código	64
4.3	Código fonte do projeto <i>Hover</i>	66
4.3.1	Diagramas de blocos	67
4.3.2	Tabelas pré-modeladas	67
4.3.3	Utilidades	68
4.3.4	Objetos procedurais	70
4.4	Geração automática de código	71

4.4.1	Conversão do modelo em código	74
4.4.2	Makefile	78
4.5	Implementação e execução do código	82
4.6	Considerações finais	85
5	O sistema operacional embarcado	87
5.1	Acessando o sistema operacional	87
5.1.1	Diretório <i>data/edu</i>	88
5.1.2	Diretório <i>bin</i> e <i>usr/bin</i>	90
5.2	Instruções via FTP com o <i>firmware</i> modificado	92
5.3	Instruções via FTP com o <i>firmware</i> nativo	94
5.3.1	Código fonte do pacote de suporte a drones	95
5.3.2	Formato dos pacotes de instrução	96
5.4	Considerações finais	99
6	Utilizando MAD para testes HIL/SIL	101
6.1	Fluxo óptico	101
6.2	Ajustando o modelo de diagrama de blocos	103
6.3	Testando o protótipo físico	106
6.4	Modificando o controlador nativo	108
6.5	Considerações finais	111
7	Conclusão	112
7.1	Trabalhos futuros	114
	Referências bibliográficas	115

Lista de símbolos e abreviaturas

$\ddot{\Omega}_{\mathcal{F}}$	Aceleração angular do corpo no sistema de coordenadas \mathcal{F} – [s] ⁻²	13
$\ddot{P}_{\mathcal{F}}$	Aceleração linear do corpo no sistema de coordenadas \mathcal{F} – [m][s] ⁻²	13
$\dot{\Omega}_{\mathcal{F}}$	Velocidade angular do corpo no sistema de coordenadas \mathcal{F} – [s] ⁻¹	13
$\dot{P}_{\mathcal{F}}$	Velocidade linear do corpo no sistema de coordenadas \mathcal{F} – [m][s] ⁻¹	13
\mathcal{F}_{ϕ}	Sistema resultante da rotação de \mathcal{F}_{θ} no eixo \vec{x}_{θ}	11
\mathcal{F}_{θ}	Sistema resultante da rotação de \mathcal{F}_v no eixo \vec{y}_v	11
\mathcal{F}_b	Sistema de coordenadas centrado acoplado aos braços do VANT	9
\mathcal{F}_i	Sistema de coordenadas inercial no solo	9
\mathcal{F}_v	Sistema de coordenadas inercial centrado no CG do VANT	9
$\Omega_{\mathcal{F}}$	Orientação do corpo no sistema de coordenadas \mathcal{F} – [rad]	10
τ	Matriz de torques aplicados no CG do corpo – [N][m]	13
R	Matriz de rotação	10
T	Matriz de transformação angular	10
b	Constante de empuxo – [kg][m]	15
F	Matriz de forças aplicadas no CG do corpo – [N]	13
J	Matriz de rotação inercial do corpo – [kg][m] ²	13
k	Constante de arrasto – [kg][m] ²	16
l	Comprimento de um braço do quadricóptero – [m]	14
M	Massa total do corpo – [kg]	13

$P_{\mathcal{F}}$	Posição do corpo no sistema de coordenadas \mathcal{F} – [m]	10
Q_i	Força de arrasto reacionária à rotação do motor i – [N]	16
T_i	Força de empuxo vertical produzida pelo motor i – [N]	15
W	Força peso do quadricóptero – [N]	16
CG	Centro de Gravidade	9
GAC	Geração Automática de Código	2
HIL	Hardware-in-the-loop	2
MAD	Modelo Aplicado no Desenvolvimento	2
MVA	Micro Veículo Aéreo	2
SIL	Software-in-the-loop	2
SO	Sistema operacional	87
VANT	Veículo Aéreo Não Tripulado	1

Lista de Tabelas

2.1	Equações e etapas do filtro de Kalman para estimar um vetor de estados desconhecidos a partir de medição de sensores ou do valor do estado inicial. .	27
3.1	Equações e etapas do filtro de Kalman para estimar a altitude e velocidade vertical do drone a partir do sensor ultrassônico, barômetro e IMU embarcado.	40
3.2	Equações e etapas do filtro de Kalman para estimar a as velocidade do drone no plano XY a partir do fluxo ótico e IMU embarcado.	42
3.3	Ganhos sintonizados para os controladores de atitude e altitude do Parrot Mambo no diagrama de blocos do Simulink.	43
5.1	Índice de comandos e argumentos dos pacotes enviados via comunicação TCP/IP na porta FTP 12391 do Parrot Mambo modificado com o <i>firmware</i> de suporte ao Simulink.	93
5.2	Índice de comandos e argumentos dos pacotes enviados via comunicação UDP na porta 6000 do Parrot Mambo com o <i>firmware</i> nativo.	97

Lista de Figuras

1.1	Diagrama de processos do MAD segundo divisão de KRIZAN et al.[9].	3
1.2	Fluxo de trabalho unindo MAD, HIL, SIL e GAC.	4
1.3	Modelo em diagrama de blocos do Parrot Mambo no Simulink.	4
1.4	Representação física do Parrot Mambo no ambiente virtual simulado no Simulink.	5
2.1	Relação do sentido de rotação de cada propulsor aos movimentos executáveis por um drone no espaço tridimensional. a – empuxo vertical. b – Ação de guinada. c – Ação de arfagem. d – Ação de rolagem.	9
2.2	Sistemas de coordenadas referenciais F_i , F_b e F_v	10
2.3	Modelo inercial do quadricóptero.	14
2.4	Fotografia do minidrone Parrot Mambo.	27
2.5	Visão inferior da placa de circuito impresso do Parrot Mambo.	28
2.6	Visão superior da placa de circuito impresso do Parrot Mambo.	28
3.1	Diagrama de blocos do Parrot Mambo no Simulink. Da esquerda para a direita, os subsistemas de comandos de voo (<i>Flight Commands</i>), sistema de controle (<i>Flight Control System - Code Generation</i>), modelo de simulação (<i>Simulation Model</i>) e visualização do voo (<i>Flight Visualization</i>).	30
3.2	Diagrama de uma trajetória simbólica do drone sobre as coordenadas w_1 , w_2 e w_3 . O VANT está inicialmente a uma distância d da reta que conecta os pontos w_1 a w_2 . O raio de transição é representado por r	32
3.3	Bloco de comandos de voo (<i>Flight Commands</i>) e referência de atitude e posição (<i>Position/Attitude Reference</i>) no modelo <i>Hover</i>	33

3.4	Barramento do sinal <i>AC Cmd</i> saída do bloco de comandos de voo <i>Flight Commands</i>	33
3.5	Subsistema de controle de voo <i>Flight Control System - Code Generation</i> . . .	34
3.6	Módulo <i>Flight Control System</i> do subsistema de controle de voo <i>Flight Control System - Code Generation</i>	36
3.7	Módulo de controle de trajetória <i>Path Planning</i> do subsistema de controle de voo <i>Flight Control System - Code Generation</i>	37
3.8	Módulo do estimador de estados <i>State Estimator</i> do subsistema de controle de voo <i>Flight Control System - Code Generation</i>	38
3.9	Barramento do sinal <i>HAL</i> do bloco de calibração <i>SensorPreprocessing</i> no estimador de estados <i>State Estimator</i>	39
3.10	Barramento dos sensores de visão <i>VisionSensors</i> do subsistema de controle de voo <i>Flight Control System - Code Generation</i>	39
3.11	Diagrama de blocos do controlador <i>Controller</i> do subsistema de controle de voo <i>Flight Control System - Code Generation</i>	41
3.12	Modelo não linear do VANT no subsistema <i>Simulation Model</i>	44
3.13	Bloco <i>AC Model</i> do modelo não linear do VANT no subsistema <i>Simulation Model</i>	45
3.14	Modelo estático do ambiente no subsistema <i>Simulation Model</i>	46
3.15	Modelo dinâmico dos sensores no subsistema <i>Simulation Model</i>	47
3.16	Bloco <i>Sensor System</i> no modelo dinâmico dos sensores do subsistema <i>Simulation Model</i>	47
3.17	Bloco <i>Simulink 3D</i> no subsistema de visualização de voo <i>Flight Visualization</i>	48
3.18	Bloco <i>Extract Flight Instruments</i> no subsistema de visualização de voo <i>Flight Visualization</i>	49
3.19	Modelo 3D do Parrot Mambo no ambiente virtual simulado a partir do diagrama de blocos do Simulink.	50
3.20	Janela da interface de controle de voo aberta após a compilação e envio do modelo em blocos do Simulink para o drone.	51
4.1	Blocos acessórios disponíveis pelo pacote de suporte a minidrones Parrot. . .	58

4.2	Janela de configuração do modelo no Simulink.	65
4.3	Árvore de arquivos do diretório <RAIZ> de um projeto <i>Hover</i>	67
4.4	Janela de configurações da geração de código no Simulink.	72
4.5	Janela de configurações das opções avançadas de implementação de <i>hardware</i> no Simulink.	73
4.6	Fluxo de execução das funções do modelo implementado no sistema embarcado. Em amarelo, estão representadas as funções ponto de entrada. Os métodos declarados em <i>flightControlSystem.c</i> estão desenhados em verde. . .	78
4.7	Captura dos terminais de comando após estabelecida a conexão telnet com o <i>firmware</i> do MATLAB (acima) e o <i>firmware</i> nativo (abaixo).	83
4.8	Fluxo de transformação do modelo em diagrama de blocos para um objeto compartilhado com o protótipo físico.	85
5.1	Diagrama das comunicação entre o sensor de fluxo óptico para a estimativa de velocidade no plano XY e o controlador por meio da camada de abstração de <i>hardware</i>	89
5.2	Diagrama das portas disponíveis no sistema com o <i>firmware</i> adaptado ao MATLAB para o envio de comandos pelo servidor.	95
5.3	Captura da tela do <i>software</i> Wireshark ao monitorar as portas de comunicação entre o servidor e o drone com o sistema operacional original.	98
5.4	Diagrama das portas disponíveis no sistema com o <i>firmware</i> nativo para o envio de comandos pelo servidor.	100
6.1	Velocidade estimada do drone no plano XY em missão simulada de <i>Hover</i> . .	104
6.2	Posição estimada do drone no plano XY em missão simulada de <i>Hover</i>	105
6.3	Conjugados τ_x e τ_y computados pelo controlador de atitude em missão simulada de <i>Hover</i>	105
6.4	Posição do drone no plano XY em uma trajetória de lemniscate.	107
6.5	Velocidade estimada do drone em uma trajetória de lemniscate.	107
6.6	Posição estimada do drone em uma trajetória de lemniscate.	107
6.7	Diagrama de blocos do controlador de atitude. O bloco da integral discreta em destaque deve ser modificado para a aproximação de Tustin.	109

6.8	Diagrama de blocos do controlador de altitude. O bloco da integral discreta em destaque deve ser modificado para a aproximação de Tustin.	109
6.9	Conjugados τ_x e τ_y calculado pelo controlador de atitude PID com discretização de Euler e de Tustin.	110
6.10	Empuxo vertical T calculado pelo controlador de altitude PID com discretização de Euler e de Tustin.	110

Capítulo 1

Introdução

Neste capítulo serão apresentados a justificativa e os objetivos do desenvolvimento desse estudo. Por fim, será descrita a organização dos capítulos e das seções deste documento.

1.1 Justificativa

A aplicação de Veículos Aéreos Não Tripulados (VANTs), comumente denominados drones, estabeleceu-se ubíqua durante a última década em virtude da autonomia, baixo custo e baixo risco operacional desses dispositivos [1]. Os VANTs já são empregados em missões militares de busca, resgate e monitoramento de áreas endêmicas ou em risco de desastres ambientais [2].

Por efeito desse amplo campo de aplicações, os VANTs são alvos recorrentes de pesquisa para o desenvolvimento de algoritmos mais específicos de controle adaptativo, planejamento de trajetória e controle colaborativo [3]. Em ambientes acadêmicos, é preferido o uso de Micro Veículos Aéreos (MVAs) por razão do seu tamanho pequeno e baixo peso [4] e fácil manobrabilidade [5].

Para acelerar o projeto e sintonia do controlador de voo de um VANT é essencial dispor de uma plataforma que permita tanto testar o *software* quanto a prototipagem em *hardware*. Nesse cenário, é apropriado o uso de Modelo Aplicado no Desenvolvimento (MAD), Geração Automática de Código (GAC), *software-in-the-loop* (SIL) e *hardware-in-the-loop* (HIL). Esse fluxo de trabalho permite o ajuste do *firmware* por meio da simulação de um modelo e,

concomitantemente, a implementação do código modelado no protótipo ainda não finalizado. Ou seja, esse método possibilita o desenvolvimento simultâneo do *hardware* e do *software* do drone, uma vez que as duas linhas de pesquisa, mesmo que dependentes, não são sequenciais.

O método MAD consiste na modelagem matemática de uma planta e a simulação desse modelo em um ambiente virtual. O MAD de VANTs proporciona o teste numérico do sistema de controle de drones sem a necessidade de voos experimentais. Além disso, facilita a depuração dos testes ao permitir a visualização de variáveis internas não acessíveis no protótipo [6]. Dessa forma, o desenvolvimento de *software* se torna independente do *hardware* e pode ser testado rigorosamente sem o risco de danificar o dispositivo.

O ambiente virtual no método MAD inclui todos os componentes físicos importantes para o comportamento do sistema. São representadas a planta e as condições normais de operação do robô. Com esse modelo, é possível realizar rapidamente testes de integração *software-hardware* em diferentes cenários e garantir que o sistema desenvolvido responde corretamente às variáveis de entrada [7].

Esse modelo inicial deve ser detalhado e constantemente testado em simulação (*software-in-the-loop*). Ao terminar a execução do fluxo, a modelagem do sistema encapsulará todo o conhecimento necessário sobre o protótipo físico. A partir dessa etapa, é possível iniciar os testes *hardware-in-the-loop* a partir de uma plataforma de GAC.

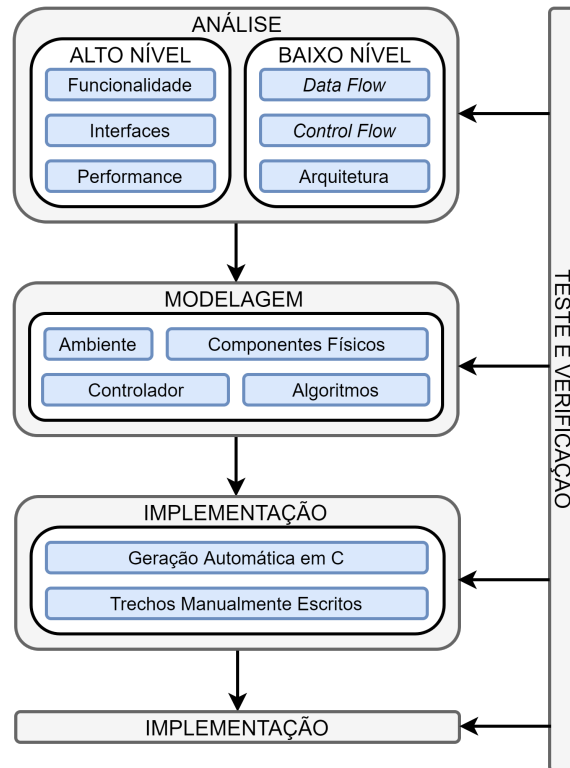
A GAC reduz a probabilidade de erro da tradução do modelo em código, uma vez que isenta do pesquisador o papel de programador. Esse código gerado automaticamente pode ser modular [8] ou compreender todo o sistema embarcado, incluindo *drivers* e *schedulers* [9]. Conforme Mosterman [6], com GAC, o foco do desenvolvimento do algoritmo é transferido da funcionalidade de cada subsistema para a arquitetura de integração, como interfaces – mapas de memória e chamada de funções – ou dependências de arquivo.

O emprego do método MAD em conjunto com a GAC é uma grande tendência na produção de robôs em larga escala. A união do MAD com a GAC acelera o fluxo de elaboração, teste e correção do produto ao criar uma linguagem comum e colaborativa entre os times de desenvolvimento.

Segundo Krizan et al. [9], MAD pode ser dividido em cinco processos conforme ilustrado na Figura 1.1: análise e descrição do comportamento físico da plataforma alvo, transfor-

mação desses requisitos funcionais em ambiente simulável, compilação do código para a plataforma alvo, implementação do código no protótipo e testes e validação do produto.

Figura 1.1: Diagrama de processos do MAD segundo divisão de KRIZAN et al.[9].



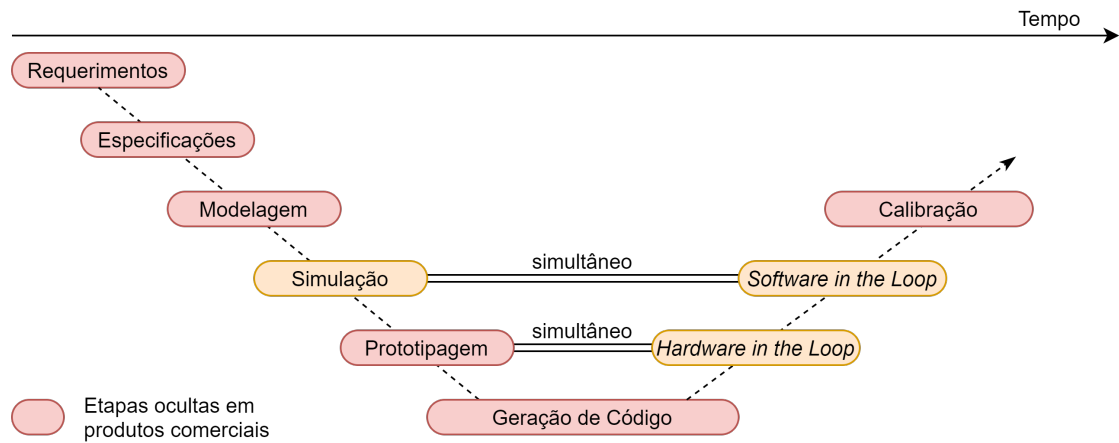
Fonte: Adaptado de KRIZAN et al.[9]

Esse fluxo de trabalho resultante da prática conjunta de GAC/HIL e MAD/SIL é efetivo para o desenvolvimento de validação de VANTs. De acordo com Mosterman [10], o fluxo MAD-GAC é representado por um “V” ilustrado na Figura 1.2, no qual as iterações de HIL ou de SIL do lado direito do “V” verificam a viabilidade e equivalência dos componentes físicos do produto final em relação ao modelo prototipado no lado esquerdo do “V”. Quando automatizado, esse fluxo de trabalho acelera a concepção e garante a qualidade do produto final. Logo, o MAD é essencial para assegurar a competitividade do fornecedor mercado atual.

1.2 Estudo de caso

O projeto descrito no presente documento trata da aplicação do método MAD no desenvolvimento de VANTs de pequeno porte. Para tal, foi adotado o quadricóptero Parrot

Figura 1.2: Fluxo de trabalho unindo MAD, HIL, SIL e GAC.

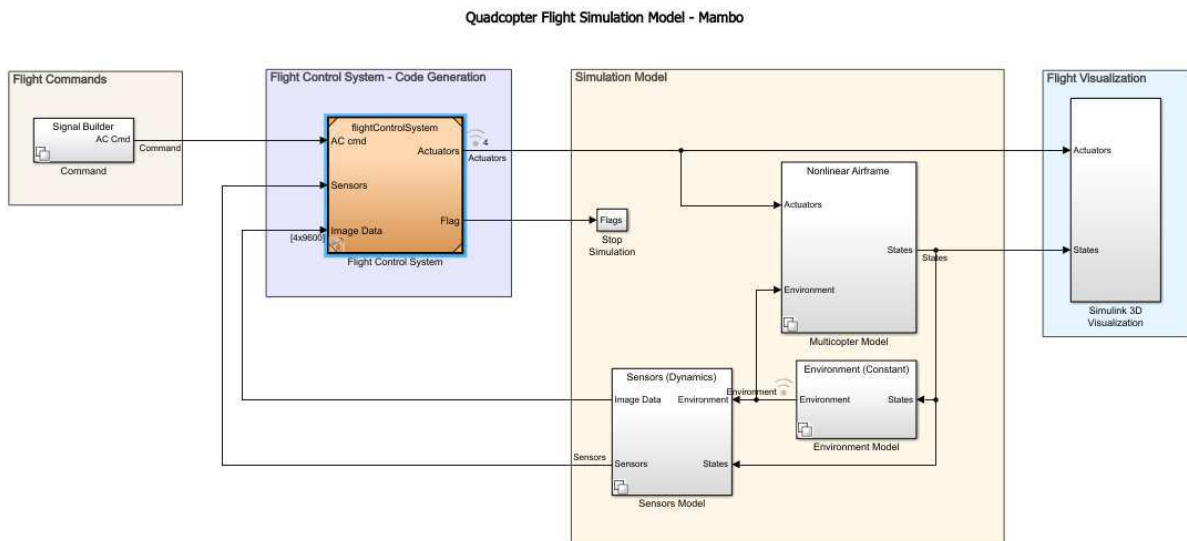


Fonte: Adaptado de MOSTERMAN, PRABHU e ERKKINEN[10]

Mambo. Essa escolha foi fundamentada na disponibilidade do suporte para HIL e SIL na ferramenta de modelagem Simulink/MATLAB [11].

Nesse ambiente de desenvolvimento, tanto os componentes físicos – tais como propulsores e sensores – quanto o sistema de controle são representados por blocos gráficos interconectados, conforme ilustrado na Figura 1.3.

Figura 1.3: Modelo em diagrama de blocos do Parrot Mambo no Simulink.

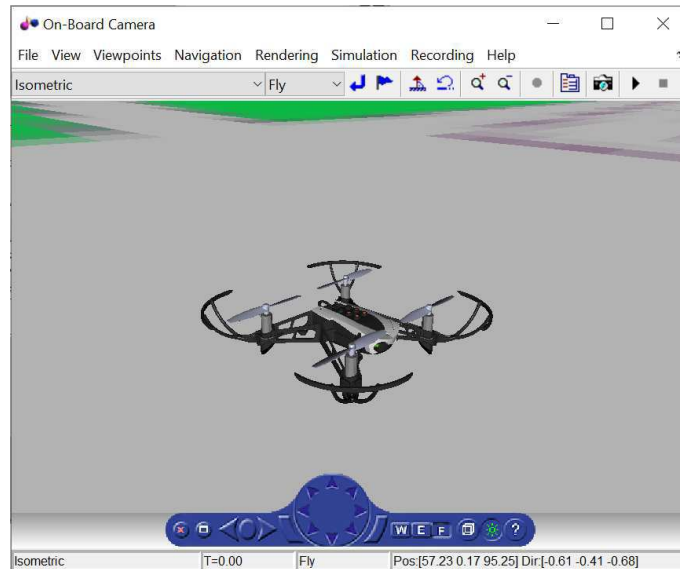


Fonte: MATHWORKS[11]

O modelo físico do minidrone pode ser representado visualmente em um ambiente virtual simulado. Nesse cenário, a o VANT em estudo é simbolizado por um conjunto de formas tridimensionais texturizadas, assim como ilustrado na Figura 1.4. Esse ambiente é repre-

sentado por campos magnéticos e gravitacionais e por correntes de ar com temperaturas e densidades personalizáveis.

Figura 1.4: Representação física do Parrot Mambo no ambiente virtual simulado no Simulink.



Fonte: MATHWORKS[11]

Essa customização do ambiente garante a aproximação entre a realidade virtual e a bancada experimental. Dessa forma, os ensaios com a planta simulada são verossímeis aos experimentos sobre o protótipo físico e conseguem antecipar falhas de *software* ou quedas que danifiquem o corpo do robô.

O pacote para o Parrot Mambo para Simulink também dispõe de simulações de missões pré-programadas e oferece a geração automática do código a partir do modelo construído pela interface gráfica mediante a licença do Simulink Coder [12].

O Simulink é um ambiente de simulação apropriado para pesquisadores sem conhecimento de programação ou de dependências de arquivo, tendo em vista a abstração dos níveis de arquitetura em linguagem de máquina. Contudo, essa ferramenta limita a modelagem dos sensores e o controle do Parrot Mambo a planos de voo pré-programados.

Dentro do contexto supracitado, é necessária uma solução baseada em código aberto a fim de compreender ou alterar componentes ou etapas ocultas no produto comercial. Portanto, este Projeto de Conclusão de Curso propõe analisar, reproduzir e sugerir possíveis melhorias à geração automática de código e à modelagem do Parrot Mambo.

1.3 Objetivo geral

Tendo em vista o método de desenvolvimento de *softwares* e prototipagem para drones e das limitações das plataformas VANTs de pequeno porte supracitadas, o objetivo geral deste Projeto de Conclusão de Curso é estudar e documentar a modelagem do Parrot Mambo e a geração de código automática no pacote de suporte ao Simulink por meio de experimentos *hardware* e *software-in-the-loop*. Esses ensaios constam na investigação, caracterização e projeto do sistema de controle de voo implementado no Parrot Mambo. Dessa forma, esse trabalho propõe dispor abertamente uma documentação do fluxo de teste e validação utilizado no desenvolvimento do sistema de controle de voo de VANTs.

1.4 Objetivos específicos

Para a realização do objetivo geral, são definidos os seguintes objetivos específicos:

- Compreender o modelo matemático de VANTs;
- Estudar e documentar o fluxo de desenvolvimento do sistema de controle de voo do Parrot Mambo em sua versão educacional e comercial;
- Executar ensaios para caracterização da comunicação e da estratégia de controle implementadas no sistema embarcado do Parrot Mambo.

1.5 Organização do documento

O capítulo 2 trata da modelagem matemática, da estratégia de controle de um quadricóptero e dos parâmetros e componentes físicos do Parrot Mambo.

O capítulo 3 aborda o modelo do Parrot Mambo na interface da ferramenta Simulink, isto é, os subsistemas e barramentos que compõem o diagrama de blocos, relacionando esta representação com o modelo matemático de um VANT.

O capítulo 4 trata do método MAD e GAC aplicado sobre o modelo do Parrot Mambo no Simulink. São detalhados os códigos-fonte do pacote e o fluxo da produção e implementação do *firmware* personalizado no drone.

O capítulo 5 trata do sistema operacional embarcado, do protocolo de comunicação com o drone sem o uso de ferramentas licenciadas e dos pacotes enviados do servidor para o drone com o modelo customizado ou original de fábrica.

O capítulo 6 aborda três experimentos. O primeiro trata do ajuste do algoritmo de fluxo óptico simulado na planta modelada no Simulink, isto é, um ensaio *software-in-the-loop*. Em seguida, o funcionamento desse sensor no protótipo físico é verificado por meio da aquisição de telemetria, ou seja, um ensaio *hardware-in-the-loop*. O último ensaio exemplifica a implementação de um novo controlador no sistema embarcado.

Por fim, o capítulo 7 trata das conclusões deste estudo e das contribuições desta documentação para futuros trabalhos.

Capítulo 2

Modelagem do quadricóptero

Este capítulo trata do modelo matemático de um quadricóptero e das especificações do MVA Parrot Mambo.

2.1 Dinâmica do quadricóptero

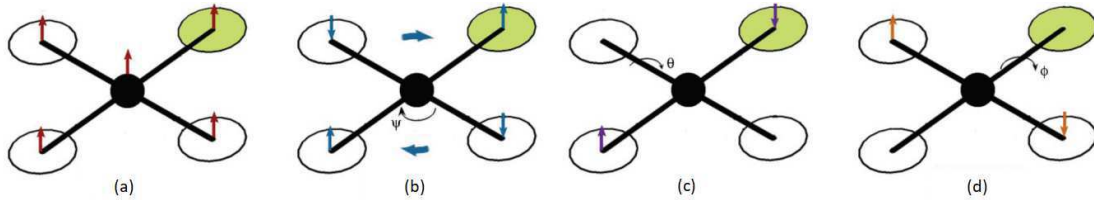
Os quadricópteros são VANTs com quatro motores dispostos em pontos extremos de uma cruz. Esses robôs são capazes de flutuação, pouso e decolagem vertical. Os quadricópteros possuem seis graus de liberdade, isto é, movimento livre no espaço tridimensional. Dessa forma, os drones transladam e rotacionam sobre três eixos perpendiculares.

O movimento de um quadricóptero é controlado a partir da variação da velocidade de rotação dos motores. Enquanto um par de motores opostos giram em um sentido, o outro par rotaciona no sentido oposto. Essa configuração é adotada para balancear o efeito do arrasto gerado por cada par de motores [13].

Para alterações de altitude, os quatro motores devem ser acionados simultaneamente para produzir uma força perpendicular ao solo. Para movimentações laterais, devem ser produzidas angulações no drone, cujos torques aceleram o corpo lateralmente [14]. Essas rotações são geradas alterando as velocidades relativas entre os motores de um par ou entre os dois pares de motores. Para manter constante a altitude durante a angulação, a aceleração imposta sobre um dos motores do par deve ser decrescida do seu motor complementar, isto é, do propulsor oposto na geometria da cruz [13].

Os entidos de rotação específicos de cada par de propulsores são associados a torques de guinada, rolagem ou arfagem e do empuxo vertical no quadricóptero, conforme observação da Figura 2.1.

Figura 2.1: Relação do sentido de rotação de cada propulsor aos movimentos executáveis por um drone no espaço tridimensional. a – empuxo vertical. b – Ação de guinada. c – Ação de arfagem. d – Ação de rolagem.



Fonte: Adaptado de RAZA e GUEAIEB[13]

Fica claro que o controle das velocidades dos propulsores em termos de forças e torques é mais adequado em oposição a velocidades. Logo, uma representação do VANT por um modelo dinâmico é mais apropriada que por um modelo cinemático.

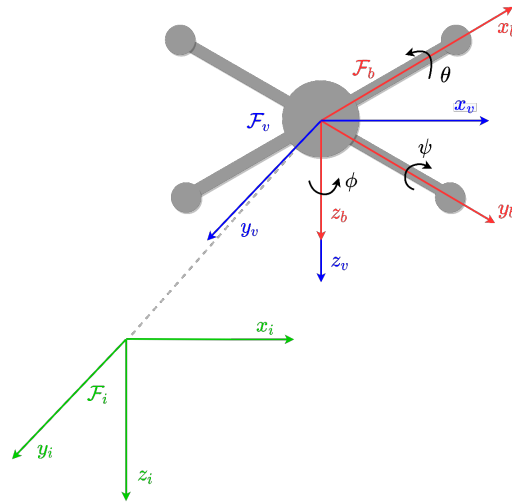
2.2 Modelo matemático do quadricóptero

2.2.1 Sistemas de coordenadas

Para a análise do modelo dinâmico do quadricóptero são adotados três sistemas de coordenadas:

- Sistema inercial $\mathcal{F}_i = (\vec{x}_i, \vec{y}_i, \vec{z}_i)$ fixo no solo para indicar a velocidade e localização do VANT no mundo;
- Sistema de coordenadas do corpo $\mathcal{F}_b = (\vec{x}_b, \vec{y}_b, \vec{z}_b)$ para o estudo das equações de movimento;
- Sistema de coordenadas inercial do veículo $\mathcal{F}_v = (\vec{x}_v, \vec{y}_v, \vec{z}_v)$ para a avaliação das forças e torques dos motores.

Os sistemas de referência supracitados podem ser observados na Figura 2.2. Tanto \mathcal{F}_b quanto \mathcal{F}_v estão centrados no centro de gravidade (CG) do quadricóptero. Os eixos \vec{x}_b e \vec{y}_b estão alinhados com os braços do drone.

Figura 2.2: Sistemas de coordenadas referenciais \mathcal{F}_i , \mathcal{F}_b e \mathcal{F}_v .


Fonte: Adaptado de RAZA e GUEAIEB[13]

A posição do quadricóptero é definida por $P_{\mathcal{F}} = [p_x \ p_y \ p_z]^T$ e a orientação do quadricóptero é anotada como $\Omega_{\mathcal{F}} = [\phi \ \theta \ \psi]^T$ para qualquer sistema de coordenadas \mathcal{F} [13].

Um vetor de deslocamento é necessário para transformar \mathcal{F}_i para o referencial inercial \mathcal{F}_v no CG do quadricóptero. Por sua vez, a transformação de \mathcal{F}_v para \mathcal{F}_b é realizada por meio da matriz de rotação $R_{\mathcal{F}_v}^{\mathcal{F}_b}$.

Para esse estudo, considera-se a relação entre as velocidades lineares e angulares no referencial local conforme representado na Equação 2.1 [15]. Já no referencial inercial, essa relação é representada conforme a Equação 2.2 [15].

$$[\dot{x}_v \ \dot{y}_v \ \dot{z}_v]^T = \mathbf{R}[\dot{x}_b \ \dot{y}_b \ \dot{z}_b]^T \quad (2.1)$$

$$[\dot{\phi}_v \ \dot{\theta}_v \ \dot{\psi}_v]^T = \mathbf{T}[\dot{\phi}_b \ \dot{\theta}_b \ \dot{\psi}_b]^T \quad (2.2)$$

para \mathbf{R} a matriz de rotação e \mathbf{T} a matriz de transformação angular.

2.2.2 Matriz de rotação

Qualquer transformação de coordenadas entre dois sistemas distintos pode ser obtida por três rotações sequenciais [15]. Para este estudo, foi adotada a sequência de rotação cardaniana ZYX descrita na Equação 2.3 [14].

$$\mathbf{R} = R_{\mathcal{F}_b}^{\mathcal{F}_v} = [R_{\mathcal{F}_v}^{\mathcal{F}_b}]^{-1} = [R_{\mathcal{F}_v}^{\mathcal{F}_b}]^T = R_z(-\psi)R_y(-\theta)R_x(-\phi) \quad (2.3)$$

Substituindo as matrizes $R_z(\psi)$, $R_y(\theta)$, $R_x(\phi)$ na Equação 2.3 referentes a rotação nos eixos \vec{x}_b , \vec{y}_b , \vec{z}_b , respectivamente, a transformação do referencial local \mathcal{F}_b para o inercial \mathcal{F}_v do veículo é escrita como exposto na Equação 2.4.

$$\begin{aligned} \mathbf{R} &= \begin{bmatrix} c\psi & -s\psi & 0 \\ s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \\ \mathbf{R} &= \begin{bmatrix} c\psi c\theta & -s\psi & c\psi s\theta \\ s\psi c\theta & c\psi & s\psi s\theta \\ -s\theta & 0 & c\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix} \\ \mathbf{R} &= \begin{bmatrix} c\psi c\theta & c\psi s\theta s\phi - s\psi s\theta c\phi & c\psi s\theta c\phi + s\psi s\phi \\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \end{aligned} \quad (2.4)$$

2.2.3 Matriz de transformação angular

Assim como a matriz de rotação $R_{\mathcal{F}_b}^{\mathcal{F}_v}$ converte as velocidades lineares do referencial local \mathcal{F}_b para o referencial inercial \mathcal{F}_v , a matriz de transformação angular $T_{\mathcal{F}_b}^{\mathcal{F}_v}$ associa as velocidades angulares $(\dot{\phi}, \dot{\theta}, \dot{\psi})$ entre esses sistemas de coordenadas [15].

Para encontrar a matriz $T_{\mathcal{F}_b}^{\mathcal{F}_v}$, considera-se os sistemas de referência:

- \mathcal{F}_θ resultante da rotação do referencial \mathcal{F}_v em um ângulo θ ao redor do eixo \vec{y}_v até que os eixos \vec{x}_v e \vec{z}_v estejam alinhados com \vec{x}_b e \vec{z}_b , respectivamente;
- \mathcal{F}_ϕ resultante da rotação do referencial \mathcal{F}_θ em um ângulo ϕ ao redor do eixo \vec{x}_θ até que os eixos \vec{x}_θ e \vec{y}_θ estejam alinhados com \vec{x}_b e \vec{y}_b , respectivamente.

Conforme discutido na Subseção 2.2.2, as matrizes de rotação $R_{\mathcal{F}_v}^{\mathcal{F}_\theta}$ de \mathcal{F}_v para \mathcal{F}_θ e $R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi}$ de \mathcal{F}_θ para \mathcal{F}_ϕ podem ser escritas conforme Equação 2.5 e Equação 2.6, respectivamente.

$$R_{\mathcal{F}_v}^{\mathcal{F}_\theta} = [R_{\mathcal{F}_\theta}^{\mathcal{F}_v}]^T = [R_y(\theta)]^T = \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix} \quad (2.5)$$

$$R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} = [R_{\mathcal{F}_\phi}^{\mathcal{F}_\theta}]^T = [R_x(\phi)]^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & s\phi \\ 0 & -s\phi & c\phi \end{bmatrix} \quad (2.6)$$

Como os sistemas \mathcal{F}_v e \mathcal{F}_b são centrados sobre o mesmo ponto, após duas rotações \mathcal{F}_v se transforma em \mathcal{F}_b . Logo, a matriz de rotação de \mathcal{F}_ϕ em relação ao sistema \mathcal{F}_b e de \mathcal{F}_ψ em relação ao sistema \mathcal{F}_v são equivalentes à matriz identidade I . Assim,

$$R_{\mathcal{F}_\phi}^{\mathcal{F}_b} = R_{\mathcal{F}_v}^{\mathcal{F}_\psi} = I$$

A relação entre o vetor de velocidades angulares $(\dot{\phi}, \dot{\theta}, \dot{\psi})$ no referencial local \mathcal{F}_b e no referencial inercial \mathcal{F}_v é exibida na Equação 2.7 [13].

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b} = R_{\mathcal{F}_\phi}^{\mathcal{F}_b} \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix}_{\mathcal{F}_\phi} + R_{\mathcal{F}_\phi}^{\mathcal{F}_b} R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix}_{\mathcal{F}_\theta} + R_{\mathcal{F}_\phi}^{\mathcal{F}_b} R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} R_{\mathcal{F}_\psi}^{\mathcal{F}_\theta} R_{\mathcal{F}_v}^{\mathcal{F}_\psi} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_\psi=\mathcal{F}_v} \quad (2.7)$$

Ao calcular os fatores separadamente, tem-se:

$$R_{\mathcal{F}_\phi}^{\mathcal{F}_b} R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} = I R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -s\phi \\ 0 & s\phi & c\phi \end{bmatrix}$$

$$R_{\mathcal{F}_\phi}^{\mathcal{F}_b} R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} R_{\mathcal{F}_\psi}^{\mathcal{F}_\theta} R_{\mathcal{F}_v}^{\mathcal{F}_\psi} = I R_{\mathcal{F}_\theta}^{\mathcal{F}_\phi} R_{\mathcal{F}_\psi}^{\mathcal{F}_\theta} = \begin{bmatrix} c\theta & 0 & -s\theta \\ s\phi s\theta & c\phi & s\phi c\theta \\ c\phi s\theta & -s\phi & c\phi c\theta \end{bmatrix}$$

Substituindo os termos acima na Equação 2.7, pode-se escrever a relação simplificada descrita na Equação 2.8.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b} = \begin{bmatrix} 1 & 0 & -s\theta \\ 0 & c\phi & s\phi c\theta \\ 0 & -s\phi & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_v} \quad (2.8)$$

Invertendo a multiplicação de matrizes em Equação 2.8, tem-se

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_v} = \begin{bmatrix} 1 & -s\phi t\theta & c\phi t\theta \\ 0 & c\phi & s\phi c\theta \\ 0 & s\phi s^{-1}\theta & c\phi s^{-1}\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b} \quad (2.9)$$

Relacionando Equação 2.9 com a Equação 2.2, conclui-se que a matriz de transformação angular \mathbf{T} é definida conforme exposto na Equação 2.10.

$$\mathbf{T} = \begin{bmatrix} 1 & -s\phi t\theta & c\phi t\theta \\ 0 & c\phi & s\phi c\theta \\ 0 & s\phi s^{-1}\theta & c\phi s^{-1}\theta \end{bmatrix} \quad (2.10)$$

2.2.4 Modelo dinâmico

O modelo dinâmico do quadricóptero pode ser deduzido utilizando-se a formulação de Newton-Euler [13] da Equação 2.11.

$$\begin{bmatrix} MI_{3 \times 3} & 0 \\ 0 & JI_{3 \times 3} \end{bmatrix} \begin{bmatrix} \ddot{P}_{\mathcal{F}_b} \\ \ddot{\Omega}_{\mathcal{F}_b} \end{bmatrix} + \begin{bmatrix} \dot{\Omega}_{\mathcal{F}_b} \times M\dot{P}_{\mathcal{F}_b} \\ \dot{\Omega}_{\mathcal{F}_b} \times J\dot{\Omega}_{\mathcal{F}_b} \end{bmatrix} = \begin{bmatrix} F_{\mathcal{F}_b} \\ \tau_{\mathcal{F}_b} \end{bmatrix} \quad (2.11)$$

para M a massa total do drone, J a matriz de rotação inercial, $F_{\mathcal{F}_b} = [f_x \ f_y \ f_z]^T$ a matriz de forças $\tau_{\mathcal{F}_b} = [\tau_y \ \tau_x \ \tau_z]^T$ a matriz de torques externos aplicados ao CG do quadricóptero.

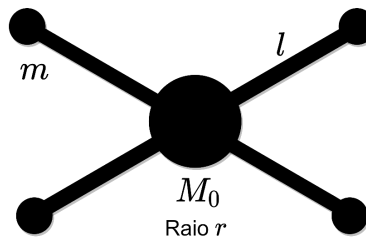
Matriz de rotação inercial

A matriz de rotação inercial J é representada por uma matriz simétrica [14], cujos termos na diagonal são os momentos de inércia e os termos fora da diagonal são os produtos de inércia. Assim, define-se J por

$$J = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

Neste trabalho, o quadricóptero é representado por quatro massas m pontuais conectadas por um braço de comprimento l a uma esfera sólida central de massa M_0 e raio r [13], conforme observa-se na Figura 2.3. As massas pontuais representam os motores, enquanto a esfera central simboliza o corpo do quadricóptero.

Figura 2.3: Modelo inercial do quadricóptero.



Fonte: Adaptado de RAZA e GUEAIEB[13]

A distribuição de massa do drone é simétrica em relação ao eixo x_v e y_v e o centro de massa no eixo z_v é equivalente ao centro geométrico de massa. Logo, os produtos de inércia são nulos [14] e a matriz de rotação inercial J do quadricóptero pode ser representada por

$$J = \begin{bmatrix} j_x & 0 & 0 \\ 0 & j_y & 0 \\ 0 & 0 & j_z \end{bmatrix} \quad (2.12)$$

O momento de inércia de uma esfera sólida é escrita por

$$I = \frac{2M_0r^2}{5}$$

Por sua vez, o momento de inércia de massa pontuais a um raio l dos eixos de rotação é dado por

$$I = ml^2$$

Logo, o momento do quadricóptero em cada eixo do sistema é descrito por

$$j_x = j_y = \frac{2M_0r^2}{5} + 2ml^2 \quad \text{e} \quad j_z = \frac{2M_0r^2}{5} + 4ml^2$$

Dessa forma, a matriz inercial da Equação 2.12 pode ser reescrita como

$$J = \begin{bmatrix} 2M_0r^2/5 + 2l^2m & 0 & 0 \\ 0 & 2M_0r^2/5 + 2l^2m & 0 \\ 0 & 0 & 2M_0r^2/5 + 4l^2m \end{bmatrix} \quad (2.13)$$

É possível desassociar a Equação 2.11 em duas equações referentes a cada linha. Enquanto a primeira equação está relacionada com os termos lineares, isto é, um modelo translacional, a segunda linha está associada aos termos angulares de posição do corpo, ou seja, um modelo rotacional.

Modelo de translação

O modelo de dinâmica translacional do quadricóptero descrito na Equação 2.14 pode ser expandido da linha referente aos termos lineares da Equação 2.11.

$$M \begin{bmatrix} \ddot{p}_x \\ \ddot{p}_y \\ \ddot{p}_z \end{bmatrix}_{\mathcal{F}_b} = \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix}_{\mathcal{F}_b} + M \begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{p}_z \end{bmatrix}_{\mathcal{F}_b}^T \times \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b}^T \implies \begin{bmatrix} \ddot{p}_x \\ \ddot{p}_y \\ \ddot{p}_z \end{bmatrix} = \frac{1}{M} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} + \begin{bmatrix} \dot{\psi} \dot{p}_y - \dot{\theta} \dot{p}_z \\ \dot{\phi} \dot{p}_z - \dot{\psi} \dot{p}_x \\ \dot{\theta} \dot{p}_x - \dot{\phi} \dot{p}_y \end{bmatrix} \quad (2.14)$$

O empuxo gerado por cada motor pode ser descrito por um vetor vertical no sentido $-\vec{z}_v$. Sendo a velocidade angular de cada motor anotada por ω_i , para $i = 1, 2, 3, 4$, o empuxo T_i de um motor é dado por

$$T_i = b\omega_i^2, \text{ para } i = 1, 2, 3, 4$$

para constante de empuxo b dependente da densidade do ar e do raio da lâmina das hélices [14]. A força de empuxo vertical total T é dada pela soma da propulsão T_i de cada motor:

$$T = \sum_{i=1}^4 T_i = T_1 + T_2 + T_3 + T_4 = b(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \quad (2.15)$$

A diferença entre as forças T_i de cada par de motores provoca um torque e, consequentemente, a rotação do veículo. Esse torque produtor do giro no eixo \vec{x} é dado por

$$\tau_x = l(T_4 - T_2) = lb(\omega_4^2 - \omega_2^2) \quad (2.16)$$

para l o a distância do motor ao CG, isto é, o comprimento do braço do quadricóptero. Por sua vez, o torque produtor do giro no eixo \vec{y} é dado por

$$\tau_y = l(T_3 - T_1) = lb(\omega_3^2 - \omega_1^2) \quad (2.17)$$

Conforme o princípio da inércia, a força reativa do arrasto aerodinâmico que atua sobre o corpo do drone [14] é expressa por

$$Q_i = k\omega_i^2, \text{ para } i = 1, 2, 3, 4$$

para a constante de arrasto k dependente dos mesmo fatores de b . Esse arrasto provoca um torque ao redor do eixo \vec{z} descrito por

$$\tau_z = -Q_1 + Q_2 - Q_3 + Q_4 = -k(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \quad (2.18)$$

para o par dos motores 1 e 3 rotacionando no sentido horário e o par dos os motores 2 e 4 girando no sentido anti-horário [13]. Por fim, resta descrever a força peso [13] que age no CG do corpo como

$$W_{\mathcal{F}_v} = Mg\vec{z}_v$$

para g a aceleração gravitacional e M a massa total do drone. Utilizando a matriz de rotação, $R_{\mathcal{F}_v}^{\mathcal{F}_b}$ é realizada a transformação da força peso $W_{\mathcal{F}_v}$ para o referencial local \mathcal{F}_b .

$$W_{\mathcal{F}_b} = R_{\mathcal{F}_v}^{\mathcal{F}_b} W_{\mathcal{F}_v} = [R_{\mathcal{F}_v}^{\mathcal{F}_b}]^T W_{\mathcal{F}_v} = \mathbf{R}^T W_{\mathcal{F}_v}$$

Conforme a Equação 2.4, tem-se

$$R_{\mathcal{F}_v}^{\mathcal{F}_b} = \mathbf{R}^T = \begin{bmatrix} c\psi c\theta & s\psi c\theta & -s\theta \\ c\psi s\theta s\phi - s\psi c\phi & s\psi s\theta s\phi + c\psi c\phi & c\theta s\phi \\ c\psi s\theta c\phi + s\psi s\phi & s\psi s\theta c\phi - c\psi s\phi & c\theta c\phi \end{bmatrix}$$

$$W_{\mathcal{F}_b} = \mathbf{R}^T \begin{bmatrix} 0 \\ 0 \\ Mg \end{bmatrix} = Mg \begin{bmatrix} -\sin\theta \\ c\theta s\phi \\ c\theta c\phi \end{bmatrix}$$

Finalmente, é preciso substituir $F_{\mathcal{F}_b}$ pela soma das forças atuantes sobre o quadricóptero $W_{\mathcal{F}_b} + T$ no modelo dinâmico translacional da Equação 2.14 [13]. Assim, encontra-se o modelo translacional do quadricóptero da Equação 2.19.

$$\begin{bmatrix} \ddot{p}_x \\ \ddot{p}_y \\ \ddot{p}_z \end{bmatrix}_{\mathcal{F}_b} = \frac{1}{M} \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}_{\mathcal{F}_b} + g \begin{bmatrix} -\sin\theta \\ c\theta s\phi \\ c\theta c\phi \end{bmatrix} + \begin{bmatrix} \dot{\psi} \dot{p}_y - \dot{\theta} \dot{p}_z \\ \dot{\phi} \dot{p}_z - \dot{\psi} \dot{p}_x \\ \dot{\theta} \dot{p}_x - \dot{\phi} \dot{p}_y \end{bmatrix}_{\mathcal{F}_b} \quad (2.19)$$

Modelo de rotação

O modelo de dinâmica rotacional descrito na Equação 2.20 do quadricóptero pode ser reescrito da linha referente aos termos angulares da Equação 2.11.

$$J \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix}_{\mathcal{F}_b} = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix}_{\mathcal{F}_b} + J \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b}^T \times \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}_{\mathcal{F}_b}^T \implies \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} + \begin{bmatrix} \dot{\phi} j_x \\ \dot{\theta} j_y \\ \dot{\psi} j_z \end{bmatrix} \times \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \right) \quad (2.20)$$

Tendo em vista a definição de J na Equação 2.13, a matriz inversa J^{-1} é definida por

$$J^{-1} = \begin{bmatrix} \frac{1}{j_x} & 0 & 0 \\ 0 & \frac{1}{j_y} & 0 \\ 0 & 0 & \frac{1}{j_z} \end{bmatrix}$$

Ao substituir J^{-1} na Equação 2.20, encontra-se no modelo rotacional da Equação 2.21.

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \tau_x/j_x \\ \tau_y/j_y \\ \tau_z/j_z \end{bmatrix} + \begin{bmatrix} \frac{j_y - j_z}{j_x} \dot{\theta} \dot{\psi} \\ \frac{j_z - j_x}{j_z} \dot{\phi} \dot{\psi} \\ \frac{j_x - j_y}{j_z} \dot{\phi} \dot{\theta} \end{bmatrix} \quad (2.21)$$

2.2.5 Espaço de estados

O espaço de estados do quadricóptero é descrito por $\dot{\mathbf{X}} = f(\mathbf{X}) + g(\mathbf{U})$ para \mathbf{X} o vetor de estados e \mathbf{U} o vetor de forças e torques de entrada. A partir das Equações 2.19 e 2.21 escreve-se:

$$\mathbf{X} = \left[\phi \quad \theta \quad \psi \quad \dot{\phi} \quad \dot{\theta} \quad \dot{\psi} \quad p_x \quad p_y \quad p_z \quad \dot{p}_x \quad \dot{p}_y \quad \dot{p}_z \right]^T$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \\ \dot{p}_x \\ \dot{p}_y \\ \dot{p}_z \\ \ddot{p}_x \\ \ddot{p}_y \\ \ddot{p}_z \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \tau_x/j_x + \frac{j_y - j_z}{j_x} \dot{\theta} \dot{\psi} \\ \tau_y/j_y + \frac{j_z - j_x}{j_y} \dot{\phi} \dot{\psi} \\ \tau_z/j_z + \frac{j_x - j_y}{j_z} \dot{\phi} \dot{\theta} \\ \dot{p}_x \\ \dot{p}_y \\ \dot{p}_z \\ \dot{\psi} \dot{p}_y - \dot{\theta} \dot{p}_z - g s \theta \\ \dot{\phi} \dot{p}_z - \dot{\psi} \dot{p}_x + g c \theta s \phi \\ \dot{\theta} \dot{p}_x - \dot{\phi} \dot{p}_y + g c \theta c \phi + \frac{T}{M} \end{bmatrix} \quad (2.22)$$

Observando o espaço de estados exposto na Equação 2.22, é possível verificar que a orientação do drone ($\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$) independe de componentes translacionais (\dot{p}_x , \dot{p}_y , \dot{p}_z). Entretanto, a translação do drone depende diretamente da mudança nas componentes rotacionais.

2.3 Controle de um quadricóptero

Uma missão é uma coleção de objetivos que devem ser executados pelo drone em um voo, tais como desviar ou rastrear obstáculos. Para executar uma missão, o drone deve percorrer uma trajetória determinada, ou seja, deslocar-se por um conjunto de posições e orientações de referência. As coordenadas de um ponto de referência são transferidas para o controlador embarcado a medida que o quadricóptero atinge o ponto anterior do conjunto.

O quadricóptero é um modelo sub-atuado com quatro atuadores em um sistema com seis

graus de liberdade [15]. Os sinais de entrada desse sistema são as forças e torques sobre o VANT, os quais atuam sobre a altitude p_z e orientação (ϕ, θ, ψ) do drone, respectivamente.

Ao alterar a rotação do seu corpo, um VANT é capaz reorientar os torques e forças [16] para alcançar a pose de referência [17]. Observando o espaço de estados na Equação 2.22, a rotação do corpo do VANT provoca um deslocamento lateral do corpo. O sistema de seis graus de liberdade é subatuado com os quatro atuadores, dado que a orientação e a posição p_x e p_y do drone estão dinamicamente acopladas [18].

Para a determinação da lei de controle, os valores de referência controláveis são p_x^* , p_y^* , p_z^* e ψ^* . Tendo em vista esse acoplamento entre a posição e orientação do corpo, é possível computar os valores de referência não controláveis ϕ^* e θ^* necessários para atingir p_x^* e p_y^* .

No drone em estudo, adquire-se a altitude p_z a partir de um sensor ultrassônico ou barométrico. A orientação ϕ , θ , ψ e as velocidades angulares $\dot{\phi}$, $\dot{\theta}$ e $\dot{\psi}$ do VANT são obtidas mediante um giroscópio e acelerômetro embarcado.

Por meio de um algoritmo de fluxo ótico, são medidos os deslocamentos do drone \dot{p}_x e \dot{p}_y . Esses estados são filtrados por um estimador, o qual retorna os valores corrigidos \hat{p}_x e \hat{p}_y . A modelagem desse estimador convergente é possível, uma vez que esses estados são observáveis [18]. No caso em estudo, o algoritmo estimador aplicado é o filtro de Kalman.

Os estados p_x , p_y não são diretamente aferidos e, por isso, são estimados a partir da integração dos valores \hat{p}_x e \hat{p}_y . Por sua vez, a velocidade \dot{p}_z é calculada pela rotação dos estados \hat{p}_x e \hat{p}_y para o referencial inercial do corpo. A altitude do drone p_z também é corrigida por um filtro de Kalman, o qual retorna o estado estimado \hat{p}_z .

Conforme as Equações 2.15, 2.16, 2.17 e 2.18, pode-se escrever a matriz de forças e torques no sistema como exposto na Equação 2.23.

$$\begin{bmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} b\omega_1^2 + b\omega_2^2 + b\omega_3^2 + b\omega_4^2 \\ -lb\omega_2^2 + lb\omega_4^2 \\ -lb\omega_1^2 + lb\omega_3^2 \\ -k\omega_1^2 + k\omega_2^2 + -k\omega_3^2 + k\omega_4^2 \end{bmatrix} = \begin{bmatrix} b & b & b & b \\ 0 & -lb & 0 & lb \\ -lb & 0 & lb & 0 \\ -k & k & -k & k \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \mathbf{A} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} \quad (2.23)$$

Visto que as constantes b , l e k são positivas não nulas, o posto da matriz \mathbf{A} é completo. Portanto, a matriz \mathbf{A} é inversível. Sendo assim, encontra-se:

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \quad (2.24)$$

para \mathbf{A} a matriz nomeada matriz de mixagem. As velocidades dos motores para que o drone atinja uma posição específica no espaço são calculadas pela multiplicação dos torques e forças pela matriz \mathbf{A}^{-1} descrita na Equação 2.24. É modelado um controlador para calcular as forças e torques necessários para que o drone se desloque para o ponto de referência desejado.

2.3.1 Lei de controle

Para computar as forças e torques necessários para que o drone alcance uma pose de referência, é empregada uma estrutura de controladores aninhados [14]. O laço mais externo é um controlador de posição, o qual fornecerá os ângulos de referência ϕ^* e θ^* com base nas coordenadas almejadas p_x^* e p_y^* . Em posse desses valores, o controlador mais interno determinará os torques e forças que deverão ser impostos ao sistema.

Conforme discutido, o controlador de posição estima os ângulos ϕ^* e θ^* para os quais o sistema atingirá as coordenadas de referência (p_x^*, p_y^*) . Nesse estudo, é modelado um controlador PD.

$$\begin{bmatrix} \phi^*(t) \\ \theta^*(t) \end{bmatrix} = K_p \begin{bmatrix} p_x^* - \hat{p}_x(t) \\ p_y^* - \hat{p}_y(t) \end{bmatrix} + K_d \begin{bmatrix} \dot{\hat{p}}_x(t) \\ \dot{\hat{p}}_y(t) \end{bmatrix}$$

Os ganhos empregados no modelo foram determinados pelas estratégias de sintonia clássicas da teoria de controle [14] e estão descritos na Tabela 3.3. Nos controladores descritos nessa seção, é incluso um filtro derivativo para garantir a estabilização da orientação do drone.

Por sua vez, controlador do laço interno determina os torques e forças para os quais o sistema atingirá a orientação de referência ϕ^* , θ^* e ψ^* . Os torques τ_x e τ_y são calculados pelo controlador de atitude, o qual é modelado por um PID.

$$\begin{bmatrix} \tau_x(t) \\ \tau_y(t) \end{bmatrix} = K_p \begin{bmatrix} \phi^* - \phi(t) \\ \theta^* - \theta(t) \end{bmatrix} + K_i \begin{bmatrix} \int [\phi^* - \phi(t)] dt \\ \int [\theta^* - \theta(t)] dt \end{bmatrix} + K_d \begin{bmatrix} \dot{\phi}(t) \\ \dot{\theta}(t) \end{bmatrix}$$

Os ângulos de referência ϕ^* e θ^* podem ser fornecidos diretamente ao controlador de atitude. Nesse caso, o laço externo do controlador é momentaneamente desativado.

O torque τ_z é calculado por um controlador PD, cuja entrada é o valor de referência ψ^* .

$$\tau_z(t) = K_p[\psi^* - \psi(t)] + K_d[\dot{\psi}(t)]$$

O empuxo vertical T é calculado pelo controlador de altitude modelado por um PID.

$$T(t) = K_p[p_z^* - \hat{p}_z(t)] + K_i \int [p_z^* - \hat{p}_z(t)]dt + K_d[\dot{\hat{p}}_z(t)]$$

Em posse dos torques τ_x , τ_y e τ_z e a força vertical T necessários para atingir a posição de referência, a velocidade de cada propulsor é calculada pela Equação 2.24. Para determinar esses valores, é necessária a relação entre a rotação ω do rotor o momento gerado no corpo, a qual está caracterizada na matriz de mixagem \mathbf{A} .

Tendo em vista que o drone modelado no Capítulo 3 está em configuração de “X”, ou seja, com os eixos \vec{x}_b e \vec{y}_b a 45 graus dos braços, é preciso rotacionar os torques τ_x , τ_y e τ_z ao redor do eixo \vec{z}_b . Não é necessário rotacionar a força vertical total T uma vez que esta possui componente apenas no eixo \vec{z} , o qual permanece fixo. Logo,

$$\begin{bmatrix} \tau_x(t) \\ \tau_y(t) \\ \tau_z(t) \end{bmatrix} = R_z(\pi/4) \mathbf{A}^* \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} -\frac{\sqrt{2}}{2}lb\omega_1^2 & -\frac{\sqrt{2}}{2}lb\omega_2^2 & \frac{\sqrt{2}}{2}lb\omega_3^2 & \frac{\sqrt{2}}{2}lb\omega_4^2 \\ -\frac{\sqrt{2}}{2}lb\omega_1^2 & \frac{\sqrt{2}}{2}lb\omega_1^2 & \frac{\sqrt{2}}{2}lb\omega_1^2 & -\frac{\sqrt{2}}{2}lb\omega_1^2 \\ -k\omega_1^2 & k\omega_2^2 & -k\omega_3^2 & k\omega_4^2 \end{bmatrix} \quad (2.25)$$

para \mathbf{A}^* a matriz equivalente a segunda, terceira e quarta linha da matriz de mixagem \mathbf{A} . Isolando as velocidades de rotação w_i na Equação 2.25, é possível redefinir a matriz de mixagem \mathbf{A} da seguinte maneira:

$$\mathbf{A} = \begin{bmatrix} b & b & b & b \\ -\frac{\sqrt{2}}{2}lb & -\frac{\sqrt{2}}{2}lb & \frac{\sqrt{2}}{2}lb & \frac{\sqrt{2}}{2}lb \\ -\frac{\sqrt{2}}{2}lb & \frac{\sqrt{2}}{2}lb & \frac{\sqrt{2}}{2}lb & -\frac{\sqrt{2}}{2}lb \\ -k & k & -k & k \end{bmatrix} \quad (2.26)$$

No modelo em estudo no Capítulo 3, a constante de empuxo b e constante de arrasto k são definidas conforme a seguinte formulação.

$$\begin{aligned} b &= C_t \rho (\pi r^4) \\ k &= C_q \rho (\pi r^5) \end{aligned} \quad (2.27)$$

para ρ a densidade do ar, r o raio da hélice de cada rotor. Substituindo a Equação 2.27 na Equação 2.26, resulta-se na seguinte definição da matriz de mixagem \mathbf{A} :

$$\mathbf{A} = [C_t \rho (\pi r^4)] \begin{bmatrix} 1 & 1 & 1 & 1 \\ -\frac{\sqrt{2}}{2}l & -\frac{\sqrt{2}}{2}l & \frac{\sqrt{2}}{2}l & \frac{\sqrt{2}}{2}l \\ -\frac{\sqrt{2}}{2}l & \frac{\sqrt{2}}{2}l & \frac{\sqrt{2}}{2}l & -\frac{\sqrt{2}}{2}l \\ -\frac{C_q r}{C_t} & \frac{C_q r}{C_t} & -\frac{C_q r}{C_t} & \frac{C_q r}{C_t} \end{bmatrix} \quad (2.28)$$

Em posse da matriz de mixagem \mathbf{A} , dos torques τ_x , τ_y e τ_z e a força vertical T , as velocidades de rotação w_i de cada propulsor são computados pela Equação 2.24. Essa informação é transmitida a partir dos controladores ESC (*Electronic Speed Control* – controle eletrônico de velocidade), os quais produzem os sinais de comando para os motores. Esses resultados são obtidos a partir da relação entre a velocidade de rotação e a corrente de entrada do rotor $A_{\omega^2}^{PWM}$, a qual é fornecida na documentação desse componente.

A partir da Equação 2.24, o sinal de comando PWM_i para cada um dos motores é então definido por:

$$\begin{bmatrix} PWM_1(t) \\ PWM_2(t) \\ PWM_3(t) \\ PWM_4(t) \end{bmatrix} = A_{\omega^2}^{PWM} \begin{bmatrix} \omega_1^2(t) \\ \omega_2^2(t) \\ \omega_3^2(t) \\ \omega_4^2(t) \end{bmatrix} = A_{\omega^2}^{PWM} \mathbf{A}^{-1} \begin{bmatrix} T(t) \\ \tau_x(t) \\ \tau_y(t) \\ \tau_z(t) \end{bmatrix} \quad (2.29)$$

Os sinais de PWM são saturados em 92% da tensão máxima de entrada dos motores. Essa etapa garante a vida útil dos motores.

É evidente que o deslocamento do sistema até a posição de referência depende do comando enviado para o motor, a velocidade de rotação e o empuxo vertical gerado pelo rotor. Sendo assim, a modelagem detalhada dos atuadores é essencial para garantir a eficiência do controlador. Quando na planta está integrado o modelo da bateria, pode-se desenvolver um

controle adaptativo que otimize a performance do drone. Por exemplo, a trajetória poderia ser replanejada ou os ganhos poderiam ser refinados para atender a tensão ainda disponível para a propulsão.

2.3.2 Estimador de estados

O filtro de Kalman é um método matemático que estima o valor de variáveis aleatórias x a partir de um conjunto de medições y contaminadas com ruídos. Nesse trabalho, esse algoritmo é empregado para estimar as velocidades \dot{p}_x e \dot{p}_y e a posição p_z do drone. O espaço de estados para um vetor de estados x é modelado por

$$\begin{aligned}\dot{x}_{[n]} &= Ax_{[n]} + Bu_{[n]} + w_n \\ y_{[n]} &= Cx_{[n]} + Du_{[n]} + v_n\end{aligned}\tag{2.30}$$

para $w_{[n]}$ o ruído do processo e $v_{[n]}$ o ruído de medição. Considerando $x_{[n]}$ o vetor de estados no tempo discreto n , é possível estimar o vetor $x_{[n+1]}$ no próximo instante $n+1$ por

$$x_{[n+1]} = Fx_{[n]} + Gu_{[n]} + w_n\tag{2.31}$$

para $u_{[n]}$ o sinal de entrada do sistema. As matrizes F e G são calculadas por métodos numéricos. Considerando um modelo cinemático e p um estado de posição ou velocidade, tem-se que

$$\begin{bmatrix} p_{[n+1]} \\ \dot{p}_{[n+1]} \end{bmatrix} = \begin{bmatrix} p_{[n]} \\ \dot{p}_{[n]} \end{bmatrix} + \Delta T \begin{bmatrix} \dot{p}_{[n]} \\ \ddot{p}_{[n]} \end{bmatrix} + w_n$$

para ΔT o tempo entre amostras. Esse modelo considera que tanto velocidade quanto a aceleração são constantes entre as amostras.

Tendo em vista que o estado estimado é contaminada com um ruído w_n e considerando $D = 0$ na Equação 2.30, é possível prever o estado atual da seguinte maneira

$$\hat{x}_{[n|n]} = \hat{x}_{[n|n-1]} + k_n(y_{[n]} - \hat{y}_{[n]}) = \hat{x}_{[n|n-1]} + k_n C(x_{[n]} - \hat{x}_{[n|n-1]}) + k_n v_n\tag{2.32}$$

para $\hat{x}_{[n|n]}$ a previsão dos estados atuais no instante n , $\hat{x}_{[n|n-1]}$ a previsão dos estados atuais

no instante $n - 1$, $\hat{y}_{[n]}$ a saída esperada no instante n , $y_{[n]}$ a saída medida no instante n e k_n o ganho de Kalman. Essa ganho depende da qualidade da estimativa, isto é, caso

- $\hat{x}_{[n|n-1]}$ seja uma boa estimativa ou a medição $y_{[n]}$ não seja confiável, k_n será pequeno;
- $\hat{x}_{[n|n-1]}$ não seja uma boa estimativa ou a medição $y_{[n]}$ seja confiável, k_n será grande.

O ganho k_n não é constante e é escolhido de modo a minimizar a variância do erro entre o estado real e sua estimativa $x_{[n]} - \hat{x}_{[n|n]}$ a cada amostra n . Essa variância $P_{[n|n]}$ é definida

$$P_{[n|n]} = \text{cov}\{x_{[n]} - \hat{x}_{[n|n]}\} = \text{cov}\{\tilde{x}_{[n|n]}\} \quad (2.33)$$

Substituindo a Equação 2.32 na Equação 2.33,

$$\begin{aligned} P_{[n|n]} &= \text{cov}\{(I - k_n C)(x_{[n]} - \hat{x}_{[n|n-1]}) + k_n v_n\} \\ &= \text{cov}\{(I - k_n C)(x_{[n]} - \hat{x}_{[n|n-1]})\} + \text{cov}\{k_n v_n\} \\ &= (I - k_n C) \text{cov}\{x_{[n]} - \hat{x}_{[n|n-1]}\} (I - k_n C)^T + k_n \text{cov}\{v_n\} k_n^T \\ &= (I - k_n C) P_{[n|n-1]} (I - k_n C)^T + k_n R_n k_n^T \end{aligned} \quad (2.34)$$

para R_n a matriz da covariância dos ruídos de medição de cada estado. Considerando os estados do vetor $x_{[n]}$ independentes, a matriz de covariância $P_{[n|n]}$ é uma matriz diagonal.

Conforme a teoria de filtros lineares ótimos, o ganho k_n escolhido deve minimizar o traço da matriz $P_{[n|n]}$ a cada nova amostra. O traço $\text{tr}(P_{[n|n]})$ é definido como a soma dos elementos da diagonal da matriz $P_{[n|n]}$. Expandindo a Equação 2.34, obtém-se

$$\begin{aligned} P_{[n|n]} &= (I - k_n C) P_{[n|n-1]} (I^T - (k_n C)^T) + k_n R_n k_n^T \\ &= (P_{[n|n-1]} - k_n C P_{[n|n-1]}) (I - C^T k_n^T) + k_n R_n k_n^T \\ &= P_{[n|n-1]} - k_n C P_{[n|n-1]} - P_{[n|n-1]} C^T k_n^T + k_n (C P_{[n|n-1]} C^T + R_n) k_n^T \end{aligned} \quad (2.35)$$

Uma vez que o traço de uma matriz equivale ao traço de sua transposta, o traço sobre a definição de $P_{[n|n]}$ desenvolvida na Equação 2.35 é

$$\begin{aligned} \text{tr}(P_{[n|n]}) &= \text{tr}(P_{[n|n-1]}) - \text{tr}(k_n C P_{[n|n-1]}) - \text{tr}(P_{[n|n-1]} C^T k_n^T) \\ &\quad + \text{tr}(k_n (C P_{[n|n-1]} C^T + R_n) k_n^T) \end{aligned}$$

$$\text{tr}(P_{[n|n]}) = \text{tr}(P_{[n|n-1]}) - 2\text{tr}(k_n C P_{[n|n-1]}) + \text{tr}(k_n (C P_{[n|n-1]} C^T + R_n) k_n^T) \quad (2.36)$$

Diferenciando a Equação 2.36 em relação ao ganho k_n para encontrar o ponto mínimo:

$$\begin{aligned} \frac{\partial(\text{tr}(P_{[n|n]}))}{\partial k_n} &= \frac{\partial(\text{tr}(P_{[n|n-1]}))}{\partial k_n} - \frac{\partial(2\text{tr}(k_n C P_{[n|n-1]}))}{\partial k_n} \\ &\quad + \frac{\partial(\text{tr}(k_n (C P_{[n|n-1]} C^T + R_n) k_n^T))}{\partial k_n} = 0 \end{aligned} \quad (2.37)$$

Tendo em conta que o traço é um operador linear, e que para uma matriz B diagonal

$$\begin{aligned} \frac{\partial(AB)}{\partial A} &= B^T \\ \frac{\partial(ABA^T)}{\partial A} &= 2AB \end{aligned}$$

é possível expandir a Equação 2.37 nos seguintes termos

$$\begin{aligned} \text{tr}\left(\frac{\partial(P_{[n|n-1]})}{\partial k_n}\right) &= 0 \\ \text{tr}\left(\frac{\partial(k_n C P_{[n|n-1]})}{\partial k_n}\right) &= (C P_{[n|n-1]})^T \\ \text{tr}\left(\frac{\partial(k_n (C P_{[n|n-1]} C^T + R_n) k_n^T)}{\partial k_n}\right) &= 2k_n (C P_{[n|n-1]} C^T + R_n) \end{aligned}$$

Substituindo na Equação 2.37, resulta-se na definição do ganho k_n a seguir.

$$\begin{aligned} k_n &= (C P_{[n|n-1]})^T (C P_{[n|n-1]} C^T + R_n)^{-1} \\ &= P_{[n|n-1]} C^T (C P_{[n|n-1]} C^T + R_n)^{-1} \end{aligned} \quad (2.38)$$

tendo em vista que a matriz $P_{[n|n-1]}$ é uma matriz simétrica. Substituindo a definição de k_n da Equação 2.38 no quarto termo do somatório na Equação 2.35, obtém-se

$$\begin{aligned} P_{[n|n]} &= P_{[n|n-1]} - k_n C P_{[n|n-1]} - P_{[n|n-1]} C^T k_n^T \\ &\quad + P_{[n|n-1]} C^T (C P_{[n|n-1]} C^T + R_n)^{-1} (C P_{[n|n-1]} C^T + R_n) k_n^T \\ &= P_{[n|n-1]} - k_n C P_{[n|n-1]} - P_{[n|n-1]} C^T k_n^T + P_{[n|n-1]} C^T k_n^T \\ &= P_{[n|n-1]} - k_n C P_{[n|n-1]} \end{aligned} \quad (2.39)$$

A Equação 2.39 é a formulação reduzida da atualização da covariância $P_{[n|n]}$. A previsão da variância do erro de estimativa no instante futuro $n + 1$ é semelhante à Equação 2.33.

$$P_{[n+1|n]} = cov\{x_{[n+1]} - \hat{x}_{[n+1|n]}\}$$

para $x_{[n+1]}$ o vetor de estados verdadeiros no instante $n + 1$ e $\hat{x}_{[n+1|n]}$ o vetor de estados previstos em $n + 1$ no instante n . Analogamente à Equação 2.32, a previsão dos estados futuros não corrigidos com o ganho k_n é dada por

$$\hat{x}_{[n+1|n]} = F\hat{x}_{[n|n]} + Gu_{[n|n]} \quad (2.40)$$

Substituindo os termos $x_{[n+1]}$ e $\hat{x}_{[n+1|n]}$ na Equação 2.39, resulta-se em

$$\begin{aligned} P_{[n+1|n]} &= cov\{Fx_{[n]} + Gu_{[n]} + w_n - F\hat{x}_{[n|n]} - Gu_{[n|n]}\} = cov\{F(x_{[n]} - \hat{x}_{[n|n]}) + w_n\} \\ &= cov\{F(x_{[n]} - \hat{x}_{[n|n]})\} + cov\{w_n\} = Fcov\{(x_{[n]} - \hat{x}_{[n|n]})\}F^T + Q_n \\ &= FP_{[n|n]}F^T + Q_n \end{aligned}$$

O conjunto das equações do filtro estimador de Kalman são escritas conforme a Tabela 2.1. Em posse de um estado $x_{[0]}$ e correlação inicial $P_{[0]}$, inicia-se correção do estado para o instante atual n e extrapolação da previsão para o próximo instante $n + 1$. Esse ciclo se repete para cada incremento de n . O erro entre estado previsto \hat{x} conhecido e o estado real x desconhecido é menor a cada iteração. As matrizes F , G , Q e R empregadas no modelo simulado nesse trabalho são apresentadas nas Tabela 3.1 e Tabela 3.2.

2.4 Características do quadricóptero em estudo

Para o estudo do fluxo de desenvolvimento de VANTs, foi utilizado um quadricóptero Parrot. O MAV adotado pertence à serie Mambo Fly e está ilustrado na Figura 2.4. Esse drone possui quatro motores, uma câmera inferior voltada para o solo e para-choques para proteger as hélices. O quadricóptero pode ser acompanhado por acessórios, como câmera superior de alta resolução, garra para segurar objetos e canhão para arremessar bolas. Os LEDs frontais indicam o nível de bateria, cuja autonomia é 9 minutos. O Parrot Mambo pesa 60

Tabela 2.1: Equações e etapas do filtro de Kalman para estimar um vetor de estados desconhecidos a partir de medição de sensores ou do valor do estado inicial.

Etapa	Equação	Formulação
Correção	Medição	$\hat{y}_{[n]} = C\hat{x}_{[n]} + Du_{[n]}$
	Ganho	$k_n = P_{[n n-1]}C^T(CP_{[n n-1]}C^T + R_n)^{-1}$
Atualização	Estado	$\hat{x}_{[n n]} = \hat{x}_{[n n-1]} + k_n(y_{[n]} - \hat{y}_{[n]})$
	Correlação	$P_{[n n]} = (I - k_nC)P_{[n n-1]}$
Previsão	Estado	$\hat{x}_{[n+1 n]} = F\hat{x}_{[n n]} + Gu_{[n n]}$
	Correlação	$P_{[n+1 n]} = FP_{[n n]}F^T + Q_n$

gramas e mede 18 centímetros de largura e de comprimento.

Figura 2.4: Fotografia do minidrone Parrot Mambo.



Fonte: MATHWORKS[11].

2.4.1 Componentes físicos do Parrot Mambo

O drone Parrot Mambo é composto por um conjunto de sensores que informam ao controlador dados de altitude, velocidade e posição. Na Figura 2.5 e na Figura 2.6 estão ilustradas a placa de circuito impresso na visão inferior e superior ao drone.

A altitude do drone é obtida por dois sensores. O primeiro é o ultrassônico apontado para o solo, em destaque na Figura 2.5. O segundo sensor é o barômetro compacto LPS22HB com comunicação I²C.

As velocidades e acelerações também são aferidas por dois sensores. O primeiro é a unidade de processamento de movimento (*Motion Processing Unit* – MPU) MPU6050. Esse dispositivo une um giroscópio, um acelerômetro e um processador digital de movimento (*Digital Motion Processor* – DPM). O segundo sensor é a câmera estéreo Aptina MT9V117

Figura 2.5: Visão inferior da placa de circuito impresso do Parrot Mambo.

Fonte: Autoria própria

Figura 2.6: Visão superior da placa de circuito impresso do Parrot Mambo.

Fonte: Autoria própria

acompanhada do controlador Etron ESP668F, o qual calcula a velocidade do VANT por meio de um algoritmo de fluxo óptico. São capturados 60 quadros por segundo.

A bateria é controlada pelo circuito integrado SMB1358. O processador ARM Cortex-A9 e é acompanhado pelo controlador de periféricos DMA PL080. Enquanto com o *firmware nativo*, o controlador é temporizado em 800MHz; com o *firmware* do pacote do MATLAB o processador é temporizado em 416MHz.

2.5 Considerações finais

A dinâmica de um quadricóptero foi matematicamente representada por um espaço de estados cujo vetor de estados é composto pela pose e pelas velocidades translacionais e rotacionais do drone e cuja entrada do sistema são os torques e a força vertical sobre o corpo. De posse dos parâmetros do corpo, tais como massa e comprimento do braço; e do ambiente, por exemplo a constante de arrasto; foi apresentado como modelar um controlador de atitude e altitude para o drone. O próximo capítulo trata como essas relações matemáticas do modelo dinâmico e do controlador são simbolizadas no diagrama de blocos do Simulink.

Capítulo 3

O ambiente de modelagem

Este capítulo analisa o método MAD no Parrot Mambo por meio da ferramenta Simulink. Primeiramente, são apresentados os diagramas de blocos que compõem o modelo do quadricóptero e, em seguida, os cenários disponíveis para a simulação no ambiente virtual.

3.1 Pacote de suporte a minidrones Parrot no Simulink

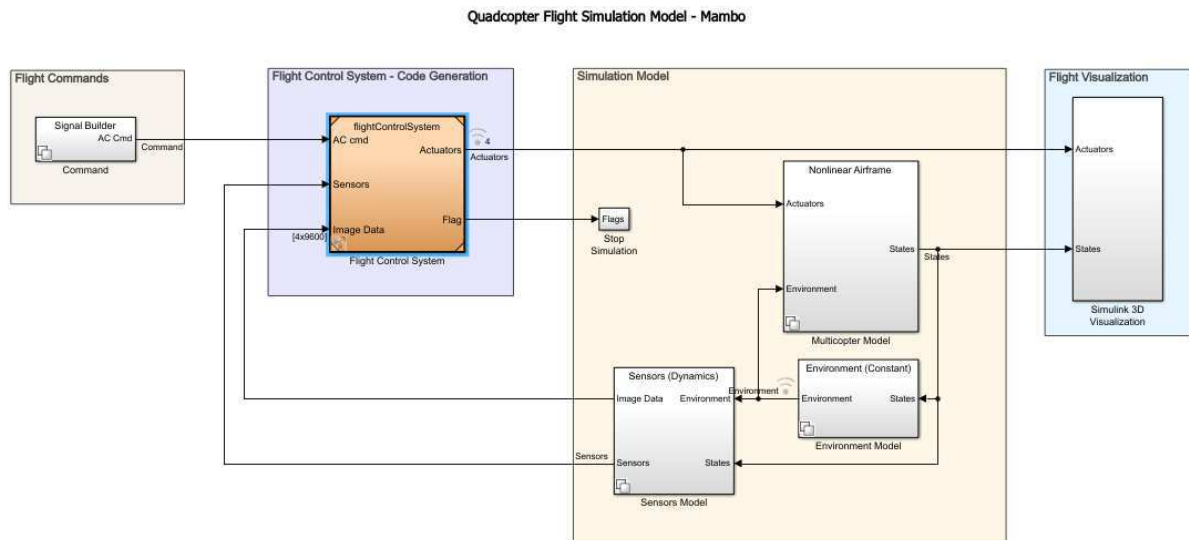
O pacote de suporte à minidrones Parrot [11] oferece um modelo do quadricóptero Parrot Mambo no Simulink. O Simulink é uma Interface de Programação de Aplicações (IPA) na qual o modelo é representado por blocos conectados mediante portas de entrada e de saída. O conjunto desses blocos interconectados é nomeado diagrama de blocos.

Essa IPA permite que outras ferramentas acessem os dados de memória do modelo, tais como o Simulink Coder [12] ou Embedded Coder. Essas duas ferramentas são responsáveis pela geração de código automática a partir da representação de blocos no Simulink.

3.2 Diagrama de blocos do Parrot Mambo

O diagrama de blocos do Parrot Mambo no Simulink pode ser observado na Figura 3.1. Esse modelo é separável em quatro módulos: comandos de voo, sistema de controle, modelo de simulação e visualização do voo. Apenas o bloco do sistema de controle é exportado para código em linguagem C e exportado para o protótipo físico.

Figura 3.1: Diagrama de blocos do Parrot Mambo no Simulink. Da esquerda para a direita, os subsistemas de comandos de voo (*Flight Commands*), sistema de controle (*Flight Control System - Code Generation*), modelo de simulação (*Simulation Model*) e visualização do voo (*Flight Visualization*).



Fonte: MATHWORKS[11]

O pacote de suporte no Simulink disponibiliza diagramas de blocos para aplicações específicas do MAV Mambo:

- *Getting Started*: esse modelo rotaciona os propulsores 1 e 3 por dois segundos, seguidos dos propulsores 2 e 4 por mais dois segundos. Há uma espera de 20 segundos entre as repetições. Tendo em vista que o drone não voa, esse modelo é indicado para testar a conexão com o quadricóptero;
- *Communication*: a rotação dos motores nesse modelo é semelhante ao *Getting Started*. Todavia, o sinal de controle dos motores é enviado por uma porta TCP/IP ilustrada por um bloco no Simulink. Os dados do acelerômetro são recebidos pelo computador por um bloco representante da conexão UDP. O modelo responsável por se comunicar com estas portas do drone é o *Communication Host*;
- *External Mode*: esse modelo proporciona a sintonia e monitoramento de parâmetros durante voo. É indicado para testes de integração de *hardware* e de *software* e prototipagem rápida;
- *Keyboard Control*: nesse modelo, caracteres ACSII são transmitidos como sinais de

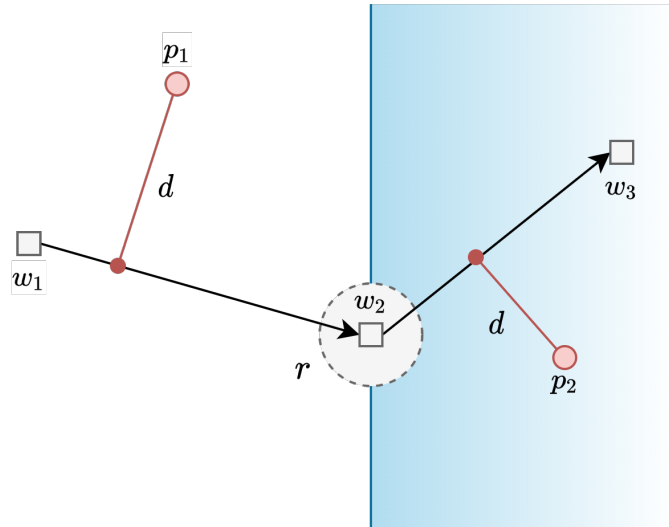
controle via comunicação TCP/IP. A configuração inicial é acionar um motor com o caractere “a” e pará-lo com o caractere “s”;

- *Vision*: esse modelo customiza o processamento da imagem da câmera inferior do drone para controle de trajetória. Esse algoritmo é executado a cada 200 milissegundos. A imagem em formato Y1UY2V é convertida em três canais RGB. Em seguida, identificam-se padrões da superfície a partir da composição de cores da imagem. Com base nessa informação, é gerada uma lógica de controle dos motores. Esse modelo também possui um modo externo equivalente (*Vision External Mode*) para visualização da imagem capturada durante simulação;
- *Hover*: nesse modelo, o drone flutua a 1,1 metro da altitude inicial. O voo perdura até o tempo especificado na simulação. Há a estimação de posição por fluxo óptico;
- *Competition*: une o modelo *Hover* com *Vision*. Em outras palavras, a trajetória é controlada por processamento de imagem e a estimação de posição é realizada por fluxo óptico. A configuração inicial do modelo é detectar uma superfície azul e planar sobre a mesma;
- *Waypoint Follower*: esse modelo planeja a trajetória do minidrone por um conjunto de pontos. Esses locais devem ser especificados em coordenadas. É ajustável o raio de transição r , isto é, a distância mínima do ponto para que o drone se movimente para o próximo ponto do conjunto.

Na Figura 3.2 observa-se uma trajetória figurativa do VANT, cujos pontos planejados são w_1 , w_2 e w_3 . Considerando o drone na posição inicial p_1 , a posição de referência transferida para o controlador é um ponto do segmento entre w_1 e w_2 a uma distância máxima d de p_1 .

Quando o drone atingir essa posição, uma nova posição de referência é informada para o controlador sobre o segmento de reta entre w_1 e w_2 . A distância entre a nova posição de referência e a posição atual do drone é d . Caso a distância entre a posição do drone e o ponto desejado w_2 for menor que r , é considerado que o veículo alcançou o ponto w_2 . Reinicia-se o ciclo para atingir o ponto w_3 . O raio r é nomeado raio de transição.

Figura 3.2: Diagrama de uma trajetória simbólica do drone sobre as coordenadas w_1 , w_2 e w_3 . O VANT está inicialmente a uma distância d da reta que conecta os pontos w_1 a w_2 . O raio de transição é representado por r .



Fonte: Autoria própria

Considerando a posição inicial do drone como p_2 , é possível verificar, observando a Figura 3.2, que a distância até o segmento w_1-w_2 é maior do que ao segmento w_2-w_3 . Nessa situação, o drone está localizado no hiperplano (região em azul) e considera-se que o drone já atingiu o ponto w_2 . Em seguida, reinicia-se o ciclo tendo em vista o próximo ponto do conjunto w , ou seja, w_3 .

Os modelos *Competition*, *Hover* e *Waypoint Follower* são os únicos que iniciam as variáveis no *workspace*. Eles são aptos a montar uma estrutura de pastas específica para a geração automática de código, dado que estão disponíveis, além do formato de diagrama de blocos *.sltx*, em arquivos compactados *.zip*. Posto que os demais modelos disponibilizam apenas do diagrama *.slx*, para executá-los é necessário previamente compilar e montar o diretório de um dos três modelos compactados.

Para o estudo documentado nesse trabalho, foi utilizada a versão MATLAB 2019b e o modelo *Hover*. Por padrão, o método numérico utilizado pelo Simulink é Bogacki-Shampine em passos fixos de 0,005 segundos.

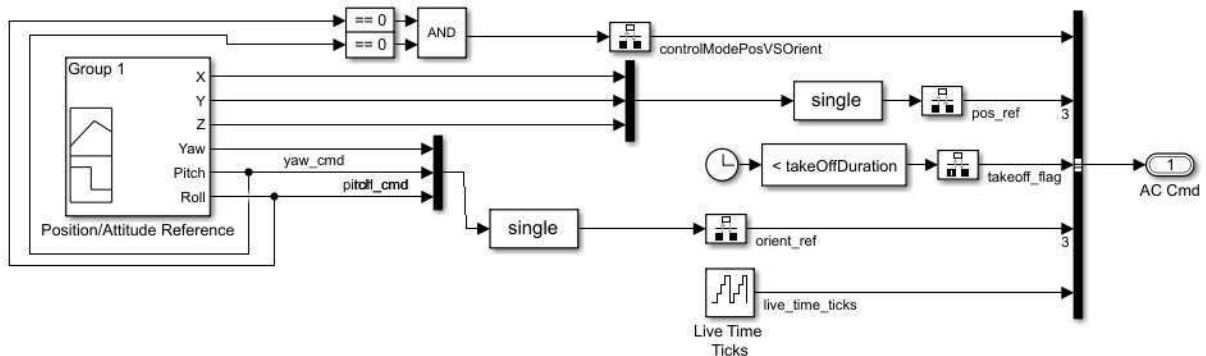
A seguir, são apresentados os quatro subsistemas que compõem o modelo do Parrot Mambo no Simulink, em concordância com os blocos ilustrados na Figura 3.1.

3.2.1 Comandos de voo

O bloco de comandos de voo (*Flight Commands*) reúne os valores de referência de posição (X, Y e Z) e orientação (arfagem, rolagem e guinada) em um barramento *AC Cmd*, o qual é encaminhado para o controlador.

Essas variáveis podem ser informadas de quatro maneiras: modificando os parâmetros do bloco de referência de atitude e posição, informando por um *joystick* ou por planilhas em formato *.mat* ou *.xlsx*. No modelo *Hover* é empregado o bloco de referência de atitude e posição *Position/Attitude Reference* ilustrado na Figura 3.3.

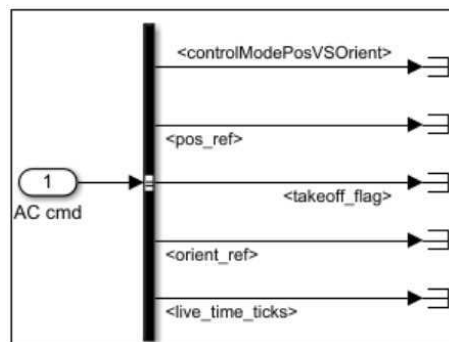
Figura 3.3: Bloco de comandos de voo (*Flight Commands*) e referência de atitude e posição (*Position/Attitude Reference*) no modelo *Hover*.



Fonte: MATHWORKS[11]

Observa-se os sinais que compõem o barramento *AC Cmd* na Figura 3.4. O booleano *controlModePosVSOrient* indica se foram informados a orientação de arfagem e guinada no vetor *orient_ref*. Caso positivo, é ignorada as referência de posição no sinal *pos_ref*.

Figura 3.4: Barramento do sinal *AC Cmd* saída do bloco de comandos de voo *Flight Commands*.



Fon200te: MATHWORKS[11]

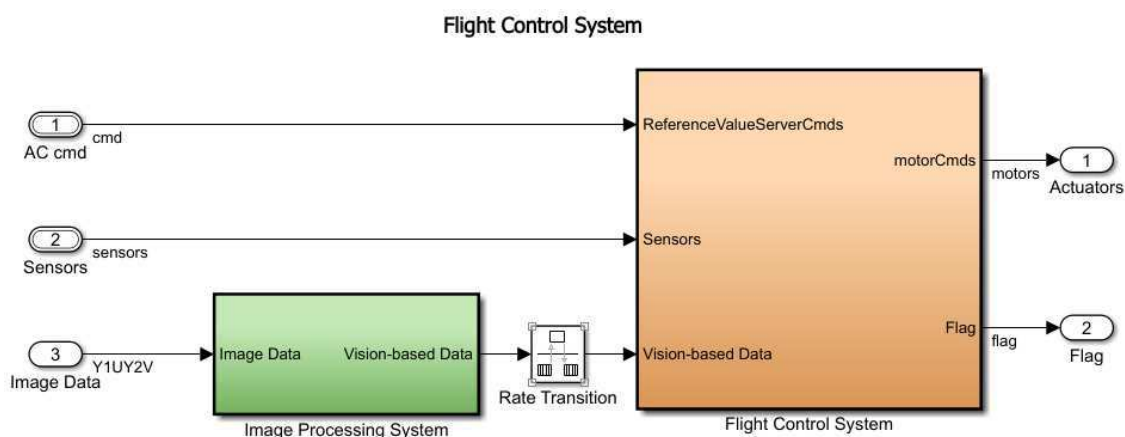
Por sua vez, o bit *takeoff_flag* comunica ao controlador que o drone está em decolagem. Há uma pausa de três segundos após a validação desse sinal como verdadeiro. Nesse intervalo, é possível implementar um algoritmo pré-vo. A última variável *live_time_ticks* representa o contador do controlador. Por padrão, ele retorna um pulso a cada 5 milissegundos, ou seja, a 200Hz.

3.2.2 Sistema de controle

No subsistema de controle de voo (*Flight Control System - Code Generation*), são fornecidos ao controlador os dados do processamento de imagem, o barramento de referência *AC Cmd* e as saídas dos sensores do drone. A saída desse subsistema são os sinais de comando aos motores, ou seja, os sinais PWM. Apenas o bloco *Flight Control System - Code Generation* é cross-compilado para o processador ARM embarcado.

O módulo de sistema de controle de voo é composto por dois blocos: *Flight Control System* e *Image Processing System*, conforme a Figura 3.5. O bloco de processamento de imagem *Image Processing System* interpreta a imagem capturada pela câmera inferior. Esse módulo identifica padrões na superfície e produz um sinal de controle (*Vision-based Data*) para o planejamento de trajetória.

Figura 3.5: Subsistema de controle de voo *Flight Control System - Code Generation*.



Fonte: MATHWORKS[11]

Por sua vez, o bloco de controle *Flight Control System* gera os sinais de controle dos motores (*Actuators*) e os bits de erro (*Flags*). Dentro deste bloco estão contidos os sistemas de planejamento da trajetória (*Path Planning*), o estimador de estados (*State Estimator*)

e o controlador (*Controller*). O diagrama de blocos do módulo *Flight Control System* está ilustrado na Figura 3.6.

Controle de trajetória

O bloco de controle de trajetória (*Path Planning*) calcula o valor da referência de posição a ser transmitido ao controlador. Como o diagrama de blocos é generalizado para diferentes aplicações, estão pré-modelados três possibilidades: por valores de referência p_x^* , p_y^* e p_z^* (*xValue*, *yValue*, *zValue*), por realimentação do espaço de estados (*EstimatedVal*) ou pelo processamento da imagem (*Vison-based Data*).

No modelo *Hover*, a pose de referência é fixa a 1,1 metro da altitude inicial do drone, conforme ilustrado na Figura 3.7. O valor de referência transmitido para o controlador é renomeado *UpdatedReferenceCmds*.

Estimador de estados

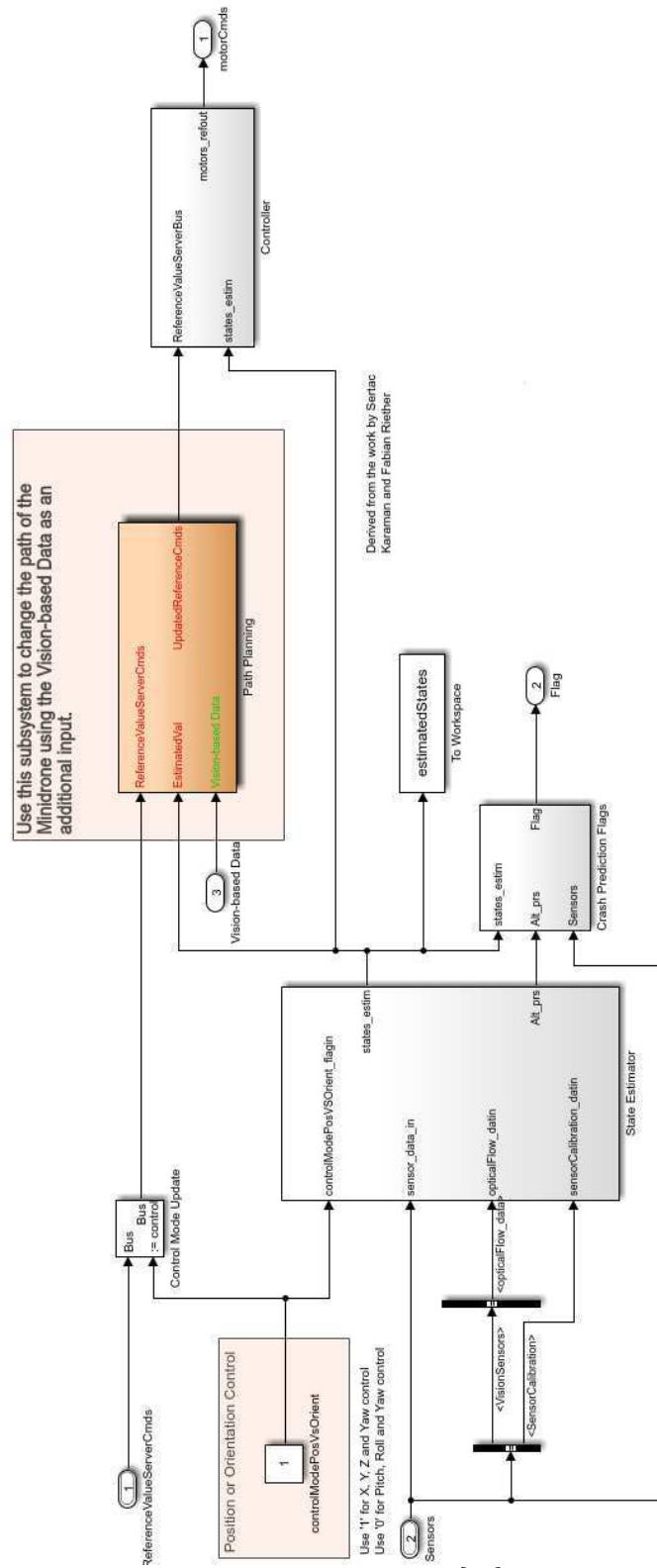
O bloco de estimação de estados *State Estimator* calibra os dados recebidos dos sensores pelo barramento *sensor_data_in*. A partir dos valores corrigidos, é estimada a posição do drone *state_estim* em relação aos três eixos. O diagrama de blocos desse módulo está ilustrado na Figura 3.8.

A calibração é realizada no bloco *SensorPreprocessing*. Nessa etapa, o barramento *sensor_data_in* é renomeado *HAL* (camada de abstração de *hardware* – *Hardware Abstraction Layer*). Observa-se os sinais que compõem esse barramento na Figura 3.9.

As variáveis *HAL_acc_SI*, *HAL_gyro_SI*, *HAL_pressure_SI*, *HAL_ultrasound_SI* e *HAL_vbat_SI* representam, respectivamente, os dados adquiridos do acelerômetro, giroscópio, barômetro, ultrassom e controlador de bateria. Em *HAL_vbat_SI*, os sinais *vbat_V* e *vbat_percentage* indicam o valor em volts e em porcentagem da bateria restante. As demais variáveis são expressas em unidades do SI. Os dados do acelerômetro e giroscópio são unidos em um único barramento denominado *IMU* (unidade de medida inercial, ou seja, *Inertial Measurement Unit*).

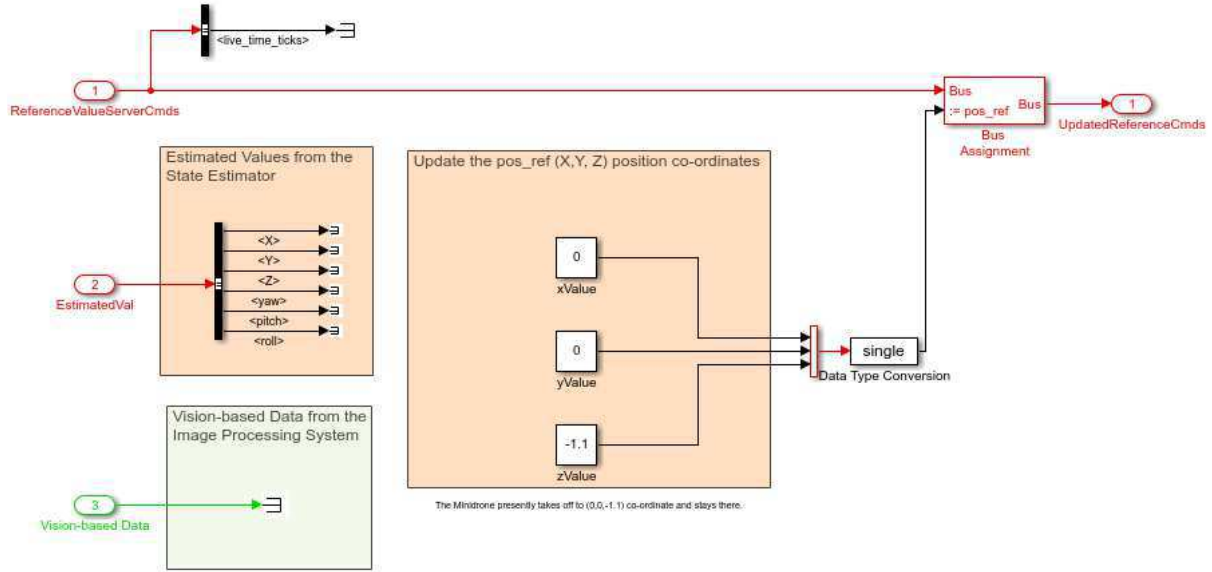
O estimador de altitude *EstimatorAltitude* estima os estados \hat{p}_z e $\hat{\dot{p}}_z$ do drone com um filtro de Kalman, cuja entradas são a distância p_z até o solo capturada pelo sensor ultrassô-

Figura 3.6: Módulo *Flight Control System* do subsistema de controle de voo *Flight Control System - Code Generation*.



Fonte: MATHWORKS[11]

Figura 3.7: Módulo de controle de trajetória *Path Planning* do subsistema de controle de voo *Flight Control System - Code Generation*.



Fonte: MATHWORKS[11]

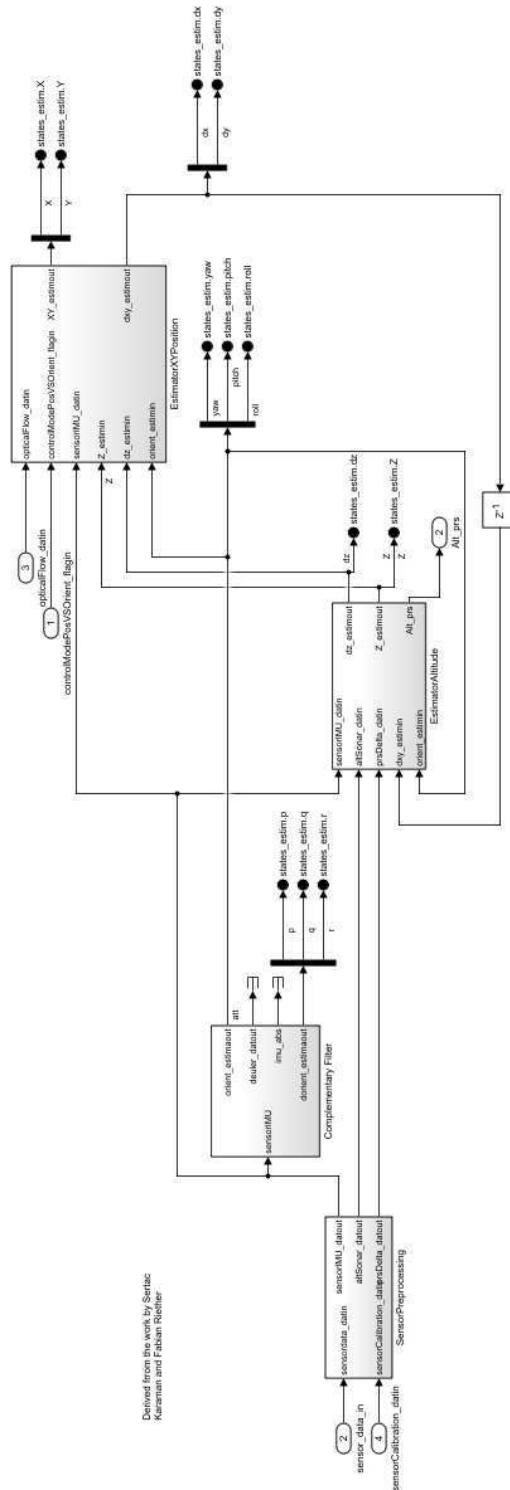
nico e a aceleração \ddot{p}_z do corpo adquirida pelo IMU. Em concordância com a representação matemática do filtro de Kalman apresentada no Capítulo 2, as equações do estimador de altitude modelado no diagrama Simulink estão apresentadas na Tabela 3.1.

Já o estimador de posição XY *EstimatorXYPosition* estima os estados \hat{p}_x e \hat{p}_y com outro filtro de Kalman, cuja entradas são as velocidades \dot{p}_x e \dot{p}_y calculadas pelo fluxo ótico e as acelerações \ddot{p}_x e \ddot{p}_y adquiridas pelo IMU. Essas estimativas de velocidades \hat{p}_x e \hat{p}_y são integradas no tempo discreto para calcular a posição estimada \hat{p}_x e \hat{p}_y . Conforme a representação matemática do filtro de Kalman apresentada no Capítulo 2, as equações do estimador de posição XY modelado no diagrama Simulink estão apresentadas na Tabela 3.2.

As estimativas dos estados de orientação ($\hat{\phi}$, $\hat{\phi}$, $\hat{\theta}$, $\hat{\theta}$, $\hat{\psi}$ e $\hat{\psi}$) são equivalentes as medições adquiridas pelo IMU. Previamente aos filtros, os dados capturados pelo IMU são rotacionados do sistema de coordenadas local \mathcal{F}_b do corpo para o sistema de coordenadas inercial do corpo \mathcal{F}_v .

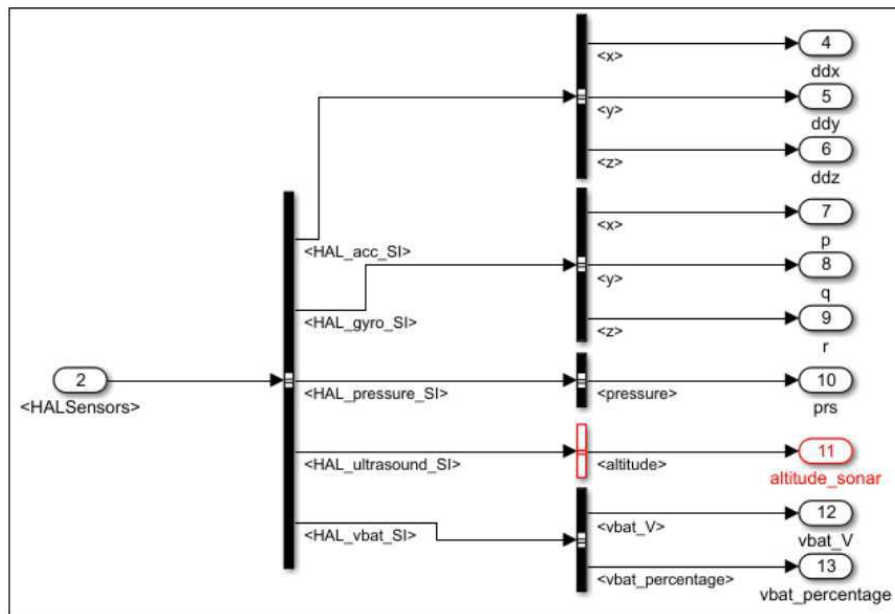
O código fonte em que são definidas as matrizes F , G , C , D , R e Q para os filtros de Kalman é discutido na Subseção 4.3.4. Já o algoritmo de fluxo ótico modelado no Simulink é detalhado no Capítulo 6. O barramento dos dados de visão é nomeado *VisionSensors* e pode ser observado na Figura 3.10.

Figura 3.8: Módulo do estimador de estados *State Estimator* do subsistema de controle de voo *Flight Control System - Code Generation*.



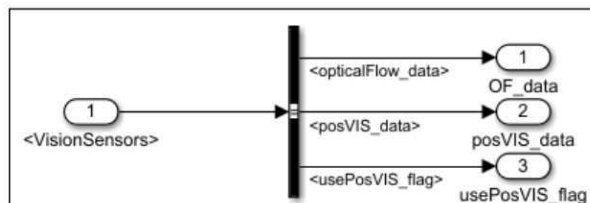
Fonte: MATHWORKS[11]

Figura 3.9: Barramento do sinal *HAL* do bloco de calibração *SensorPreprocessing* no estimador de estados *State Estimator*.



Fonte: MATHWORKS[11]

Figura 3.10: Barramento dos sensores de visão *VisionSensors* do subsistema de controle de voo *Flight Control System - Code Generation*.



Fonte: MATHWORKS[11]

Tabela 3.1: Equações e etapas do filtro de Kalman para estimar a altitude e velocidade vertical do drone a partir do sensor ultrassônico, barômetro e IMU embarcado.

Etapa	Equação	Formulação
Correção	Medição	$\hat{y}_{[n]} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{p}_z[n] \\ \dot{\hat{p}}_z[n] \end{bmatrix} + [0] \ddot{p}_z[n]$
	Ganho	$k_n = \begin{bmatrix} var\{\tilde{p}_z[n n-1]\} \\ 0 \end{bmatrix} (var\{\tilde{p}_z[n n-1]\} + 0.1)^{-1}$
Atualização	Estado	$\begin{bmatrix} \hat{p}_z[n n] \\ \dot{\hat{p}}_z[n n] \end{bmatrix} = \begin{bmatrix} \hat{p}_z[n n-1] \\ \dot{\hat{p}}_z[n n-1] \end{bmatrix} + k_n(p_z[n] - \hat{y}_{[n]})$
	Correlação	$P_{[n n]} = (I - k_n \begin{bmatrix} 1 & 0 \end{bmatrix}) \begin{bmatrix} var\{\tilde{p}_z[n n-1]\} & 0 \\ 0 & var\{\tilde{p}_z[n n-1]\} \end{bmatrix}$
Previsão	Estado	$\begin{bmatrix} \hat{p}_z[n+1 n] \\ \dot{\hat{p}}_z[n+1 n] \end{bmatrix} = \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{p}_z[n n] \\ \dot{\hat{p}}_z[n n] \end{bmatrix} + \begin{bmatrix} 0 \\ T_s \end{bmatrix} \ddot{p}_z[n n]$
	Correlação	$P_{[n+1 n]} = \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} P_{[n n]} \begin{bmatrix} 1 & 0 \\ T_s & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0.0005 \end{bmatrix}$

No barramento *VisionSensors*, a variável *opticalFlow_data* representa o valor do deslocamento horizontal do drone \dot{p}_x e \dot{p}_y e é entrada do módulo *EstimatorXYPosition*, conforme ilustrado na Figura 3.8.

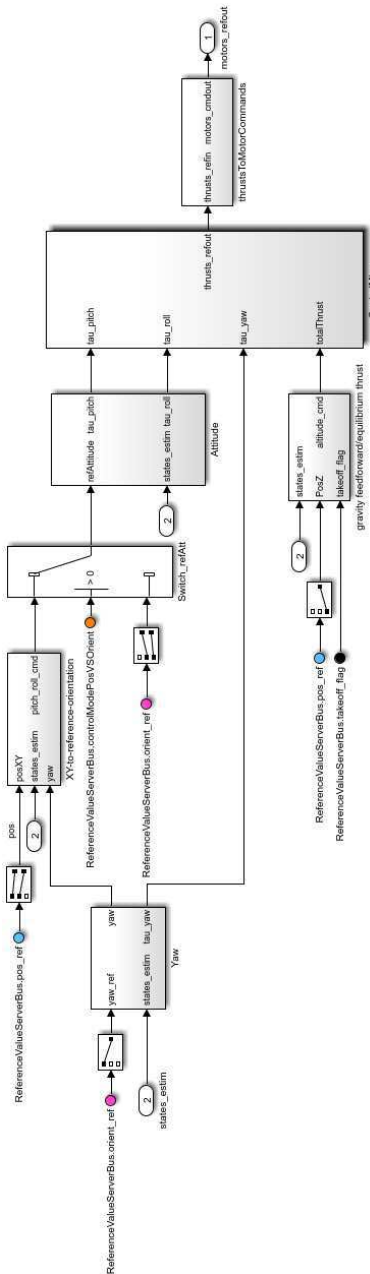
O vetor *posVIS_data* é um vetor nulo que informa a posição estimada do drone ao estimador quando o algoritmo de fluxo óptico não consegue reconstruir a posição do drone. Nessa situação, é acionado a *flag usePOSVIS_flag*, a qual indica que os dados informados pela visão computacional não são válidos.

Controlador

Internamente ao subsistema do controlador (*Controller*), estão modelados um controlador de atitude e outro controlador de altitude. O diagrama de blocos do subsistema *Controller* está ilustrado na Figura 3.11. Os sinais de controle de rotação de cada motor são computados conforme os torques τ_x , τ_y , τ_z e o empuxo vertical *totalThrust* necessários para atingir os valores de referência de p_x^* , p_y^* , p_z^* e ψ^* .

O torque *tau_yaw* (τ_z) é calculado no bloco *Yaw* com base na referência ψ^* . Já o controlador de atitude (*Attitude*) recebe os valores de referência de θ^* e ϕ^* e calcula os torques *tau_pitch* (τ_x) e *tau_roll* (τ_y) necessários para atingir a posição de referência.

Figura 3.11: Diagrama de blocos do controlador *Controller* do subsistema de controle de voo *Flight Control System - Code Generation*.



Fonte: MATHWORKS[11]

Tabela 3.2: Equações e etapas do filtro de Kalman para estimar a as velocidade do drone no plano XY a partir do fluxo óptico e IMU embarcado.

Etapa	Equação	Formulação
Correção	Medição	$\hat{y}_{[n]} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{p}_{x[n]} \\ \hat{p}_{y[n]} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \ddot{p}_{x[n]} \\ \ddot{p}_{y[n]} \end{bmatrix}$
	Ganho	$k_n = \begin{bmatrix} var\{\tilde{p}_{x[n n-1]}\} & 0 \\ 0 & var\{\tilde{p}_{y[n n-1]}\} \end{bmatrix} \left(\begin{bmatrix} var\{\tilde{p}_{x[n n-1]}\} & 0 \\ 0 & var\{\tilde{p}_{y[n n-1]}\} \end{bmatrix} + \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix} \right)^{-1}$
Atualização	Estado	$\begin{bmatrix} \hat{p}_{x[n n]} \\ \hat{p}_{y[n n]} \end{bmatrix} = \begin{bmatrix} \hat{p}_{x[n n-1]} \\ \hat{p}_{y[n n-1]} \end{bmatrix} + k_n \left(\begin{bmatrix} \dot{p}_{x[n]} \\ \dot{p}_{y[n]} \end{bmatrix} - \hat{y}_{[n]} \right)$
	Correlação	$P_{[n n]} = \left(I - k_n \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} var\{\tilde{p}_{x[n n-1]}\} & 0 \\ 0 & var\{\tilde{p}_{y[n n-1]}\} \end{bmatrix}$
Previsão	Estado	$\begin{bmatrix} \hat{p}_{x[n+1 n]} \\ \hat{p}_{y[n+1 n]} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{p}_{x[n n]} \\ \hat{p}_{y[n n]} \end{bmatrix} + \begin{bmatrix} T_s & 0 \\ 0 & T_s \end{bmatrix} \begin{bmatrix} \ddot{p}_{x[n]} \\ \ddot{p}_{y[n]} \end{bmatrix}$
	Correlação	$P_{[n+1 n]} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} P_{[n n]} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0.09 & 0 \\ 0 & 0.09 \end{bmatrix}$

O controlador de altitude (*gravity feedforward/equilibrium thrust*) rastreia a altitude p_z^* do corpo e computa o empuxo T vertical *altitude_cmd* necessário para atingir a posição de referência. A variável T é renomeada *totalThrust*.

O modelo oferece a possibilidade de, ao invés de rastrear as variáveis p_x^* , p_y^* , referenciar o controlador a partir dos ângulos de orientação θ^* e ϕ^* . Essa opção é informada pelo bit *controlModePosVsOrient* quando verdadeiro. Nesse caso, os ângulos de referência ϕ^* e θ^* são estimados a partir da posição XY por um controlador (bloco *XY-to-reference-orientation*). Quando o booleano *controlModePosVsOrient* é falso, a referência de orientação é o vetor *orient_ref* do barramento *AC Cmd*. A variável *controlModePosVsOrient* é externa ao controlador e pode ser encontrada no bloco *Flight Control System* ilustrado na Figura 3.6.

Em concordância com a representação matemática da lei de controle apresentada na Seção 2.3, os ganhos sintonizados para cada controlador no modelo do Simulink estão apresentadas na Tabela 3.3.

Conforme a Equação 2.29, os sinais PWM de cada atuador são resultado da multiplicação dos torques e forças computados pelos controladores por uma matriz de mixagem e uma constante que relaciona a velocidade de rotação do rotor com o *buffer* de comando do motor. Essa operação é realizada no bloco *Control Mixer* e *ThrustsToMotorCommands*, ilustrados

Tabela 3.3: Ganhos sintonizados para os controladores de atitude e altitude do Parrot Mambo no diagrama de blocos do Simulink.

Saída	K_p	K_i	K_d
$[\phi^* \ \theta^*]^T$	$[-0.24 \ 0.24]$	-	$[0.1 \ -0.1]$
$[\tau_x \ \tau_y]^T$	$[0.013 \ 0.01]$	$[0.01 \ 0.01]$	$- [0.002 \ 0.003]$
τ_z	0.004	-	-0.0012
T	0.8	0.1	-0.5

na Figura 3.11. No bloco *Control Mixer*, os torques e forças são multiplicados pelo inverso da matriz descrita na Equação 2.28. Em seguida, esse resultado é multiplicado pelo termo em evidência $[C_t \rho(\pi r^4)]^{-1}$ e pela constante $A_{\omega^2}^{PWM}$ no bloco *ThrustsToMotorCommands*. O código fonte em que são definidas as constantes C_t , C_q , l , b , k , $A_{\omega^2}^{PWM}$ e a matriz de mixagem A é discutido na Subseção 4.3.4.

3.2.3 Modelo de simulação

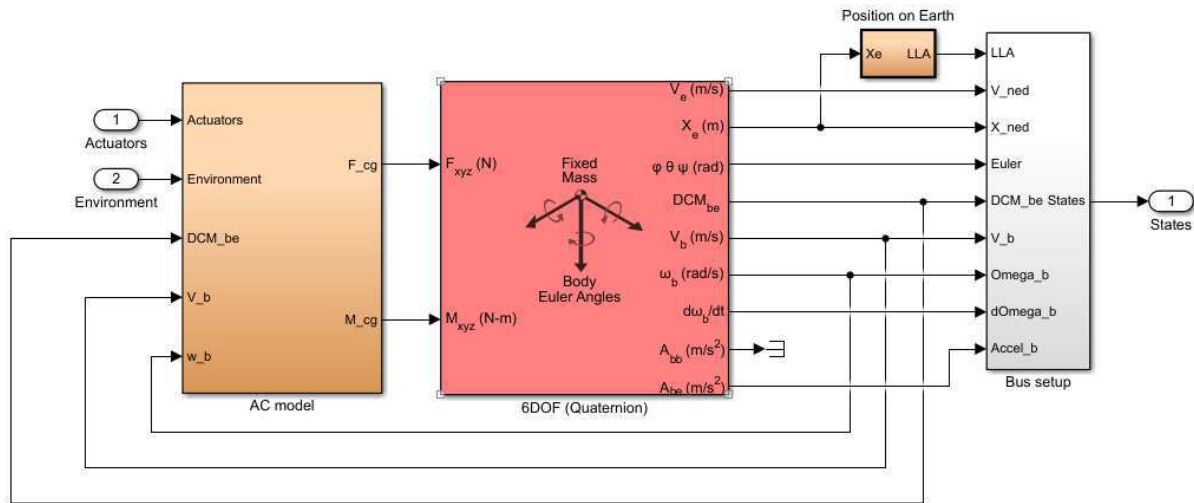
O subsistema de modelo de simulação *Simulation Model* é composto por três blocos: modelagem do VANT, modelagem do ambiente e modelagem dos sensores. Estão disponíveis blocos do modelo linear e não-linear do VANT, do modelo dinâmico e estático do ambiente e do modelo dinâmico ou *feedthrough* dos sensores. Para esse estudo, foram usados os blocos de modelagem não linear do VANT, modelagem estática do ambiente e modelagem dinâmica dos sensores.

Modelo não linear do VANT

O bloco do modelo não linear é formado pelo espaço de estados não linear (*6DOF*) e o modelo das forças e torques aplicadas sobre o corpo (*AC Model*). Esse bloco está ilustrado na Figura 3.12

Observa-se o bloco *AC Model* na Figura 3.13. As saídas F_cg e M_cg representam o total das forças e toques aplicados sobre o CG do drone, respectivamente. Enquanto F_cg é composto pela soma da força peso, força de arrasto aerodinâmico e a força empuxo vertical dos motores; M_cg é constituído pela soma dos torques dos motores.

Figura 3.12: Modelo não linear do VANT no subsistema *Simulation Model*.

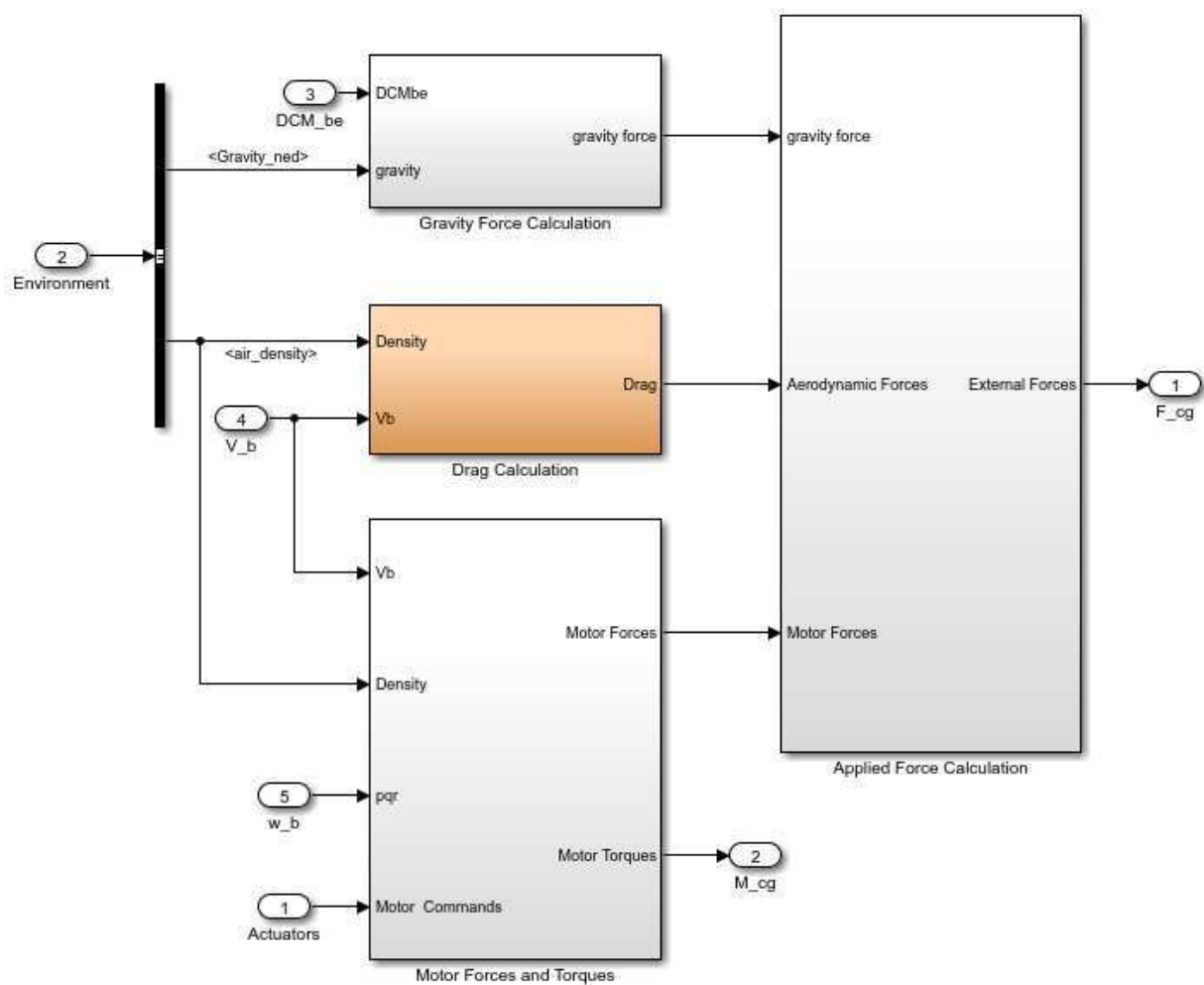


Fonte: MATHWORKS[11]

Conforme descrito na Equação 2.22, o espaço de estados possui como entrada os estados atuais, as forças e toques no centro de gravidade do drone e, como saída, o vetor dos novos estados. Conforme ilustrado na Figura 3.12, os sinais do barramento de saída *State* são:

- *LLA*, representa a matriz de conversão das posições no sistema inercial no mundo para coordenadas geodésicas (latitude, longitude e altitude). Essas coordenadas são utilizadas para localizar o drone no ambiente virtual 3D;
- $V_{ned} = [\dot{X}_e \ \dot{Y}_e \ \dot{Z}_e]$, é o vetor de velocidades no sistema inercial no mundo;
- $X_{ned} = [X_e \ Y_e \ Z_e]$, é o vetor de posições no sistema inercial no mundo;
- $Euler = [\phi \ \theta \ \psi]$, é o vetor de orientação no referencial local do corpo;
- $DCM_{be} = \mathbf{R}$, é a matriz de rotação do sistema inercial do mundo para o referencial local do corpo;
- $V_b = [\dot{X}_b \ \dot{Y}_b \ \dot{Z}_b]$, é o vetor de velocidades no sistema local do corpo;
- $\omega_b = [\dot{\phi} \ \dot{\theta} \ \dot{\psi}]$, é o vetor de velocidades angulares no sistema local do corpo;
- $d\omega_b = [\ddot{\phi} \ \ddot{\theta} \ \ddot{\psi}]$, é o vetor de acelerações angulares no sistema local do corpo;
- $A_b = [\ddot{X}_b \ \ddot{Y}_b \ \ddot{Z}_b]$, é o vetor de acelerações no sistema local do corpo em relação ao referencial inercial no mundo;

Figura 3.13: Bloco *AC Model* do modelo não linear do VANT no subsistema *Simulation Model*.

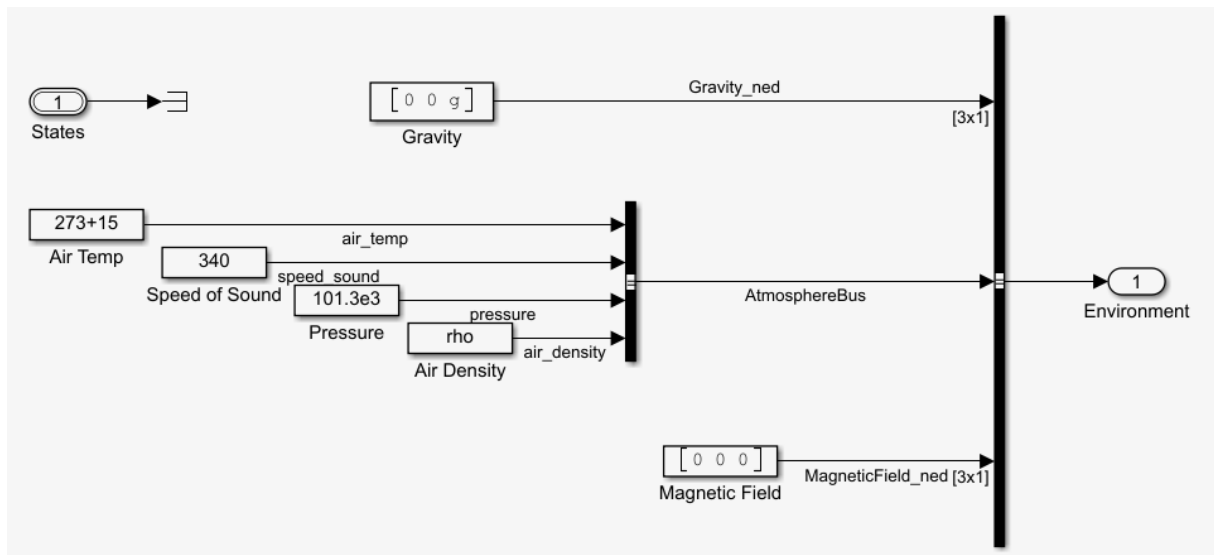


Fonte: MATHWORKS[11]

Modelo estático do ambiente

Na Figura 3.14, está apresentado o diagrama de blocos da modelagem estática do ambiente. O barramento *Environment* engloba os sinais de temperatura (*Air Temp*), velocidade do som (*Speed of Sound*), pressão (*Pressure*), densidade do ar (*Air density*), aceleração da gravidade (*Gravity*) e campo magnético (*Magnetic Field*). Todos esses valores são pré-definidos nos códigos discutidos no Capítulo 4.

Figura 3.14: Modelo estático do ambiente no subsistema *Simulation Model*.



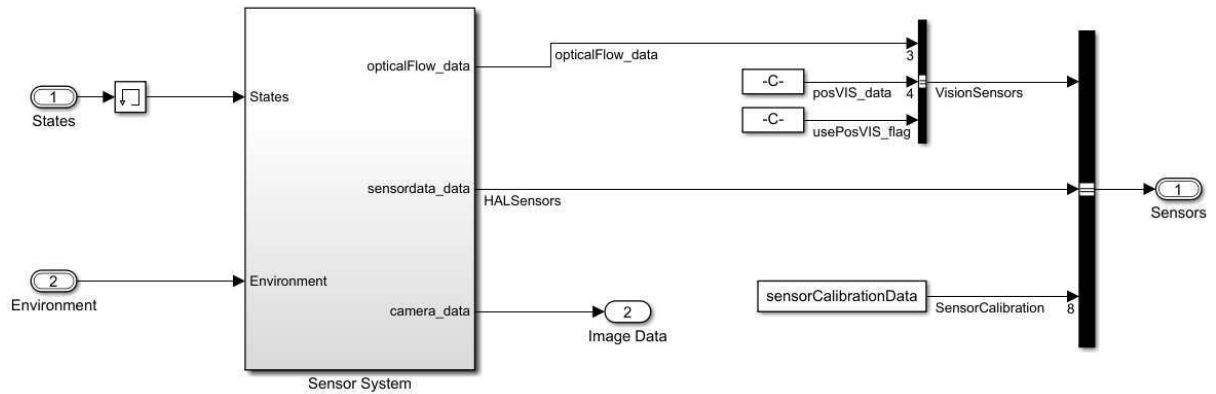
Fonte: MATHWORKS[11]

Modelo dinâmico dos sensores

O modelo dinâmico dos sensores reúne, em um único barramento, os sensores de visão (*VisionSensors*) e o barramento *HAL*. Esse bloco transfere a imagem *camera_data* para o módulo de processamento de imagem (*Image Processing System*) no sistema de controle de voo (*Flight Control System*). Essa imagem é renomeada *Image Data*, conforme ilustrado na Figura 3.15.

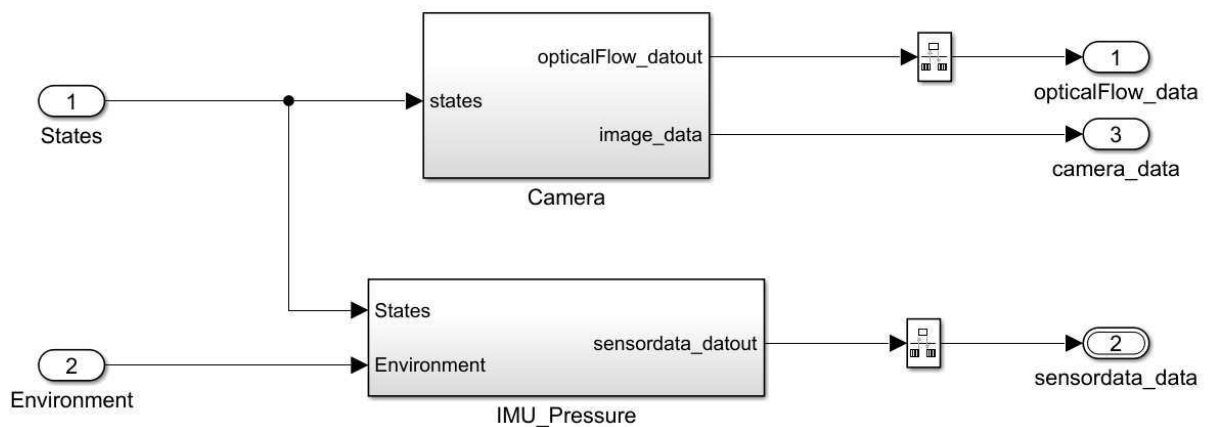
O bloco *Sensor System* compõe o barramento IMU/Barômetro e da câmera, conforme apresentado na Figura 3.16. No bloco *Camera*, está contido o algoritmo de fluxo óptico e a captura da imagem *camera_data* da câmera inferior do drone. Já o bloco *IMU_Pressure* unifica no barramento *HAL* os dados do acelerômetro, giroscópio, barômetro. Esses dados são gerados a partir da calibração dos estados com correções de BIAS e adição de variâncias.

Figura 3.15: Modelo dinâmico dos sensores no subsistema *Simulation Model*.



Fonte: MATHWORKS[11]

Figura 3.16: Bloco *Sensor System* no modelo dinâmico dos sensores do subsistema *Simulation Model*.



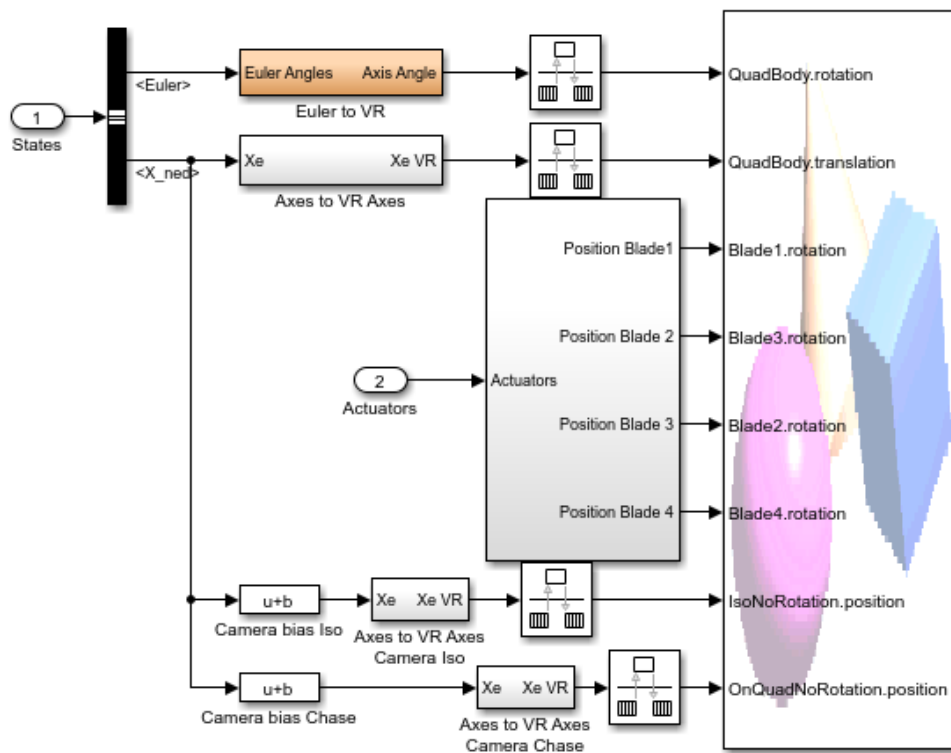
Fonte: MATHWORKS[11]

3.2.4 Visualização do Voo

O módulo de visualização do voo *Flight Visualization* é formado por dois blocos: *Simulink 3D* e *Extract Flight Instruments*. O bloco *Simulink 3D* encaminha os dados do diagrama de blocos para a representação 3D virtual. Já o bloco *Extract Flight Instruments* proporciona a visualização, em blocos *display*, das variáveis encaminhadas para o modelo 3D.

O bloco *Simulink 3D* está ilustrado na Figura 3.17. Nesse bloco, os ângulos e posições do espaço de estados 6DOF (seis graus de liberdade – *Degrees Of Freedom*) são transformados para o referencial inercial no corpo por matrizes de rotação e transformação angular, conforme discutido no Capítulo 2.

Figura 3.17: Bloco *Simulink 3D* no subsistema de visualização de voo *Flight Visualization*.



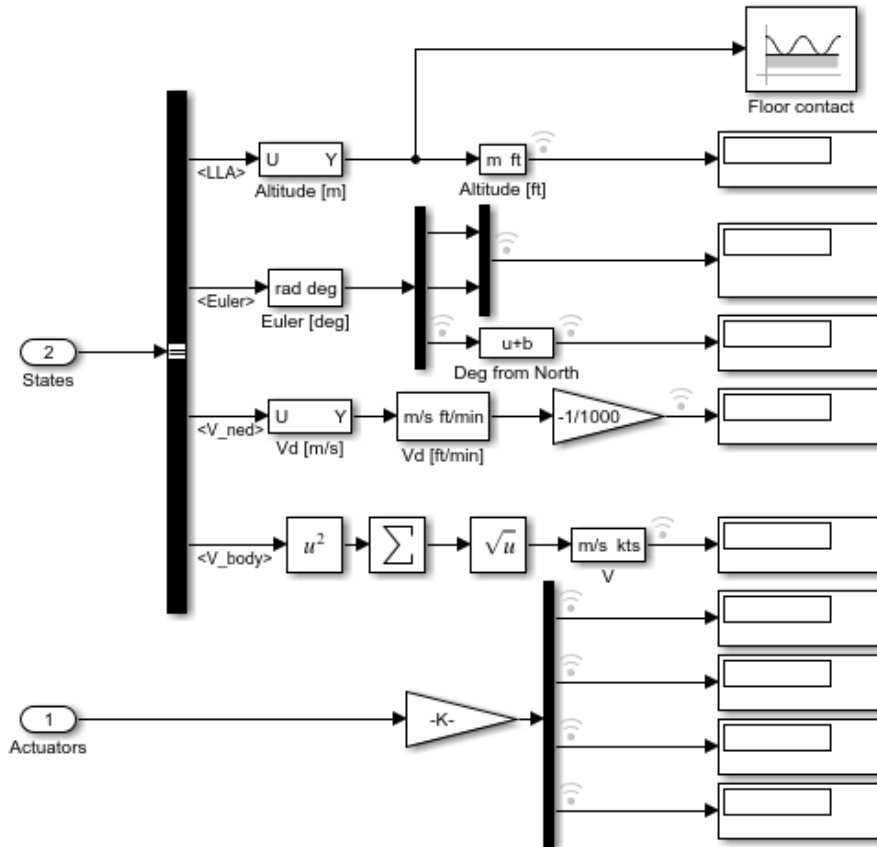
Fonte: MATHWORKS[11]

Os sinais de comando para os atuadores são convertidos para sinais de posição da hélice. Observando a Figura 3.17, é evidente que os estados no sistema de coordenadas inerciais do mundo e os sinais de comando para os atuadores são enviados para a simulação 3D.

O bloco *Extract Flight Instruments*, apresentado na Figura 3.18, expõe em *displays* a

altitude, a angulação, as velocidades de cada propulsor e a velocidade do corpo no referencial inercial do mundo e local do corpo. É possível visualizar esses dados no Simulink Data Inspector e gerar arquivos de log automaticamente.

Figura 3.18: Bloco *Extract Flight Instruments* no subsistema de visualização de voo *Flight Visualization*.

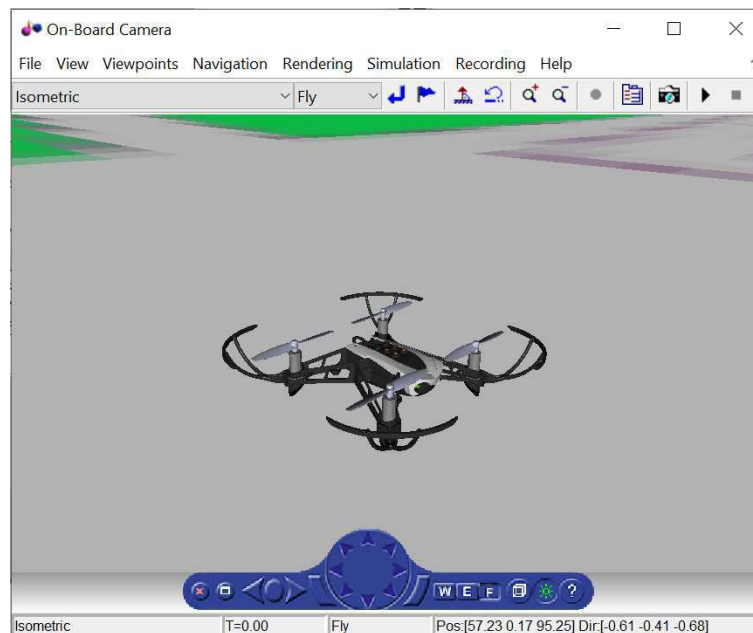


Fonte: MATHWORKS[11]

O modelo 3D simulado do Parrot Mambo está ilustrado na Figura 3.19. É possível construir ambientes mais complexos. Para tal, pode-se adicionar objetos em declaração PROTO ou pré disponíveis pelo MATLAB no cenário da simulação 3D.

Entretanto, é importante destacar que a interface do Simulink não é tão intuitiva quanto as ferramentas ROS e Gazebo, as quais são ambientes de simulação e modelagem de robôs consolidados. Dado que a Parrot disponibiliza um simulador código aberto baseado em Gazebo denominado SPHINX, essa é a ferramenta mais indicada para testes de performance do Parrot Mambo em cenários mais intrincados.

Figura 3.19: Modelo 3D do Parrot Mambo no ambiente virtual simulado a partir do diagrama de blocos do Simulink.



Fonte: MATHWORKS[11]

3.3 Simulação e implementação

No método MAD, a planta modelada é inicialmente simulada e, depois, implementada no protótipo. No Simulink, o diagrama de blocos pode ser simulado a partir do botão “Run”, na barra superior da interface. É aberta a visualização do mundo virtual, semelhante a apresentada na Figura 3.19. O modelo desenhado é executado até o tempo final de simulação, também definido na barra superior da interface.

O modelo já testado em um ambiente virtual pode ser implementado por meio do botão “Build, Run and Deploy”, junto ao botão “Run”. Após o envio do modelo para o protótipo físico, é aberta a janela de interface de voo.

3.3.1 Janela de interface de voo

A janela da interface de voo está ilustrada na Figura 3.20. Por padrão, ela é aberta automaticamente após a compilação e envio do modelo para o drone.

A variável *powerGain* é determinada pelo controle deslizante. Quando clicado, o botão *Start* inicia a execução do código compilado no sistema embarcado. A partir da decolagem,

Figura 3.20: Janela da interface de controle de voo aberta após a compilação e envio do modelo em blocos do Simulink para o drone.



Fonte: MATHWORKS[11]

pode-se clicar no botão *Stop* para finalizar o voo e produzir os arquivos de log.

É importante ressaltar que os registros só são adquiridos via portas TCP após o pouso. Dessa forma, a captura de informações do drone em tempo real pela interface do Simulink é restrita ao modelo *External Mode*. Nos demais modelos, é possível observar, em tempo real, apenas os sensores do drone simulado.

Maiores detalhes sobre a conexão ao drone ou as etapas de comunicação com a interface do Simulink são pormenorizadas no guia do pacote de suporte à minidrones Parrot.

3.4 Considerações finais

Nesse capítulo, foi apresentado como o modelo dinâmico do quadricóptero matematicamente descrito no Capítulo 3 é representado em diagrama de blocos no Simulink. Foi apresentado o ambiente virtual de simulação e a interface da aplicação MAD com o usuário. Também foram detalhados os barramentos de comunicação entre a planta simulada e o controlador, o qual é compilado e exportado para o protótipo físico. O próximo capítulo trata desse fluxo de geração de código para o Parrot Mambo, ou seja, dos algoritmos executados para a geração e implementação automática do controlador modelado no drone real.

Capítulo 4

A geração automática de código

Este capítulo trata da geração de código automática para o modelo em diagrama de blocos do Parrot Mambo no Simulink. Primeiramente, é apresentado o processo de compilação cruzada do diagrama de blocos e, em seguida, como o código gerado é implementado e executado no sistema embarcado.

4.1 Simulink Coder e Embedded Coder

As ferramentas empregadas para a geração de código automática no pacote de suporte a minidrones Parrot são Simulink Coder e Embedded Coder. Enquanto aquele produz o código em linguagem C correspondente aos blocos do diagrama Simulink, esse último compila o código em C para a plataforma embarcada.

Tanto os códigos em linguagem C quanto a arquitetura do processador e dos barramentos devem ser informadas ao Embedded Coder pelo pacote. Sendo assim, são apresentadas a seguir as estruturas do pacote de suporte a minidrones Parrot e do projeto *Hover* no Simulink.

4.2 Código fonte do pacote de suporte a drones Parrot

O código fonte do pacote de suporte para minidrones Parrot está localizado no caminho `<MATLAB>/SupportPackages/R2019b/toolbox/target/supportpackages/parrot`, para `<MATLAB>` a pasta de instalação (por exemplo, `C:/ProgramData/MATLAB`). Esse diretório é a raiz

da árvore de arquivos do Embedded Coder e Simulink Coder.

4.2.1 Compilação cruzada

Na pasta raiz estão localizados os arquivos de configuração *arm_linux_drone.m* e *rtwTargetInfo.m*. Em *arm_linux_drone.m* são informadas as configurações da *toolchain* e diretivas de compilação e linkagem para o processador ARM. Esses parâmetros da *toolchain* são:

- *Build Artifact*: especifica o tipo de *makefile* que irá automatizar do processo de compilação do código embarcado. No pacote, é empregado *gmake*;
- *Supported Languages*: indica as linguagens do código gerado a partir do diagrama de blocos. No pacote, é definido como C e C++;
- *Linker* e compilador: define as ferramentas para linkagem e compilação cruzada. Tendo em vista que o controlador embarcado possui arquitetura ARM, o *linker* adotado foi *arm-none-linux-gnueabi-gcc*.

Em *rtwTargetInfo.m* é criada a *toolchain* conforme as arquiteturas do sistema embarcado e do sistema nativo executando o MATLAB. A sigla *rtw* provém de *Real-Time Workshop*, denominação anterior do Simulink Coder.

4.2.2 Comunicação com o protótipo

No diretório `<RAIZ>/+codertarget/+parrot/+internal` estão os códigos de comunicação do drone com a IPA do MATLAB. Cada um dos arquivos *.m* possui uma versão binária privada *.p*. Esses arquivos são:

- *DroneConnect.m*: descreve as funções de criar e apagar o canal de comunicação *telnet* com o drone. Nesse arquivo estão definidas as funções para escrever e receber os dados do drone via portas TCP/IP. Um bit de erro é ativado após 10 segundos caso não haja resposta do protocolo, isto é, o *handshake*;
- *ParrotConstants.m*: enumera os drones suportados pelo pacote (Parrot Mambo e Rolling Spider), os endereços bluetooth e as portas de comunicação. Para o Mambo

e Rolling Spider, os IPs são 192.168.3.1 e 192.168.3.5, respectivamente. A porta TCP/IP para ambos é numerada 12391. Além disso, é definido como 192.168.3.2 o endereço do servidor, ou seja, do computador que executa o MATLAB;

- *PostFlightAnalysis.m*: plota em gráficos os dados de trajetória, velocidade, altitude adquiridos dos sensores. Esses valores são armazenados por um arquivo *.mat* durante o voo. Após o pouso, essa planilha de dados é enviada para o MATLAB via TCP. Para que o arquivo *.mat* seja produzido, é necessário previamente entrar com a instrução `set_param('flightControlSystem', 'MatFileLogging', 'on')` no terminal de comando do MATLAB.
- *Telnet.m*: configura o tamanho do *buffer* de entrada e saída da comunicação *telnet* com o drone. São definidos a resposta esperada para o *handshake* e o tempo máximo de espera (*timeout*) pela resposta. Por fim, esse código descreve as funções para abrir ou fechar a conexão *telnet* e enviar ou receber dados por esse canal;
- *Utility.m*: apresenta as funções de conexão com o ponto de acesso bluetooth do drone. Primeiramente, tenta-se a conexão com o último endereço MAC conectado. Caso não seja possível, são buscados os endereços bluetooth de drones Rolling Spider e Mambo, nessa ordem. Caso conectado, são validados o nome, o endereço bluetooth e a posição de escrita de memória. Se houver erro, um bit de aviso é lançado para a interface de voo no Simulink;
- *addCompilerPath.m*: adiciona o caminho do compilador e das bibliotecas às variáveis de ambiente no MATLAB;
- *getCodeSourceryRoot.m*: especifica o diretório de instalação do compilador ARM GNU C++;
- *getPARROTBaseRoot.m*: retorna o caminho do diretório raiz da *toolbox* de suporte a *hardwares* Parrot, isto é, `<MATLAB>/SupportPackages/R2019b/toolbox/`;
- *getSpPkgRootDir.m*: retorna o caminho do diretório raiz `<RAIZ>` do pacote de suporte a minidrones Parrot;

- *uiVisibilityCallback.m*: configura se a interface de voo apresentada na Subseção 3.3.1 é aberta após o clique do botão “*Build, Load and Run*”;
- *loadAndRun.m*: define a lógica de envio e execução do modelo compilado para a arquitetura ARM no drone. Esse modelo compilado é referido como objeto compilado no decorrer deste estudo. O algoritmo *loadAndRun.m* é executado quando clicado o botão *Start* na interface de voo apresentada na Subseção 3.3.1;
- *onAfterCodeGenHook.m*: declara os bits de aviso que são alocados no sistema embarcado. Os eventos monitorados e o nome das respectivas *flags* são
 - Presença do bloco *Image Processing System* de processamento de imagem no modelo Simulink (`MW_HAS_IMAGE_PROCESSING`);
 - Cálculo válido do fluxo óptico (`MW_HAS_OPTICALFLOW_LOGIC`);
 - Pouso do quadricóptero (`MW_HAS_LAND_DRONE`);
 - Geração de logs (`MW_LOGGING`).

Nesse arquivo são nomeados os dos canais da imagem YUV e RGB e criada uma variável para representar fim do tempo de simulação (`STOP_TIME`);

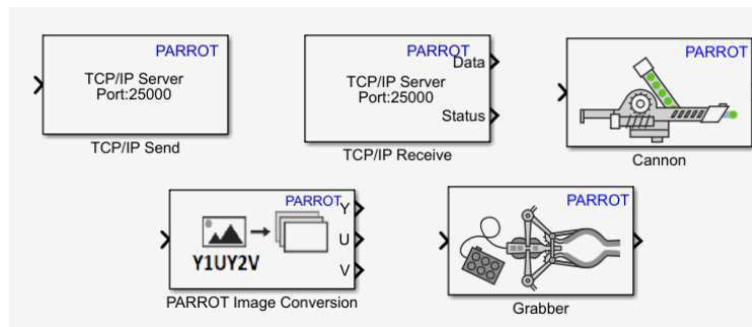
- *parrotio.m*: contém propriedades privadas do subsistema *Flight Control System*, como as etapas do controlador e os comandos hexadecimais para a porta TCP do drone. Essas etapas estão descritas na Subseção 4.2.7. Inclui as funções de *takeoff*; de leitura e escrita dos estados e sensores e de captura do registro de voo *.mat* e logs via TCP;
- *processImage.m*: converte a imagem simulada em YUV para RGB;
- *parrotminidrone.m*: define a classe do objeto *parrotminidrone*. As propriedades da classe são os valores de referência para ϕ , θ , ψ e z ; a constante de ganho dos motores *powerGain* e o endereço IP bluetooth *BluetoothIPAddress*. Esse código apresenta as funções de início e fim de voo chamadas na janela de interface de voo ou por comandos ASCII do teclado. Esses caracteres e suas respectivas respostas são:
 - “d” e “a” para acréscimo e redução de 0,05 graus em ϕ , respectivamente;

- “w” e “s” para acréscimos e redução de 0,05 graus em θ , respectivamente;
 - “j” e “l” para acréscimos e redução de 0,2 graus em ψ , respectivamente;
 - “i” e “k” para movimentações de menos e mais 0,2 metros na altitude, respectivamente;
 - “y” e “h” para movimentações de menos e mais 0,6 metros na altitude, respectivamente;
 - “r” para alterar a posição de referência para a origem do sistema de coordenadas inercial no mundo;
 - “e” para abortar o voo.
- *Callbacks*: caso um bloco específico esteja no diagrama de blocos, os códigos dessa categoria verificam e implantam uma lógica especial. Esses códigos são:
 - *cannonCallback.m*: indica erro se o acessório de canhão de bolas está sendo usado simultaneamente ao acessório de garras;
 - *checkImageBlocks.m*: verifica se o processamento de imagens é utilizado para o controle de trajetória. Caso positivo, o código ativa a propriedade de *multitasking*, a qual permite a execução da *thread* de 60Hz do fluxo óptico e de captura de imagens em paralelo à *thread* de 200Hz do controlador;
 - *fcsSubsystemCallback.m*: informa ao subsistema controle de voo (*Flight Control System – fcs*) que deve-se abortar o voo caso o algoritmo de fluxo óptico não consiga reconstruir o deslocamento do drone (*usePOSVIS_flag = 1*);
 - *hostKeyPressCallback.m*: retorna erro caso dois caracteres ASCII sejam comandados simultaneamente no modelo *Keyboard Control*;
 - *imageFormatCallback.m*, *ipSubsystemCallback* e *inportCallback*: verifica se há, no modelo, algum bloco *Camera*, *Image Processing Subsystem* e *In-Port* (no modelo *Vision*). Caso positivo, é elevado o bit `MW_HAS_IMAGE_PROCESSING` em *onAfterCodeGenHook.m*, o qual verificará se as *threads* de fluxo óptico e captura de imagens foram criadas.

4.2.3 Blocos acessórios

O diretório `<RAIZ>/+parrot` contém os códigos dos blocos Simulink específicos do pacote Parrot ilustrados na Figura 4.1. Os blocos *Grabber*, *Cannon*, *Image Conversion*, *TCP Receive* e *TCP Send* são compilados juntamente com o subsistema de controle de voo. Dessa forma, os parâmetros das portas de comunicação, tamanho das imagens ou comando do canhão e garra são atualizados em cada geração de código. Para tanto, é necessário que os novos valores informados na interface do Simulink sejam transferidos para os códigos fontes em C. Essa transmissão é realizada pelos códigos *Cannon.m*, *Grabber.m*, *ImageProcess.m*, *TCPReceive.m* e *TCPSend.m*.

Figura 4.1: Blocos acessórios disponíveis pelo pacote de suporte a minidrones Parrot.



Fonte: MATHWORKS[11]

Esses blocos são mapeados para a respectiva biblioteca no Simulink pelos arquivos *getBlockHelpMapNameAndPath.m* e *sblocks.m* localizados no diretório `<RAIZ>/blocks`. Enquanto esse último define o nome e a hierarquia da biblioteca *parrotlib.slx*, aquele mapeia os códigos fontes *Grabber*, *Cannon*, *Image Processing*, *TCP Receive* e *TCP Send* para os respectivos blocos criado na interface do Simulink.

4.2.4 Análise da performance

Por meio da GAC, é possível analisar a performance temporal das três *threads* que executam no drone: a do controlador (*control*), a do fluxo óptico (*optical flow – OF*) e a do processamento de imagem (*vision*). A contagem do tempo é realizada por um objeto *ptimer* (*profiling timer*), cujas funções-membro escrevem tabelas de tempo em arquivos *.txt*.

Os códigos que definem essa classe estão localizados no diretório `<RAIZ>/+parrot/+util`.

Eles são:

- *downloadPtimes.m*, que recupera arquivos de texto das tabelas temporais via FTP;
- *importPtimes.m*, o qual importa os dados dos arquivos de texto como matrizes para o *workspace* do MATLAB;
- *getInOutTimes.m*, que configura o formato das matrizes temporais. Essas matrizes armazenam os instantes de início e fim que cada *thread* ocupa o processador e são criadas no código *ptimer.m*;
- *PTimesAnalyzer.m*, responsável por plotar os dados de tempo de cada *thread* em gráficos.

4.2.5 Modelos e recursos dos diagrama de blocos

O diretório `<RAIZ>/parrotexamples` contém os modelos Simulink *Getting Started*, *Communication*, *External Mode*, *Keyboard Control*, *Vision*, *Competition* e *Waypoint Follower* apresentados na Seção 3.2. O modelo *Hover* está situado no diretório `<RAIZ>/templates`. Nas pastas `<RAIZ>/parrotexamples/html` e `<RAIZ>/resources` estão presentes as imagens dos tutoriais apresentados no guia do pacote.

4.2.6 Atualização do *firmware* nativo

Antes da execução e implementação do diagrama de blocos no drone real, o pacote para minidrones Parrot guia o usuário por uma interface para substituir o *firmware* nativo por um adaptado. Esse novo *firmware* apresenta novas portas de comunicação TCP em relação ao *firmware* original. No diretório `<RAIZ>/+codertarget/+parrot/+setup` estão os códigos fechados pre compilados dessa interface de instalação.

Tendo em vista a necessidade de conectar o drone via USB nessa etapa, supõe-se que a substituição do *firmware* ocorra de modo semelhante a instalação do sistema operacional nativo. Logo, presume-se que essa interface envia via USB um arquivo *.plf* para a o diretório raiz do drone. Em seguida, um *script* embarcado verifica a existência do arquivo de atualização no sistema e troca o *firmware*.

No diretório `<RAIZ>/lib` estão os arquivos copiados para o drone via USB durante a atualização de *firmware*. Em `<RAIZ>/lib/EDUfirmwareSYS` há os arquivos *.plf* para os drones Mambo e Rolling Spider. Ainda há, em `<RAIZ>/lib/EDUfirmwareFILES`, os scripts bash (*MamboFlight.sh* e *dragon-prog*). Esses arquivos são detalhados no Capítulo 5.

4.2.7 Código fonte em linguagem C

O diretório `<RAIZ>/include` inclui os cabeçalhos *.h* dos códigos fontes em linguagem C localizados em `<RAIZ>/src`. É importante compreender que esses programas são executados no drone real. As arquivos em linguagem C são:

- *controlCommand.c*: descreve as respostas do sistema embarcado no drone aos comandos recebidos via TCP/IP do servidor. Os eventos reconhecidos e suas respectivas respostas são:
 1. Pedido de leitura dos sensores
 - São enviados os dados do barramento *HAL* do protótipo físico para o servidor que está executando o Simulink;
 2. Recebimento do comando de decolagem
 - As variáveis internas do drone nomeadas *flightDuration* e *throttle* são atualizadas com os valores de *STOP_TIME* e *powerGain* da interface Simulink, respectivamente. É levantada a flag *isMotorOn*. Após, é enviada uma autenticação (*acknowledge*) à porta remetente;
 3. Recebimento do comando de pouso
 - É desligada a flag *isMotorOn*. Após, é enviada uma autenticação (*acknowledge*) à porta remetente;
 4. Recebimento do comando de parada
 - É desligada a flag *isMotorOn* e *run_flag*. Após, é enviada uma autenticação (*acknowledge*) à porta remetente;
 5. Recebimento das posição e orientação de referência

- São atualizados os valores de referência da memória do sistema embarcado lida pelo controlador. A resposta é autenticada pela porta remetente do servidor;

6. Pedido de captura de imagem

- A flag `capture` da *thread* de visão responsável pela captura de imagens é levantada;
- *mw_extrathreads.c*: esse código cria e define as prioridades das *threads*. Quando no modelo *Keyboard Control*, é criada a *thread* para escutar a interface do teclado do servidor. Já no modelo *External Mode*, é adicionada uma *thread* para enviar ao servidor os valores dos sensores. A prioridade máxima sempre é atribuída a *thread* de controle;
- *ptimer.c*: cria, abre e escreve em arquivos o intervalo de tempo em microsegundos para a execução de cada *thread*. Os novos arquivos são nomeados `/tmp/edu/ptimes/pt_<NOME DA THREAD>.txt` e escritos dentro do sistema operacional do drone. As funções declaradas nesse código são chamadas nos arquivos *rsedu_control.c*, *rsedu_of.c* e *rsedu_image.c* para a análise, respectivamente, da *thread* de controle, do fluxo óptico e da captura de imagens;
- *rsedu_image.c*: preenche os *buffers* YUV ou RGB do MATLAB conforme as variáveis alocadas em *onAfterCodeGenHook.m*. Esse cálculo é realizado por hardware via operações de soma e deslocamentos de bit;
- *rsedu_of.c*: esse código recebe as estimações de velocidade do fluxo óptico implementado por *hardware* no drone e transfere-as para uma fila. Essa fila está disponível para o controlador e é atualizada em uma frequência de 60Hz;
- *rsedu_vis.c*: cria e executa a *thread* de visão computacional caso a flag `capture` esteja ativa. A imagem capturada pelo drone é enviada para o Simulink via TCP. Até que a imagem seja processada e o novo sinal de controle seja produzido, a *thread* de visão hiberna.

- *rsedu_control.c*: responsável pelo controle de estabilidade e medição dos sensores. Define as respostas dos controlador aos os bits de decolagem, pouso ou processamento de imagem modificados nos demais arquivos C. Nele, são descritos quatro estágios de voo:

1. Configuração das *threads*

- (a) Aguardo da conexão com o servidor;
- (b) Alocação das variáveis dos barramentos *HAL* de sensores, de comandos dos motores, de comandos do teclado e dos sensores de visão na memória do servidor;
- (c) Início do rastreo da performance temporal pelo objeto *ptimer*;
- (d) Criação da planilha *.mat*, caso ativa a produção de log;
- (e) Criação da *thread* de controle, início do temporizador e conexão com a *thread* de fluxo óptico;
- (f) Criação da *thread* de visão, caso exista processamento de imagem no modelo compilado;
- (g) Início da comunicação entre as *threads* por meio do renomeio de variáveis;
- (h) Criação da fila de fluxo óptico;
- (i) Espera pelo comando de decolagem;
- (j) Caso a flag `isMotorOn` esteja baixa, são transmitidas a duração do voo e ganho dos motores para o controlador.

2. Calibração

- (a) Cálculo dos valores de BIAS e variância para os sensores;
- (b) Esvaziamento da fila de fluxo óptico;
- (c) Alimentação dos motores com PWM nulo.

3. Início da dinâmica de voo

- (a) Calibração dos sensores;
- (b) Verificação das condições de voo pela inclinação da superfície e pressão indicada pelo barômetro;

- (c) Verificação se fila de fluxo óptico está vazia;
- (d) Alimentação dos motores com os sinais PWM de decolagem.

4. Voo

- (a) Início da decolagem e desativação do controle de altitude;
- (b) Atualização da altitude de referência para 1,1 metro do solo após a decolagem;
- (c) Verificação do nível de bateria e aborto do voo se carga bateria está baixa;
- (d) Ativação do controle de altitude e chamada às respostas aos comandos do servidor definidas em *controlCommand.c*;
- (e) Envio do bit das posições de referência *controlModePosVSOrient* para o controlador;
- (f) Aborto do voo caso a carga da bateria esteja baixa, caso seja atingido o tempo final de simulação, caso a fila do fluxo óptico esteja vazia ou caso ocorra uma mudança brusca de velocidade (colisão);
- (g) Execução da *thread* de visão computacional, caso haja planejamento de trajetória baseado na imagem capturada pela câmera inferior;
- (h) Transferência das velocidades estimadas do fluxo óptico para o barramento *Vision Sensors* pela função declarada *rsedu_of.c*;
- (i) Aborto do voo se as velocidades estimadas pelo fluxo óptico diferirem do estimador de estados;
- (j) Envio dos comandos PWM para os motores.

5. Fim de voo e geração de logs

- (a) Parada dos motores;
- (b) Armazenamento dos dados adquiridos durante o voo na planilha *.mat*;
- (c) Exclusão da fila de fluxo óptico;
- (d) Parada do rastreamento de performance da classe *ptimer* e escrita dos respectivos logs;
- (e) Finalização das *threads*.

A cada ciclo do controlador é verificada a flag `run_flag`, a qual é levantada na emissão do comando de aborto do voo. Caso ela esteja ativa, é chamado o método `stop_flight`, o qual anula o sinal de alimentação dos propulsores. Para o projeto *Hover* na versão 2019b, não há lógica de pouso implementada.

4.2.8 Configurações da geração automática de código

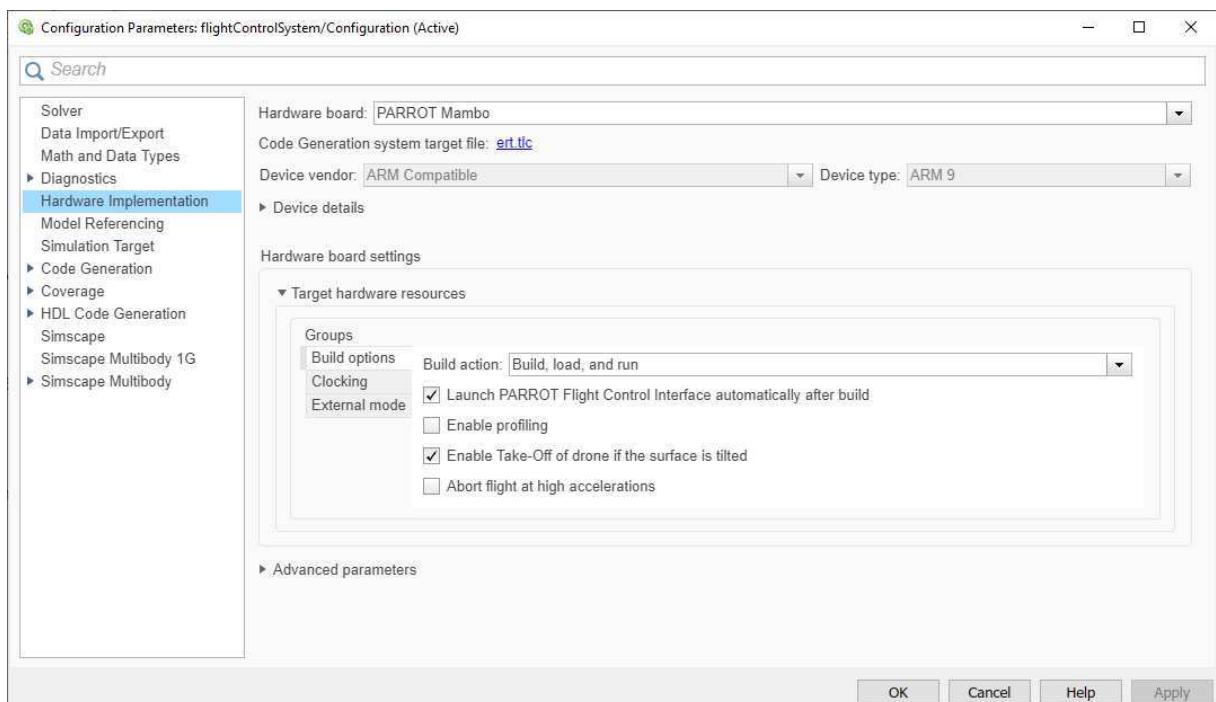
O diretório `<RAIZ>/registry` comporta os arquivos de configuração XML da geração automática de código do Simulink Coder. Em `parrotMambo_ParameterInfo.xml` estão especificadas as opções padrão da compilação. Essas configurações podem ser modificadas via interface, conforme a janela ilustrada Figura 4.2.

Essa janela pode ser aberta pelo atalho ***Crtl+E*** e clicando no tópico *Hardware Implementation* no menu lateral. Ela só está disponível quando o bloco *Flight Control System* é aberto como modelo topo (*Top Model*). Para abrir o bloco *Flight Control System* como modelo topo, clica-se com o botão direito sobre o subsistema no diagrama. As opções informadas em `parrotMambo_ParameterInfo.xml` são:

- *Runtime.BuildAction*: determina os passos da geração de código. Por padrão é compilar, implementar e executar (*Build, load, and run*);
- *UI.Launch*: especifica se a interface de voo deve ser aberta automaticamente após a compilação;
- *Config.EnableProfiling*: ativa a geração de logs `.mat`. Pode ser determinada pela linha de código `set_param('flightControlSystem', 'MatFileLogging', 'on');`;
- *Config.EnableOpticalFlow*: habilita a execução do fluxo óptico;
- *Config.EnableImageVision*: permite o processamento da imagem capturada;
- *Config.UseImageVisionForPosition*: notifica ao modelo que a imagem processada é empregada no controle de trajetória;
- *Config.UseLookup*: usa tabelas `.xlsx` para indicar as referências em *AC Cmd*;
- *Config.NoSafety*: permite a decolagem mesmo em superfícies inclinadas;

- *Config.Abort*: aborta o voo em caso de acelerações bruscas, isto é, colisões;
- *Config.MaxAcceleration*: especifica o limite de aceleração tolerável antes de abortar o voo. O padrão é 60 metros por segundo;
- *Config.ImageLogging*: habilita a transmissão de imagens em tempo real;
- *Clocking.cpuClockRateMHz*: define a frequência do contador do controlador para 416MHz.

Figura 4.2: Janela de configuração do modelo no Simulink.



Fonte: MATHWORKS[11]

Apenas as opções *Runtime.BuildAction*, *UI.Launch Config.NoSafety*, *Config.Abort*, *Config.EnableProfiling*, e *Clocking.cpuClockRateMHz* são customizáveis pela interface gráfica do Simulink.

Em `<RAIZ>/registry/attributes/parrotMambo_AttributeInfo.xml` são informados à *toolchain* os caminhos das bibliotecas *.h* em `<RAIZ>/include`, dos códigos em C em `<RAIZ>/src` e dos arquivos *onAfterCodeGenHook.m* e *onHardwareSelect.m*. Nesse XML são definidos os atributos da comunicação do modelo *External Mode* quando selecionado o modo de compilação *Build, load, and run*. Eles são:

- Interface de Comunicação: padronizada em TCP/IP;
- IP e porta de comunicação: definida como 192.168.3.1 porta 17725.

Finalmente, o arquivo `targethardware/parrotMambo_TargetHardwareInfo.xml` informa a *toolchain* os locais dos parâmetros (`parrotMambo_ParameterInfo.xml`) e dos atributos (`attributes/parrotMambo_AttributeInfo.xml`) de compilação. Ainda, é comunicado que as etapas de implementação e execução (*run*) estão descritas em `loadAndRun.m` no diretório `<RAIZ>/+codertarget/+parrot/+internal`.

Os demais arquivos omitidos nessa seção não são necessários para compreender o processo de geração de código automático. Para obter maiores detalhes, pode-se pesquisar a descrição do código no diretório fonte do pacote.

4.3 Código fonte do projeto *Hover*

A estrutura de arquivos alicerce da geração de código automática é construída quando aberto um projeto dos modelos *Competition*, *Waypoint Follower* ou *Start*. O arquivo de projeto, cuja extensão é `.prj`, organiza e compartilha o código fonte do modelo no Simulink.

Um novo projeto *Hover* é copiado no diretório `<PASTA DO USUÁRIO>/Matlab/examples` quando entrada a seguinte linha de comando no terminal do MATLAB.

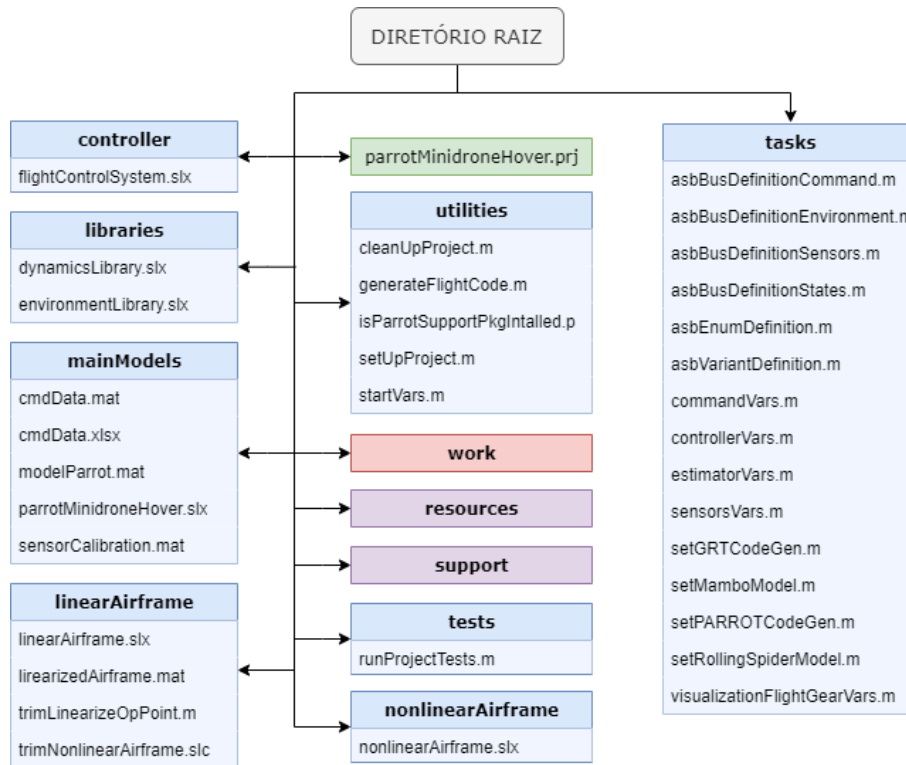
```
>> parrotMinidroneHoverStart
```

para `<PASTA DO USUÁRIO>` o diretório `<DISCO>/Users/<NOME DO USUÁRIO>`.

Esse projeto é composto pela árvore de arquivos do fluxograma ilustrada na Figura 4.3. A estrutura de diretório corresponde ao arquivo descompactado `.zip` referente a cada modelo *Competition*, *Waypoint Follower* ou *Start*, conforme discutido na Seção 3.2.

Observando a Figura 4.3, os diretórios cuja cor é roxa não são descritos nesse documento. Visto que seus arquivos representam configurações XML ou texturas para o cenário da simulação, essas pastas não são essenciais na compreensão da geração automática de código. O diretório em cor vermelha é produzido após a primeira compilação do projeto e é descrito na Seção 4.4.

Figura 4.3: Árvore de arquivos do diretório <RAIZ> de um projeto *Hover*.



Fonte: Autoria própria.

4.3.1 Diagramas de blocos

Os arquivos *flightControlSystem.slx*, *linearAirframe.slx* e *nonLinearAirframe.slx* correspondem, respectivamente, aos diagramas de blocos do subsistema de controle de voo, do modelo linear e do modelo não linear do espaço de estados. O arquivo *parrotMinidroneHover.slx* equivale ao diagrama de blocos do ambiente completo ilustrado na Figura 3.1, isto é, inclui o diagrama de blocos da modelagem do controlador, dos sensores e do cenário de voo.

Por sua vez, o modelo *environmentLibrary.slx* contém os blocos do modelo estático do ambiente e o modelo dinâmico do ambiente. Por fim, o arquivo *dynamicLibrary.slx* define o bloco *Rotot Dynamics* responsável por transformar os estados em forças e conjugados, conforme o modelo dinâmico discutido no Subseção 2.2.4.

4.3.2 Tabelas pré-modeladas

No diretório **mainModels** estão localizadas as planilhas *cmdData.mat* e *cmdData.xlsx*. Esses arquivos são modelos para a entrada das posições de referência, conforme discutido na

Subseção 3.2.1. Nessa mesma pasta, as tabelas *sensorCalibration.mat* e *modelParrot.mat* contém, respectivamente, os parâmetros de calibração dos sensores e a cadeia de caracteres “Mambo” representante da série do drone.

4.3.3 Utilidades

No diretório `utilities` estão localizados os códigos para abertura do projeto, configuração da chamada da GAC e limpeza da área de trabalho após fechamento do projeto. Ao abrir o arquivo *parrotMinidroneHover.prj*, são executados *startVars.m* e *setUpProject.m*. Ao encerrar o projeto, é executado o script *cleanUpProject.m*. Os arquivos na pasta `utilities` são:

- *startVars.m*: registra as variáveis na área de trabalho antes do carregamento do projeto. Nesse código são definidas as variáveis inteiras que representam as opções de comando de entrada *VSS_COMMAND*, modelagem dos sensores *VSS_SENSORS*, modelagem do ambiente *VSS_ENVIRONMENT* e modelagem do espaço de estados *VSS_VEHICLE*. Esses parâmetros e os possíveis valores atribuídos são:
 - *VSS_COMMAND*: indica se posição de referência é encaminhada pelo bloco de referência de altitude e posição (0), por um *joystick* (1) ou por planilhas em formato *.mat* (2) ou *.xlsx* (3);
 - *VSS_SENSORS*: indica se a modelagem dos sensores é *feedthrough* (0) ou dinâmica (1);
 - *VSS_ENVIRONMENT*: indica se a modelagem do ambiente é estática (0) ou dinâmica (1);
 - *VSS_VEHICLE*: indica se a modelagem do espaço de estados do quadricóptero é linear (0) ou não linear (1);

São também definidas as variáveis *Ts* e *TFinal*, representantes, respectivamente, do tempo de amostragem da simulação (padronizado em 0,005 segundos) e do tempo total de simulação (padronizado em 100 segundos). O período de amostragem da *thread* de visão *VTs* é personalizada para 0,2 segundos e o tempo de espera para a decolagem é 1 segundo.

Em seguida, são determinadas as velocidades, posições, ângulos e estados iniciais do drone no ambiente de simulação. Também são definidas as constantes ambientais, tais como gravidade e densidade do ar.

Nesse arquivo são declarados os barramentos nomeado nos códigos `tasks/ asbEnumDefinition.m` e `tasks/asbBusDefinition<BARRAMENTO>.m`, para `<BARRAMENTO>` equivalente a *Command*, *Sensors*, *Environment* ou *States*. Também é carregada a cadeia de caracteres em `modelParrot.mat`, e os objetos iniciados em `tasks/<BARRAMENTO>Vars.m`, para `<BARRAMENTO>` equivalente a *command*, *sensors*, *environment* ou *states*. Esses objetos são descritos discutidos Subseção 4.3.4. Quando o projeto é encerrado, as variáveis da área de trabalho são salvas nas planilhas caso haja modificações.

- *setUpProject.m*: cria o diretório *work* e configura essa pasta como local do cachê de simulação e da geração de código;
- *generateFlightCode.m*: chama o comando *rtwbuild* sobre o subsistema de controle de voo *Flight Control System*. Essa função é responsável por gerar o código do modelo Simulink e um Makefile e é discutida na Seção 4.4. Também é desativado a produção do relatório de execução da função *rtwbuild*. Para modificar essa opção, é preciso alterar o comando para

```
>> set_param(model, 'GenerateReport', 'On');
```

A imagem do diagrama de blocos para o sistema embarcado não é automaticamente gerada por razão da opção *GenCodeOnly* estar desativada. Caso acionada, é criado um arquivo *.so* (*shared object*, isto é, objeto compartilhado com a plataforma embarcada) no diretório `work`. Esse arquivo é detalhado na Seção 4.4. Pode-se configurar a opção “*Generate code only*” pela linha de comando

```
>> set_param('flightControlSystem', 'GenCodeOnly', 'on')
```

- *cleanUpProject.m*: limpa as variáveis da área de trabalho e redefine o diretório de geração de código após o encerramento do projeto.

4.3.4 Objetos procedurais

Na pasta `tasks` pode-se encontrar os seguintes scripts referidos pelos códigos do diretório `utilities`:

- *asbBusDefinitionStates.m*, *asbBusDefinitionSensors.m*, *asbBusDefinitionCommand.m* e *asbBusDefinitionEnvironment.m*: configuram a ordem e o nome das variáveis dos barramento *StateBus*, *SensorsBus*, *CommandBus* e *EnvironmentBus* na área de trabalho, respectivamente;
- *asbEnumDefinition.m*: enumera os códigos de comando do ultrassom e dos motores enviados via bluetooth. Para mais detalhes, é indicado consultar o arquivo;
- *vehicleVars.m*: declara os valores das propriedades físicas do drone, tais como massa, momento de inércia, comprimento do braço, altura e diâmetro do corpo, número de lâminas por hélice, massa da hélice, momento de inércia da hélice, constantes b e k para o cálculo das forças sobre o corpo, entre outros. Essas variáveis são armazenadas em um objeto *Vehicle*;
- *controllerVars.m*: declara a matriz de mixagem e os ganhos de transformação das velocidades do rotores em sinais de comando para os propulsores. Essas variáveis são armazenadas em um objeto *Controller* e chamadas na interface de blocos do controlador no Simulink;
- *estimatorVars.m*: declara as constantes de calibração do sensor IMU e as matrizes e erros de medição empregados nos filtros de Kalman no bloco do estimador. Essas variáveis são armazenadas em um objeto *Estimator* e chamadas na interface de blocos do estimador Simulink;
- *sensorsVars.m*: declara os valores dos ruídos e viés modelados para cada sensor e aplicados na planta virtual. Essas variáveis são armazenadas em um objeto *Sensors* e chamadas na interface de blocos da calibração dos sensores no Simulink;
- *setPARROTCodeGen.m*: configura uma função privada responsável por verificar o modelo antes da geração automática de código. Primeiramente, é apurado se o bloco

Flight Control System está aberto como modelo topo (*Top Model*). Conforme a cadeia de caracteres em *modelParrot.mat*, é definido o *hardware* alvo da geração de código como Mambo. Essa informação é transferida para o pacote Parrot com o intuito de estabelecer o IP e os comandos conforme a série do drone. Após, esse código chama as funções em *inportCallback.m* e *ipSubsystemCallback.m* no diretório fonte do pacote. Caso seja verificado que há blocos de processamento de imagem no modelo topo, é criado *buffer* usado por *rsedu_vis.m* para enviar a imagem capturada pelo drone para o Simulink.

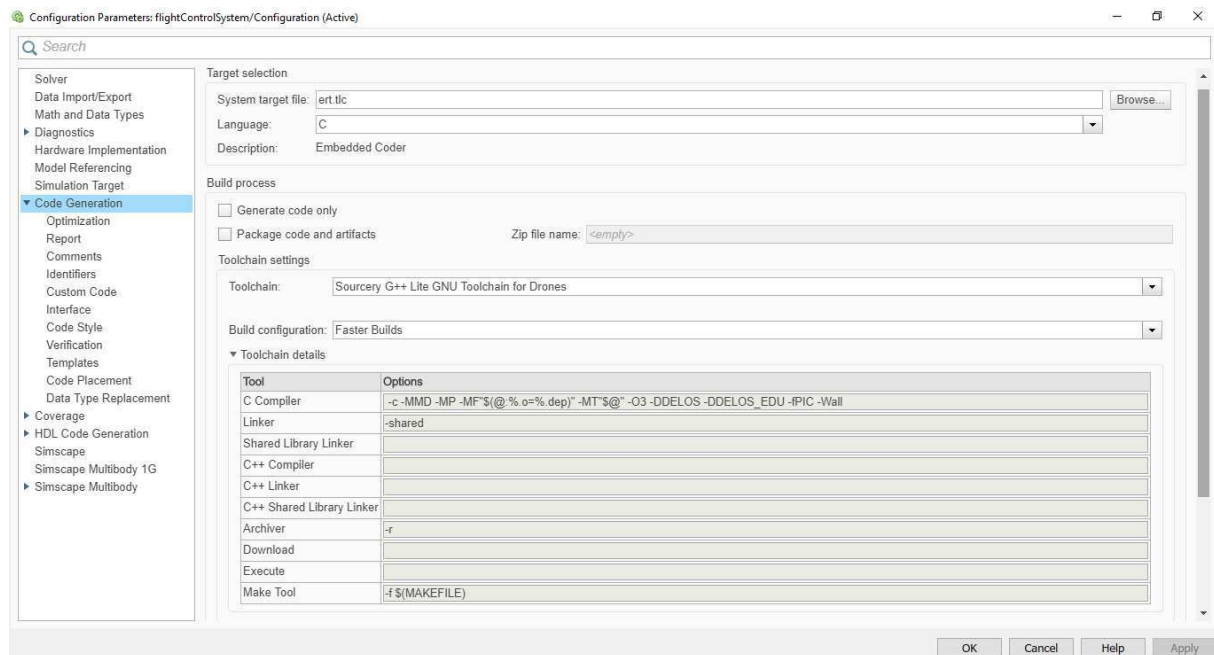
- *setMamboModel.m*: define as variáveis representantes das propriedades físicas do drone em *vehicleVars.m* para os valores do Parrot Mambo Fly. Esse código estabelece a cadeia de caracteres em *modelParrot.mat* para “Mambo”. Há um código equivalente para a série Rolling Spider.

4.4 Geração automática de código

A geração do código para a plataforma embarcada inicia por meio da função `rtwbuild`. Essa função transforma o diagrama de blocos do Simulink em códigos C/C++, gera o Makefile da compilação do objeto compartilhado e implementa-o na plataforma alvo.

A função `rtwbuild` pertence ao pacote Embedded Coder e compila, a cada execução, os códigos fontes dos blocos alterados em relação à última execução. Dessa forma, reduz-se o tempo para executar o Makefile. Pode-se empregar a função `rtwrebuild`, a qual compila apenas o Makefile considerando que o modelo foi mantido constante.

Durante a execução da função `rtwbuild` é criado um diretório de compilação no interior da pasta `work`. O nome desse diretório segue o padrão `<MODELO TOPO>_<PLATAFORMA ALVO>_rtw`. Dessa forma, para o pacote Parrot, a nova pasta é denominada `flightControlSystem_ert_rtw`. A sigla *ert* equivale a “código para embarcado em tempo real” (*Embedded Real Time Code*). A opção de plataforma alvo é determinada na interface de configuração do modelo ilustrada na Figura 4.4, acessível pelo atalho **Ctrl+E**. Ao final da geração de código, no diretório `flightControlSystem_ert_rtw` são encontrados os objetos `.o` produtos dos códigos `.c` pelo Makefile.

Figura 4.4: Janela de configurações da geração de código no Simulink.

Fonte: MATHWORKS[11]

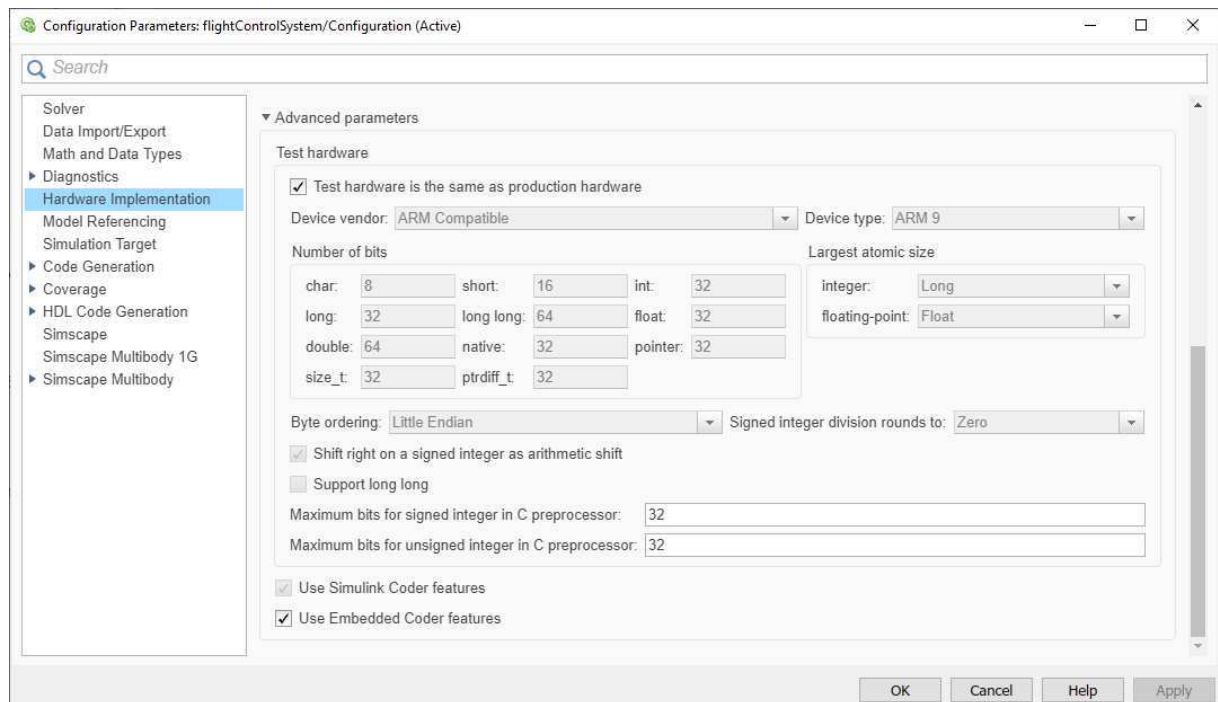
A configuração *ert* permite ao usuário escolher a linguagem alvo da compilação do modelo, nesse caso, C. Na aba *Code Generation* ilustrada na Figura 4.4, é possível personalizar as diretrizes de compilação e linkagem da toolchain. Em concordância com a Seção 4.2, a composição padrão dessas diretivas estão informadas no arquivo *arm_linux_drone.m*.

Para visualizar a aba *Code Generation* no menu, é necessário ativar na seção “*Advanced Parameters*” do tópico *Hardware Implementation* a opção “*Use Simulink Coder features*”, conforme apresentado na Figura 4.5. Essa opção é importante para a análise da geração de código para o Mambo, visto que impede a limpeza da pasta `flightControlSystem_ert_rtw` ao fim da geração de código.

Tendo em vista que o modelo *Flight Control System* refere a blocos modelados em outros arquivos, é também criado um diretório `slprj` (Simulink Project) com subpastas específicas para cada subsistema. A estrutura de pastas criada por `rtwbuild` não é atualizada pela função `rtwrebuild`.

É importante diferenciar as funções `rtwbuild` e `slbuild`. Essa última pertence ao pacote Simulink Coder e define a plataforma alvo *grt*, isto é, código generalizado em tempo real (*Generalized Real Time Code*). O objeto compartilhado `.so` produzido por essa ferramenta

Figura 4.5: Janela de configurações das opções avançadas de implementação de *hardware* no Simulink.



Fonte: MATHWORKS[11]

não é específico para uma plataforma embarcada. Em outras palavras, a geração automática do Simulink Coder não é hábil para alocar os espaços de memória no sistema embarcado.

Entretanto, essa compilação é indicada para verificar o novo *firmware* em simulação, visto a dispensabilidade de compilação cruzada. A função `slbuild` é uma alternativa adequada para contornar a licença do Embedded Coder.

Com o intuito de executar testes rápidos do *software* em simulação, é essencial interromper o fluxo MAD antes da implementação do objeto `.so` no protótipo. Para tal, pode-se ativar a opção “*Generate code only*” na interface de configuração apresentada na Figura 4.4. Dessa forma, são produzidos os códigos C equivalentes ao modelo *Flight Control System*, porém o Makefile não é executado. Sendo assim, haverá a geração de código automática sem que ocorra a compilação cruzada e implementação do objeto compartilhado `.so` no drone.

O pacote Parrot impede que o Embedded Coder construa o objeto compartilhado caso o drone não seja identificado nas conexões bluetooth do servidor. Esse problema é solucionado ativando a opção “*Use Embedded Coder features*” na seção “*Advanced Parameters*” do tópico *Hardware Implementation*, conforme ilustrada na Figura 4.5. Essa ferramenta executa o

Makefile, todavia não parte para a fase de comunicação com o drone. Logo, pode-se testar a ferramenta de GAC antes dos verificar com o *hardware*.

4.4.1 Conversão do modelo em código

A seguir, são analisados os arquivos do diretório `flightControlSystem_ert_rtw`. É importante ressaltar que esses arquivos são gerados automaticamente pelo Simulink Coder. Os códigos não essenciais para compreender o fluxo MAD não são discutidos nesse documento.

São quatro os arquivos referentes ao modelo topo:

- `flightSystemControl.c` implementa o modelo topo em linguagem C;
- `flightSystemControl.h` contém as definições das variáveis representantes dos parâmetros ou estados dos blocos no diagrama Simulink;
- `flightSystemControl_private.h` inclui as declarações dos protótipos das funções;
- `flightSystemControl_types.h` declara as estruturas empregadas no código C.

Esses arquivos são interdependentes. Dessa forma, são descritos inicialmente os arquivos de declaração dos tipos, seguidos dos protótipos das funções. Por fim, é discutido o arquivo em linguagem C.

Tipos customizáveis

Em `flightSystemControl_types.h` são declaradas as estruturas `struct` das variáveis definidas pelo servidor. Em outras palavras, esse cabeçalho declara os tipos instanciados nos códigos `tasks/asbEnumDefinition.m` e `tasks/asbBusDefinition<BARRAMENTO>.m`, para `<BARRAMENTO>` equivalente a `Command`, `Sensors`, `Environment` ou `States`. Por exemplo, o vetor nomeado `UpdateReferenceCmds` é do tipo estrutura `CommandBus`, declarada como:

```

1 typedef struct {
2     boolean_T controlModePosVSorient;
3     real32_T pos_ref[3];
4         boolean_T takeoff_flag;
5         real32_T orient_ref[3];
6         uint32_T live_time_ticks;

```

```
7 } CommandBus;
```

As estruturas *CommandBus*, *SensorsBus*, *motors_outport* e *flag_outport* são exportadas como variáveis globais. Os objetos dessas estruturas são instanciados nos códigos *rseu_control.c* e *controlCommand.c* descritos na seção Seção 4.2. Esses objetos criam uma correspondência entre os locais de memória no drone e as variáveis no servidor, ou seja, são *buffers* de comunicação pelos quais o servidor envia os comandos dos motores e recebe do drone os dados adquiridos pelos sensores.

Protótipos das funções

Em *flightSystemControl_private.h* são declarados os protótipos das funções externas definidas em *flightSystemControl.c*. Nesse cabeçalho é importado o apontador global para o *buffer* da imagem recebida do drone.

Variáveis de parâmetros e estados

Os tipos estruturais dos sinais de entrada de cada bloco estão declaradas em *flightSystemControl.h*. Nesse código, são alocadas as estruturas de “estado” de cada bloco, isto é, as variáveis internas que são operadas no próximo ciclo de relógio.

Essas variáveis são nomeadas conforme o diagrama de blocos no Simulink. Por exemplo, o vetor nomeado *estimatedStates* no bloco *Flight Control System* é declarado tipo estrutura *ToWorkspace_PWORK* a seguir:

```
1 typedef struct {void *LoggedData;
2 } ToWorkspace_PWORK;           /* '<S1>/To Workspace' */
```

No final desse cabeçalho, são enumerados os subsistemas e blocos do diagrama Simulink. O formato dessa listagem é `<NOME DO SISTEMA>/<NOME DO BLOCO>`. Essa associação permite fácil localização das variáveis no diagrama de blocos. Por exemplo, a enumeração do bloco controlador do subsistema de comandos de voo é:

```
1 /* '<S3>': `flightControlSystem/Flight Control System/Controller' */
```

Por fim, em *flightSystemControl.h* são exportadas as funções de ponto de entrada (*entry-point*). O ponto de entrada é a localização da transferência do controle do programa, isto é, onde uma função é chamada. Por exemplo, em um código C, a função *main* é o ponto

de entrada para a execução da imagem compilada. Na geração automática de código, é necessário definir as funções ponto de entrada do código C do modelo Simulink.

Essas funções de ponto de entrada não possuem parâmetros e se comunicam modificando as estruturas globais armazenadas na memória compartilhada. É importante ressaltar que as funções de ponto de entrada são geradas automaticamente pelo Simulink Coder. Esse processo mecânico pode ser configurado no subtópico “*Interface*” em “Code Generation” nas configurações do modelo Simulink.

Código do subsistema de controle de voo

As funções de ponto de entrada definidas em *flightSystemControl.c* são:

- *flightControlSystem_initialize*: é chamada unicamente para iniciar a geração de código. Aloca estaticamente as estruturas de dados globais na memória. Essa função transfere o controle de execução para o código *rt_logging.c*, responsável pela geração de logs, e para a função *flightControlSystem_FlightControlSystem_SetupRTR*.

Por sua vez, a função *flightControlSystem_FlightControlSystem_SetupRTR* aloca os parâmetros do pacote *Real Time Workspace*, tais como os vetores que devem ser exportados para o log *.mat* e as opções do modelo definidas *startVars.m*. Ela é executada uma vez ao final da compilação, independentemente da quantidade de simulações. Dessa forma, essa função é essencial para evitar que o código, caso inalterado, seja recompilado após o término de cada simulação.

Em seguida, a função *flightControlSystem_initialize* chama a função *flightControlSystem_FlightControlSystem_Init*, a qual atribui os valores iniciais às variáveis de estado do modelo.

Diferentemente dos barramentos de entrada e saída dos blocos declarados em *flightControlSystem_initialize*, os estados alocados na função *flightControlSystem_FlightControlSystem_Init* são as variáveis intermediárias dos blocos. Essa função é executada em todas as simulações.

- *flightControlSystem_step0* e *flightControlSystem_step1*: computam os valores correntes do bloco *Flight Control System*. Essas funções atualizam os arquivos de log e e

finalizam quando o tempo de execução alcança o parâmetro T_{Final} .

Observando a Figura 3.5, diferencia-se a função *step0* da *step1*. Enquanto aquela implementa o algoritmo do bloco *Flight Control System* descrito na função *flightControlSystem_FlightControlSystem*, a última computa o bloco *Image Processing System* a partir da função *MW_Build_RGB*. Essa função *MW_Build_RGB* é definida no arquivo *rsedu_vis.c* no diretório do pacote discutido na Seção 4.2.

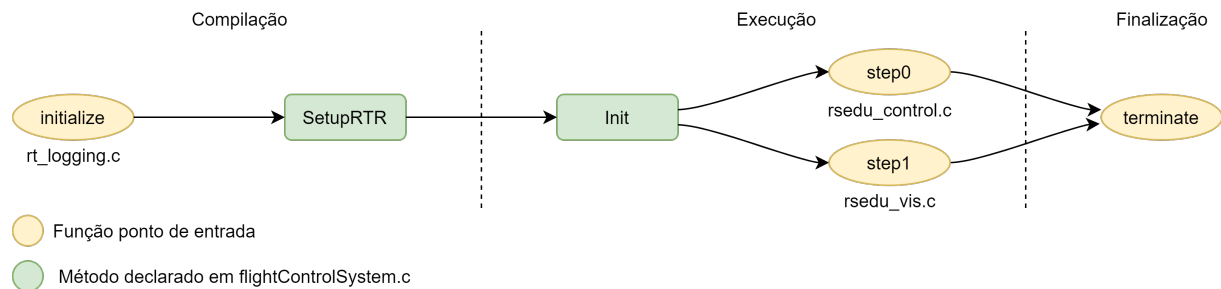
Cada uma das funções *step0* e *step1* possui sua própria *thread* amostrada pelo processador em T_s e VT_s , respectivamente. Destaca-se que o código implementado nessas funções não representa o cálculo dos parâmetros (e.g. matriz de mixagem, constante de empuxo), mas as operações entre esses parâmetros (e.g. mixagem da saída do controlador, cálculo das forças sobre o corpo). A computação desses parâmetros envolvidos na dinâmica do drone está descrita nos arquivos *controllerVars.m* e *vehicleVars.m*.

Caso a função em *setPARROTCodeGen.m* retorne que não há bloco *Image Processing Data* no modelo Simulink, a função *step1* não é gerada. Desse modo, o multiprocessamento é desligado visto que há uma única *thread*.

- *flightControlSystem_terminate*: para o pacote Parrot, não há protocolos para serem executados ao fim da simulação. As ações de retirada de log são realizadas manualmente para interface de voo apresentada na Figura 3.20. A limpeza da área de trabalho e atualização das variáveis é realizada pelo código *cleanUpProject.m*. Nesse caso em estudo, a função é vazia.

O fluxo de execução do *firmware* modelo para o Parrot Mambo pode ser observado na Figura 4.6. Como discutido, a função ponto de entrada *flightControlSystem_initialize* inicia a compilação do código e chama, nessa ordem, as funções *flightControlSystem_FlightControlSystem_SetupRTR* e *flightControlSystem_Init*. Depois que todas as variáveis são iniciadas, o fluxo de controle é executado por meio das funções *flightControlSystem_step0* e *flightControlSystem_step1*. Quando o voo é finalizado, o fluxo é transferido para a função *flightControlSystem_terminate*. A fronteira entre as etapas de compilação e execução separa as funções que são executadas unicamente ou repetidas vezes a cada simulação.

Figura 4.6: Fluxo de execução das funções do modelo implementado no sistema embarcado. Em amarelo, estão representadas as funções ponto de entrada. Os métodos declarados em *flightControlSystem.c* estão desenhados em verde.



Fonte: Autoria própria.

Arquivos de Dados

No diretório `flightControlSystem_ert_rtw` pode-se encontrar o arquivo *flightControlSystem_data.c*. Esse código define o objeto global *flightControlSystem_P*. Os membros dessa estrutura são as variáveis listadas ao fim de *flightSystemControl.h*. Esse objeto é atualizado a cada ciclo de relógio da simulação.

Arquivos de Utilidade

Esses arquivos são códigos em C copiados dos diretórios `src` do diretório fonte do pacote Parrot e *Real Time Workshop*. Como os códigos essenciais para a compreensão da geração automática de código foram supracitados, mais detalhes devem ser buscados nos relatórios *html* produzidos no diretório `flightControlSystem_ert_rtw/html`.

É importante ressaltar que os relatórios só são produzidos caso o arquivo *setUpProject.m* discutido na Subseção 4.3.3 seja alterado. Após a compilação, pode-se acessar o relatório geral pelo comando

```
>> rtw.report.open('flightControlSystem', '<DIRETORIO DO PROJETO HOVER>\
work\flightControlSystem_ert_rtw')
```

4.4.2 Makefile

Quando ativada as opções “Generate Code Only” e o “Use Embedded Code features” estudadas na Seção 4.4, o Makefile é produzido e executado pela função `rtwbuild`. Outras

alternativas são a função `rtwrebuild` ou a ação “*Build, load and run*” acessível a partir do atalho ***Ctrl+B*** no diagrama Simulink.

O Makefile é gerado automaticamente conforme as diretivas em `arm_linux_drone.m` ou no tópico “*Code Generation*” na interface de configuração do modelo. Essa interface é acessada pelo atalho ***Ctrl+E***, conforme discutido na Seção 4.4.

A seguir, são descritos os trechos do Makefile indispensáveis para a compreensão da implementação do código gerado na plataforma física.

Diretórios fonte

Primeiramente, é necessário conhecer o caminho dos diretórios acessados pelo Makefile. As variáveis referentes ao endereços dos arquivos são `MATLAB_ROOT`, `PKG_PATH` e `START_DIR` definidas por

```

1 MATLAB_ROOT = C:/PROGRA~1/POLYSP~1/R2019b
2 PKG_PATH    = C:/ ProgramData/MATLAB/SupportPackages/R2019b/toolbox/
3             target/supportpackages/parrot/src/
4 START_DIR   = C:/Users/<USUARIO>/MATLAB/Projects/examples/
5             parrotMinidroneHover/work

```

Arquivos Fontes

Os arquivos fontes são os códigos em linguagem C no diretório `src` do pacote Parrot e na pasta `flightControlSystem_ert_rtw` do projeto *Hover*. A variável do Makefile que referencia esses arquivos é `SRCS` declarada por

```

1 ## SOURCE FILES
2 SRCS = $(PKG_PATH)/rsedu_image.c
3       $(MATLAB_ROOT)/rtw/c/src/rt_logging.c
4       $(START_DIR)/flightControlSystem_ert_rtw/flightControlSystem.c
5       $(START_DIR)/flightControlSystem_ert_rtw/
6       flightControlSystem_data.c
7       $(START_DIR)/flightControlSystem_ert_rtw/rtGetInf.c
8       $(START_DIR)/flightControlSystem_ert_rtw/rtGetNaN.c
9       $(START_DIR)/flightControlSystem_ert_rtw/rt_nonfinite.c
10      $(PKG_PATH)/rsedu_control.c

```

```

11     $(PKG_PATH)/rsedu_of.c
12     $(PKG_PATH)/rsedu_vis.c
13     $(PKG_PATH)/ptimer.c
14     $(PKG_PATH)/controlCommand.c
15     $(PKG_PATH)/mw_extrathreads.c

```

Arquivos objetos .o

A partir de cada arquivo .c é compilado um objeto .o. Essa compilação cruzada é direcionada para a arquitetura ARM Cortex 9, em concordância com as especificações do drone expostas Subseção 2.4.1. No Makefile, as diretivas de compilação são definidas por

```

1  # "Faster Builds" Build Configuration
2  CFLAGS = -c \
3         -MMD -MP -MF "$(@:%.o=%.dep)" -MT "$@" -O3 \
4         -DDELOS \
5         -DDELOS_EDU \
6         -fPIC \
7  ## BUILD TOOL COMMANDS
8  # C Compiler: Sourcery G++ Lite GNU Toolchain for Drones C Compiler
9  CC_PATH = $(PARROT_GCCSourceryLite)
10 CC      = $(CC_PATH)/arm-none-linux-gnueabi-gcc

```

Observando a Figura 4.4 e segundo a Seção 4.2, percebe-se que as diretivas CFLAGS são as definidas em *arm_linux_drone.m*. Conforme discutido na Seção 4.2, o caminho *PARROT_GCC SourceryLite* é declarado como variável de ambiente em *addCompilerPath.m*. Os alvos intermediários .o são produzidos a partir do trecho do Makefile a seguir.

```

1  # SOURCE-TO-OBJECT
2  %.o : %.c
3     $(CC) $(CFLAGS) -o "$@" "$<"

```

Esses objetos estão localizados no diretório `flightControlSystem_ert_rtw`. A partir deste momento, é definida a variável *OBJS* análoga a *SRCS*, porém referindo aos arquivos .o. As linhas referentes a essa etapa são:

```

1  ## OBJECTS
2  OBJS = rsedu_image.o rt_logging.o flightControlSystem.o

```



```

3      flightControlSystem_data.o rtGetInf.o rtGetNaN.o
4      rt_nonfinite.o rsedu_control.o rsedu_of.o rsedu_vis.o
5      ptimer.o controlCommand.o mw_extrathreads.o

```

Arquivo objeto compartilhado *.so*

O arquivo de objeto compartilhado *.so* é gerado pela linkagem dos arquivos objetos *.o*. As informações do local e nome do arquivo de saída *.so* são definidas no Makefile pelas seguintes linhas:

```

1  ## OUTPUT INFO
2  PRODUCT = ../flightControlSystem.so

```

As diretivas da *toolchain* e do linker são definidas também em *arm_linux_drone.m*. No Makefile, essas diretivas são representadas pelas variáveis `TOOLCHAIN_LIBS` e `LD`.

```

1  ## MACROS
2  TOOLCHAIN_LIBS = -lm -lm
3  # Linker: Sourcery G++ Lite GNU Toolchain for Drones Linker
4  LDFLAGS = -shared
5  LD_PATH = $(PARROT_GCCSourceryLite)
6  LD      = $(LD_PATH)/arm-none-linux-gnueabi-gcc

```

Por fim, a geração do objeto compartilhado *flighControlSystem.so* é realizado pelas linhas do Makefile a seguir:

```

1  ## FINAL TARGET
2  # Create a standalone executable
3  $(PRODUCT) : $(OBJS)
4      @echo "### Creating standalone executable $(PRODUCT) ..."
5      $(LD) $(LDFLAGS) -o $(PRODUCT) $(OBJS) $(TOOLCHAIN_LIBS)
6      @echo "### Created: $(PRODUCT)"

```

O arquivo *.so* está na pasta `work` do diretório do projeto. Os comandos `echo` de script `bash` no trecho de código anterior estimulam a hipótese que um sistema operacional Linux esteja embarcado no drone.

4.5 Implementação e execução do código

Para os testes de prototipagem, o objeto compartilhado *flightControlLib.so* gerado no diretório `work` deve ser enviado para o drone e executado. O código *loadAndRun.m* descrito na Subseção 4.2.2 é executado para implementar o `.so` gerado no sistema embarcado. Quando executado pelo sistema operacional do quadricóptero, esse objeto compartilhado inicia o fluxo ilustrado na Figura 4.6.

Primeiramente, o programa *loadAndRun.m* procura conectar o servidor com o quadricóptero via bluetooth. Em caso de erro, a operação é abortada. Caso contrário, o servidor abre uma conexão telnet com o quadricóptero a partir de um objeto da classe *Telnet*. Essa classe é declarada no código *Telnet.m*, conforme discutido na Seção 4.2. O segmento de código a seguir pertence ao arquivo *loadAndRun.m*.

```
1 telnetObj = codertarget.parrot.internal.Telnet(droneIPAddress);
2 telnetObj.open();
3
4 telnetObj.cmd('cd /data/edu');
5 telnetObj.cmd('rm *so');
6
7 disp(message('parrot:setup:killDragonProg').getString);
8 telnetObj.cmdAndWait('kk', 'RS.edu');
```

Os métodos *open* e *close* da classe *Telnet* abrem e fecham a conexão com o quadricóptero, respectivamente. O IP depende da série do drone (Mambo ou Rolling Spider) informado pelo arquivo *modelParrot.m* no diretório `task` do projeto. Por sua vez, os IPs estão definidos no código *parrotminidrone.m*, conforme discutido na Seção 4.2.

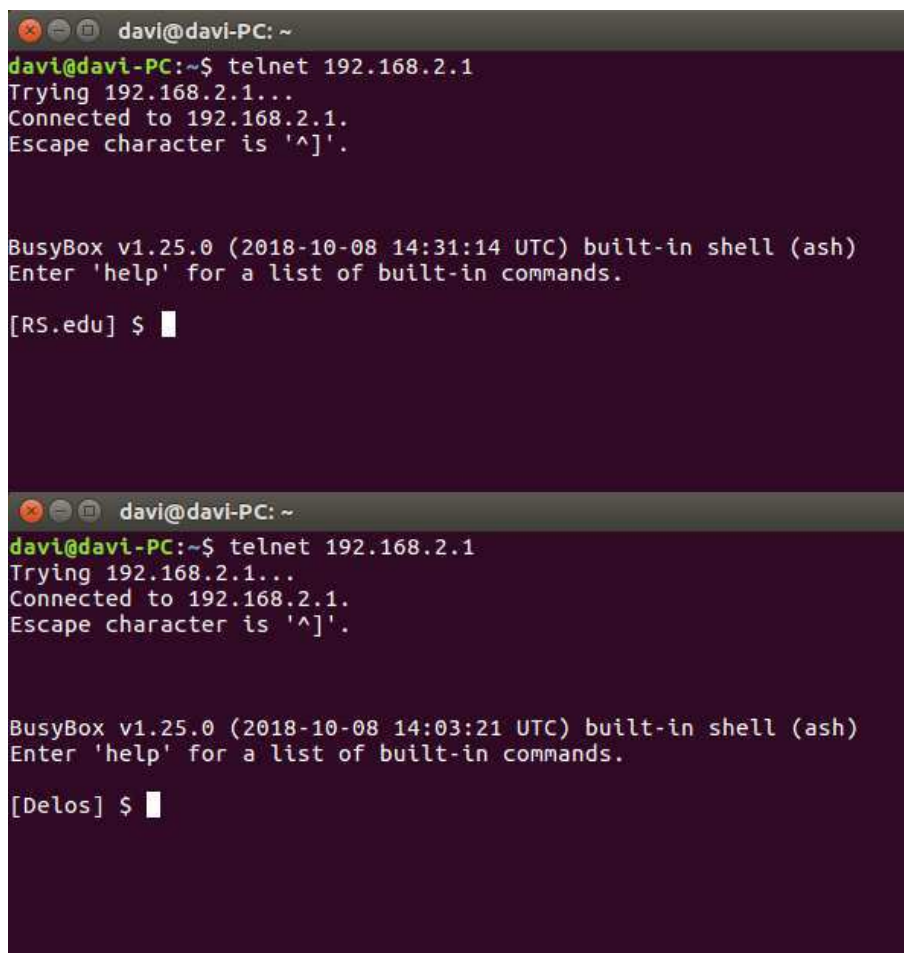
O método *cmd* envia para o buffer da porta de comunicação telnet do drone comandos. Esses comandos possuem sintaxe UNIX. Dessa forma, comprovou-se que o sistema operacional embarcado é Linux.

Imediatamente após a conexão telnet ser aberta, é enviado o comando de mudança de diretório `cd`. Tendo em vista que a comunicação via protocolo telnet inicia no diretório raiz, o comando `rm` é executado dentro do diretório `/data/edu` do drone. A instrução `rm`, por sua vez, remove o objeto compartilhado `.so` existente na pasta. Essa etapa garante que o novo arquivo *flightControlSystem.so* seja transferido para o drone sem conflitos.

Em seguida, o servidor envia o comando `kk`. Essa instrução representa um mnemônico para fechar os processos correntes no drone, com exceção aos processos do sistema operacional. O comando `kk` é detalhado na Capítulo 5.

Logo após, o servidor lê a string de resposta do terminal do drone. Na Figura 4.7 são apresentadas as capturas do terminal do sistema embarcado com o *firmware* nativo e o do MATLAB após o estabelecimento da comunicação telnet. É perceptível que para o *firmware* nativo, a resposta é `[Delos]`. Contudo, para o *firmware* do MATLAB, a resposta é `[RS.edu]`. Esse passo de verificação garante que todos os processos do controlador de voo foram terminados antes acrescentar o novo arquivo `.so`.

Figura 4.7: Captura dos terminais de comando após estabelecida a conexão telnet com o *firmware* do MATLAB (acima) e o *firmware* nativo (abaixo).



```
davi@davi-PC: ~
davi@davi-PC:~$ telnet 192.168.2.1
Trying 192.168.2.1...
Connected to 192.168.2.1.
Escape character is '^]'.

BusyBox v1.25.0 (2018-10-08 14:31:14 UTC) built-in shell (ash)
Enter 'help' for a list of built-in commands.

[RS.edu] $ █

davi@davi-PC: ~
davi@davi-PC:~$ telnet 192.168.2.1
Trying 192.168.2.1...
Connected to 192.168.2.1.
Escape character is '^]'.

BusyBox v1.25.0 (2018-10-08 14:03:21 UTC) built-in shell (ash)
Enter 'help' for a list of built-in commands.

[Delos] $ █
```

Fonte: Autoria própria

Após garantir que o diretório de destino do arquivo `.so` está limpo, o servidor envia esse objeto compartilhado via comunicação FTP. Caso a porta definida em *DroneConnect.m* não

esteja aberta, o processo é abortado. Esse código de lançamento do erro em *loadAndRun.m* está apresentado a seguir.

```

1 disp(message('parrot:setup:connectingToFTP', droneIPAddress).getString);
2     try
3         drone_ftp = ftp(droneIPAddress);
4     catch
5         error(message('parrot:setup:noFTP', droneIPAddress,
6             targetMinidrone).getString);
7     end

```

Desde que a conexão FTP esteja estabelecida, o arquivo *flightControlSystem.so* é renomeado para *librsedu.so* e enviado para o drone pela função `mput`. O trecho de código abaixo apresenta essa etapa no código *loadAndRun.m*. A variável *modelName* é definida como *flightControlSystem* pela função `fileparts`, a qual retorna o vetor `[<DIRETORIO DO ARQUIVO .SO>, <NOME DO MODELO TOPO>, <EXTENSÃO DO ARQUIVO>]`.

```

1 [~, modelName, ~] = fileparts(executableFile);
2 sharedObjName = [ modelName '.so'];
3 mput(drone_ftp, executableFile);
4 rename(drone_ftp, sharedObjName, 'librsedu.so');

```

Logo após, o servidor envia o arquivo *MamboFlight.sh* pela mesma comunicação FTP. Esse arquivo inicia a execução do objeto *librsedu.so* no drone. A transferência desse script é realizada pela instrução a seguir.

```

1 mput(drone_ftp, fullfile(codertarget.parrot.internal.getSpPkgRootDir,
2     'lib', 'EDUfirmwareFILES', 'MamboFlight.sh'));

```

Em seguida, o servidor retorna à comunicação telnet para garantir propriedade de execução do script bash *MamboFlight.sh* por meio da instrução `chmod +x`. Em seguida, o servidor envia o comando para executar *MamboFlight.sh* e armazenar o seu log de saída no arquivo *droneFlight.txt*. Esse log é posteriormente recuperado do diretório `/data/edu/` pelo código *parrotio.m* apresentado na Seção 4.2. Por fim, a conexão telnet é fechada após 10 segundos de ociosidade. O segmento de código equivalente a essa etapa no arquivo *loadAndRun.m* está indicado a seguir.

```

1 telnetObj.cmd('cd /data/edu');
2 telnetObj.cmd('chmod +x MamboFlight.sh');

```

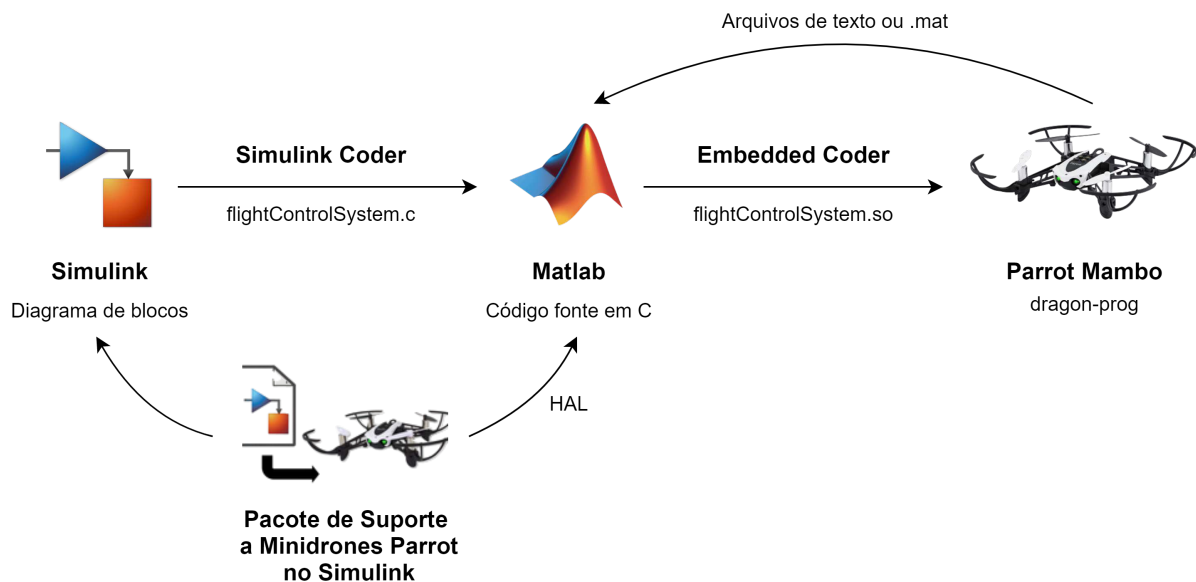
```

3 telnetObj.cmd('./MamboFlight.sh > /data/edu/droneFlight.txt &');
4 telnetObj.waitForTimeout(10);
5 close(drone_ftp);

```

Na Figura 4.8 está ilustrado o fluxo de execução MAD/GAC do pacote de suporte a minidrones Parrot no Simulink [11]. Por meio da análise do código fonte, foi caracterizada a conversão do diagrama de blocos em linguagem C pelo Simulink Coder e, em seguida, para o formato *.so* pelo Embedded Coder. Esse processo é executado por um *Makefile*, o qual, ao final da implementação, chama um script *bash* no sistema embarcado. Por sua vez, esse SO possui dependência de arquivos Linux.

Figura 4.8: Fluxo de transformação do modelo em diagrama de blocos para um objeto compartilhado com o protótipo físico.



Fonte: Autoria própria

4.6 Considerações finais

O estudo do código fonte do pacote de suporte a minidrones Parrot no Simulink possibilitou a compreensão dos comandos enviados pelo servidor e interpretados pelo controlador embarcado. Também foi esclarecida as configurações de compilação do sistema de controle e dos canais de comunicação entre o drone e o MATLAB. Já o estudo sobre o código fonte do projeto esclareceu a transformação do desenho personalizado em blocos Simulink para o

código em C e, posteriormente, para o processador ARM. A execução do código customizado no sistema embarcado é detalhado no próximo capítulo, o qual trata do sistema operacional e dos comandos nativos interpretados pelo quadricóptero.

Capítulo 5

O sistema operacional embarcado

Este capítulo trata sobre como se comunicar com o sistema operacional (SO) do drone. Primeiramente, é apresentado como foi realizado o acesso ao SO do quadricóptero e, em seguida, como enviar instruções diretamente para o drone por meio da porta FTP.

5.1 Acessando o sistema operacional

Para adentrar o sistema operacional embarcado, é necessário acessar o IP do drone via telnet. O endereço MAC do Parrot Mambo foi discutido no código *DroneConnect.m* na Seção 4.2.

Os comandos apresentados nesse capítulo podem ser enviados por um terminal no Windows ou Linux. A diferença está na necessidade daquele de acessar a rede pontual do drone, enquanto o último dispensa essa conexão. Com o intuito de abrir a conexão telnet com o drone, digita-se a seguinte instrução no terminal:

```
>> telnet 192.168.3.1
```

Por padrão, tenta-se alcançar a porta 23 do IP 192.168.3.1. Essa numeração de porta, contudo, é aberta apenas para o *firmware* modificado e instalado pela interface discutida na Subseção 4.2.6.

Para um drone com o sistema embarcado original, a porta 23 da comunicação telnet só é aberta enquanto o drone está em uma sessão de depuração e conectado ao servidor via USB, isto é, a partir do IP 192.168.2.1. Essa sessão é criada ao apertar quatro vezes seguidas

o botão de ligar. As luzes irão piscar e um novo disco será reconhecido pelo computador quando o quadricóptero é conectado via USB.

Quando a conexão é bem sucedida, é aberto o terminal conforme ilustrado na Figura 4.7. Como pode ser observado nas primeiras linhas do terminal, o drone executa uma imagem Linux BusyBox. Esse *software* implementa os comandos de usuário e *bootloader* UNIX e é comumente usado em aplicações de memória restrita, tal como o Mambo. A estrutura de diretórios embarcada é semelhante a disposição do SO Linux. Dessa forma, na raiz há as pastas *bin*, *dev*, *etc*, *usr*, *proc* e *var*. No *firmware* alterado pelo Simulink, há um novo diretório na raiz, nomeado *data/edu*.

5.1.1 Diretório *data/edu*

O *firmware* nativo e o modificado pelo MATLAB diferem na presença do diretório `data/edu`. Na pasta `data/edu` estão presentes os códigos *MamboFlight.sh* e *librsedu.so*, copiados dos diretórios `<RAIZ DO PACOTE>/lib/EDUfirmwareFILES/` e `<RAIZ DO PROJETO>/work`, conforme apresentado na Seção 4.2 e Seção 4.3.

O arquivo *MamboFlight.sh* inicia o fluxo de controle no drone após o recebimento do objeto *librsedu.so*. Assim como descrito na Seção 4.4, a execução desse script é a última etapa da implementação da geração de código automática. Todas as *strings* exibidas em tela durante a execução desse script são armazenadas no arquivo de texto *droneFlight.txt*. Esse arquivo é posteriormente recuperado a partir da interface de comando de voo da Figura 3.20.

Em *MamboFlight.sh*, primeiramente são criadas as filas do fluxo óptico e processamento de imagem. Em seguida, são criados os diretórios para armazenar as imagens capturadas e os dados de log da classe *ptimes*.

Logo após, o script *MamboFlight.sh* chama o arquivo compilado *dragon-prog* presente na pasta `usr/bin`. Esse programa é responsável por escutar os comandos enviados pelo servidor por meio da porta FTP 12391, visto que qualquer dado enviado a esse canal é ignorado até a execução do *dragon-prog*.

Até finalizar, o processo *dragon-prog* os gera arquivos de log do script *MamboFlight.sh* na pasta temporária `tmp`. Quando o botão “STOP” da interface de voo discutida na Subseção 3.3.1 é acionado, é executado no sistema embarcado o comando `kk`. Como discutido na

Seção 4.5, essa instrução finaliza o processo do *dragon-prog*.

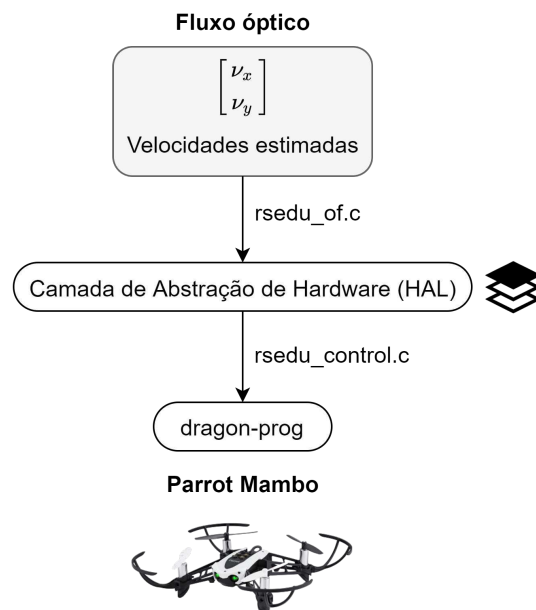
Foi proposta a hipótese desse binário representar o controlador do drone. Para validar essa suposição, buscou-se quais programas são chamados pelo arquivo *dragon-prog* já compilado. Para tal, usou-se o comando `strings` a seguir, o qual retorna as cadeias de caracteres presentes no binário:

```
>> strings /usr/bin/dragon-prog
```

Para o *firmware* modificado, na saída da instrução anterior é encontrada a *string* “*librsedu.do*”. Em outras palavras, conclui-se que é realizada uma chamada ao objeto compartilhado *librsedu.so* e às funções *step0* e *step1* que o compõe. Assim, comprova-se a hipótese que esse programa representa o controlador.

A partir da consideração que o processo *dragon-prog* representa o laço de controle e das observações dos códigos *rsedu_control.c* e *rsedu_of.c* descritas na Seção 4.2, é possível ilustrar a comunicação entre o sensor de fluxo óptico e o controlador. Esse fluxo está apresentado na Figura 5.1.

Figura 5.1: Diagrama das comunicação entre o sensor de fluxo óptico para a estimativa de velocidade no plano XY e o controlador por meio da camada de abstração de *hardware*.



Fonte: Autoria própria

Contudo, no executável *dragon-prog* do *firmware* nativo, não há essa referência à *string* “librsedu.do” do objeto compartilhado. Logo, infere-se que a caixa preta do controlador é diferente para as duas versões do *firmware*. Tal desigualdade pode ser justificada pela necessidade de novas portas de comunicação entre o programa *dragon-prog* e o diagrama compilado do Simulink. Essas novas conexões permitem a atualização dos parâmetros e lógica de controle de atitude.

Essa alteração de portas não é necessária para o *firmware* nativo. Quando esse SO está sendo empregado, o objetivo do experimento não é modificar o método de controle de trajetória. Pelo contrário, o intuito do drone programado com o *firmware* nativo é responder apenas a atualizações da posição de referência enviadas por um dispositivo externo, como um *joystick*.

Por fim, o script *MamboFligh.sh*, move as imagens adquiridas no diretório `tmp` (temporário) para o `data/edu`. Essa pasta é a raiz da comunicação FTP do drone, pela qual irão ser recuperados as imagens capturadas e o arquivo de log *droneFlight.txt*. Esse arquivo de texto contém as saídas do script *rsedu_control.c* apresentado na Subseção 4.2.7, o qual é chamado pelo *dragon-prog* iniciado em *MamboFligh.sh*.

Também são copiados os arquivos produzidos pelo objeto *ptimes* `pt_<NOME DA THREAD>.txt` da pasta `/tmp/edu/ptimes/` para `data/edu`, para `<NOME DA THREAD>` equivalente a `RSEDU_OF`, `RSEDU_CONTROL` e `RSEDU_VISION`.

5.1.2 Diretório *bin* e *usr/bin*

Na raiz há o diretório *bin*. Nessa pasta estão localizados os programas de funcionalidade mínima do sistema acessíveis por todos os usuários.

Há também a pasta *usr/bin*, a qual contém os comandos disponíveis apenas para o usuário com privilégios de administrador. Os códigos presentes nesse último diretório não podem ser requisitados na inicialização ou reparação do SO. Em *usr/bin* está localizado o binário *dragon-prog*.

Em `bin`, estão presentes scripts para inicialização e manutenção do SO. São encontrados também executáveis de depuração, os quais apresentam comandos de alimentação de motores. Um exemplo é o bash *init_motors.sh*, cujo trecho está apresentado a seguir.

```
1 #!/bin/sh
2 source /bin/user_gpio.sh
3 gpio_out MOTOR_FAULT 0
4 for i in 0 1 2 3; do
5     valid_pwm 10000 0 ${i} &
6     usleep 200000
7     killall -9 valid_pwm
8 done
```

Esse segmento do código é responsável por ligar os quatro motores em sequência. A instrução `valid_pwm` está definida em `usr/bin` e envia um sinal PWM para cada um dos motores. Experimentalmente, constatou-se que os parâmetros desse comando são a frequência do sinal, o ciclo de trabalho e o número do motor alvo do sinal, respectivamente.

No diretório `bin` está localizado o comando `kk`. Assim como discutido na Seção 4.5, essa instrução é um mnemônico. A linha de código que esse comando representa é:

```
1 #!/bin/sh
2 pstop dragon-prog
```

O comando `pstop` pertence à pasta `proc`, a qual contém as informações dos processos em execução no momento. Essa instrução finaliza o processo referenciado por argumento, o qual, no caso, é o *dragon-prog*.

Ainda, em `bin/onoffbutton`, encontra-se os scripts *shortpress_1.sh*, *shortpress_4.sh* e *verylongpress_0.sh*. Esses códigos iniciam, respectivamente, os protocolos para desligar, entrar no modo de depuração e redefinir o drone para as condições de fábrica.

A título de informação, *shortpress_1.sh* chama o script *delos_shutdown.sh* na pasta `bin`. Esse programa encerra o *dragon-prog*, esvazia a pasta temporária `tmp` e desliga a alimentação do drone. Por sua vez, *shortpress_4.sh* chama os scripts *delos_gadgetmode_stop.sh* e *delos_usb_debug_mode_switch.sh*, os quais montam a imagem de disco do drone para reconhecimento USB e configuram a interface para o IP `192.168.2.1`.

Por fim, em *verylongpress_0.sh* é chamado o script *delos_reset_factory.sh*. Esse último código finaliza o processo corrente do *dragon-prog* e apaga o interior da pasta `data` localizada no diretório raiz. Os demais *scripts* não são tratados nesta documentação. Para mais detalhes, é indicado acessar o sistema embarcado e consultar os comandos.

5.2 Instruções via FTP com o *firmware* modificado

Um método para comandar o drone sem a necessidade do MATLAB é enviando pelo terminal os pacotes para porta FTP 12391. Contudo, diferentemente dos comandos pelo terminal do SO Busy Box, essa estratégia não pode ser executada sobre o *firmware* nativo.

O drone com SO original não responde aos comandos de alimentação dos motores quando conectado ao computador por um cabo USB. Visto que o *firmware* original apenas disponibiliza a comunicação telnet quando o quadricóptero está conectado fisicamente ao computador, o Mambo não responderá aos pacotes enviados via FTP para decolagem e pouso. Há outras estratégias de comunicação via FTP com o sistema embarcado original, porém elas são discutidas na Seção 5.3.

Os pacotes enviados para o servidor foram descobertos a partir do estudo das funções `takeoff` e `abort` no arquivo `parrotio.m` apresentado na Seção 4.2. Para a decolagem, é chamada a função `writeToDrone(ID, PowerGain*100)`, cujos parâmetros são o ID hexadecimal da decolagem (`x01`) e o ganho dos motores, respectivamente. Um terceiro argumento pode ser atribuído para definir o tempo de duração de voo. Por sua vez, para o pouso é chamada a função `writeToDrone(ID)`, para o argumento ID o hexadecimal `x03` referente ao comando de pouso.

A função `writeToDrone` chama a função `sendData(Porta TCP, Pacote)`, cujos parâmetros são o objeto da classe `DroneConnect` e o pacote de mensagem. A classe `DroneConnect` está definida em `DroneConnect.m`, conforme discutido na Seção 4.2. O pacote será transferido para a porta TCP.

Esse pacote é um vetor resultado da concatenação dos argumentos de `writeToDrone`. Porém, antes de enviá-lo, esse vetor é convertido para inteiros de 32 bits. O pacote pode ser produzido e enviado manualmente para a porta por meio da instrução `netcat`. Por exemplo, o pacote referente ao processo de decolagem foi produzido com o seguinte código no MATLAB:

```
1 message = fopen("TakeOff", "w");
2 power = 20;
3 fwrite(message, [1 power], 'integer*4');
4 fclose(message);
```

A variável *power* representa a porcentagem de ganho dos motores. Nesse caso, foi utilizado 20%. O parâmetro *integer* simboliza 8 bits, os quais multiplicados por 4 resultam nos 32 bits de conversão do pacote. Analogamente, para o pouso, o pacote pode ser gerado manualmente a partir do seguinte código:

```

1 message = fopen("Land", "w");
2 fwrite(message, [3], 'integer*4');
3 fclose(message);

```

O envio desses pacotes para o drone é realizado por meio da instrução `netcat`. Para a emissão, deve-se entrar no terminal com o comando a seguir.

```
>> cat TakeOff | netcat 192.168.3.1 12391
```

Alterna-se a palavra *TakeOff* para *Land* para a execução do pouso. Outros pacotes podem ser encontrados no código *parrotio.m*. Um sumário dos possíveis comandos e argumentos estão dispostos na Tabela 5.1.

Tabela 5.1: Índice de comandos e argumentos dos pacotes enviados via comunicação TCP/IP na porta FTP 12391 do Parrot Mambo modificado com o *firmware* de suporte ao Simulink.

Comando	Função	ID hexadecimal	Argumento primário	Argumento secundário
Modificar a velocidade dos motores	<i>takeoff</i>	x01	1	PowerGain
Parar os motores	<i>land</i>	x02	2	—
Alterar o ângulo de giro	<i>writePitch</i>	x0B	11	(Pitch+10)*1000
Alterar o ângulo de rolagem	<i>writeRoll</i>	x0C	12	(Roll+10)*1000
Alterar o ângulo de guinada	<i>turn</i>	x0D	13	(Yaw+10)*1000
Informar nova altura em relação ao solo	<i>writeHeight</i>	x0E	14	ZPos*100
Informar nova posição XY de referência	<i>writePosition</i>	x0F	15	XPos*100, YPos*100
Ler dos sensores para as variáveis no servidor	<i>read</i>	x0A	10	—
Capturar imagem da câmera inferior	<i>snapshot</i>	x10	16	—
Parar os motores e sinalizar emergência	<i>abort</i>	x03	3	—

Na Tabela 5.1, o parâmetro `PowerGain` representa o ciclo de trabalho do sinal PWM enviado aos quatro motores. Já as variáveis `Pitch`, `Roll` e `Yaw` equivalem aos ângulos em radianos de arfagem, rolagem e guinada, respectivamente. A soma com fator 10 é justificada pela necessidade de um valor angular positivo para o drone.

Ainda na Tabela 5.1, `XPos`, `YPos` e `ZPos` representam a posição de referência em metros para o controlador do drone. É importante atentar que o referencial do eixo Z é negativo. Logo, para que o drone se mantenha a 30 centímetros do solo, por exemplo, `ZPos` deve ser equivalente a $-0,3$. As multiplicações por fatores de 10 ajustam o número de dígitos de precisão.

Esses novos comandos podem ser enviados modificando o segundo argumento da função `fwrite` para o vetor da mensagem. Tal vetor é composto pela ID hexadecimal e os argumentos da função apresentados a Tabela 5.1. A ordem desses parâmetros não pode ser alterada. É necessário destacar que o MATLAB forma os arquivos considerando os parâmetros inteiros. Logo, a ID hexadecimal deve ser convertida para a base decimal, conforme a coluna “Argumento primário” na Tabela 5.1.

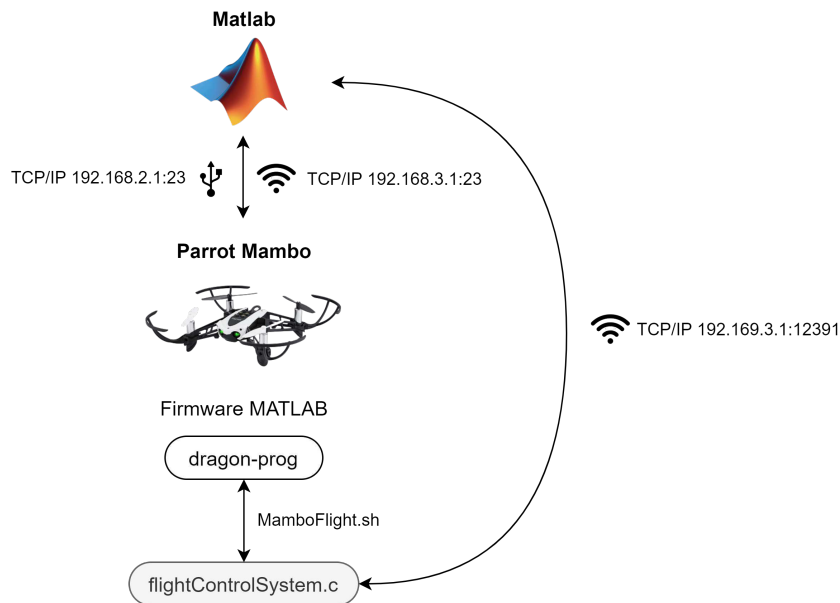
Essa compreensão dos comandos interpretados pelo drone permite o abandono das ferramentas licenciadas do MATLAB. Tendo em consideração que as instruções de alimentação dos motores e informação da pose de referência ao controlador são a base para a movimentação do drone, tornou-se possível ordenar uma trajetória específica por meio do terminal.

Logo, o servidor se torna dispensável e pode-se desenvolver uma plataforma aberta para controle do drone. Além de contornar a necessidade dos pacotes Simulink e Embedded Coder, essa plataforma pode abordar uma modelagem diferente e mais próxima do *firmware* nativo que o Simulink. Na Figura 5.2 estão ilustradas as portas de comunicação disponíveis para o envio de instruções para um drone com o *firmware* adaptado ao MATLAB.

5.3 Instruções via FTP com o *firmware* nativo

Como discutido na Seção 5.2, não é possível acionar os motores do drone com o sistema embarcado original por meio da comunicação USB. Contudo, é possível acessar o IP 192.168.99.3 do *firmware* nativo e enviar pacotes FTP diretamente ao quadricóptero utilizando o pacote de suporte a drones Parrot no MATLAB [19].

Figura 5.2: Diagrama das portas disponíveis no sistema com o *firmware* adaptado ao MATLAB para o envio de comandos pelo servidor.



Fonte: Autoria própria

Esse pacote é diferente do suporte a drones no Simulink [11], uma vez que emprega o controlador original do quadricóptero. Ele é composto por um conjunto de comandos para o terminal MATLAB e não possui interface gráfica.

Para utilizar esse pacote é necessária a câmera FVP acessória, a qual possui um ponto de acesso Wifi. A partir dessa rede UDP é possível conectar-se ao IP 192.168.99.3 do SO original e enviar os comandos. Por isso, é necessário estudar o código fonte do pacote para documentar a comunicação direta com o SO original.

5.3.1 Código fonte do pacote de suporte a drones

O código fonte do pacote de suporte para drones Parrot no MATLAB está localizado no caminho `<MATLAB>/SupportPackages/R2019b/toolbox/matlab/hardware/supportpackages/parrotio`, para `<MATLAB>` a pasta de instalação (por exemplo, `C:/ProgramData/MATLAB`). Nesta seção, essa pasta será renomeada `<RAIZ>`.

Será analisado o código `parrot.m` localizado em `<RAIZ>/@parrot`, o qual é o único script aberto do pacote. Nesse arquivo estão declaradas a classe `parrot` e funções de comunicação com o drone, tais como pouso, decolagem, rotação, translação e captura de imagens.

É importante destacar que esse pacote comunica com dois drones Parrot: o Mambo e o Bebop. Dessa forma, é preciso configurar o servidor para o quadricóptero que se deseja alcançar. Essa alteração é realizada ao criar o objeto *p* da classe *parrot* com o argumento "Mambo", conforme a linha a seguir:

```
>> p = parrot("Mambo");
```

O construtor da classe *parrot* chama a função *initProperties*, a qual define as portas de comunicação entre o servidor e o drone. Elas são:

- 44444, para a conexão TCP com o drone;
- 43210, referente à porta UDP do servidor no IP 192.168.99.3;
- 6000, referente à porta UDP do servidor drone no IP 192.168.99.1;

Ainda em *initProperties*, é definida a variável *ARSDKFileName* como 'minidrone'. Essa variável relaciona as mensagens construídas pelo MATLAB à tabela de instruções descrita no arquivo *minidrone.xml*¹, que está localizado em `<RAIZ>/+parrotio/+internal/`.

5.3.2 Formato dos pacotes de instrução

A associação entre a tabela *minidrone.xml* e os pacotes enviados para o drone é feita pelo método *readCommandPacket* da classe *parrot*, cujos argumentos são os nomes das classes declaradas em *minidrone.xml*. A exemplo, será exposta a seguir a chamada desse método na função membro de decolagem *takeoff*.

```
1 command = readCommandPacket(obj.XMLParser, obj.ARSDKFileName,  
2                               'Piloting', 'Takeoff');
```

Em *minidrone.xml*, a classe 'Piloting' está associada ao hexadecimal 0x00, ao passo que a classe 'TakeOff' está associada ao hexadecimal 0x01. É perceptível a semelhança com índice documentado na Tabela 5.1. A variável *command* é posteriormente enviada para o drone por meio do método *write*. Exibindo em tela a variável *command*, tem-se

```
>> command = 0x02 0x00 0x01 0x00;
```

¹Esse código é aberto e disponível para o desenvolvimento de plataformas de controle de minidrones Parrot. Acessível em <https://github.com/Parrot-Developers/arsdk-xml/blob/master/xml/minidrone.xml>

Essa é a mensagem enviada pelo servidor para o drone. O primeiro hexadecimal refere-se ao nome da classe topo 'minidrone' da hierarquia em *minidrone.xml*, seguido pelos valores de 'Piloting' e 'TakeOff'. Para completar os 32 bits do buffer, é adicionado o hexadecimal nulo ao fim da mensagem.

Um sumário das mensagens compreendidas pelo drone está apresentada na Tabela 5.2. As instruções que precisam de parâmetros, tais como atualizações da posição de referência, são compostas pela ID da mensagem de 32 bits concatenada com os parâmetros necessários convertidos em 8, 16 ou 32 bits.

Tabela 5.2: Índice de comandos e argumentos dos pacotes enviados via comunicação UDP na porta 6000 do Parrot Mambo com o *firmware* nativo.

Comando	Função	ID hexadecimal	Argumentos
Decolagem	<i>takeoff</i>	x02 x00 x01 x00	-
Pouso	<i>land</i>	x02 x00 x03 x00	-
Translação ou rotação	<i>move</i>	x02 x00 x02 x00	[1 roll pitch rotationSpeed verticalSpeed 0x00]
Rotação ao redor do eixo z	<i>turn</i>	x02 x04 x01 x00	angle
Cambalhota	<i>flip</i>	x02 x04 x00 x00	<ul style="list-style-type: none"> • Frente: x00 x00 x00 x00 • Trás: x01 x00 x00 x00 • Direita: x02 x00 x00 x00 • Esquerda: x03 x00 x00 x00

Na Tabela 5.2, o parâmetro **angle** representa o ângulo de rotação em radianos no sentido horário. Ele deve ser convertido para 16 bits. Já as variáveis **pitch**, **roll** e **equivaler** aos ângulos em radianos de arfagem e rolagem, respectivamente. Esses valores são convertidos para inteiros de 8 bits.

Ainda na Tabela 5.2, **rotationSpeed** e **verticalSpeed** representam, em 8 bits, a porcentagem da velocidade máxima de rotação e vertical, respectivamente, da movimentação do drone. O valor 1 do argumento de *move* é um único bit, e informa ao controlador que deverá acionar o controle de atitude.

Diferentemente do *firmware* discutido na Seção 5.2, o SO original só compreende os pacotes enviados com cabeçalho completo. Experimentalmente, constatou-se que o formato do pacote completo compreendido pelo controlador nativo é:

```
[Ack cmdNumber tag 0x00 0x00 0x00 0x00 ID data]
```

para ID e data referente às colunas “ID hexadecimal” e “Argumentos” da Tabela 5.2.

O inteiro Ack é composto por 16 bits e assume, na devida ordem, os valores 0x040b ou 0x020a se o drone responder à porta remetente do servidor ou não. Já o hexadecimal tag equivale a 0x0b para as instruções *takeoff* e *land*, 0x0d para *turn*, 0x0f para *flip* e 0x14 para *move*. O hexadecimal cmdNumber representa a numeração do comando enviado para o drone. Como os comandos não são obrigatoriamente recebidos em ordem pelo drone, essa variável informa ao controlador a ordem dos pacotes emitidos pelo servidor no canal UDP.

O monitoramento dos pacotes enviados entre o servidor e o drone foi realizado a partir do *software* Wireshark. Na Figura 5.3 está ilustrada a captura de tela do programa quando configurado para observar a comunicação com o sistema embarcado. Em amarelo, estão destacados os pacotes de configuração da conexão enviados via protocolo TCP. Em vermelho, estão realçados os pacotes de comando de enviados pelo servidor para a porta 6000 do drone. Por fim, as linhas evidenciadas em azul são referentes às instruções trocadas no período de “*stand-by*”, ou seja, entre as transferências de comandos de controle pela porta 6000.

Figura 5.3: Captura da tela do *software* Wireshark ao monitorar as portas de comunicação entre o servidor e o drone com o sistema operacional original.

1	0.000000	192.168.99.32	192.168.99.3	TCP	66 55911 → 44444 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SAC...
2	0.002777	192.168.99.3	192.168.99.32	TCP	66 44444 → 55911 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 ...
3	0.002886	192.168.99.32	192.168.99.3	TCP	54 55911 → 44444 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4	0.048527	192.168.99.32	192.168.99.3	TCP	131 55911 → 44444 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=77
5	0.050626	192.168.99.3	192.168.99.32	TCP	54 44444 → 55911 [ACK] Seq=1 Ack=78 Win=5840 Len=0
6	0.086926	192.168.99.3	192.168.99.32	TCP	132 44444 → 55911 [PSH, ACK] Seq=1 Ack=78 Win=5840 Len=78
7	0.100793	192.168.99.3	192.168.99.32	TCP	54 44444 → 55911 [FIN, ACK] Seq=79 Ack=78 Win=5840 Len=0
8	0.100859	192.168.99.32	192.168.99.3	TCP	54 55911 → 44444 [ACK] Seq=78 Ack=80 Win=65536 Len=0
9	0.140181	192.168.99.32	192.168.99.3	UDP	61 60025 → 6000 Len=19
10	0.142277	192.168.99.32	192.168.99.3	TCP	54 55911 → 44444 [FIN, ACK] Seq=78 Ack=80 Win=65536 Len=0
11	0.144632	192.168.99.3	192.168.99.32	TCP	54 44444 → 55911 [ACK] Seq=80 Ack=79 Win=5840 Len=0
12	0.220496	192.168.99.32	192.168.99.3	UDP	63 43210 → 6000 Len=21
13	0.220597	192.168.99.32	192.168.99.3	UDP	65 43210 → 6000 Len=23
14	0.235873	192.168.99.32	192.168.99.3	UDP	53 43210 → 6000 Len=11
15	0.240741	192.168.99.3	192.168.99.32	UDP	69 40737 → 43210 Len=27
16	0.244460	192.168.99.3	192.168.99.32	UDP	50 40737 → 43210 Len=8
17	0.245078	192.168.99.3	192.168.99.32	UDP	50 40737 → 43210 Len=8
18	0.301558	192.168.99.3	192.168.99.32	UDP	66 40737 → 43210 Len=24
19	0.301890	192.168.99.32	192.168.99.3	UDP	50 60024 → 6000 Len=8
20	0.326666	192.168.99.3	192.168.99.32	UDP	63 40737 → 43210 Len=21

Fonte: Autoria própria

Essa engenharia reversa foi empregada apenas para o monitoramento do protocolo de comunicação do sistema nativo. A documentação das instruções interpretadas pelo *firmware* educacional foi realizada a partir do código fonte do pacote de suporte a minidrones.

Por meio do MATLAB, pode-se enviar os pacotes manualmente para o drone. Por exemplo, com o código a seguir foram construídos e enviados os comandos de decolagem, rotação de 45 graus ao redor do eixo z e pouso.

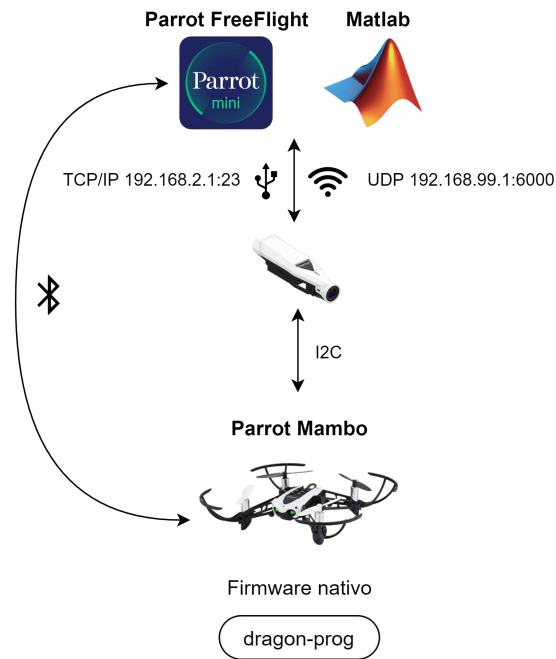
```
1 cmdNumber = 9;
2 angle     = 45;
3 takeOff   = [0x04 0x0b uint8(cmdNumber)
4             0x0b 0x00 0x00 0x00
5             0x02 0x00 0x01 0x00];
6 land     = [0x04 0x0b uint8(cmdNumber+1)
7             0x0b 0x00 0x00 0x00
8             0x02 0x00 0x03 0x00];
9 turn     = [0x04 0x0b uint8(cmdNumber+2)
10            0x0d 0x00 0x00 0x00
11            0x02 0x04 0x01 0x00
12            typename(int16(angle), 'uint8')];
13 p = parrot("Mambo");
14 u = udp('192.168.99.3', 6000, 'LocalPort', 54321);
15 fopen(u);
16 fwrite(u, takeOff, 'int8');
17 fwrite(u, turn, 'int8');
18 fwrite(u, land, 'int8');
19 fclose(u);
```

O objeto da classe `udp` estabelece a conexão com a porta 6000 do IP 192.168.99.3 da rede Wifi da câmera. Analogamente à Seção 5.2, as mensagens são enviadas por meio da função `fwrite`. Na Figura 5.4 estão ilustradas as portas de comunicação disponíveis para o envio de instruções para um drone com o *firmware* nativo.

5.4 Considerações finais

A comunicação direta com o drone é importante para o desenvolvimento de novas aplicações em ambientes acadêmicos. Tendo em vista a independência de licenças ou programas específicos, a documentação realizada nesse capítulo é essencial para ampliar os testes de algoritmos personalizados no Parrot Mambo. Como exemplo da comunicação com o protótipo simulado e físico, no próximo capítulo discute-se sobre dois ensaios com o sistema embarcado. O primeiro experimento é um aperfeiçoamento da planta modelada, enquanto o segundo verificará a conexão e leitura do registro de dados do protótipo físico. Em outras palavras, é um ensaio de *software-in-the-loop* e um ensaio de *hardware-in-the-loop*. Também

Figura 5.4: Diagrama das portas disponíveis no sistema com o *firmware* nativo para o envio de comandos pelo servidor.



Fonte: Autoria própria

será demonstrado como realizar uma modificação no controlador embarcado.

Capítulo 6

Utilizando MAD para testes HIL/SIL

Este capítulo descreve dois ensaios sobre o algoritmo de fluxo óptico implementado no drone. Primeiramente, é discutida a modelagem matemática desse algoritmo. Em seguida, apresentam-se os experimentos de SIL e HIL, nessa ordem. Por fim, é descrito um experimento modificando o controlador para exemplificar o fluxo MAD para o Parrot Mambo.

6.1 Fluxo óptico

O fluxo óptico é um método de estimação o deslocamento de um ponto em relação a câmera a partir da comparação de duas imagens. Esse algoritmo representa o movimento relativo entre um ponto P e espectador no espaço tridimensional por meio de vetores.

Considerando $P = [X, Y, Z]^T$ e $p = [x, y, z]^T$ as coordenadas de um ponto da imagem em unidades no mundo e no plano focal da câmera, respectivamente, a relação de escala entre P e p para f a distância focal da câmera é expressa por:

$$p = f \frac{P}{Z} \quad (6.1)$$

Nesse caso, as coordenadas de P e p não representam a posição absoluta do ponto, mas a posição do ponto em relação a imagem capturada no ciclo anterior.

A velocidade $\nu = [\nu_x, \nu_y, \nu_z]^T$ do ponto p , isto é, do ponto P no plano focal da lente é determinada por

$$\nu = \frac{dp}{dt} \quad (6.2)$$

Substituindo a Equação 6.1 na Equação 6.2, resulta-se em

$$\nu = \frac{f}{Z^2} \left[Z \frac{dP}{dt} - P \frac{dZ}{dt} \right] = \frac{f}{Z^2} \left[ZV - PV_z \right] \quad (6.3)$$

Por sua vez, a velocidade relativa $\nu = [\nu_x, \nu_y, \nu_z]^T$ entre o ponto P e o espectador, ou seja, a câmera, em unidades do mundo é dada por

$$V = -T - \omega \times P \quad (6.4)$$

para $T = [T_x, T_y, T_z]^T$ a componente translacional e $\omega = [\omega_x, \omega_y, \omega_z]^T$ a componente angular da velocidade. Desenvolvendo a Equação 6.4, resulta-se em

$$V = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} = - \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} - \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} -T_x - \omega_y Z + \omega_z Y \\ -T_y - \omega_z X + \omega_x Z \\ -T_z - \omega_x Y + \omega_y X \end{bmatrix} \quad (6.5)$$

Substituindo Equação 6.5 em Equação 6.3, encontra-se

$$\nu = \begin{bmatrix} \nu_x \\ \nu_y \\ \nu_z \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} -T_x - \omega_y Z + \omega_z Y \\ -T_y - \omega_z X + \omega_x Z \\ -T_z - \omega_x Y + \omega_y X \end{bmatrix} + \frac{f}{Z^2} (T_z + \omega_x Y - \omega_y X) \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

No Parrot Mambo, o fluxo óptico é empregado sobre as imagens capturadas pela câmera inferior, a qual aponta para o plano XY. As componentes em ν_x e ν_y da velocidade relativa do ponto p em relação à câmera são

$$\begin{aligned} \nu_x &= \frac{f}{Z^2} T_z(X) - \frac{f}{Z} T_x - f\omega_y + \frac{f}{Z} \omega_z(Y) + \frac{f}{Z^2} \omega_x(XY) - \frac{f}{Z^2} \omega_y(X^2) \\ \nu_y &= -\frac{f}{Z} T_y - \frac{f}{Z} \omega_z(X) + f\omega_x + \frac{f}{Z^2} T_z(Y) + \frac{f}{Z^2} \omega_x(Y^2) - \frac{f}{Z^2} \omega_y(XY) \end{aligned} \quad (6.6)$$

Essa aproximação do fluxo óptico é nomeada método de Horn–Schunck [20]. Ao evidenciar a distância focal f na Equação 6.6, obtém-se

$$\frac{\nu_x}{f} = \frac{X}{Z^2} T_z - \frac{T_x}{Z} - \omega_y + \frac{Y}{Z} \omega_z + \frac{XY}{Z^2} \omega_x - \frac{X^2}{Z^2} \omega_y \quad (6.7)$$

$$\frac{\nu_y}{f} = \frac{Y}{Z^2}T_z - \frac{T_y}{Z} + \omega_x - \frac{X}{Z}\omega_z + \frac{Y^2}{Z^2}\omega_x - \frac{XY}{Z^2}\omega_y \quad (6.7)$$

Os termos no membro esquerdo da Equação 6.7 representam os dados de saída do fluxo óptico modelado no Parrot Mambo. Contudo, a modelagem no diagrama de blocos do Simulink é simplificada e não corresponde aos membros do lado direito da Equação 6.7. A equação modelada no pacote é descrita na Seção 6.2.

Como a câmera inferior do Parrot Mambo possui foco automático, a distância focal f é equivalente à distância do drone ao plano XY, ou seja, a altitude Z . Nesse caso, a partir da Equação 6.1, conclui-se que

$$\nu_z = V_z$$

6.2 Ajustando o modelo de diagrama de blocos

No modelo em diagrama de blocos do Parrot Mambo, o fluxo óptico estima a velocidade do drone no plano XY dividida pela distância focal f , em concordância com a Equação 6.7. No bloco do estimador apresentado na Seção 3.2.2, esse valor é multiplicado pela altitude estimada Z , isto é, a distância focal, e um fator de correção (constante equivalente a 1.15). Desse modo, são encontrados os termos ν_x e ν_y descritos na Equação 6.6.

Esses resultados alimentam o bloco *EstimatorXYPosition*, o qual é responsável por informar ao controlador a posição estimada p_x e p_y do drone por meio da integral das velocidades estimadas \dot{p}_x e \dot{p}_y . É importante destacar que os estados estimados de velocidade (\dot{p}_x , \dot{p}_y , \dot{p}_z) e altitude (p_z) do modelo cinemático são produzidos após a filtragem de Kalman das medidas do IMU, do sensor ultrassônico e do fluxo óptico.

Conforme discutido na Seção 6.1, o fluxo óptico modelado no Simulink é simplificado. O modelo nativo do diagrama de blocos é semelhante a aproximação de Lucas-Kanade [21]e. As equações representadas no diagrama de blocos do pacote são:

$$\begin{aligned} \frac{\nu_x}{f} &= -T_x + 0,8\omega_y Z \\ \frac{\nu_y}{f} &= -T_y - 0,8\omega_x Z \end{aligned} \quad (6.8)$$

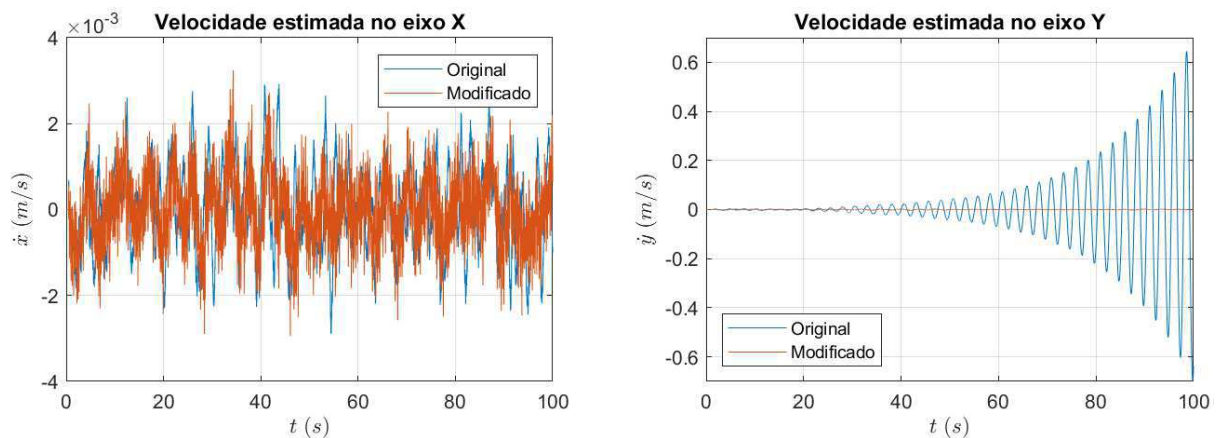
Comparando-as com a Equação 6.6, fica claro que são representados no modelo Simulink apenas o segundo e terceiro termo da estimativa. Essa síntese prejudica o fluxo SIL, uma vez que não representa a realidade mais aproximada do modelo físico.

Em uma missão *Hover*, o drone decola para uma altitude de 1,1 metros e mantém-se parado até o fim do tempo de simulação (100 segundos). Contudo, ao simular o projeto, observou-se que havia uma instabilidade dos estados estimados de velocidade \dot{p}_y . Essa oscilação se propaga para a estimação da pose atual do drone informada ao controlador.

Com o intuito de otimizar a precisão do algoritmo do fluxo óptico em simulação, foi implementada a Equação 6.7 no diagrama de blocos do Simulink.

A velocidade estimada após o filtro de Kalman utilizando a Equação 6.8 e Equação 6.7 estão apresentados na Figura 6.1. É perceptível que, no eixo Y, há um erro cumulativo na estimativa v_y , o qual é justificado pelo sinal negativo no termo ω_x modelado na Equação 6.8. Com o modelo aprimorado, o valor \dot{p}_y variou menos do que no equacionamento original.

Figura 6.1: Velocidade estimada do drone no plano XY em missão simulada de *Hover*.

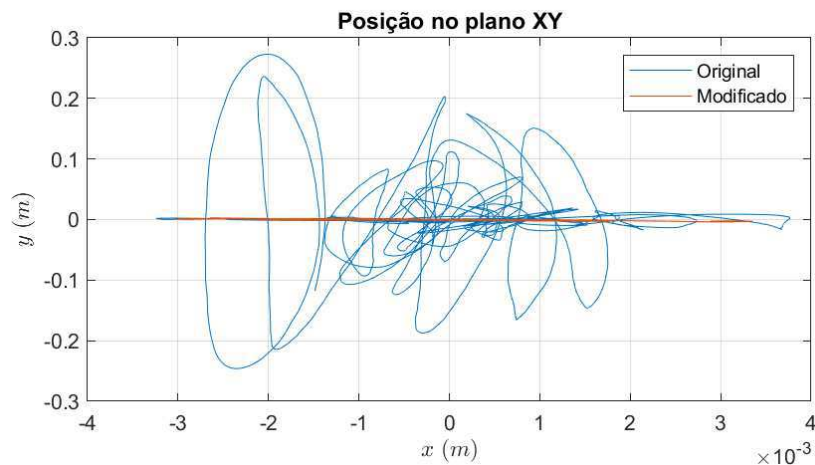


Fonte: Autoria própria.

O erro entre as curvas de velocidade, quando integrado, provoca variações na ordem de 20 centímetros da posição real do drone. Os estados p_x e p_y estimados pela integração dos resultados do fluxo óptico originalmente modelado no drone e do algoritmo ajustado estão ilustrados na Figura 6.2. É perceptível que o novo modelo de fluxo óptico está mais próximo da posição real do drone, ou seja, planando sobre a origem dos eixo X e Y do que o algoritmo original.

Essa oscilação da velocidade no plano XY também atinge o cálculo dos conjugados pelo

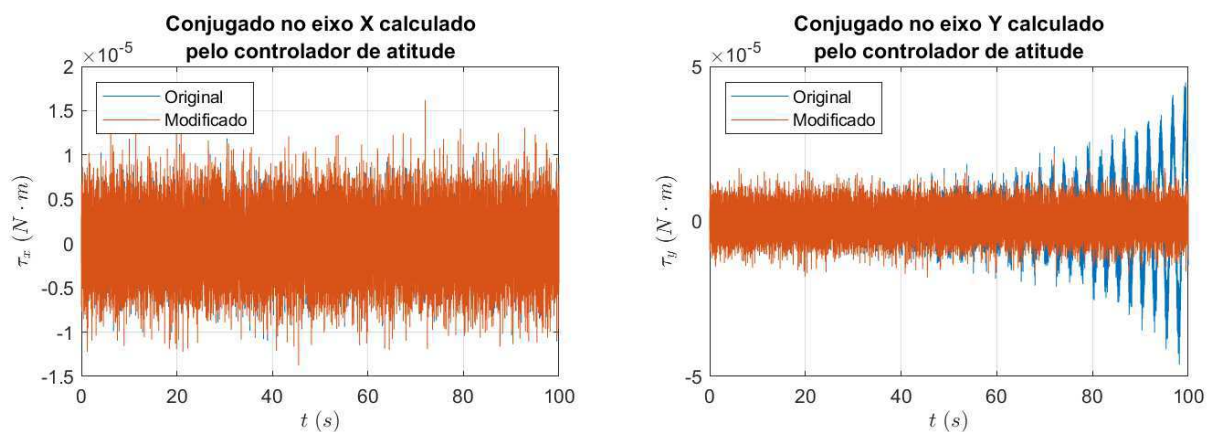
Figura 6.2: Posição estimada do drone no plano XY em missão simulada de *Hover*.



Fonte: Autoria própria.

controlador de atitude, cuja referência é erro entre a posição almejada e posição estimada pelo fluxo óptico. Experimentalmente, a partir de 80 segundos decorridos da simulação do modelo original, há uma oscilação do torque τ_y , conforme ilustrado na Figura 6.3.

Figura 6.3: Conjugados τ_x e τ_y computados pelo controlador de atitude em missão simulada de *Hover*.



Fonte: Autoria própria.

Essa oscilação do conjugado é da ordem de $4 \cdot 10^{-5}$ Nm para o modelo do fluxo óptico original do pacote Simulink [11], enquanto que o valor máximo de τ_y para o fluxo óptico modificado é $1,5 \cdot 10^{-5}$ Nm. Logo, é possível concluir que o modelo proposto nesse experimento é mais estável que o implementado inicialmente pelo MATLAB.

6.3 Testando o protótipo físico

A alteração do fluxo óptico em simulação não pode ser implementada no protótipo físico, visto que a compilação cruzada não abrange o modelo do drone simulado. Com o intuito de verificar se o fluxo óptico implementado no drone estima corretamente o deslocamento, foi modificado o código *rstedu_of.c* para exibir, no registro, os valores de ν_x/f e ν_y/f encaminhados ao estimador. Junto a esses valores, foi exportada a marca temporal de cada exibição em tela e a altitude medida pelo ultrassônico inferior. Esse dado foi amostrado em duas taxas: 60Hz, equivalente à *thread* de visão; e 200Hz, equivalente à *thread* de controle.

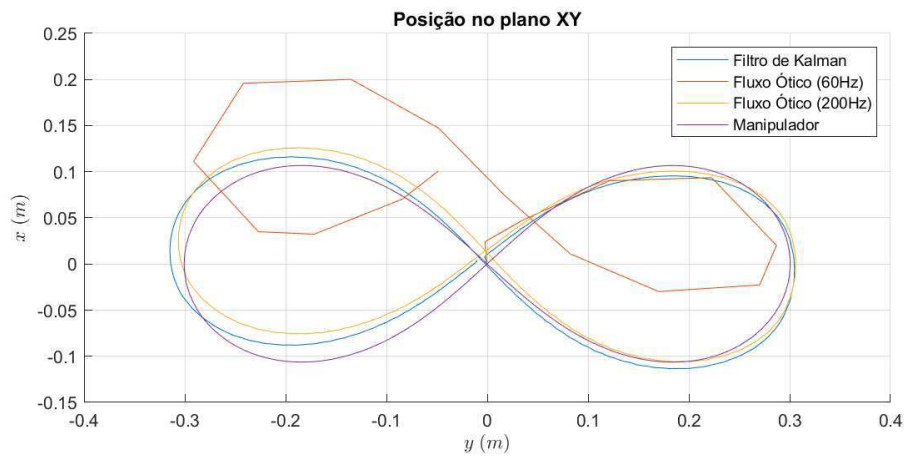
A partir dos valores capturados, foi possível reconstruir a trajetória produzida pelo drone. Primeiramente, multiplicou-se a saída do fluxo óptico pela altitude estimada z após o filtro de Kalman, obtendo as velocidades estimadas ν_x e ν_y . Em seguida, esses resultados foram corrigidos pelo fator 1.15 e integrados no tempo para determinar a posição estimada x e y do drone.

Para o experimento, foi planejada a trajetória de uma lemniscata com 60 centímetros de comprimento. Para garantir a precisão do movimento, o drone foi acoplado a um manipulador robótico à 60 centímetros da superfície XY, o qual percorria a trajetória planejada. Com o intuito de garantir a melhor performance do algoritmo de fluxo óptico, a superfície sobre a qual o drone se movimentava foi personalizada com faixas coloridas não regulares.

Na Figura 6.4 está apresentada a trajetória estimada do drone a partir das velocidades coletadas do fluxo óptico, da posição corrigida pelo estimador e da real posição do manipulador ao qual o drone estava acoplado. Observando a Figura 6.4, é claro que a trajetória corrigida pelo filtro de Kalman é mais próxima da posição real do manipulador que a posição calculada pelo integração das velocidades estimadas pelo fluxo óptico.

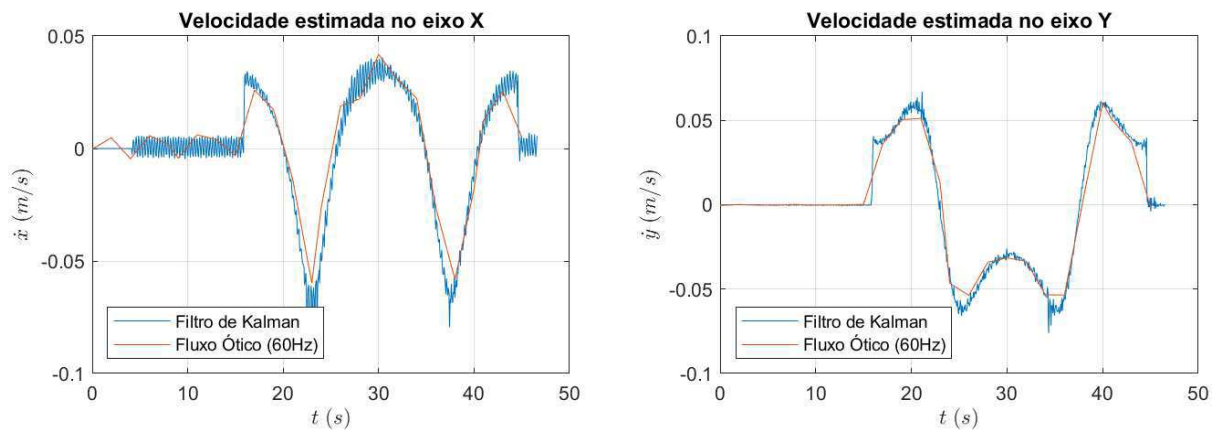
A diferença entre as trajetórias baseadas no fluxo óptico amostrado em 60Hz ou em 200Hz é justificada pela integral da velocidade estimada no tempo. Tendo em vista a curva da velocidade amostrada em 60Hz ilustrada na Figura 6.5, há apenas uma amostra negativa para cada mínimo local da lemniscata no eixo X. Dessa forma, ao integrar o dado amostrado a 60Hz, não há um acúmulo suficiente de valores negativos para reconstruir o primeiro mínimo local da trajetória em X. Esse erro é observável na Figura 6.6 por meio da diferença entre a posição estimada com a amostragem de 60Hz e 200Hz.

Figura 6.4: Posição do drone no plano XY em uma trajetória de lemniscate.



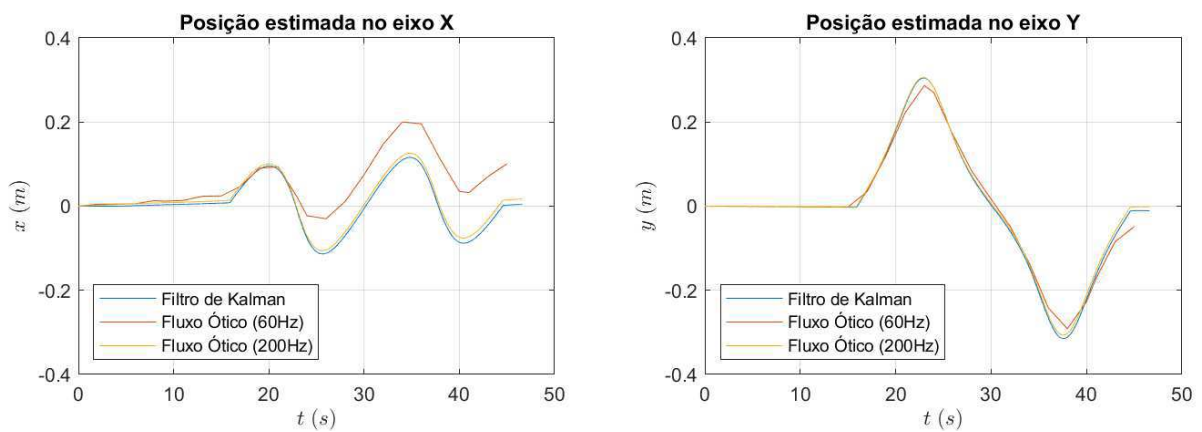
Fonte: Autoria própria.

Figura 6.5: Velocidade estimada do drone em uma trajetória de lemniscate.



Fonte: Autoria própria.

Figura 6.6: Posição estimada do drone em uma trajetória de lemniscate.



Fonte: Autoria própria.

O erro médio absoluto entre o caminho percorrido pelo braço robótico e a trajetória reconstruída pelo fluxo óptico implementado no protótipo físico é 4 milímetros. Esse erro ainda é maior que a diferença entre trajetória medida em simulação com o equacionamento modificado do fluxo óptico, o qual foi 0,7 milímetros. Dessa forma, é possível concluir que o fluxo óptico no drone real, mesmo sendo mais preciso que a planta originalmente simulada no pacote Simulink, não é tão exato quanto o equacionamento completo ajustado na Seção 6.2.

6.4 Modificando o controlador nativo

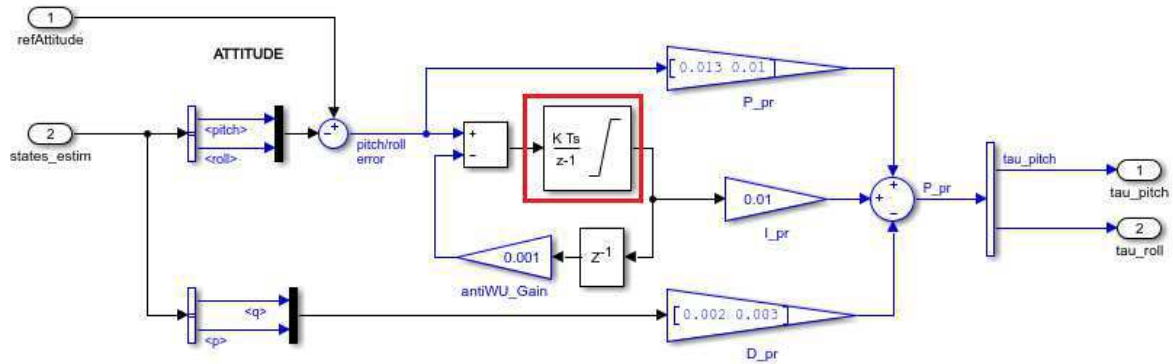
Conforme descrito no Capítulo 5, a diferença entre os dois pacotes oferecidos para a ferramenta MATLAB consiste no acesso do Simulink e Embedded Coder ao controlador do VANT. Com o intuito de exemplificar o fluxo de desenvolvimento de um controlador de voo para o Parrot Mambo utilizando o pacote de suporte de drones para o Simulink [11], essa seção trata da implementação do controlador de altitude e atitude utilizando aproximação de Tustin. Por padrão do diagrama de blocos, a integral do PID é originalmente discretizada conforme a aproximação de Euler.

A alteração da aproximação do controlador PID pode ser realizada modificando o método de integração do bloco do controlador de altitude e atitude. Esses controladores estão representados na Figura 3.11 como *gravity feedforward/equilibrium thrust* e *Attitude*, respectivamente. Clicando duas vezes sobre o bloco da integral discreta em destaque na Figura 6.7 e Figura 6.8, é possível alterar o parâmetro “*Integrator method*” de *Foward Euler* para o método integrativo de Tustin *Trapezoidal*.

Tendo em vista que o termo proporcional é estático no tempo [22], não é necessário modificar o ramo P. A operação derivativa não precisa ser aproximada pelo método de Tustin, posto que o caminho derivativo é calculado diretamente com os estados estimados de velocidade \dot{p}_z , $\dot{\phi}$ ou $\dot{\theta}$.

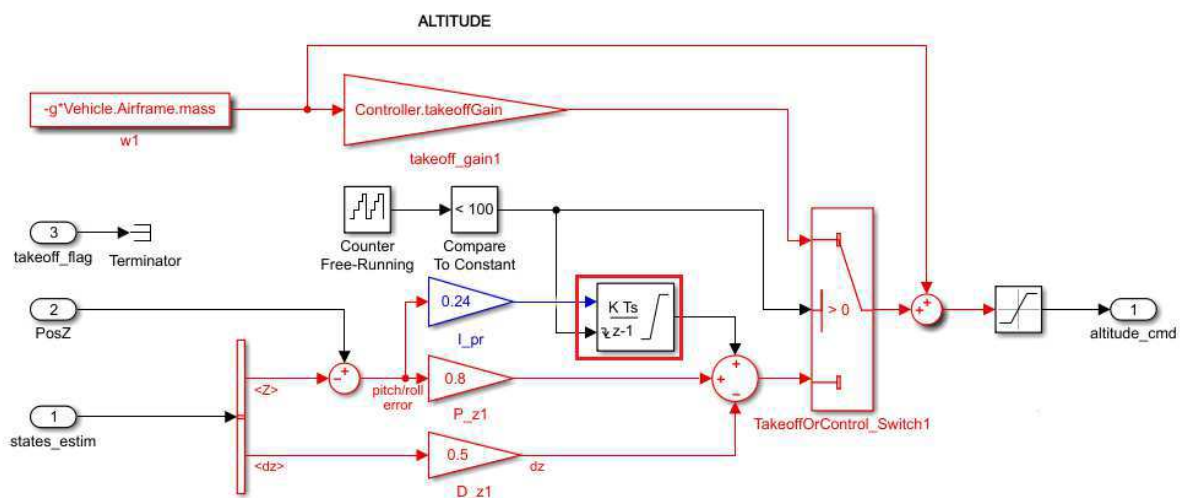
Considerando o modelo de fluxo óptico otimizado, foram simulados dois ensaios: no primeiro, foi simulado o modelo com os controladores aproximados pela formulação de Euler e, no segundo, com aproximação de Tustin. Durante as simulações, foram adquiridos os conjugados τ_x e τ_y computados pelo controlador de atitude e o empuxo vertical T calculado pelo controlador altitude, respectivamente ilustrados na Figura 6.9 e Figura 6.10.

Figura 6.7: Diagrama de blocos do controlador de atitude. O bloco da integral discreta em destaque deve ser modificado para a aproximação de Tustin.



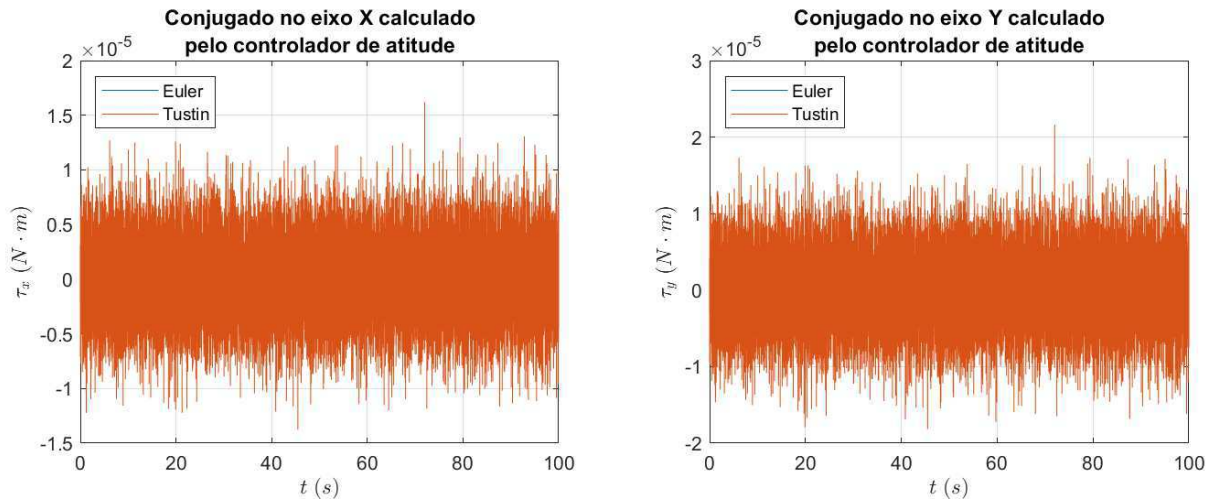
Fonte: MATHWORKS[11].

Figura 6.8: Diagrama de blocos do controlador de altitude. O bloco da integral discreta em destaque deve ser modificado para a aproximação de Tustin.



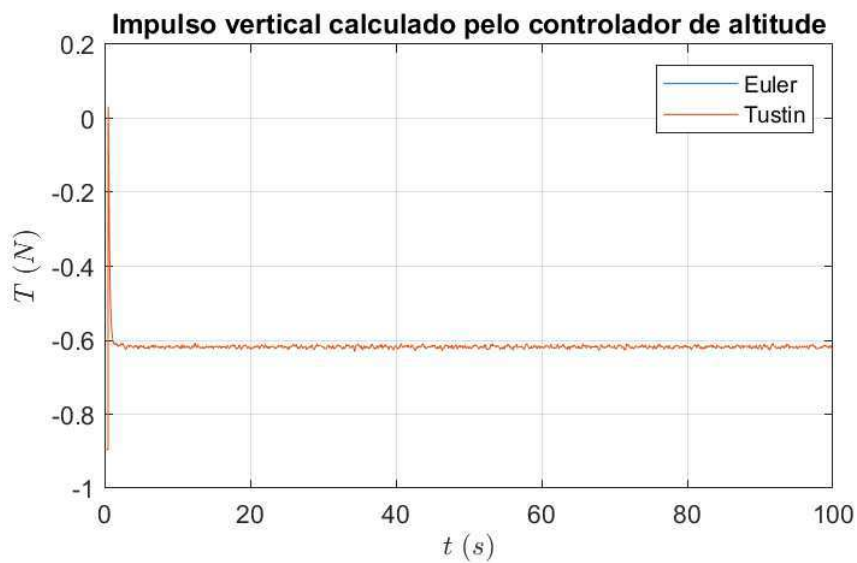
Fonte: MATHWORKS[11].

Figura 6.9: Conjugados τ_x e τ_y calculado pelo controlador de atitude PID com discretização de Euler e de Tustin.



Fonte: Autoria própria.

Figura 6.10: Empuxo vertical T calculado pelo controlador de altitude PID com discretização de Euler e de Tustin.



Fonte: Autoria própria.

Observando as Figura 6.9 e Figura 6.10, é evidente que os resultados são semelhantes entre as aproximações pois não atingem a sintonia dos ganhos dos controladores. O erro quadrático médio entre a discretização do integrador pelo método de Euler e de Tustin para o empuxo vertical é $8,30 \cdot 10^{-11}$ N. Por sua vez, o erro quadrático médio entre a discretização pela aproximação de Euler e de Tustin para os conjugados τ_x e τ_y são, respectivamente, $3,59 \cdot 10^{-18}$ Nm e $1,35 \cdot 10^{-18}$ Nm.

6.5 Considerações finais

O fluxo óptico é o principal método de estimação de posição do Parrot Mambo. Verificou-se que o equacionamento modelado no pacote original era simplificado e provocava oscilações no drone. Com o aprimoramento da planta no Simulink, encontrou-se curvas de trajetória simulada mais próximas da realidade que o modelo original dos sensores. Com o intuito de verificar se o equacionamento implementado no *hardware* também era simplificado, o drone foi acoplado a um manipulador robótico, o qual percorreu uma trajetória bem definida. Para avaliar a proximidade das curvas com a trajetória simulada e real executada pelo drone, utilizou-se o erro médio absoluto. No ambiente virtual, o erro médio absoluto foi 0,7 milímetros, enquanto no experimento com o protótipo físico, os dados indicaram um erro médio absoluto de 4 milímetros. A missão executada sobre a planta original do pacote resultou em um erro médio absoluto de 20 centímetros. Com esses resultados, foi constatado que o fluxo óptico modelado no pacote Simulink e implementado no drone é correto, porém simplificado. O capítulo 7 apresenta as conclusões e os trabalhos futuros que podem empregar a documentação desse estudo para aprimorar mais componentes modelados no pacote de suporte ao Parrot Mambo no Simulink.

Capítulo 7

Conclusão

Neste trabalho de conclusão de curso, foram discutidos os modelo dinâmico de um quadricóptero e as estratégias de controle para que esse dispositivo percorra a trajetória planejada. Esses pontos foram base para a análise da planta virtual simulada e a compreensão do sistema embarcado e do protocolo de comunicação entre o drone e o servidor. Com essa documentação, foi possível avaliar a aplicação do método MAD/GAC no desenvolvimento e teste de um VANT.

O quadricóptero Parrot Mambo é uma plataforma de teste de pequeno porte adequada para produções acadêmicas. Esse drone possui sensores de pressão, altitude, aceleração e fluxo óptico. O sistema operacional nativo é fechado, impedindo a alteração da estratégia de controle ou aquisição de telemetria em testes. Para desviar esse obstáculo, foi documentado o Modelo Aplicado em Desenvolvimento oferecido pelo pacote de suporte a minidrones Parrot no Simulink. Esse *firmware* alternativo é aberto, porém possui uma performance mais lenta. Essa menor eficiência foi justificada pela modelagem inadequada da planta virtual, uma vez que o controlador desenhado possuía boa performance em simulação, mas não era adequado para o protótipo físico.

O modelo matemático do espaço de estados e do controlador discutidos no capítulo 2 foram relacionados com a representação por blocos na ferramenta licenciada Simulink. Essa apresentação em diagrama facilita a criação de novas estratégias de controle, abstraindo a geração de código. Contudo, foi comprovado que essa representação impede alterações em linguagem de máquina, uma vez que limita a comunicação do *firmware* dedicado ao servidor

MATLAB. Dessa maneira, foi investigado o protocolo de comunicação entre o Parrot Mambo e o usuário, com o intuito diversificar das modificações possíveis no sistema embarcado.

A geração de código automática do sistema de controle do drone foi documentada, o que proporcionou o traçado do fluxo de controle do sistema embarcado. Com essa análise, foi possível definir quais são os comandos nativos do sistema operacional embarcado, as portas de comunicação e as instruções interpretadas pelo drone tanto na sua configuração original, quanto o formato adaptado para o servidor MATLAB. Foi constatado que, no *firmware* nativo, não há possibilidade de alterar os parâmetros do controlador, mas as variedades de comandos compreendidos pelo drone é maior do que no *firmware* adaptado para o pacote Simulink. Foi também certificado que a diferença entre esses *firmwares* reside no fato deste último abrir uma porta de comunicação direta com o controlador, a qual é essencial para depuração da nova estratégia de controle.

Em posse da numeração de portas e dos comandos de pouso, decolagem e deslocamento enviados para as duas versões do sistema embarcado, foi tabelado o protocolo de comunicação entre o drone e qualquer servidor. Essa documentação possibilita a independência das ferramentas licenciadas e oportuna o desenvolvimento de projetos *open-source*.

Para exemplificar o subdesempenho da planta modelada no ambiente de teste virtual, foi verificada a oscilação da ordem de 20 centímetros da posição estimada no eixo Y originada da velocidade estimada pelo fluxo óptico. Para contornar esse problema, foi implementada a equação completa do fluxo óptico no diagrama de blocos. É importante destacar que esse algoritmo, no Mambo, retorna a divisão da velocidade estimada pela distância focal da câmera inferior. Como esse valor foi multiplicado pela altitude do drone, foi constatada que a distância focal equivale à distância do drone ao plano XY, ou seja, sua altitude.

Com o conhecimento dos códigos-fonte do pacote Simulink e da planta modelada no ambiente virtual, foi modificado o sistema operacional embarcado para informar os dados capturados pelo fluxo óptico implementado no drone. O Mambo foi acoplado a um manipulador robótico, o qual percorreu a trajetória de uma lemniscata. Foram comparadas as curvas da trajetória real descrita pelo manipulador e da posição estimada do drone pelo algoritmo de fluxo óptico simulado e implementado. O erro médio absoluto entre a trajetória simulada e a do manipulador foi 0,7 milímetros, enquanto entre a trajetória capturada pela telemetria do drone real e a do manipulador foram 4 milímetros. Assim, é possível afirmar

que a simulação com a modelagem corrigida do fluxo óptico é correta e que o algoritmo implementado no drone é aproximado e não representa o equacionamento completo. Isso demonstra que o Mambo possui espaço para otimização, a qual pode ser efetivada a partir desta documentação do sistema embarcado anteriormente desconhecido.

7.1 Trabalhos futuros

Como sugestão de trabalhos futuros nessa linha de pesquisa, é possível citar:

- incluir modelo da bateria e dos atuadores do Parrot Mambo no diagrama Simulink;
- refinar ou modelar novos controladores para o modelo Simulink do Parrot Mambo considerando condições ambientais não ideais;
- desenvolver aplicações abertas de comunicação com o Mambo para envio de instruções e captura da telemetria, sem a inclusão de bibliotecas ou ferramentas licenciadas.

Referências bibliográficas

- 1 AL-RADAIDEH, A.; AL-JARRAH, M. A.; JHEMI, A. UAV Testbed building and development for research purposes at the American University of Sharjah. *7th International Symposium on Mechatronics and its Applications*, Abril 2010.
- 2 FORSTER, C.; PIZZOLI, M.; SCARAMUZZA, D. SVO: Fast semi-direct monocular visual odometry. *IEEE International Conference on Robotics and Automation*, Maio 2014. ISSN 1050-4729.
- 3 LI, Z.; LIU, Z.; DING, B.; LIAO, X.; HU, H. Explore the rapid prototyping of advanced flight control algorithms: From simulation to actual flight. *5th International Conference on Automation, Control and Robotics*, p. 413–416, Abril 2019. ISSN 2251-2446.
- 4 HO, H. W.; CROON, G. C. de; CHU, Q. Distance and velocity estimation using optical flow from a monocular camera. *International Journal of Micro Air Vehicles*, v. 9, n. 3, p. 198–208, Janeiro 2017.
- 5 KAPLAN, M. R.; ERASLAN, A.; BEKE, A.; KUMBASAR, T. Altitude and Position Control of Parrot Mambo Minidrone with PID and Fuzzy PID Controllers. *11th International Conference on Electrical and Electronics Engineering*, p. 785–789, Novembro 2019.
- 6 MOSTERMAN, P. J. Automatic Code Generation: Facilitating New Teaching Opportunities in Engineering Education. *36th Annual Proceedings Frontiers in Education Conference*, Outubro 2006. ISSN 2377-634X.
- 7 AARENSTRUP, R. *Managing Model-Based Design*. 1. ed. Natick: MathWorks, 2015. 4-12 p. ISBN 978-1512036138.
- 8 NETLAND, O.; SKAVHAUG, A. Software Module Real-Time Target: Improving Development of Embedded Control System by Including Simulink Generated Code Into Existing Code. *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, p. 232–235, Setembro 2013. ISSN 2376-9505.
- 9 KRIZAN, J.; ERTL, L.; BRADAC, M.; JASANSKY, M.; ANDREEV, A. Automatic code generation from Matlab/Simulink for critical applications. *IEEE 27th Canadian Conference on Electrical and Computer Engineering*, Maio 2014. ISSN 0840-7789.
- 10 MOSTERMAN, P. J.; PRABHU, S.; ERKKINEN, T. An industrial embedded control system design process. *Proceedings of the Canadian Engineering Education Association*, Agosto 2011.

- 11 MATHWORKS. *Parrot Minidrones Support from Simulink*. Natick, 2019. Disponível em: <https://www.mathworks.com/help/pdf_doc/supportpkg/parrot/parrot_ug.pdf>. Acesso em: 20 mai. 2020.
- 12 MATHWORKS. *Simulink Coder*. Natick, 2019. Disponível em: <https://www.mathworks.com/help/pdf_doc/rtw/rtw_ug.pdf>. Acesso em: 20 mai. 2020.
- 13 RAZA, S. A.; GUEAIEB, W. *Motion Control*. 1. ed. Olajnic: In-Tech, 2010. 253-263 p. ISBN 9789537619558.
- 14 CORKE, P. *Robotics, Vision and Control*. 1. ed. Berlin: Springer Tracts, 2011. 78-81 p. ISBN 9783319544137.
- 15 SABATINO, F. *Quadrotor control: modeling, nonlinear control design, and simulation*. Dissertação (Mestrado em Engenharia Elétrica) — KTH Royal Institute of Technology, Estocolmo, 2015.
- 16 DEANS, C. A. *Simulation & Integration of a 6-DOF Controllable Multirotor Vehicle*. Dissertação (Mestrado em Engenharia Aeroespacial) — Virginia Polytechnic Institute and State University, Blacksburg, 2020.
- 17 KOTARSKI, D.; PILJEK, P.; KRZNAR, M. Mathematical Modelling of Multirotor UAV. *International Journal of Theoretical and Applied Mechanics*, v. 1, p. 233–238, 2016. ISSN 2367-8992.
- 18 BRUNTON, S. L.; KURTZ, J. N. *Data-Driven Science and Engineering: Machine learning, dynamical systems, and control*. 1. ed. Cambridge: Cambridge University Press, 2019. 287-289 p. ISBN 9781108422093.
- 19 MATHWORKS. *Parrot Drone Support from MATLAB*. Natick, 2020. Disponível em: <https://www.mathworks.com/help/pdf_doc/supportpkg/parrotio/parrotio_ug.pdf>. Acesso em: 20 mai. 2020.
- 20 HORN, B. K. P.; SCHUNCK, B. G. Determining optical flow. *Proceedings Techniques and Applications of Image Understanding*, Novembro 1981.
- 21 LUCAS, B. D.; KANADE, T. An interative image registration technique with application to stereo vision. *Proceedings DARPA Image Understanding Workshop*, p. 121–130, Abril 1981.
- 22 ASTRÖM, K. J.; WITTENMARK, B. *Computer Controlled Systems: Theory and design*. 3. ed. Tsinghua: Prentice Hall, 1997. 306-319 p. ISBN 730200082.