ESCOLA NAVAL

talant de bien faire

Departamento de Ciencias e Tecnologia

Ricardo Maia Nunes

Controlo de um Veículo Autónomo Submarino Biomimético

Dissertação para obtenção do grau de Mestre em Ciências Militares Navais, na especialidade de Engenheiros Navais Ramo de Armas e Eletrónica

Alfeite

2020

ESCOLA NAVAL

talanteobiefaire

Ricardo Maia Nunes

Controlo de um Veículo Autónomo Submarino Biomimético

Dissertação para obtenção do grau de Mestre em Ciências Militares Navais, na especialidade de Engenheiros Navais Ramo de Armas e Eletrónica

Orientação de: Professor Duarte Bruno Damas

O aluno mestrando,

_____

ASPOF Maia Nunes

O orientador,

_____

Prof Bruno Damas

Alfeite

2020

Epígrafe

"Well done is better than well said."

Benjamin Franklin

Agradecimentos

Agradeço ao meu orientador Professor Bruno Duarte Damas pela sua disponibilidade, ajuda e acompanhamento prestado.

Aos meus pais, irmão e familiares, por todo o apoio incondicional, paciência que sempre me deram e todos os ensinamentos e valores que foram transmitidos ao longo de toda a minha vida.

À minha namorada, por toda a compreensão e amor que foi demonstrando ao longo do tempo. Um especial agradecimento por toda a força e confiança transmitidas.

Resumo

O papel dos *Unmanned Underwater Vehicles* (UUVs) tem vindo a ganhar, progressivamente, um maior destaque nos últimos anos. A sua constante evolução capacita-os da possibilidade de serem utilizados nas mais variadas missões, seja para fins militares ou civis. Apesar das suas capacidades, este tipo de veículos, por recorrerem a uma propulsão através de hélices, faz com que possuam algumas desvantagens, tais como: baixa eficiência energética, baixa manobrabilidade em espaços confinados e altos valores de ruído. Em resposta a estes problemas foram criados um novo tipo de veículos chamados *Biomimetics Underwater Vehicles* (BUVs), que têm como objetivo o uso de sistemas que repliquem o movimento de animais, colmatando as falhas acima mencionadas.

A presente dissertação inclui-se no projeto internacional SABUVIS em parceria com a Escola Naval Polaca, Universidade de Tecnologia da Cracóvia entre outros, e a Faculdade de Engenharia da Universidade do Porto e a *Oceanscan* como parceiros portugueses. Este projeto tem como objetivo a construção e desenvolvimento de um veículo biomimético capaz de executar operações de *Intelligence, Surveillance and Reconnaissance* (ISR) através de uma cauda inspirada em peixes com duas secções ou uma cauda inspirada na foca.

Assim, o trabalho desenvolvido na presente dissertação visa o desenvolvimento de um controlador para a locomoção do veículo biomimético, usando uma cauda de imitação de um peixe com uma secção e duas barbatanas laterais. Este controlador segue uma arquitetura hierárquica de dois níveis, no controlador de alto nível são implementadas diversas funções, como o veículo ser capaz de se movimentar para um dado ponto ou seguir o movimento de outro veículo. No controlador de baixo nível são calculados os parâmetros de oscilação da cauda principal e das barbatanas laterias tendo em conta os comandos recebidos dos controladores de alto nível. Toda a validação do controlador desenvolvido foi baseada na implementação dos controlos num simulador desenvolvido para um veículo semelhante ao estudado.

Palavras chave: BUV, SABUVIS, ISR, Controlo.

Abstract

In recent times, the role of Unmanned Underwater Vehicles (UUVs) has increasingly gained importance. Its constant evolution enables them to be used in the most varied missions, whether for military or civilian purposes. Despite its capabilities, this type of vehicles, due to propulsion through propellers, has some disadvantages, such as: low energy efficiency, low manoeuvrability in confined spaces and high noise values. In response to these problems, a new type of vehicle was created - Biomimetics Underwater Vehicles (BUVs) -, which aim to use systems that replicate the movement of animals, filling those flaws.

This dissertation is part of the international project SABUVIS, in partnership with the Polish Naval School, Krakow University of Technology among others, and Faculdade de Engenharia da Universidade do Porto and Oceanscan as Portuguese partners. This project aims to build and develop a biomimetic vehicle capable of performing Intelligence, Surveillance and Reconnaissance (ISR) operations through a fish-inspired tail with two sections or a seal inspired tail.

Thus, the work developed in this dissertation aims at the development of a controller for locomotion of the biomimetic vehicle, using a fish-like tail with one section and two pectoral fins. This controller follows a two-level hierarchical architecture: in the high-level controller several functions are implemented, such as the vehicle being able to move to a given point or to follow the movement of another vehicle. In the low-level controller, oscillation parameters of the main tail and side fins are calculated, considering the commands received from the high-level controllers. All the validation of the controller was based on the implementation of the controls in a simulator developed for a vehicle like the one studied.

Keywords: BUV, SABUVIS, ISR, Control.

Índice

Lista de Abreviaturas e Acrónimos

AUV	*Autonomous Underwater Vehicle*
BCF	*Body/Caudal Fin*
BUV	*Biomimetics Underwater Vehicles*
COTS	*Commercial off-the-shelf*
HIL	*Hardware-in-the-loop*
IMC	*Inter-module Comunication Protocol*
ISR	*Intelligence, Surveillance and Reconnaissance*
LAUV	*Light Autonous Underwater Vehicle*
LSTS	Laboratório de Sistemas e Tecnologia Subaquática
MIT	*Massachusetts Institute of Technology*
MPF	*Median/Paired Fin*
NECC	*Navy Expeditionary Combat Command*
ROS	*Robot Operating System*
ROV	*Remotely Operated Underwater Vehicle*
RPM	Rotações Por Minuto
SABUVIS	*Swarm of Biomimetic Underwater Vehicle for Underwater ISR*
USCG	*United States Coast Guard*
UUV	*Unmanned Underwater Vehicles*

Índice de Figuras

Índice de Tabelas

Introdução

Enquadramento Geral

Atualmente, é recorrente a utilização de veículos submarinos não tripulados *Unmanned Underwater Vehicles* (UUVs), podendo serem controlados remotamente ou autonomamente, não necessitando de recorrer a um operador a bordo, possibilitando a capacidade de desempenhar funções de grande risco, com capacidade de recolha de informação ou até mesmo, de realização de trabalhos em áreas de difícil acesso ou de perigo para o ser humano.

Este tipo de veículos, encontram-se associados a diversas operações e tipos de missões subaquáticas, tais como missões de *Intelligence, Surveillance and Reconnaissance* (ISR). No entanto, é neste tipo de missões que os UUVs convencionais, que utilizam uma propulsão através de hélices, possuem as maiores desvantagens. Uma vez que são facilmente detetados através do ruído produzido. A principal fonte de ruído num veículo subaquático deve-se à utilização de hélices, campos eletromagnéticos dos motores, fontes mecânicas, como lemes de profundidade, mudanças de direção no plano horizontal e distúrbios de fluxo. Como tal, e tendo em conta que tem existido uma evolução dos sensores que captam o som destes veículos, torna-se crucial evoluir no sentido de criar veículos mais silenciosos utilizando uma tecnologia de baixa percetividade (tecnologia furtiva) e visibilidade (idealmente invisível) (Listewnik, 2013).

Originalmente, o termo *Stealth* é aplicado quando o veículo é projetado de acordo com uma tecnologia que torna difícil a deteção por radar ou sonar, atualmente, sendo referido em casos de sensores, tanto no meio aéreo, como no meio subaquático. Os sensores acústicos têm vindo a ser desenvolvidos com tecnologia mais avançada de deteção subaquática, sendo que o seu alcance, em alguns casos, pode chegar a doze quilómetros, o que faz com que a construção de veículos com baixa deteção acústica seja a principal tarefa, e que não se aplique apenas a tecnologia militar, mas também a técnicas de pesquisa de recursos marinhos (Listewnik, 2013).

Foram, então, desenvolvidos os *Biomimetics Underwater Vehicles* (BUVs), veículos que conseguem replicar os organismos subaquáticos, como os peixes, e que utilizam a cauda e

barbatanas como meio de propulsão, conseguindo, assim, diminuir a sua assinatura acústica.

Os UUVs e os BUVs apresentam grandes diferenças em termos de locomoção. O BUV apresenta um tipo de atuação gerada por impulsos através das barbatanas, apresentando algumas desvantagens, tais como, menor intensidade de impulso quando comparado com as hélices e embora a velocidade de rotação das hélices possa ser alterada numa fração de segundo, o mesmo não acontece com o impulso proveniente de uma barbatana, uma vez que depende da amplitude e frequência gerada através de uma função complexa, que engloba parâmetros de oscilação, modos de deflexão, oscilações harmónicas e outras variáveis (Salumae, Chemori, & Kruusmaa, 2019).

O presente documento tem como objetivo estudar e aplicar mecanismos de controlo de movimento para um veículo biomimético desenvolvido no âmbito do projeto SABUVIS (Swarm of Biomimetic Underwater Vehicle for Underwater ISR). O grande objetivo do projeto SABUVIS é o de construir um veículo submarino biomimético com uma propulsão ondulatória silenciosa para operações ISR. É pretendido que o veículo seja capaz de operar autonomamente, ou semi autonomamente, e comunicar com outros veículos que se encontrem em conjunto na mesma missão (*Project SABUVIS Technical Report on Milestone No . 5*, 2017) .

Âmbito, Objetivos e Metodologia da Dissertação

A presente dissertação, proposta pela Escola Naval, apresenta-se como a continuação do trabalho iniciado em anos anteriores no projeto SABUVIS e possui como objetivos o desenvolvimento de conceitos de operação para veículos biomiméticos e a sua respetiva análise através de testes executados em simulador, fornecendo às Forças Armadas métodos de controlo de um protótipo BUV que possa ser usado operacionalmente tirando partido das vantagens que este tipo de veículos possui face aos que usam hélices.

De maneira a adquirir conhecimentos para a realização da presente dissertação, foi executada uma pesquisa e estudo de artigos científicos com o objetivo de fundamentar a dissertação, tendo como método de desenvolvimento a procura de informação, desenvolvimento e implementação do controlador no simulador, e o seu aperfeiçoamento através da análise dos resultados obtidos em ambiente simulado.

A criação do controlador para o veículo biomimético, foi efetuada em conformidade com o simulador já existente, criado para um veículo semelhante, ambos em linguagem C++.

Estrutura da Dissertação

A presente dissertação de mestrado é composta por capítulos, iniciada por um enquadramento teórico do tema em discussão, de seguida é apresentado o problema e o modelo de resolução apoiado por testes e por fim será apresentada a conclusão e o trabalho futuro.

Inicialmente são demonstradas as vantagens na utilização de BUV's em vez de UUV's e a sua importância nos dias de hoje para a realização do mais variado tipo de missões, apresentando o objetivo desta dissertação de mestrado.

A primeira abordagem passará por estudar o movimento dos peixes, de maneira a ser possível compreender o seu modo de movimento. Entendendo o seu movimento, são apresentados vários veículos biomiméticos de maneira a compreender quais as várias áreas de desenvolvimento deste tipo de veículos, tanto em aplicações civis como militares.

Entendendo as capacidades e vantagens no uso deste tipo de veículos, é então apresentado o veículo a ser controlado, sendo apresentada uma descrição do mesmo, referindo os seus componentes de hardware e a constituição de todo o software desenvolvido pelo Laboratório de Sistemas e Tecnologias Subaquáticas (LSTS) e utilizado pelo veículo, é ainda descrito e analisado o simulador usado para testar o controlador desenvolvido.

Depois de entender o movimento dos peixes e toda a constituição e características do veículo a ser controlado, é passada à fase de desenvolvimento e implementação do controlador criado. Inicialmente é explicado todo o processo de escolha da arquitetura implementada sempre apoiada pelos resultados obtidos nos testes realizados através do simulador analisando quais as melhores implementações.

Conclui-se a dissertação com uma breve síntese do trabalho realizado e as propostas de trabalho futuro.

Todo o código pertencente ao desenvolvimento dos controladores encontra-se disponibilizado no Github através do link https://github.com/ricardomn23/Simulador.

1 Estado de Arte

1.1 Estudo do Movimento

Com o passar do tempo, os corpos dos peixes têm evoluído, possuindo capacidades que os veículos subaquáticos comuns não são capazes de possuir. Algumas dessas capacidades, dependendo do tipo de peixe, passam por apresentar uma grande capacidade de manobra aliada a uma grande eficiência de energia e serem capazes de manter a sua posição independentemente das perturbações do meio envolvente, existindo algumas espécies de peixes são capazes de combinar todas estas capacidades (Colgate & Lynch, 2004).

Como primeiro passo, para o desenvolvimento de um veículo biomimético, será necessário compreender o contexto de utilização do veículo, ou seja, se será um veículo usado em zonas litorais ou em zonas de grande profundidade, se será para percorrer pequenas distancias mas num curto período de tempo ou se terá que percorrer grandes distancias, não sendo prioritário o tempo. A grande finalidade deste tipo de veículos, não passa por serem o mais semelhante possível a um peixe, mas sim tirar partido ao máximo das suas diferentes capacidades. Para tal, é necessário recorrer ao estudo fundamental da hidrodinâmica, os efeitos de diferentes tipos de design de barbatanas e as suas capacidades de manobra e estabilidade (Colgate & Lynch, 2004).

Algumas das considerações a ter em conta no âmbito do desenho são o design do robô e dos seus propulsores, tal como, o seu formato, localização, propriedades mecânicas e por fim o tipo de movimento. Com base nas considerações anteriores, é possível entender quais as duas características gerais a ter em conta. A primeira característica é baseada no seu modo de propulsão, se é feito através de um movimento ondulatório ou um movimento oscilatório. A segunda característica tem em conta as estruturas do corpo que são usadas na produção de impulso (Colgate & Lynch, 2004).

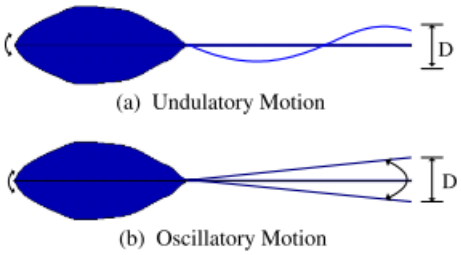

Figura 1: Diferentes modos de movimento, movimento ondulatório (a) e movimento oscilatório (b) (Kim, Kim, Jung, & Park, 2005).

Os diferentes modos de propulsão, representados na Figura 1, são diferenciados no seu tipo de movimento. Ou seja, quando um peixe movimenta o seu corpo em forma de onda, é considerado que possui um movimento ondulatório. No entanto quando o peixe gera propulsão através da oscilação de uma determinada parte do seu corpo relativamente à sua estrutura, é considerado que possui um movimento oscilatório (Masoomi, Gutschmidt, Chen, & Sellier, 2015).

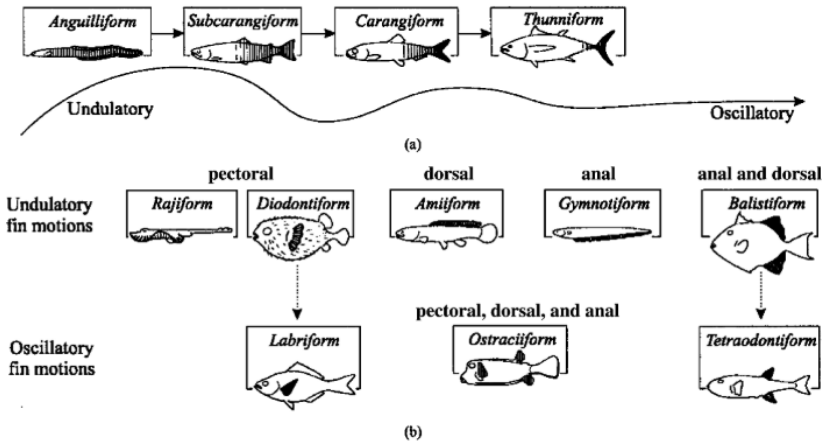

Figura 2: Modos de locomoção (a) propulsão do tipo BCF (b) tipo de propulsão MPF. Áreas a sombreado indicam a zona de produção de propulsão (Colgate & Lynch, 2004).

Como é possível observar pela Figura 2, peixes que produzem uma propulsão principalmente através do seu corpo ou cauda são generalizados como *Body/Caudal Fin Propulsion* (BCF). Como por exemplo o tipo *Anguiliform*, que usa a maioria do seu corpo para produzir propulsão, passando uma onda no sentido transversal desde a sua cabeça até à sua cauda. Necessitando de possuir um corpo bastante flexível. Este tipo de propulsão possibilita o movimento no sentido contrário, através do movimento da onda com sentido da cauda para a cabeça. (Colgate & Lynch, 2004).

A locomoção do tipo *Caranguiform*, envolve a ondulação de todo o corpo, com maior amplitude na secção da cauda. Os peixes deste tipo recorrem a uma cauda em forma bifurcada de grandes dimensões proporcionando velocidades maiores. Os peixes mais rápidos do tipo *Caranguiform* são frequentemente passados para a categoria adjacente com o nome de *Thunniform*. Este tipo de peixe, que inclui animais como, o Atum e alguns tubarões, têm formas corporais de muito baixo atrito, possuindo uma cauda em forma de "C" com a ligação ao restante corpo bastante estreita (Colgate & Lynch, 2004).

Apesar das barbatanas da cauda serem as que proporcionam uma maior eficiência a nível de propulsão, existe outro tipo de peixes, do tipo *Median/Paired Fin Propulsion* (MPF), que recorrem principalmente a barbatanas peitorais e medianas (como por exemplo, dorsal e anal). Dentro deste tipo, a classe de maior relevância apresenta-se como classe *Labriform* que recorre à oscilação criada pelas barbatanas peitorais e não ao movimento corporal. Mesmo sendo a classe *Labriform* a que apresenta menor eficiência em velocidade de cruzeiro, quando comparada com a classe *Caranguiform*, é possível concluir que a classe *Labriform* demonstra uma maior eficiência em velocidades baixas e consideravelmente maior manobrabilidade. Estas vantagens verificam-se, uma vez que as duas barbatanas peitorais podem ser controladas independentemente e serem capazes de criar propulsão no sentido contrário normal do movimento. Adicionalmente a classe *Labriform* apresenta vantagens em permanecer em posição estática com correntes, devido a não possuir oscilações provenientes da barbatana caudal, reduzindo o recuo e o deslizamento lateral (Colgate & Lynch, 2004).

Apesar do veículo estudado apresentar duas barbatanas laterais, grande parte do seu movimento horizontal é produzido pela sua cauda principal. Fazendo com que apresente características de movimento provenientes da classe *Thunniform*, tendo como vantagens a capacidade de atingir grandes velocidades cruzeiro aliadas uma boa eficiência energética, mas com as desvantagens de possuir uma manobrabilidade inferior e incapacidade de reverter o seu sentido de movimento.

1.2. BUV´s Existentes

Recentemente diversas abordagens e estudos foram realizados, existindo vários trabalhos quando o tópico se trata de veículos biomiméticos. Este capítulo tem como finalidade relatar os principais avanços e o rumo de desenvolvimento deste tipo de veículos através do estudo de trabalhos já desenvolvidos.

Cada vez mais, os sistemas robóticos sofrem adaptações de forma a poderem integrar inúmeras missões complexas. Atualmente, e dependendo do tipo de missões, alguns veículos subaquáticos mostram ser pouco eficientes no seu movimento, uma vez que apresentam uma reduzida capacidade de manobra e recorrerem a hélices para a sua propulsão.

Países que possuem um grande número de portos comerciais e instalações de movimentação tanto de carga como de passageiros, possuem a capacidade de acomodar os mais variados tipos de materiais, tais como, embarcações, balsas, navios de carga e navios oceânicos de passageiros. Apresentando um elevado risco de ameaça do mais variado tipo, podendo ser, contrabando, dispositivos explosivos improvisados através do meio aquático e outros tipos de ações, tais como, alertas de nadadores terroristas e navios que, agindo sozinhos ou em equipas, colocam explosivos ou minas nos cascos dos navios, pilares de pontes, barragens e até mesmo plataformas de petróleo (Conry et al., 2013).

Estabelecer a segurança em todas estas áreas de perigo acarreta grandes custos para o país a diferentes níveis, tais como, mão de obra, material e navios para o efeito. Sendo que o grande desafio passa pela capacidade de cobrir a maior área possível com o menor custo associado (Conry et al., 2013).

Com o evoluir das tecnologias, engenheiros e biólogos desenvolveram BUV´s inspirados nos mais diversos tipos possíveis de animais submarinos. Ou seja, não recorrendo apenas aos peixes, mas sim a animais como a manta, polvo e cobras. Estes avanços apenas são possíveis através do movimento ondulatório produzido por várias articulações ou corpos moles, demonstrando constantemente as suas vantagens.

Estes tipos de veículos, em complemento com as técnicas tradicionais de uso de materiais rígidos, fornecem soluções que não seriam possíveis quando comparados com veículos que recorrem apenas a hélices. Atualmente, apesar de todos os avanços tecnológicos, existe um vasto espaço de evolução em termos de locomoção corporal

através de corpos moles que recorram a uma atuação através de funções de oscilação e ondulação de deformação de estruturas.

No que à eficiência e manobrabilidade diz respeito, existem poucas vantagens para os veículos submarinos que recorrem a hélices para se movimentar, tendo sido provado que a sua eficiência não passa dos 70%, reduzindo o seu possível alcance. Estudos realizados a peixes, mostram que o seu modo de locomoção emite menos ruído, é mais eficiente e apresenta maior capacidade de manobra do que as tradicionais hélices. De facto, os peixes possuem uma capacidade excelente de mudar de direção, sendo capazes de o fazer em apenas 1/10 do espaço do comprimento do seu corpo - o que não acontece com os veículos submarinos tradicionais, tendo em conta que estes requerem uma maior distância para abrandar e rodar o seu corpo (Yang & Ryuh, 2013).

Uma das grandes vantagens é a sua eficiência, foi comprovado que a utilização de mecanismos de ondulação é 20% mais eficiente que a utilização de hélices. Já no ano de 1994, uma equipa de pesquisa do *Massachusetts Institute of Technology* (MIT) conduziu experiências de comparação de desempenho entre uma cauda semelhante à de um atum, e uma hélice convencional. Tendo sido concluído que a cauda apresentava uma maior eficiência, sendo capaz de produzir maior propulsão que as laminas da hélice. Apresentando uma eficiência energética de cerca de 87% em comparação com 70% recorrendo a uma hélice (Yang & Ryuh, 2013).

Em termos de ruído produzido entre os dois tipos de veículo, é possível observar quais as diferenças através do trabalho realizado em Listewnik (2013). O teste consistiu em colocar dois tipos diferentes de veículos dentro de um tanque de água isolado com borracha absorvente de som e colocar equipamentos de aquisição de som em conjunto com programas de análise. O teste possibilita concluir qual o tipo de veículo que produz maior ruído na sua locomoção (Listewnik, 2013).

O teste foi realizado utilizando 3 veículos, os veículos *Cyber-fish1* e o *Cyber-fish2* (Figura 3) e um veículo do tipo ROV (AC-ROV model 100) (Figura 4). Para a realização do teste foram selecionados dois tipos de movimento, o movimento no plano horizontal e o movimento no plano vertical. Foram escolhidos estes dois tipos de movimento devido ao facto de se encontrarem relacionados com a atuação de diferentes partes do veículo, por exemplo, no caso do veículo de propulsão a hélice quando executado o movimento

horizontal, são usados propulsores horizontais e para o movimento vertical, são usados propulsores verticais (Listewnik, 2013).

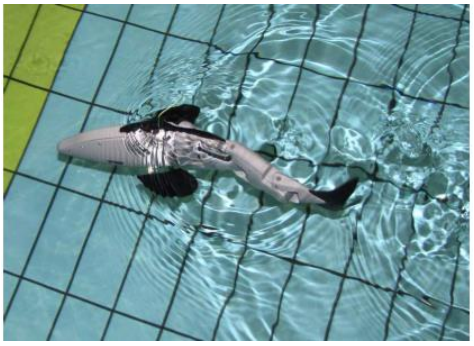

Figura 3: Cyber-fish2 durante testes na piscina (Listewnik, 2013).

Figura 4: AC-ROV model 100 (Listewnik, 2013).

A abordagem mais simples, a nível de comparação de ruído produzido por diferentes veículos, é a comparação do valor eficaz (RMS) da pressão sonora, tendo os resultados obtidos na Figura 5. Como é possível observar, o maior nível de ruído pertence ao ROV, o segundo maior pertence ao *Cyber-fish2*, que apesar de ser uma versão mais recente do *Cyber-fish1* e possuir uma velocidade maior, o primeiro possui uma propulsão ondulatória mais eficaz em termos de ruído (Listewnik, 2013).

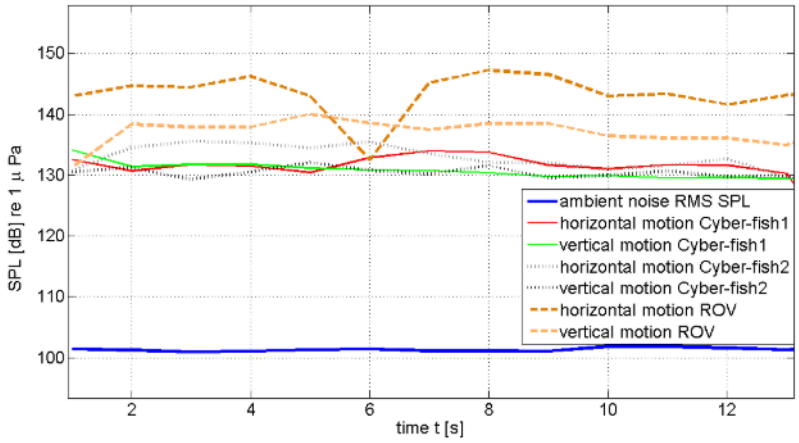

Figura 5: Resultados dos testes realizados, com indicação do ruído ambiente e ruído de cada veículo em cada tipo de movimento (Listewnik, 2013).

Com o passar dos anos e com o aumento de testes que demonstram as vantagens da utilização de veículos biomiméticos submarinos fez com que empresas, faculdades e instituições militares se dedicassem cada vez mais a desenvolver BUV's. Existindo uma

grande variedade de veículos já desenvolvidos, como podemos observar nos exemplos apresentados de seguida.

Desenvolvido pela Coreia do Sul, o 'ICHTHUS V5.6' (Figura 6) que à semelhança do veículo estudado, possui uma cauda com várias secções ligadas entre si. Este veículo tem o objetivo de parecer o máximo possível com um peixe. O seu design foi desenvolvido de maneira a possuir a menor resistência possível entre o fluxo de água e o seu corpo. Foram ainda implementados cinco atuadores capazes de criar um movimento ondulatório, tornando o seu movimento o mais semelhante possível com o dos peixes (Yang & Ryuh, 2013).

Figura 6: Peixe robo 'ICHTHUS V5.6' (Yang & Ryuh, 2013).

O corpo foi desenvolvido de maneira a possuir uma forma curva ligando os vários componentes longitudinalmente. A plataforma foi também desenvolvida com o objetivo de ser possível montar e desmontar cada peça facilmente, sendo que todos os componentes inseridos no veículo tenham sido estudados de maneira a caberem em locais pré-determinados, tal como, servo-atuadores, sensores, comunicações e controladores. O tamanho e a forma de cada um destes componentes é estudado e determinado de maneira a ser colocado no local ideal.

Como é possível observar pela Figura 7, em cada articulação os servomotores são ligados ao corpo principal do veículo, sendo através do controlo da frequência, fase e amplitude que é possível alterar a sua velocidade de propulsão.

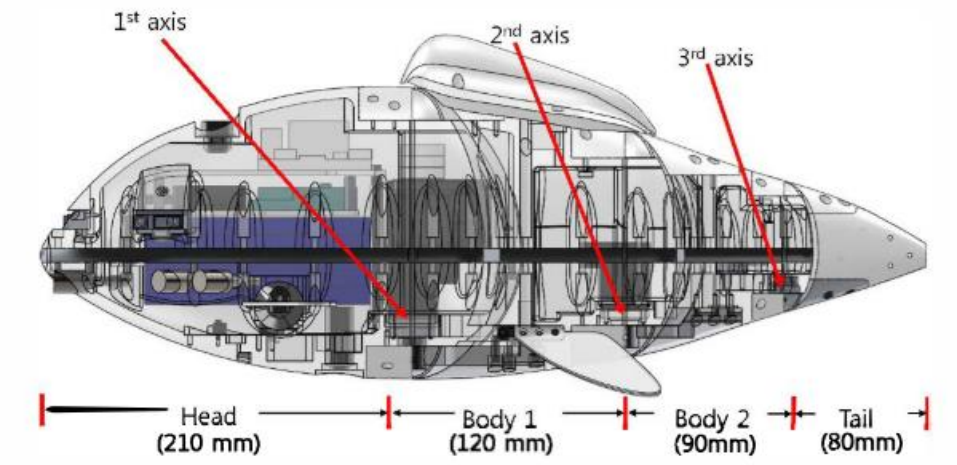

Figura 7: Design mecânico do 'Ichthus V5.6'. (Yang & Ryuh, 2013)

Este veículo possui um vasto número de sensores, tais como sensores infravermelhos, sonar, GPS, sensores de pressão de água e sistema de navegação inercial. Quando o veículo se encontra à superfície da água, utiliza as informações recebidas por GPS, quando os sensores de pressão de água enviam a informação de que este se encontra debaixo de água, o veículo passa a utilizar o sistema de navegação inercial, possibilitando o seu controlo (Yang & Ryuh, 2013).

Existem ainda modelos como o inspirado no Ituí-cavalo ou Fantasma-negro ou ainda Knifefish, que tem o seu modo de locomoção baseado num corpo rígido com uma barbatana ondulatória inferior capaz de efetuar a sua locomoção (Figura 8). Como o corpo principal não realiza qualquer movimento, não existe qualquer limitação de espaço para alojar todos os componentes elétricos e mecânicos. O veículo possui ainda um tanque onde é capaz de controlar a sua flutuabilidade chamada de bexiga nadatória. O seu controlo é feito através de uma série de elos ligados entre si, simulando o movimento do corpo do peixe (Cloitre, Arensen, Patrikalakis, Youcef-Toumi, & Valdivia Y Alvarado, 2014).

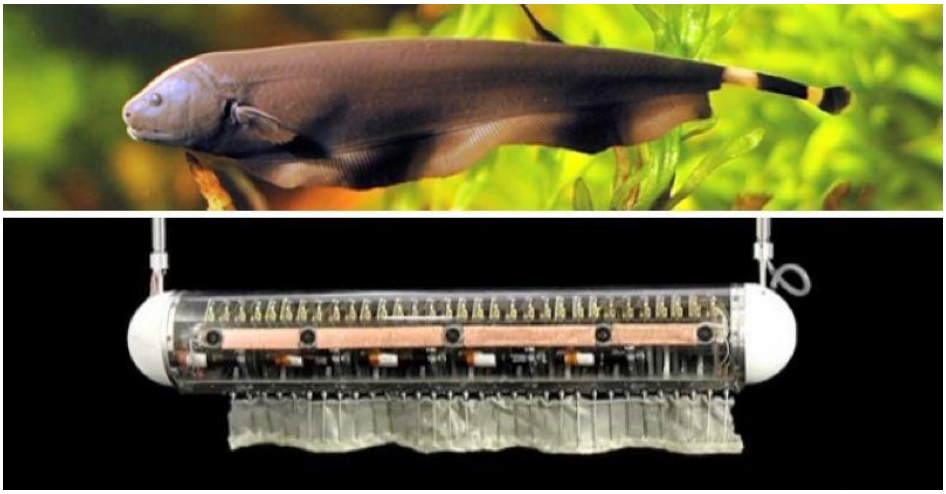

Figura 8: Exemplo do knifefish (Neveln2013)

Outro exemplo de um veículo biomimético foi o desenvolvido pelo MIT com o objetivo de imitar a locomoção de um peixe da família da raia da espécie *Myliobatiformes* (Figura 9). Este veículo tem como objetivo ser usado em missões que envolvam espaços de manobra confinados. As vantagens passam por possuir uma boa eficiência energética a baixas velocidades e uma grande agilidade. Como a parte central de uma raia não é usada para a sua locomoção, é possível colocar toda a eletrónica numa caixa localizada na parte central do seu corpo. Para o seu desenvolvimento foi adotada uma abordagem de utilização de materiais flexíveis cujas suas propriedades são capazes de reproduzir os requisitos dinâmicos da raia (Cloitre et al., 2014).

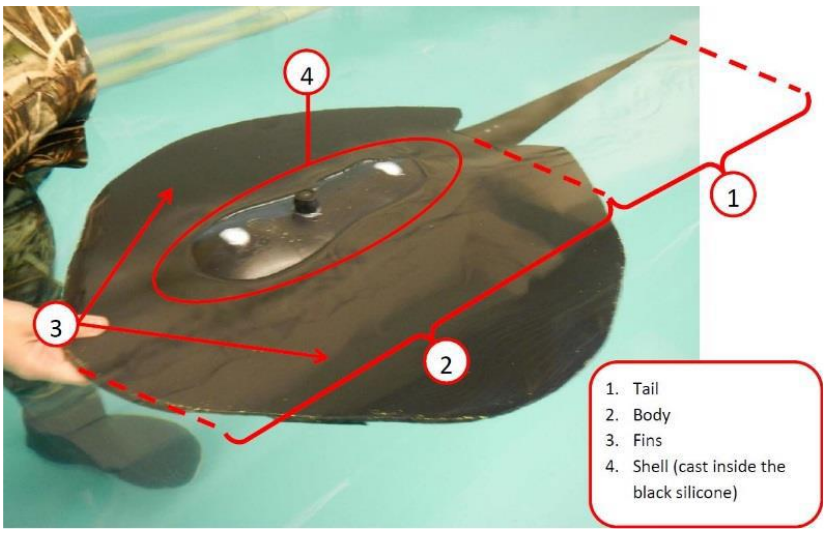

Figura 9. Veículo com a forma de uma manta e os seus correspondentes componentes (Cloitre et al., 2014)

Existe ainda outro tipo de veículos que recorrem a uma utilização de barbatanas flexíveis e são inspirados no choco e na lula. Através da imitação de músculos, foi possível reproduzir os movimentos pertencentes a estes seres e implementar nos veículos através de ligas com memoria de forma. As barbatanas flexíveis simulam os músculos transversais que fornecem o movimento, causando compressões laterais que atuando em movimentos ondulatórios originam movimento (Wang, Hang, Li, Wang, & Xiao, 2008).

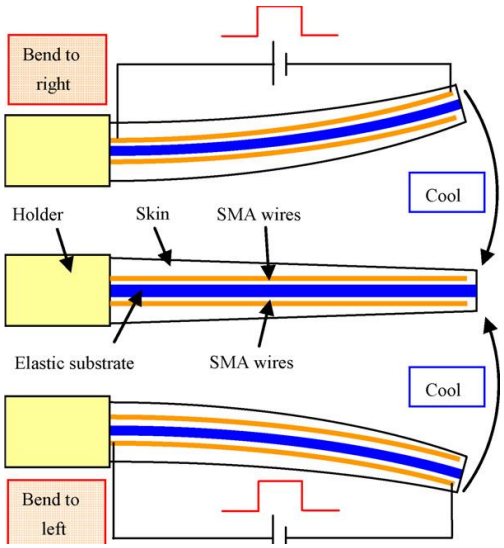

Figura 10: Estrutura e esquema do movimento da barbatana (Wang, Hang, Li, Wang, & Xiao, 2008).

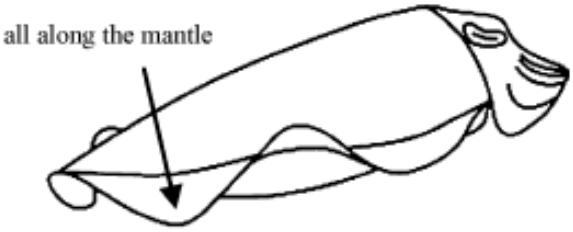

Figura 11: Representação do choco (Wang, Hang, Li, Wang, & Xiao, 2008).

Na Coreia do Sul foi desenvolvido um modelo de peixe recorrendo a corpos moles. A tecnologia é baseada num mecanismo de seis bombas e válvulas de modo a ser gerada uma pressão hidráulica ao longo do corpo mole criando movimento. O corpo é usualmente feito através de um tipo de borracha elástica, onde uma ou mais cavidades desta borracha são pressurizadas de maneira a ser possível dobrar, expandir ou dilatar causando deformação no corpo. A combinação de uma fonte causadora de pressão com um corpo mole cria um novo tipo de atuador, trazendo soluções para situações onde não seria possível utilizar partes rígidas como meio de locomoção (Katzschmann, De Maille, Dorhout, & Rus, 2016).

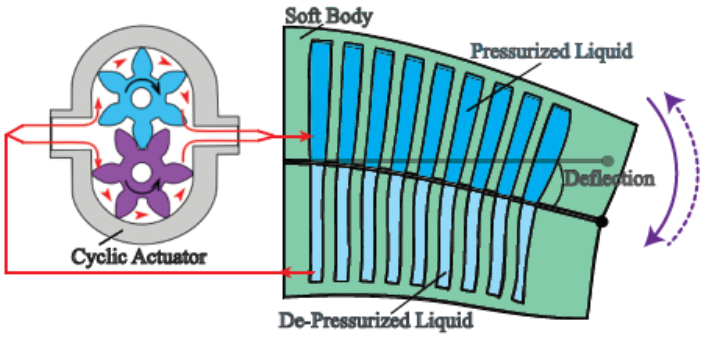

Figura 12. Atuação hidráulica de um corpo mole por uma bomba produzindo movimentos ondulatórios (Katzschmann et al., 2016).

Um dos grandes desafios deste tipo de locomoção deve-se ao facto de a longevidade e resistência do sistema não serem suficientes. Este tipo de locomoção apresenta alguns problemas e limitações, como por exemplo a propulsão, devido ao seu design, possui uma eficiência energética baixa devido às reversões constantes do motor DC da bomba. Consequentemente possui uma autonomia baixa, ruído elevado, devido às engrenagens e ainda uma atuação complexa em termos de manobrabilidade (Katzschmann et al., 2016).

Grande parte dos protótipos atuais, de maneira a ser possível transformar a estrutura e adaptar-se aos ambientes, recorrem a mecanismos rígidos ligados através de articulações fazendo com que se tornem estruturas complicadas e de grandes dimensões. Alguns dos veículos aquático-aéreo são inspirados em animais com capacidade de recolher e expelir as suas asas ou barbatanas, tal como, os Peixes Voadores ou até mesmo a ave marinha Albatroz, entre outros. Este tipo de construção permite que o veículo seja capaz de trocar rapidamente entre a locomoção por meio aquático ou por meio aéreo, no entanto a grande maioria deste tipo de veículos é construído com recurso a materiais de ligações rígidas, como dobradiças, que ligam as barbatanas e asas, tornando a sua construção complexa, acabando por ficar bastante diferente dos seus modelos biológicos (Hou et al., 2019).

Como podemos observar pela Figura 13 a lula voadora possui características distintas em comparação com outros organismos anfíbios. Possui a capacidade de sair da água através da propulsão a jato, sendo capaz de voar entre 25-33 metros e voltar a mergulhar na água, tendo a sua locomoção pode dividida em quatro etapas: (Hou et al., 2019)

- Impulso;
- Propulsão a jato;
- Voo planar;
- Mergulho;

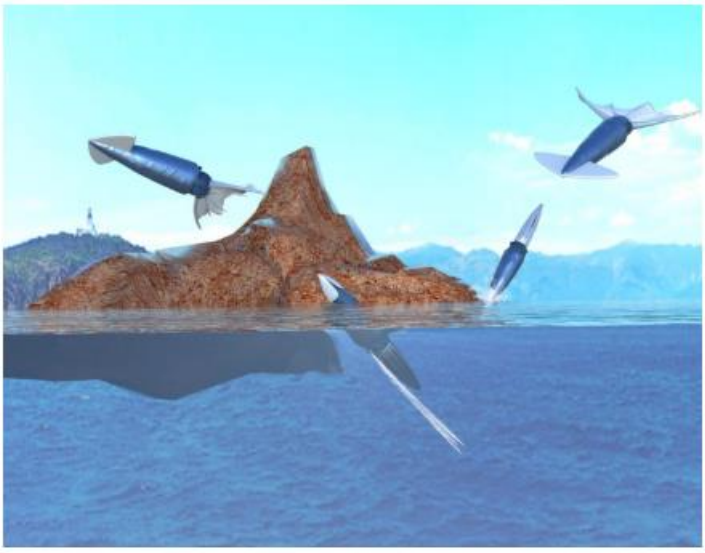

Figura 13: Demonstração das várias fazes de locomoção da lula voadora (Hou et al, 2019).

Este tipo de locomoção confere a este animal velocidades na ordem dos 11.2 m/s ou 43.9-49.7 comprimentos do corpo por segundo, sendo que o valor máximo dos peixes é de 25 comprimentos do corpo por segundo. Esta grande capacidade acontece devido às suas asas e barbatanas flexíveis aliadas ao poderoso jato de água. Na fase de impulso, as barbatanas são colocadas junto ao corpo da lula e as suas asas enroladas à volta do corpo sendo possível passar a superfície da água durante a fase de impulso. No momento em que a velocidade seja de 10 m/s, a lula irá abrir as suas asas e barbatanas de maneira a gerar força de sustentação conseguindo então planar. Este tipo de locomoção implementado em veículos poderia ser usado em ambientes de deteção de poluição, SAR entre outros.

Para ser possível recriar este tipo de locomoção foi necessário recorrer a materiais flexíveis, tendo sido usadas barbatanas e asas feitas de silicone flexível, controladas por um atuador pneumático. Ao contrário de veículos anteriormente vistos, estas barbatanas e asas possuem a capacidade de se enrolar rapidamente à volta do corpo do veículo exatamente como a Lula Voadora. Esta capacidade permite que o corpo mantenha uma grande aerodinâmica na saída da água e posteriormente abra as barbatanas e asas de maneira a produzir força de elevação ao planar, originando o deslocamento rápido entre o a água e o ar (Hou et al., 2019).

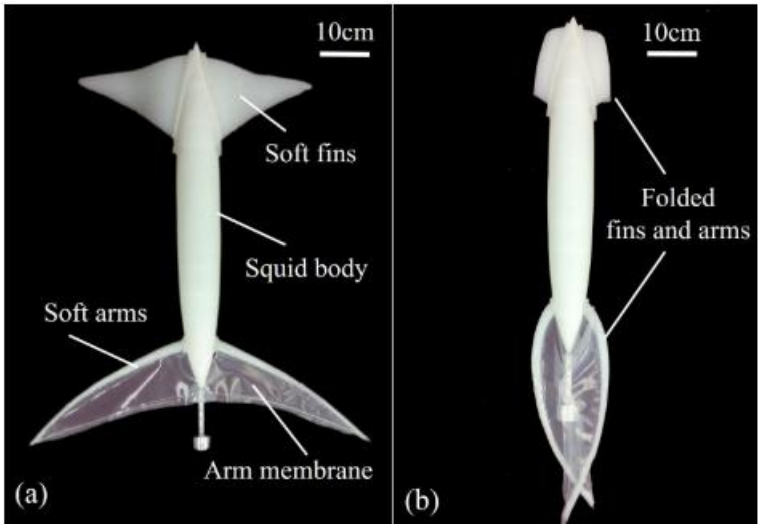

Figura 14: Protótipo da Lula Voadora e os seus membros flexíveis (a) modelo da Lula com barbatanas e asas abertas (b) modelo da Lula com os membros flechados (Hou et al., 2019).

É possível observar o protótipo desenvolvido na Figura 14. Através do controlo do sistema pneumático, é possível executar a compressão e descompressão das cavidades alojadas nos membros, permitindo ao protótipo, abrir e fechar as suas barbatanas e asas.

O sistema pneumático fornece energia tanto aos atuadores das barbatanas como à propulsão a jato. O protótipo utiliza o jato como meio de propulsão através de gás comprimido armazenado num cilindro, fazendo com que o gás crie impulsão através de um orifício. Sendo possível, tal como a lula real, controlar diferentes estados, tais como, velocidade lenta subaquática, voo a jato médio e voo planar (Hou et al., 2019).

Este tipo de design proporciona velocidades altas entre a água e o ar, sendo que como não necessita de articulações possui uma estrutura reduzida e simples na sua maneira de atuação. Os resultados obtidos nos testes realizados concluem que o design do veículo representa da melhor maneira o aspeto da lula real e o seu modo de locomoção,

comprovando a utilidade deste tipo de materiais na exploração de estratégias de locomoção a utilizar (Hou et al., 2019).

A Boston Engineering desenvolveu o veículo biomimético com o nome BIOSwimmer, para o Departamento de Segurança Interna dos EUA, com o objetivo de executar vários tipos de missões em portos, apoio a navios, entre outras. Este veículo, desenvolvido com inspiração biológica, foi projetado para se movimentar através de uma cauda articulada, tornando o raio de giração bastante pequeno (cerca de metade do comprimento do seu corpo) mesmo em velocidades de cruzeiro (Conry et al., 2013).

Este veículo tem como tarefas atribuídas em áreas de difícil acesso, inspeção de cascos de navios, tanques de lastro, tanques de água potável, inspeção do cais. Todos estes trabalhos podem ser realizados ou por mergulhadores ou por ROVs, mas a grande vantagem do BIOSwimmer é que ele é capaz de chegar a locais que outros não conseguem (Conry et al., 2013).

O BIOSwimmer é um veículo testado e altamente manobrável que usa apenas sistemas de sensores e comunicações de origem *Commercial-off-the-shelf* (COTS) é portátil e de fabrico de baixo custo, tornando-o bastante económico. Apresenta uma grande capacidade de manobra e eficiência, sendo possível executar mais que uma missão com apenas uma carga, sendo capaz de realizar trânsitos tanto em águas fechadas como abertas, desde o seu ponto de lançamento até ao seu destino (Conry et al., 2013).

Veículos que utilizam propulsores com eficiência de cerca de 45%, grande volume de casco e um perfil em forma de torpedo, originam curtos períodos de missão e problemas de controlo. O BIOSwimmer fornece maior eficiência devido ao seu baixo peso, secção traseira flexível e perfil em forma de peixe (Figura 15). Com o aumento da eficiência geral, o BIOSwimmer é capaz de executar missões por maiores períodos de tempo, atingir velocidades superiores e possuir uma capacidade de carga superior. É ainda equipado com um sonar de fiscalização sendo capaz de examinar o casco dos navios para buscas de contrabando ou engenhos explosivos (Conry et al., 2013).

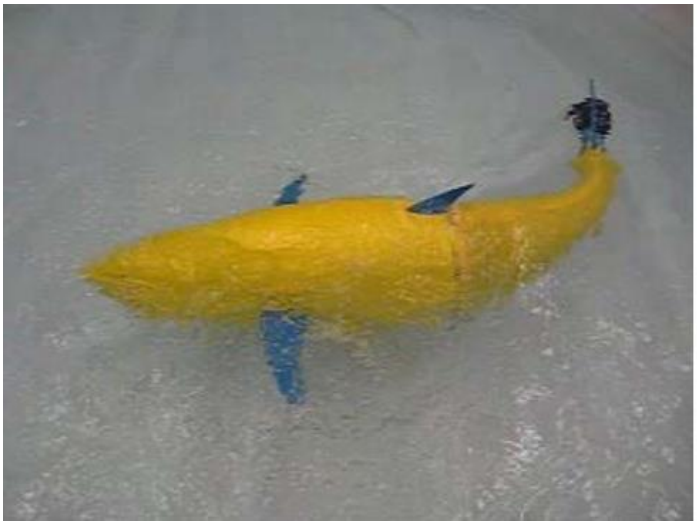

Figura 15: BIOSwimmer, a sua flexibilidade e o seu propulsor localizado no final da cauda.

A cauda do BIOSwimmer utiliza apenas um atuador, o motor tem a força necessária para aguentar a massa de água movida durante as manobras e o atrito hidrodinâmico. Sendo capaz de fornecer uma alta resposta transitória reagindo rapidamente aos controlos recebidos, girando rapidamente. O seu controlador de profundidade é feito através das barbatanas peitorais, articulando o angulo em relação ao seu corpo. O tempo de resposta da cauda desde o seu ponto neutro até ao seu ponto máximo é de 0,5 segundos, facilitando o desvio de obstáculos (Conry et al., 2013).

1.3. Aplicação Militar

Existe uma grande procura de veículos biomiméticos para ambientes militares, devido às vantagens faladas anteriormente, tais como, grande eficiência energética, baixa emissão de ruído e grande capacidade de agilidade. A Boston Engineering, mesma empresa que desenvolveu o BIOSwimmer, desenvolveu também o GhostSwimmer (Figura 16) apoiando-se, como referido anteriormente, na grande quantidade de dados experimentais existentes que comprovam a eficiência propulsora, de cerca de 87%, deste tipo de veículos, possuindo a capacidade de manobrar, tanto em altas, como, em baixas velocidades. Tudo através de um corpo flexível e hidrodinâmico impulsionado por uma cauda oscilatória e um conjunto de barbatanas peitorais (Rufo, 2010).

Figura 16: Protótipos GhostSwimmer PH I (Veículo de propulsão (esquerda) veículo de manobra (direita)). (Rufo, 2010)

O GhostSwimmer representa um grande avanço para a tecnologia dos UUVs, ao seguir o design biomimético resolve grande parte dos problemas de manobrabilidade e eficiência. O grande fator de distinção do GhostSwimmer e os outros veículos biomiméticos, como o BIOSwimmer, é a sua relevância tática. Este veículo foi construído para ser funcional, útil, capaz de transportar sensores, robusto, fácil de usar e otimizado para o desempenho de cada missão (Rufo, 2010)

Devido a todas estas características, o GhostSwimmer é direcionado para missões costeiras e outras. Aliado à sua modularidade, capacidade de carga e potencial de baixo custo, prova ser bastante útil para missões perto de terra, do tipo ISR, contramedida de minas, descarte de material explosivo, deteção de embarcações, vigilância persistente, missões secretas de patrulha, proteção de infraestruturas e inspeção de cascos (Rufo, 2010).

Com o potencial de ser capaz de operar durante longos períodos de tempo de missão, bem como alcançar altas velocidades, tanto em linha reta como durante manobras, o GhostSwimmer é ideal para todo o tipo de missões previstas e pretendidas por qualquer tipo de cliente, incluindo a United States Coast Guard (USCG), Navy Expeditionary Combat Command (NECC) entre outros (Rufo, 2010).

As grandes especificações para veículos deste tipo em aplicações militares, são:

a) Serem flexíveis, podendo ser usados nas mais variadas aplicações e missões;
b) Serem o mais ágeis possível, para a sua atividade em ambientes marítimos litorais;
c) Serem energeticamente eficientes, permitindo a atribuição de missões mais longas e missões autónomas.

De seguida, podemos observar algumas das áreas em que o GhostSwimmer se destaca:

- Missões de longa duração, missões de grande manobrabilidade (tal como, segurança de portos, longos períodos de localização e destruição de minas;
- Missões em águas pouco profundas, ambientes complexos (tais como, missões de ISR, eliminação de material explosivo);
- Missões de velocidades altas e movimentos rápidos (tais como, situações de desordem, operações de salvamento ou ambientes instáveis);
- Operações silenciosas, operações secretas (tais como, ISR, marcação de alvos, guerra antissubmarina, não deteção);
- Missões em águas profundas;

Para responder a novos tipos de ameaças, as forças navais necessitam de obter uma maior flexibilidade entre missões e o GhostSwimmer possui uma boa resposta a essa flexibilidade, sendo capaz de ser colocado em várias áreas de interesse (Rufo, 2010).

As grandes aplicações existentes, passam por ser possível a realização de operações perto de margens, devido à sua capacidade de ultrapassar problemas como manobrabilidade em espaços confinados, mudanças nas correntes e a necessidade de resposta rápida a mudanças de direção. Acrescentando a possibilidade de o veículo ser capaz de subir rios adiciona a capacidade de desempenhar missões de recolha de amostras de água ou missões de ISR em áreas altamente hostis ou onde possa existir evoluções de ameaças, como operações nucleares ou químicas por nações hostis, possuindo a capacidade de se aproximar de alvos em tempo real (Rufo, 2010).

Outra grande aplicação, é a capacidade de enviar vários veículos, deste tipo, para a frente de navios com o objetivo de investigar ou reagir, da maneira mais rápida e eficiente, a ameaças recebidas. Uma unidade do tipo GhostSwimmer pode ser usada para executar manobras bruscas aumentando a probabilidade de intervenção de ameaças letais que se aproximem do navio (Rufo, 2010).

Foram realizadas comparações entre o GhostSwimmer e o REMUS 100 (Figura 17), sendo a plataforma standard para este tipo de missões, apresentando tamanho e peso semelhantes ao GhostSwimmer. Com o desenvolver das missões e com as suas áreas de acesso a serem cada vez mais complicadas, veículos como o REMUS 100 apresentam falta

de, velocidade, capacidade de manobra e furtividade para executarem missões perto de costa (Rufo, 2010).

Figura 17: REMUS 100 Fonte: https://www.nauticexpo.com/pt/prod/kongsberg-maritime/product-31233-373860.html

Devido ao facto de cada vez mais existirem trabalhos em áreas críticas, como por exemplo entre hélices de grandes navios à procura de engenhos explosivos ou ambientes complexos com vários obstáculos, todas estas operações exigem uma grande capacidade de manobra e boas respostas de velocidade por parte dos veículos não sendo cumpridas pelos UUV's existentes, existindo a necessidade de desenvolver BUV's capazes de executar estes trabalhos tanto em aplicações de nível civil como de nível militar.

2. Descrição do Veículo a Controlar

2.1. Objetivos

O objetivo do projeto SABUVIS passa por desenvolver veículos biomiméticos subaquáticos (BUV) com propulsão ondulatória silenciosa para missões de *Intelligence, Surveillance and Reconnaissance* (ISR). O desejado é que os BUV's sejam capazes de operar autonomamente ou semi-autonomamente e consigam comunicar entre si para operações em conjunto com vários veículos em rede.

2.2. Hardware

Toda a informação apresentada nesta secção teve como referência o *Technical Report* nº6 do projeto SABUVIS que apresenta toda a constituição do BUV3. O design do BUV3 foi baseado na plataforma já existente *Light Autonous Underwater Vehicle* (LAUV) (Figura 18) e encontra-se dividido em quatro secções principais, o modulo de propulsão, casco principal, barbatanas peitorais e nariz. Cada secção é unida através de parafusos e O-rings, o espaço interior é usado para instalar componentes elétricos, baterias e sensores dos mais variados tipos (*Project SABUVIS Technical Report on Milestone No . 6*, 2017)

Figura 18: LAUV desenvolvido pela FEUP. Fonte: https://www.oceanscan-mst.com/

A secção do nariz possui vários tipos diferentes de sensores e funcionalidades:

- Sensor de navegação *Attitude and Heading Reference System* (AHRS) dando informações de altitude, rumo, rotação, inclinação e guinada;
- Sensor com o objetivo de detetar se o veículo se encontra dentro ou fora de água;
- Sensor de pressão usado na navegação;

- Ping acústico de emergência para, caso ocorra uma falha geral seja possível localizar o veículo.

As barbatanas laterais (Figura 19) usadas para o movimento vertical do veículo, encontram-se localizadas entre o nariz e o casco principal.

Figura 19: Secção do nariz e barbatanas laterais do BUV3 (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

Cada uma das barbatanas laterais é atuada através de um servomotor BLS-H50B, controlados por uma placa SCRTv4 (Figura 20) desenvolvida pelo LSTS. A placa recebe os pontos de posição através do computador principal por uma ligação RS232 gerando o sinal *Pulse-Width Modulation* correspondente.

Figura 20: Placa de controlo SCRTv4 desenvolvido pelo LSTS, controla o segundo segmento da cauda principal e das barbatanas laterias (Project SABUVIS Technical Report on Milestone No . 7, 2017).

Este veículo foi desenvolvido com o objetivo de versatilidade possuindo a capacidade de mudar facilmente a sua propulsão entre três tipos diferentes, podendo usar uma hélice (Figura 21), uma cauda de peixe com duas articulações (Figura 22) ou ainda duas caudas colocadas lado a lado semelhante a uma cauda de foca "*Seal type*" (Figura 23). O veículo possui adicionalmente duas barbatanas laterais com o objetivo de controlar o movimento no eixo vertical.

A comutação entre os diferentes tipos de propulsão apresenta-se como uma tarefa simples, sendo apenas necessário alterar a secção final que liga ao corpo principal do veículo, as ligações serão feitas por dois conectores RS232 e uma ligação às baterias.

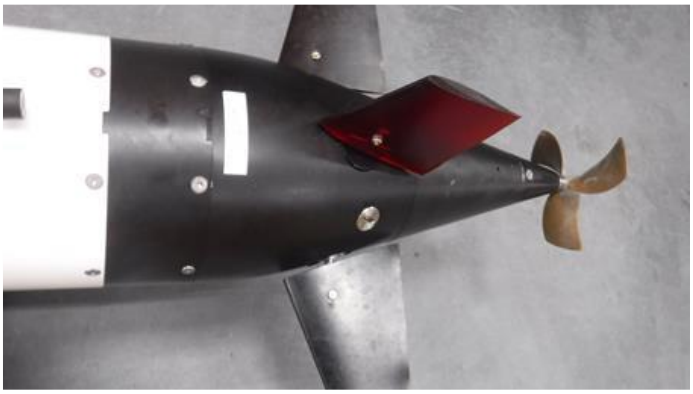

Figura 21: Secção de propulsão através de hélice (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

Figura 22: Secção de propulsão através de cauda (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

Figura 23: Secção da cauda do tipo foca (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

A secção de propulsão através de uma hélice consiste no uso de um motor Maxon EC 90W 36V ligado a uma hélice, sendo controlado por uma placa de controlo BROOM V1.1.2 desenvolvida pelo LSTS, em adição esta secção de cauda possui quatro aletas independentes acionadas por quatro servomotores Bluebird BMS-705MG para fins de controlo de direção.

Na secção de cauda do tipo peixe, existem dois segmentos, o primeiro segmento é acionado por um motor 496660 Maxon EC, acoplado a uma caixa redutora com uma razão de 285:1, o seu controlo é feito por um controlador EPOS4 Compact 50/5 CAN (Figura 24) através de uma porta RS232. O segundo segmento é atuado através do servomotor BLS-H50B controlado por uma placa SCRTv4.

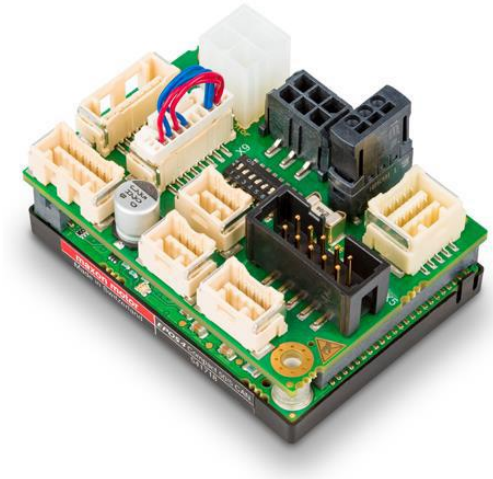

Figura 24: Controlador EPOS4 Compact 50/5 Can usado para o controlo dos motores Maxon do primeiro segmento da cauda do tipo peixe e das duas secções da cauda do tipo foca (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

A secção de cauda dupla do tipo foca é acionada por dois motores 496660 Maxon EC e tal como na cauda de peixe, ambos são acoplados a uma caixa redutora Maxon com uma razão de 285:1, sendo controlados, cada um, por um controlador CAN EPOS4 Compact 50/5.

O casco principal (Figura 25) é feito de alumínio e é no seu interior que é colocado o CPU, comunicações, sensores, baterias de Li-ion, no exterior é onde se localiza a porta de carregamento das baterias e um mastro flexível equipado com luzes de navegação, GPS, GSM e antenas Wi-Fi. É também colocado no fundo do casco um trilho com um peso de maneira a ser usado para o ajuste de equilíbrio do BUV para operações em água salgada e água doce.

Figura 25: Casco principal e o trilho com o peso para controlo de equilíbrio do BUV. Fonte: (*Project SABUVIS Technical Report on Milestone No . 7*, 2017)

Figura 26: BUV3 com as diferentes secções de cauda possíveis de utilizar (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

O controlador principal instalado no BUV3 é um AMD Geode LX (Figura 28) e tem como objetivo correr o sistema operativo Linux e o software DUNE, este processador apresenta um intervalo de temperatura de funcionamento grande sendo possível de ser aplicado em ambientes severos. Aliado ao controlador principal está um expansor de portas série (Figura 27) permitindo um maior número de sensores.

Figura 28: CPU AMD Geode LX (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

Figura 27: Expansor de portas série (*Project SABUVIS Technical Report on Milestone No . 7*, 2017).

A placa de comunicações usada é a Huawei GSM MG323 (Figura 29) e liga GSM e modulo Iridium. O GPS faz a ligação com a placa e caso as comunicações falhem com o CPU, a placa de GSM irá controlar o GPS e enviar um sinal da sua posição por qualquer meio disponível, sendo este o último recurso para a recolha do veículo caso seja perdido.

Figura 29: Placa de comunicações Huawei GSM M323. (*Project SABUVIS Technical Report on Milestone No . 7*, 2017)

A fonte de alimentação usada é uma PCTL desenvolvida pelo LSTS (Figura 30), esta fonte de alimentação é ligada diretamente às baterias e a sua função principal passa por gerir a energia, possuindo diferentes tipos de canais, que podem ser mudados em tempo

real, ativando diferentes tipos de carga, possui também sensores para a temperatura, corrente e voltagem dando uma estimativa de consumo das baterias é ainda capaz de detetar fugas de água através de um sensor que deteta a diferença de condutividade quando exposto a água.

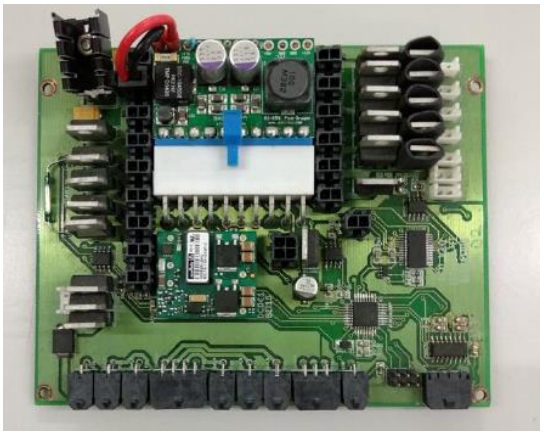

Figura 30: Fonte de energia PCTL desenvolvida pelo LSTS.

Para o sensor de comunicações acústicas, foi utilizado um micro modem WHOI (Figura 31), baseado no processador de sinais Blackfin ADSP-BF548BBCZ-5a, tendo sido utilizado anteriormente pelo LSTS. É compatível com *Phase Shift Keying* (PSK) e *Frequency Shift Keying* (FSK). É considerado, para efeitos práticos, que a velocidade de transmissão máxima de FSK é de 80 bit/s.

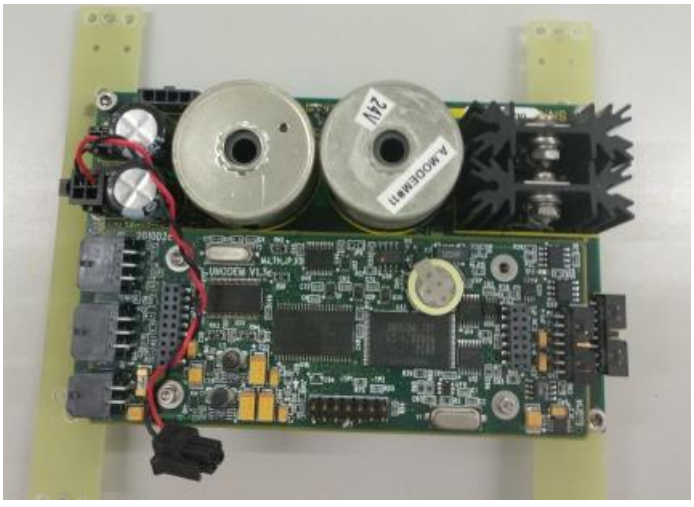

Figura 31: Micromodem de comunicações utilizado.

O GPS usado, é o modelo Ublox Evk-5 (Figura 32), ligado à placa de comunicações e à antena no mastro. O recetor possui 50 canais e tem 4 Hz de atualização com 2.5 metros de precisão. A ligação é feita através de porta-série para o CPU. O protocolo usado é o *National Marine Electronics Association* (NMEA).

Figura 32: GPS - Ublox Evk-5.

O sistema inercial usado é um ADIS16488 iSensor® (Figura 33), constituído por um giroscópio, um acelerómetro, um magnetómetro e um sensor de pressão. Devido às comunicações serem em *Serial Peripheral Interface* (SPI), como o CPU não possui esse protocolo, foi necessário desenvolver um conversor de protocolo LIMU desenvolvido pela LSTS (Figura 34) de maneira a ser possível a ligação em série entre o sensor inercial e o CPU.

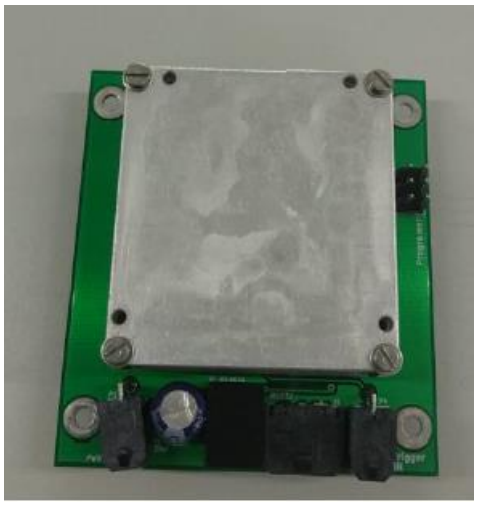

Figura 33: Sensor inercial ADIS 16488.

Figura 34: Conversor de protocolo LIMU.

As baterias usadas são células de tipo SAFT 71SF (Figura 36) com 6800mAh de capacidade, constituídas por 3 packs ligados em paralelo cada um com 7 células em serie. Cada pack é carregado individualmente. A única entrada externa é uma porta MCBH-8-FS-SS de 8 conectores (Figura 35) que tem como objetivo, carregar as baterias e estabelecer a comunicação com o CPU.

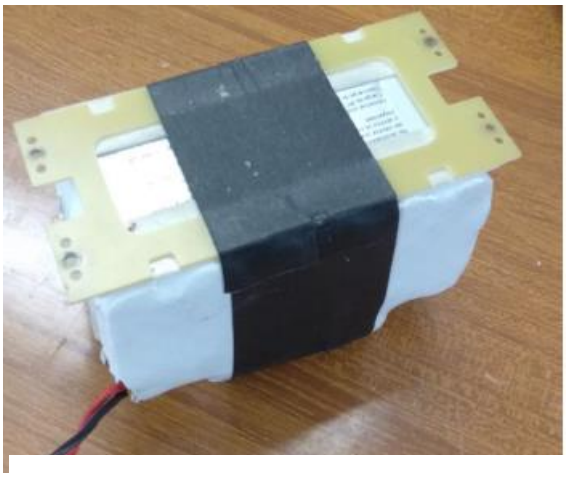

Figura 36: Células de bateria do tipo SAFT 71SP.

Figura 35: Porta de ligação externa com 8 pins

2.3. Cadeia de Ferramentas do Software LSTS

Existindo, cada vez mais, a tendência de expandir a sustentabilidade das operações marítimas, por meio de veículos autónomos, acaba por também, existir a diversificação dos tipos de veículos usados. Com a intensa evolução, vem também, o cada vez maior aumento de complexidade no controlo e gestão operacional dos veículos. De forma a diminuir essa complexidade, são criadas ferramentas que ajudam a diminuir a dificuldade de controlo, como o software LSTS Neptus-IMC-Dune (Ferreira, Pinto, Dias, & De Sousa, 2017).

Devido ao facto de o desenvolvimento do veículo ter sido em parceria com o LSTS, foi então implementada a sua cadeia de ferramentas. Esta cadeia de ferramentas foi desenvolvida com vários objetivos. Os veículos da frota LSTS são desenvolvidos para participar em operações em rede com vários veículos e usar componentes modulares de hardware e software de maneira a facilitar o desenvolvimento, manutenção e operação. A cadeia de ferramentas permite que os operadores nas estações de controlo comandem e controlem todos os tipos de veículos de uma maneira uniforme (Ferreira et al., 2017).

Esta cadeia é apresentada como um conjunto de software, de código aberto, para controlo de veículos autónomos com o objetivo de ser utilizado em rede com vários veículos, sendo controlados simultaneamente e comunicando entre si (Erstorp, 2015).

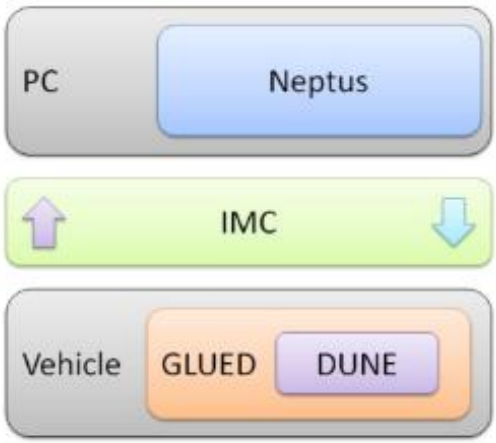

Figura 37: Cadeia de ferramentas LSTS (Erstorp, 2015).

Para ser possível controlar todos os veículos em rede, são necessários cuidados em criar o software e os protocolos, sendo capazes de ser usados em vários dispositivos para operações em rede, devendo cumprir com várias necessidades (José Pinto et al., 2012):

- Seja capaz de gerir os sensores e atuadores a bordo do veículo e o uso dos mesmos em ambientes autónomos;
- Estabelecer comunicações standard entre veículos, portas de comunicação e consolas de operação;
- Apresentar interface gráfica para interações entre o operador e o veículo, incluindo planeamentos de missão e análises;
- Ser capaz de adaptar soluções e controladores já existentes em novos veículos.

De maneira a serem cumpridas todas as necessidades, foi necessário desenvolver várias ferramentas diferentes, cada uma com o seu propósito e agindo por camadas. Tal como, DUNE para o software a bordo, Neptus como software para o comando e controlo e o *Inter-module Comunication Protocol* (IMC) como protocolo de comunicações.

Apesar do veículo possuir já implementado o software DUNE e não ser possível alterar devido à parceria com o LSTS, este software apresenta algumas vantagens face a outros já existentes, como o *Robot Operating System* (ROS).

Algumas dessas vantagens, são (José Pinto et al., 2012):

- O software Neptus fornece uma interface configurável, capaz de se adaptar a cada tipo de veículo autónomo, enquanto que o ROS apenas apresenta uma interface para todos os tipos;
- O software Neptus foi utilizado várias vezes através de testes reais por entidades com formação, académica, industrial e militar;
- O software DUNE necessita de um espaço bastante pequeno para correr (16MB). Já a compilação em ROS necessita de maiores esforços;
- Por outro lado, apesar de serem ambos, software de código aberto, o ROS possui grandes comunidades que contribuem bastante para o seu desenvolvimento o que não acontece com a cadeia do LSTS.

2.3.1. Ferramentas Software LSTS Neptus-IMC-DUNE

Neptus passa por ser um comando e controlo para operações com veículos, de sistemas e de operadores humanos em rede. Consegue suportar todas as fases do ciclo de vida da missão, tais como (Ferreira et al., 2017):

- Planeamento;
- Simulação;
- Execução;
- Análise pós missão.

O *Inter-module Comunication Protocol* (IMC) apresenta-se como um protocolo de comunicações que estabelece o controlo das mensagens por todos os tipos de nós interligados (veículos, consolas ou sensores), permitindo assim, a troca de informação de maneira standard, por entre todos os componentes (Ferreira et al., 2017).

O DUNE foi desenvolvido como um sistema dedicado aos próprios veículos, ou seja, encontra-se localizado no software do próprio veículo. Tem como grande objetivo, servir de plataforma para ser aplicado o código genérico do software dentro do veículo, como por exemplo, códigos de controlo, navegação ou acesso a sensores e atuadores. (Ferreira et al., 2017).

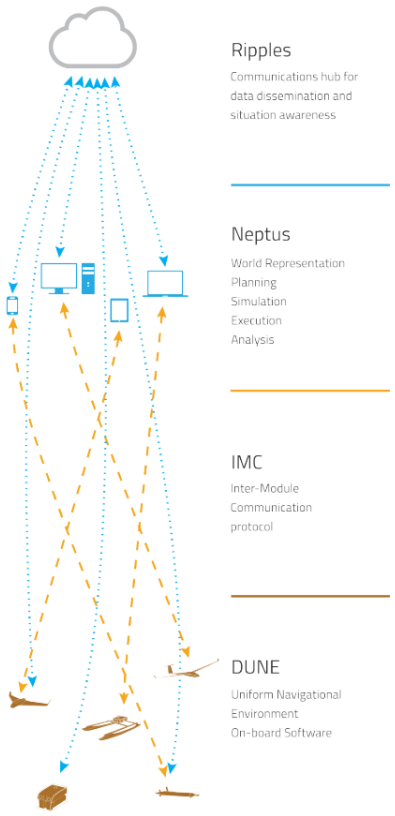

Figura 38: Estrutura da cadeia desenvolvida pelo LSTS (Ferreira et al., 2017)

O modulo *Ripples* é como um ponto central de comunicações para a propagação e perceção da situação dos dados. Funciona como um ponto de entrada de armazenamento de informação do ponto de situação, com informações de posição, especificações da missão e recolha de informação. Todos os dados são carregados em tempo real e atualizados nos vários dispositivos, no caso de se tratar de um dispositivo móvel, também será disponível a sua posição em tempo real, podendo ser utilizadas diferentes fontes com capacidade de adicionar informações meteorológicas como camadas adicionais de informação. O sistema *Ripples* é usado para reunir informações de diferentes fontes e manter uma situação global do estado do sistema (Ferreira et al., 2017).

Em termos de camadas de comando e controlo existem dois tipos diferentes de aplicações, uma direcionada para ambientes em computadores e outra desenvolvida para ambientes android (Ferreira et al., 2017).

2.3.2. Software a Bordo *DUNE*

O Software DUNE apresenta vantagens no âmbito das suas capacidades de modularidade, perfis de configuração e a facilidade de implementação em diferentes arquiteturas de controlo.

Em termos de modularidade, o DUNE funciona como um mecanismo de passagem de mensagens, em que tarefas independentes são executadas em caminhos separados, tendo o exemplo na Figura 39. As tarefas comunicam entre si usando apenas um barramento de mensagens responsáveis por encaminhar mensagens *IMC* do produtor para todos os seus recetores registados. Cada tarefa segue um ciclo de vida comum e possui manipuladores de métodos para todas as mensagens que consome (Jose Pinto et al., 2013).

Um exemplo de tarefa, como podemos observar pela Figura 40, é por exemplo, um driver de um sensor ser capaz de publicar uma mensagem com informações do sensor que esta a ser lido. Essas informações podem, posteriormente, ser lidas por uma tarefa com o objetivo de guiar o veículo no espaço tridimensional. O mesmo princípio é valido para qualquer outra tarefa que funcione, por exemplo, como um controlador de motor. Caso seja instalado um novo sensor ou testado um novo controlador, apenas será necessário ativar ou desativar algumas tarefas ou no caso de ser desejado simplesmente substituir uma tarefa, basta apenas que a nova tarefa envie o mesmo tipo de mensagem que a anterior (José Pinto et al., 2012) (Holsen, 2015).

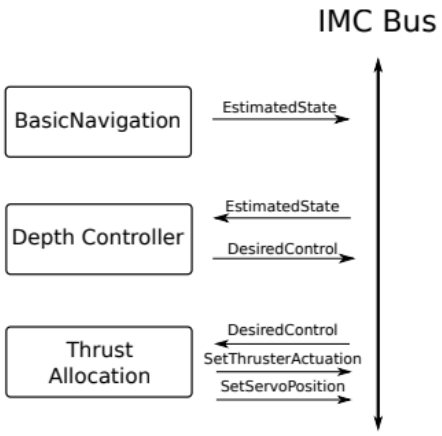

Figura 39: Passagem de mensagens através do IMC (Holsen, 2015).

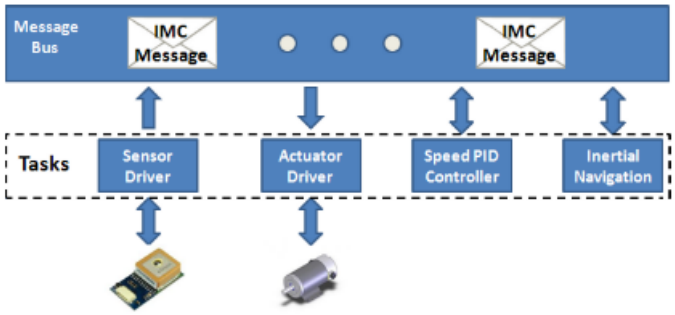

Figura 40: Conceito de transmissão de mensagens pela implementação de tarefas DUNE (José Pinto et al., 2012).

No que diz respeito aos vários perfis de configuração no DUNE, uma tarefa pode ser comum a mais do que um veículo seja subaquático ou aéreo, apenas é configurado de modo diferente. Os parâmetros que ajustam uma tarefa a funcionar de modo diferente são determinados pelo esquema de configuração. Existindo a facilidade de que essas configurações possam ser alteradas, sem ser necessário recorrer a uma recompilação de software. Permitindo ao DUNE ser executado com perfis distintos, que através do mecanismo de configuração, é possível ativar ou desativar diferentes conjuntos de tarefas dependendo do perfil selecionado (José Pinto et al., 2012).

Estas tarefas podem ser consideradas como pequenos subprogramas configuráveis que são executados em paralelo com outras tarefas, podendo ser do mais variado tipo, como (Jose Pinto et al., 2013):

- ***Sensors***, são tarefas de *drivers* do dispositivo, associadas a algum hardware que mede o meio;
- ***Actuators***, são *drivers* de dispositivos para o hardware que permitem ao veículo interagir com o meio e mover-se;
- ***Estimators***, são tarefas que agregam informações de sensores em estimativas de estado. Um bom exemplo de estimador é a tarefa de navegação;
- ***Controllers***, são tarefas que manipulam comandos de alto nível e os transformam em comandos ou ações de nível inferior, de acordo com o estado estimado atual. Por exemplo, todas as manobras programadas têm um controlador de manobra associado;

- ***Monitors***, são tarefas que recebem informações de outras tarefas e podem alterar o estado do veículo de acordo com essas informações. Por exemplo, o monitor de *Operational Limits* irá alterar o modo do veículo para "*blocked*" sempre que os limites operacionais forem violados;

- ***Supervisors***, são tarefas que ativam ou desativam outras tarefas de acordo com o estado atual do veículo. Por exemplo se o veículo entrar em modo "*blocked*", o supervisor do veículo impedirá o controlador de manobra atual de enviar comandos;

- ***Transports***, são tarefas encarregues de transportar mensagens para dentro e para fora do barramento de mensagens. O *Logging* é uma tarefa de transporte especial que escuta um conjunto de mensagens e regista o seu estado para armazenamento.

2.3.3. Configurações de Perfis DUNE

Como falado anteriormente, todas as instâncias DUNE partilham a mesma base de código sendo apenas executadas com diferentes configurações. Uma configuração é uma descrição de várias tarefas disponíveis e os seus parâmetros associados, podendo ser mudados mesmo em execução, por um utilizador ou um comando de tarefa superior (Jose Pinto et al., 2013).

Os arquivos de configuração DUNE descrevem as tarefas que serão ativadas inicialmente e os parâmetros associados. Estes arquivos podem ser partes de outros arquivos utilizando um mecanismo de referência. Permitindo a criação de arquivos de menores dimensões, suscetíveis a menos erros e específicos para um determinado veículo (Jose Pinto et al., 2013).

Para saber se uma tarefa será ativada através de um arquivo de configuração, o operador deve selecionar em quais perfis a tarefa será ativada por padrão. O DUNE usa perfis para permitir que várias configurações típicas, sejam definidas num único arquivo. Alguns exemplos desses perfis são (Jose Pinto et al., 2013):

- *Hardware*, esta tarefa estará disponível apenas quando o *DUNE* se encontra ligado aos sensores e atuadores reais do dispositivo;
- *Simulation*, estas tarefas são ativadas apenas quando o DUNE se encontra em execução sem ligação com os sensores e atuadores reais do hardware. As versões

simuladas dos sensores e atuadores irão corresponder a tarefas de simulação executadas apenas neste perfil produzindo dados simulados;

- *HIL*, este perfil é usado no hardware real, mas parte dos atuadores e sensores apresentarão entradas e saídas simuladas. Por exemplo, a tarefa *Thruster,* neste modo, será executada numa fração dos comandos de rotações por minuto (RPM's) dados de maneira a poder ser usado em segurança quando fora de água.

2.3.4. Mecanismos de Segurança DUNE

O DUNE apresenta um conjunto de tarefas que verificam constantemente partes vitais do sistema. Caso alguma dessas tarefas entre num estado de erro, um supervisor irá fechar ou abrir outras tarefas. Seguem alguns exemplos das defesas para veículos subaquáticos e aéreos (Jose Pinto et al., 2013):

- ***Fuel Level***: caso esta tarefa detete que o nível de combustível se apresente baixo, o supervisor do veículo irá interromper a execução do controlador ativado substituindo-o por um controlador de manobra de segurança;
- ***Leaks***: caso seja detetada uma fuga a bordo de um A*utonomous Underwater Vehicle* (AUV) o supervisor do veículo encerrará toda a carga de transmissão de dados e passará a executar uma manobra de segurança de modo a trazer o veículo para a superfície.
- ***Operational Limits***: caso os limites definidos ao utilizador sejam violados pelo controlador, o mesmo será impedido de controlar o veículo e apenas o supervisor do veículo irá ter acesso a manobras de segurança e operação.
- ***Communication***: caso o veículo perca a comunicação com a estação base, durante um intervalo de tempo definido pelo utilizador, o veículo irá interromper o plano atual e executará um plano predefinido que o trará de volta para perto da base.

2.4. Modelo Matemático Usado no Simulador

Como referido anteriormente, devido à situação de pandemia mundial, não foi possível trabalhar com o veículo real tendo sido necessário recorrer a um ambiente simulado de maneira a ser possível desenvolver um controlador para o veículo e executar

os seus testes de comportamento. O simulador usado e já existente, foi criado em MATLAB pela Escola Naval Polaca para a utilização de outro veículo com uma forma semelhante ao do veículo estudado, mas com a particularidade de utilizar uma cauda com uma secção e não duas como no veículo estudado. O facto de apenas ser usada uma secção leva a que a sua atuação seja, relativamente, mais simples, não influenciando em demasia o comportamento do veículo caso fossem utilizadas duas secções de cauda, uma vez que a segunda secção apenas iria acompanhar o movimento ondulatório da secção anterior, tornando o movimento mais suave e proporcionando maior força propulsora.

Apesar do simulador apresentar diferentes parâmetros cinemáticos e hidrodinâmicos, como a forma do veículo é semelhante à do veículo a controlar, a utilização deste simulador permite desenvolver os controladores e testá-lo obtendo conclusões válidas, não sendo dispensável executar alguns ajustes nos diferentes parâmetros na altura da sua implementação real.

O veículo estudado é representado como um corpo rígido com várias características, tais como, possuir três planos de simetria, movimentar-se em seis graus de liberdade, como representado na Figura 41 e mover-se a velocidades lentas em meios viscosos.

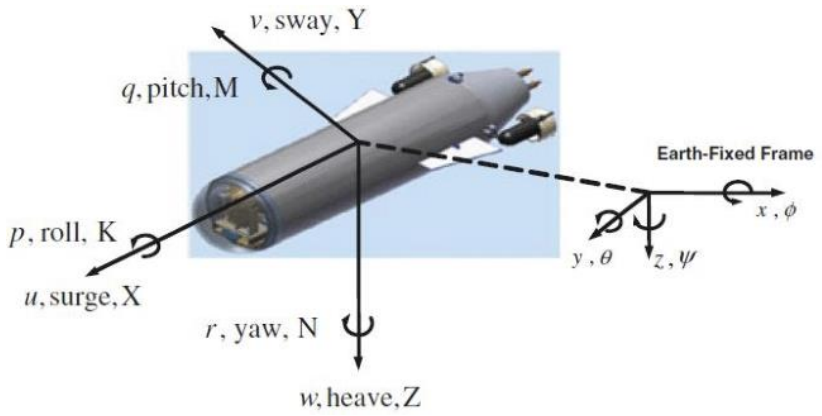

Figura 41: Representação dos seis graus de liberdade de um veículo submarino (Al-Mahturi, Santoso, Garratt, & Anavatti, 2019).

O movimento do veículo é descrito através de dois sistemas de referência, representados na Figura 41, através das coordenadas móveis associadas ao sistema do veículo $X_0Y_0Z_0$ e o sistema de coordenadas imoveis associado ao xyz da terra (Szymak, 2017).

A origem do sistema de coordenadas moveis O corresponde ao centro de gravidade do veículo, enquanto que os seus eixos são definidos como:

- X_0 é o eixo longitudinal direcionado da popa para a proa;
- Y_0 é o eixo transversal direcionado para estibordo;
- Z_0 é o eixo transversal perpendicular direcionado de cima para baixo.

Alterações da posição do sistema de coordenadas moveis $X_0Y_0Z_0$ são relativas ao sistema de coordenadas xyz associado à terra. Devido à velocidade alcançada ser considerada baixa, a aceleração de pontos na superfície da terra é ignorada e o sistema de coordenadas xyz é considerado estacionário, não sendo necessário ter em conta as forças centrifugas e centrípetas e os momentos de força causados pela rotação da terra (Szymak, 2017).

Tendo em conta todas as características e de maneira a ser simulado o movimento do BUV, foi utilizado um modelo linear do modelo subaquático em 6 graus de liberdade referido em Fossen (2011) representado na forma de matriz compacta:

$$M\dot{v} + D(v)v + g(\eta) = \tau \quad (1)$$

Onde:

v – Representa o vetor de velocidades lineares e angulares no sistema móvel;

η – Vetor das coordenadas de posição do veículo e os seus ângulos de Euler no sistema imóvel;

M – Matriz de inércia (soma das matrizes do corpo rígido e das massas acompanhantes);

$D(v)$ – Matriz hidrodinâmica de amortecimento;

$g(\eta)$ – Vetor de forças restauradoras e momentos de força (gravidade e flutuabilidade);

τ – Vetor de sinais de controlo (a soma do vetor de forças e momentos de força gerados pelo sistema de propulsão τ_d e pelas perturbações do meio τ_d).

O objetivo principal será o de calcular o vetor τ_p gerado pelo tipo de sistema de propulsão usado pelo veículo.

$$\tau_p = [X, Y, Z, K, M, N] \quad (2)$$

Onde,

- X, Y, Z – representam as forças que atuam, respetivamente, nos eixos de simetria longitudinal, transversal e vertical;
- K, M, N – representam os momentos de força que atuam, respetivamente, nos eixos de simetria longitudinal, transversal e vertical.

Para o cálculo de τ_p é necessário ter em consideração a configuração de propulsão do veículo. Como exemplificado na Figura 42, é possível observar o projeto 3D do BUV que utiliza uma propulsão ondulatória especifica através de uma barbatana móvel de duas secções na extremidade final e duas barbatanas laterais independentes (Szymak, 2017).

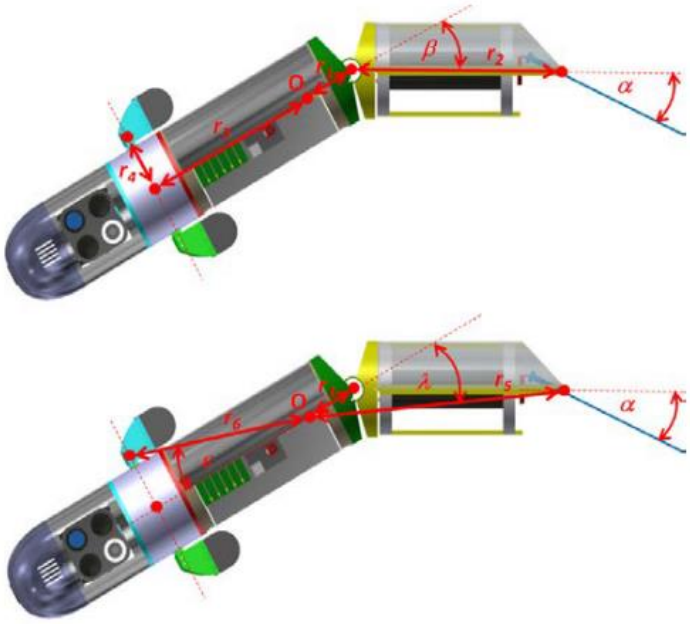

Figura 42: Desenho BUV2 visto de cima. (Szymak, 2017)

Cada uma das barbatanas laterais e a cauda são representadas a nível de atuação pela seguinte equação:

$$\theta(\mathrm{t}) = A + B\cos(wt) \tag{3}$$

Onde, A representa a deflexão, B representa a amplitude e w representa a frequência para a cauda referente.

O impulso gerado pelas diferentes barbatanas é referenciado à origem da gravidade O através de dependências simples na transformação do vetor, como:

$$X = X_t + X_l + X_r - X_d \tag{4}$$

$$Y = Y_t \tag{5}$$

$$Z = Z_l + Z_r \tag{6}$$

$$K = 0 \tag{7}$$

$$M = M_l + M_r \tag{8}$$

$$N = N_t + N_l - N_r - N_d \tag{9}$$

Os índices inferiores l, t e r referem-se, respetivamente, à operação das aletas da cauda (t), do lado esquerdo (l) e lado direito (r), o índice (d) é referente ao amortecimento causado pela aleta do lado esquerdo ou direito que trabalha como popa, sendo que é ajustada perpendicularmente ao eixo longitudinal de simetria.

Os componentes do vetor (como por exemplo X_t, Y_t e N_t) são calculados levando em consideração a localização das barbatanas em relação à origem da gravidade, usando as seguintes formulas:

$$X_t = cos\beta . T_t \tag{10}$$

$$Y_t = sin\beta . T_t \tag{11}$$

$$N_t = -cos\lambda \, . r_5 \, . Y_t \tag{12}$$

$$X_l = cos\delta_l . T_l \tag{13}$$

$$Z_l = sin\delta_l . T_l \tag{14}$$

$$N_l = r_4 . X_l \tag{15}$$

$$M_l = r_3 . Z_l \tag{16}$$

$$N_d = \cos(90 - \varepsilon) . r_6 . X_d \tag{17}$$

Onde,

T_t, T_l, T_r – representam os impulsos gerados respetivamente pelas barbatanas da cauda, esquerda e direita.

β, λ, δ_l, ε – representam os ângulos presentes na Figura 42.

r_1- Distância entre o centro de gravidade e o centro de rotação do modulo de cauda.

r_2- Distância entre o centro de rotação do modulo da cauda e o centro de rotação da barbatana caudal.

r_3- Distância entre o centro de gravidade e o centro de rotação das barbatanas laterias.

r_4- Distância entre o centro de rotação das barbatanas laterais e o centro de rotação para o lado esquerdo proporcionado pela aleta direita, de modo geral, representa a distância entre o centro de rotação de cada aleta lateral e o eixo longitudinal de simetria.

r_5- Distância entre a origem da gravidade e o centro de rotação da barbatana caudal.

r_6- Distância entre a origem da gravidade e o centro de rotação da barbatana esquerda ou direita.

2.5. Funcionalidades Presentes no Simulador

2.5.1. Controlos Básicos do Veículo

Para o desenvolvimento de um controlador de um determinado veículo, torna-se necessário perceber qual o comportamento básico de algumas das atuações possíveis de aplicar e compreender a melhor forma de as usar.

O veículo dispõe de três barbatanas disponíveis de atuar, uma cauda principal, com apenas uma secção e duas barbatanas laterais. Cada barbatana tem disponível três tipos diferentes de atuação seguindo a logica da seguinte equação:

$$\theta(t) = A + B\,cos(wt)$$

Onde se encontram os valores de frequência (w), amplitude (B) e deflexão (A), sendo que cada uma destas atuações cria um tipo diferente de resposta por parte do veículo.

Em termos de frequência, esta atuação corresponde ao movimento oscilatório da cauda e das barbatanas do veículo estando diretamente relacionado com a velocidade do mesmo. Os valores de frequência são dados em Hertz.

No que diz respeito à deflexão, representada na Figura 43, quando aplicada à cauda principal, tem como objetivo mudar o rumo do veículo no plano horizontal e quando aplicado às barbatanas laterais o de variar a sua profundidade.

Atuação em amplitude, representada na Figura 43, apresenta-se como o valor em graus a que as barbatanas irão variar o seu movimento em relação ao seu angulo de deflexão. Podemos observar um exemplo de algumas destas atuações na cauda principal, tal como a deflexão e a amplitude na Figura 43, sendo a atuação semelhante nas barbatanas laterais.

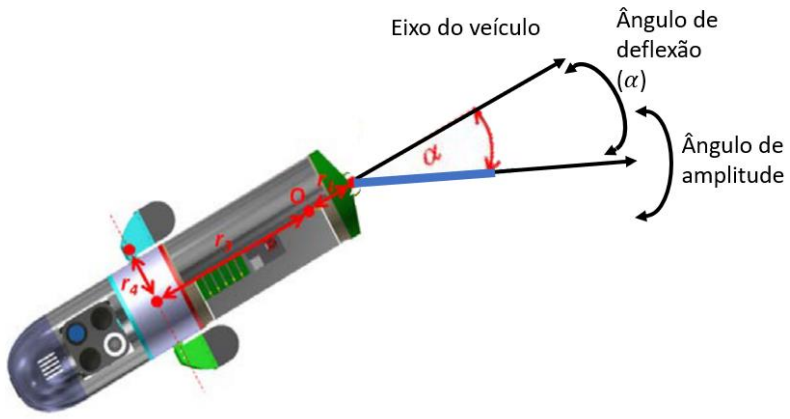

Figura 43: Imagem do veículo visto de cima, representando o angulo de deflexão e de amplitude da cauda principal (Szymak2017).

Como podemos observar pela Figura 44, a atuação da amplitude corresponde ao angulo entre o valor estipulado de deflexão da cauda ou das barbatanas laterais, ou seja, estando um valor de 0° de deflexão da cauda principal e um valor de 30° de amplitude, a cauda irá fazer ângulos de 30° graus para cada um dos lados, criando movimento em linha reta.

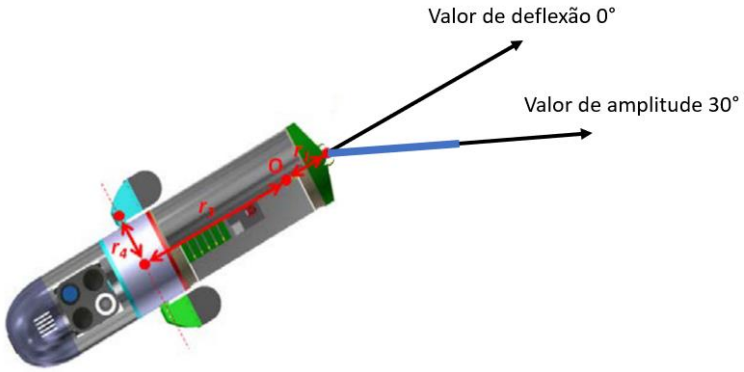

Figura 44: Demonstração da atuação de amplitude no veículo (Szymak, 2017).

2.5.2. Flutuabilidade

O simulador possui alguns parâmetros, tais como a flutuabilidade diferente de zero, esta característica poderá ser usada dependendo do peso e comportamento do veículo quando implementados diferentes sensores, equipamentos e dependendo da salinidade da água. É possível observar qual o comportamento do veículo quando a sua flutuabilidade é diferente de zero e todas as atuações iguais a zero na Figura 45 e a correspondente velocidade Figura 46, na Figura 47 é possível observar qual o comportamento do veículo quando a sua flutuabilidade se apresenta diferente de zero em conjunto com movimento horizontal.

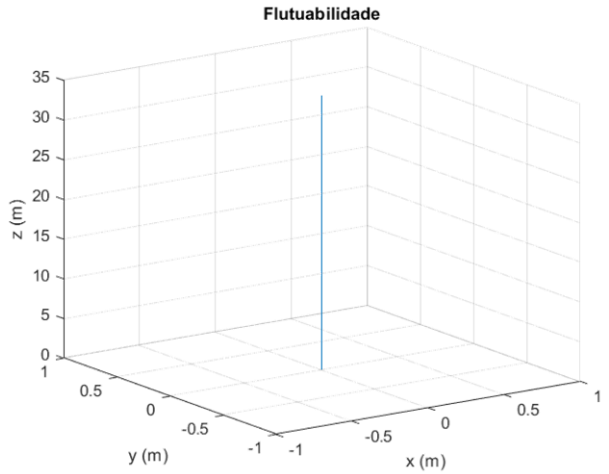

Figura 45: Exemplo do comportamento do veículo quando a flutuabilidade é diferente de zero e não possui atuação de nenhuma das barbatanas.

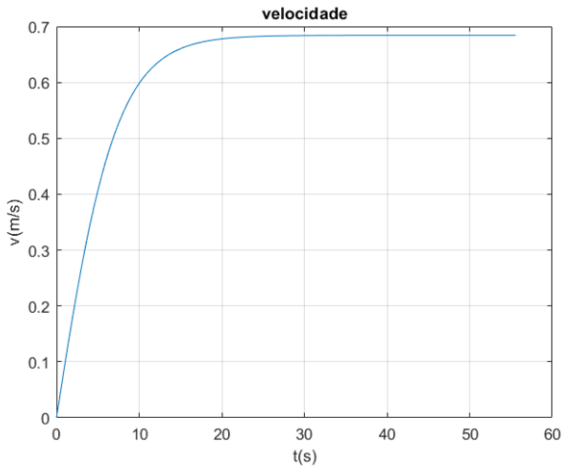

Figura 46: Velocidade vertical do veículo quando implementada a função de flutuabilidade.

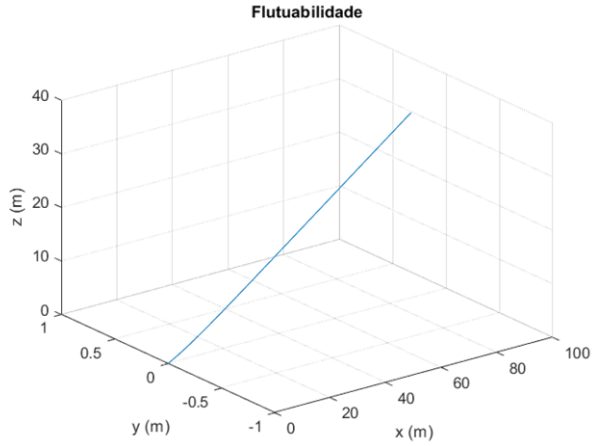

Figura 47: Exemplo do comportamento do veículo quando a flutuabilidade é diferente de zero, mas com atuação da cauda principal num movimento em linha reta.

2.5.3. Correntes

Outra das funcionalidades disponíveis no simulador, é a possibilidade de adicionar uma corrente do meio, ou seja, é capaz de simular forças exteriores capazes de influenciar o movimento do veículo. Podemos observar alguns dos exemplos da sua atuação com uma corrente de valor (0.0, 2.0, 2.0), tendo sido dada uma atuação apenas à barbatana principal de maneira ao veículo se deslocar ao longo do eixo Ox. Através dos resultados obtidos (Figura 48 e Figura 49) é possível verificar o valor de corrente imposto influencia o movimento do veículo ao longo do eixo Oy e do eixo Oz.

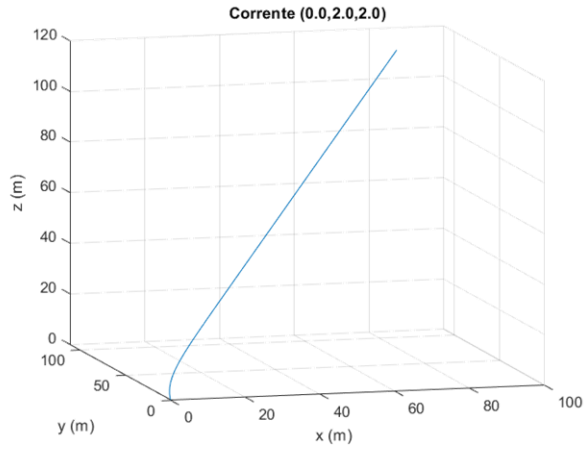

Figura 48: Gráfico do trajeto com a influência da corrente, com valor de (0.0, 2.0, 2.0) e o movimento do veículo no ao longo do eixo Ox.

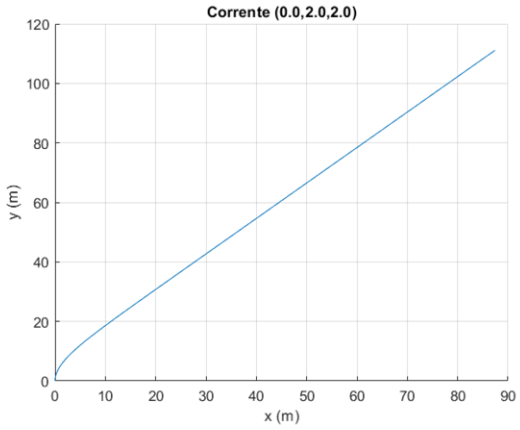

Figura 49: Gráfico do trajeto resultante entre a influência da corrente, de valor (0.0, 2.0, 2.0) e o movimento ao lindo do eixo Ox.

2.6. Movimentos Básicos do Veículo

Em termos de dinâmica de curva do veículo, foram feitos alguns testes, de maneira a entender quais as dinâmicas respetivas a cada tipo de atuação. Na Figura 50 temos uma atuação na cauda principal correspondente a uma frequência de 3Hz, deflexão de 30° e amplitude de 20° e na Figura 51 foram aplicadas as mesmas atuações, mas com o valor de deflexão de -30°.

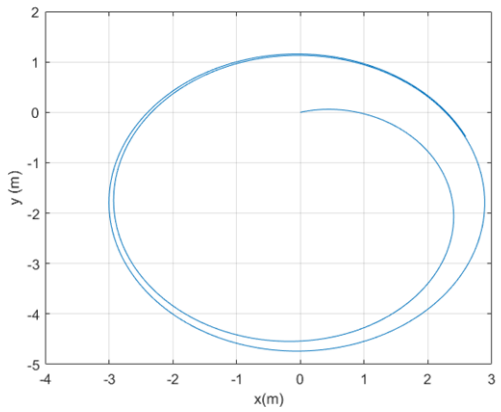

Figura 50: Representação de curva usando apenas a cauda principal, frequência de 3Hz, deflexão de 30° e amplitude de 20°.

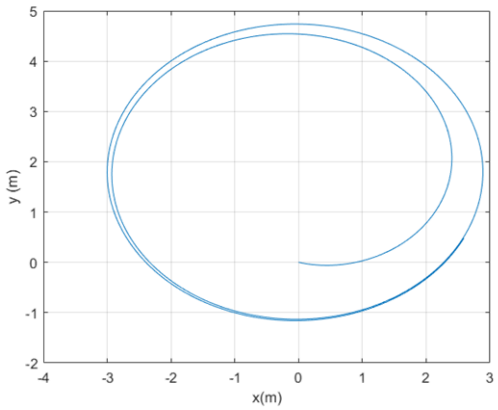

Figura 51: Representação de curva usando apenas a cauda principal, frequência de 3Hz, deflexão de -30° e amplitude de 20°.

Através da análise das figuras acima, podemos concluir que ângulos de deflexão positiva fazem com que o veículo curve para estibordo e ângulos de deflexão negativa fazem com que o veículo guine para bombordo.

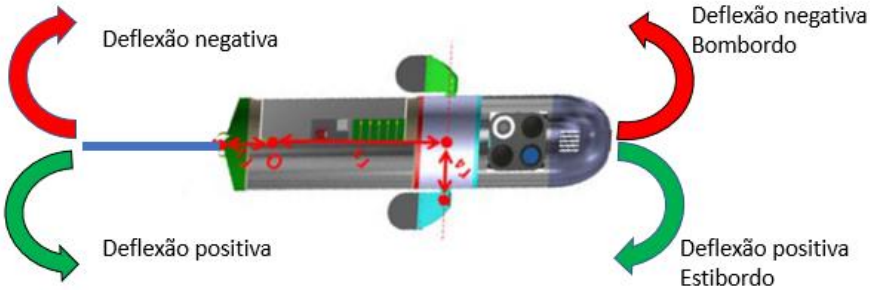

Figura 52: Representação da direção de curva em relação ao angulo de deflexão.

Seguidamente foram efetuados testes com o objetivo de estudar quais as atuações que influenciam diretamente o raio de curvatura do veículo. Na seguinte figura, foram aplicadas as mesmas atuações usadas anteriormente, mas com uma redução para metade num dos valores de atuação de frequência, amplitude ou deflexão. Representado na Figura 53 temos a atuação com um valor de frequência de 1.5Hz, a atuação com um valor de amplitude de 10° e a atuação com uma deflexão de 15°.

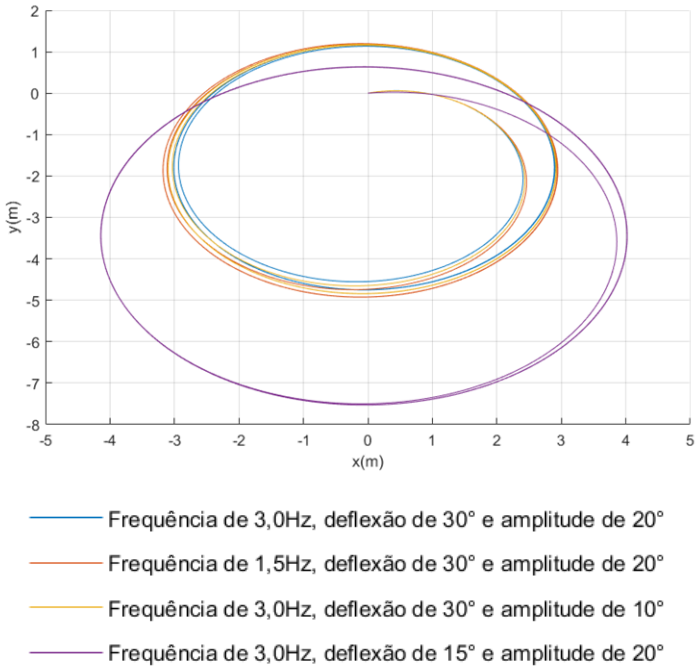

Figura 53: Testes com as várias mudanças de atuação com o objetivo de entender qual o parâmetro mais determinante para a curvatura da trajetória do veículo.

Como podemos observar pela Figura 53, a trajetória com uma atuação diferente em frequência, para metade e a trajetória com uma atuação diferente em amplitude, para metade apresentam, aproximadamente, o mesmo raio de curvatura de cerca de 3,0 metros, sendo apenas a trajetória com uma atuação diferente em deflexão, para metade a que demonstra diferenças no raio de giração, cerca de 4,0 metros. Com os dados recolhidos é possível concluir que o parâmetro de deflexão é o mais determinante para a curvatura da trajetória efetuada pelo veículo.

2.7. Testes de Curva com Diferentes Velocidades

Outros dos testes executados ao comportamento do simulador, foi averiguar se a velocidade do veículo poderá mudar o comportamento em curva, ou seja, se o raio de curvatura do veículo será o mesmo com uma velocidade inicial baixa, ou se irá mudar o seu raio de curva quando possui velocidades mais altas. De modo a realizar o teste, primeiramente foi analisado o gráfico de curva quando aplicadas as mesmas atuações ao veículo, mas com uma velocidade inicial de 0 m/s e no segundo teste feito com uma velocidade inicial de 0.7 m/s.

Através da análise das figuras seguintes, o resultado do raio de curva com velocidade inicial zero apresenta um raio de curva de 3 metros e o gráfico de curva com velocidade diferente de zero, apresenta uma elipse com o raio a variar entre os 3 e os 4 metros, concluindo que, apesar de ser pequena, a velocidade do veículo influencia o seu comportamento em curva.

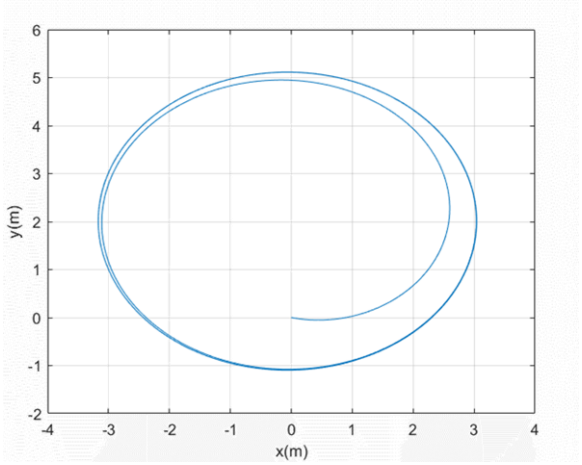

Figura 54: Raio de curva do veículo quando aplicadas as atuações de frequência 3Hz, deflexão 30° e amplitude 20° e com velocidade inicial de 0 m/s.

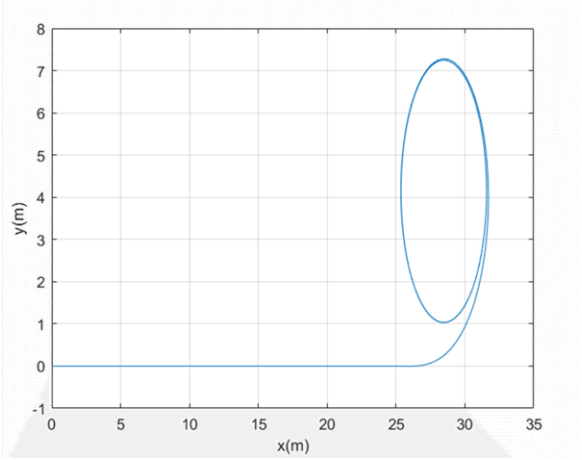

Figura 55: Raio de curva do veículo quando aplicadas as atuações de frequência 3Hz, deflexão 30° e amplitude 20° aos 25 metros com uma velocidade de 0,7 m/s.

2.8. Influencia das Barbatanas Laterais na Velocidade

Encontra-se presente nas funcionalidades do simulador, as interferências na velocidade causadas pelo atrito das barbatanas laterais, como tal, foram realizados testes com o objetivo de estudar qual a influencia do atrito causado pelas barbatanas laterais na velocidade do veículo, quando ambas atuadas em 90° de deflexão.

Foi realizado um teste, onde foram dadas atuações para a cauda principal, fazendo o veículo seguir numa linha reta, frequência 3Hz, deflexão 30° e amplitude 20°, quando este estabilize a sua velocidade são então aplicadas zero atuações em todos as barbatanas.

Foi decidido realizar um teste que consiste em dar um dado ponto no eixo Ox e quando a distância ao ponto, se apresenta como metade, são acionadas as barbatanas laterais com deflexão igual a 90°, ou seja, na posição perpendicular ao movimento do veículo, sendo possível verificar se a sua velocidade é alterada. Para tal, foi dado um determinado ponto no eixo Ox de maneira a ser possível ao veículo estabilizar a sua velocidade antes de serem acionadas as barbatanas laterais (x=25 metros) e depois de serem acionadas (x=50 metros).

Na Figura 56 podemos observar o resultado obtido, em termos de velocidade. Primeiramente é possível observar qual a velocidade a que o veículo se desloca sem a influencia do atrito causado pelas barbatanas laterais e após os 25 metros o decréscimo de velocidade associado ao seu atrito tendo um valor aproximadamente de 0.05 m/s.

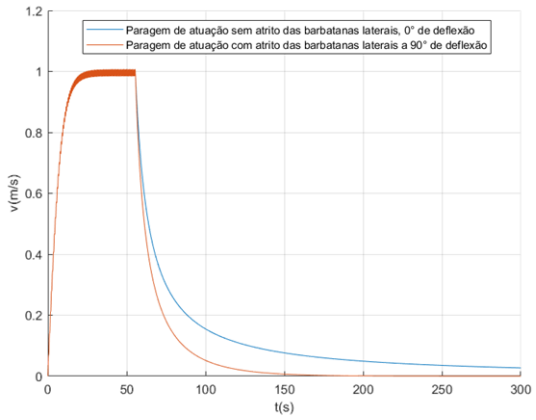

Figura 56: Teste do decréscimo de velocidade causado pelo atrito das barbatanas laterais quando atuadas em 90° de deflexão.

3 Arquitetura de Controlo

A Hierarquia de controlo de dois níveis segue a logica de separação entre os controladores de alto nível, encarregues de calcular quais os erros de angulo e velocidade entre o veículo e o ponto e os controladores de baixo nível encarregues das atuações no veículo.

Inicialmente é desenvolvido o controlador de alto nível capaz de implementar diferentes comportamentos e implementado o controlador Proporcional Integral Derivativo (PID), com o objetivo de melhorar as respostas, como o controlo de trajetória, profundidade e velocidade, seguindo uma logica de funcionamento independente do tipo de veículo a ser implementado. Este controlador envia os valores de correção como, variação de velocidade, variação do angulo de proa e profundidade para o controlador de baixo nível.

Tendo recebido os valores, o controlador de baixo nível irá aplicar os parâmetros de atuação a cada barbatana, correspondestes com o objetivo de diminuir os valores de erro recebidos pelo controlador de alto nível.

3.1 Controladores de Baixo Nível

Estes controladores ao receber os erros vindos do controlador de alto nível terá a função de os traduzir em valores de atuação, como tal e aliado ao estudo efetuado anteriormente dos controlos básicos do veículo é possível entender quais as atuações indicadas para diminuir cada um dos erros recebidos em rumo, pitch e velocidade.

Como tal, o controlador de baixo nível, ao receber os valores de erro de rumo irá traduzir esses erros para valores de deflexão da cauda principal. O erro de pitch é traduzido para valores de deflexão das barbatanas laterais, sendo que estas possuem um valor simétrico em relação ao erro de pitch, ou seja, para um valor positivo de erro será atribuído um valor negativo de deflexão. Por fim, para o valor de erro em velocidade seriam possíveis dois tipos de atuação da cauda principal, em frequência ou em amplitude, tendo sido realizado um estudo de maneira a concluir qual a atuação que teria uma maior relevância neste parâmetro.

3.1.1 Controlador de Velocidade

Como referido anteriormente, no caso das atuações em frequência e em amplitude da cauda principal, encontram-se ambas diretamente ligadas ao movimento do veículo no plano horizontal. Sendo que foi necessário realizar um estudo preliminar com o objetivo de entender qual das duas atuações teria uma maior relevância no movimento horizontal.

Como, tal, foram realizados testes com o objetivo de entender qual a melhor atuação a ser usada para a criação do controlador de velocidade. Os testes consistiram na atribuição do ponto (x,y,z)=(100, 0, 0) de maneira a que seja possível ao veículo atingir a velocidade máxima correspondente às diferentes atuações de frequência e amplitude atribuídas. No decorrer dos testes executados, presentes na Tabela 1, foi observado que o veículo demora, em média, 25 a 30 segundos até atingir a velocidade máxima correspondente com qualquer uma das atuações apresentadas.

Para a escolha de qual a atuação a ser variada para o desenvolvimento do controlador de velocidade, foi criada a seguinte Tabela 1 que relaciona as velocidades com a variação de frequência e amplitude e a correspondente diferença entre o maior e o menor valor de velocidade para cada tipo de atuação, sendo possível concluir qual o tipo de atuação com maior influencia na velocidade do veículo.

Tabela 1: Valores de velocidade em relação à frequência e amplitude da cauda principal.

f(Hz) **Amp**	**0,5Hz**	**1,0Hz**	**1,5Hz**	**2,0Hz**	**2,5Hz**	**3,0Hz**	**3,5Hz**	$(v_{max} - v_{min})$
5°	0,15	0,26	0,45	0,70	0,78	0,80	0,84	0,69
10°	0,22	0,37	0,65	1,00	1,08	1.14	1,20	0,98
15°	0,26	0,47	0,78	1,22	1,35	1,40	1,45	1,19
20°	0,30	0,52	0,90	1,42	1,54	1,60	1,67	1,37
25°	0,33	0,60	1,02	1,59	1,72	1,82	1,87	1,54
30°	0,34	0,66	1,10	1,72	1,89	1,98	2,06	1,72
$(v_{max} - v_{min})$	0,19	0,40	0,65	1,02	1,11	1,18	1,22	

Depois de observados os valores obtidos nos testes de velocidade, apresentados na Tabela 1, é possível concluir qual a atuação que proporciona uma maior taxa de variação na velocidade em relação às restantes. Neste caso, a atuação que apresenta uma maior taxa de variação será a de amplitude fixa de 30° com a variação apenas da frequência, mas foi considerado que uma amplitude de 30° seria demasiado alta, interferindo com o angulo máximo possível atuação em deflexão. Segundo Yu, Tan, Wang, & Chen (2004) apesar de

peixes reais utilizarem amplitude e a frequência para o controlo da sua velocidade, testes demonstram que devido a um aumentado da amplitude, em conjunto com frequências altas, poderá originar o mau funcionamento, devido a uma possível limitação de rotação dos motores. Foi então escolhida a amplitude fixa de 20° sendo o terceiro valor mais alto de taxa de variação de velocidade $(\boldsymbol{v}_{max} - \boldsymbol{v}_{min})$ de 1,37 (Yu, Tan, Wang, & Chen, 2004).

3.1.2 Limitador de Ângulos

Este controlador possui implementado um limitador de ângulos para as diferentes barbatanas. Tanto a barbatana principal como as laterais, possuem limitações físicas dos seus ângulos possíveis de operação, sendo que grande parte do tempo esses limites podem ser ultrapassados consoante a localização do objetivo, devido a esta hipótese, foi necessário criar um limitador de ângulos de deflexão e amplitude da cauda e barbatanas laterais.

Foi decidido que o ângulo máximo possível para a cauda e as barbatanas laterais, seria de 85°, tendo sido usados dois tipos de métodos para a construção do limitador, primeiramente foi criado um limitador que tem em conta a soma dos ângulos de deflexão e amplitude e o excesso de 85° seria retirado apenas na amplitude, ou seja, caso a soma do angulo total fosse de 110° o excesso, de valor 25°, seria retirado apenas ao valor da atuação de amplitude.

Devido ao facto de ao ser diminuído o valor de amplitude poder existir a um decréscimo de velocidade do veículo, foi mais tarde implementado um segundo limitador que limita a deflexão e a amplitude, ou seja, tendo o mesmo caso que anteriormente, com um valor de excesso de 25°, este limitador fará com que o valor seja dividido em duas partes iguais, neste caso 12,5° e retirado aos valores de deflexão e amplitude.

Para o teste do funcionamento do limitador, foi escolhido um ponto no eixo Ox com um valor negativo de 50.0 metros, devido ao veículo iniciar sempre o trajeto com orientação para o eixo positivo Ox, este ponto irá criar um erro de angulo superior ao angulo máximo possível pela barbatana principal, sendo possível observar qual o comportamento do veículo entre os dois tipos de limitadores de ângulos desenvolvidos.

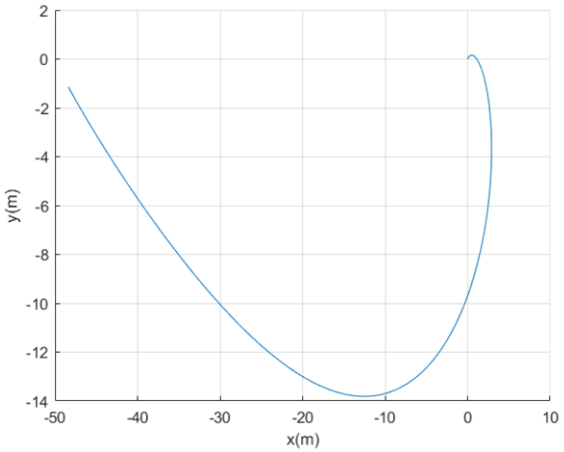

Figura 58: Resultado obtido utilizando o limitador de graus em amplitude e deflexão para o ponto x=-50.0.

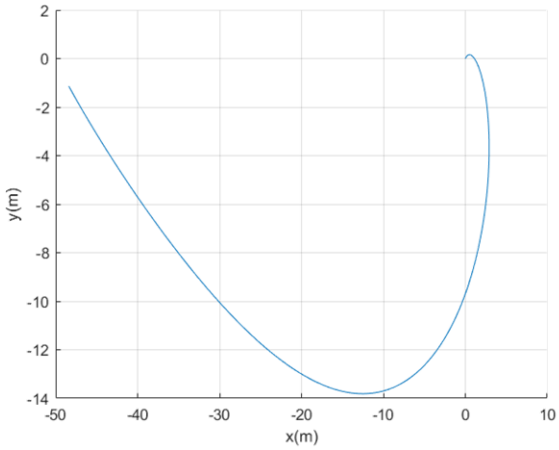

Figura 57: Resultado obtido utilizando o limitador de graus apenas em amplitude para o ponto x=-50.0.

Através da análise da Figura 57 e Figura 58, é possível concluir que em ambos os casos o comportamento do veículo será bastante semelhante, tendo sido efetuado o teste à velocidade e concluído que também não existe nenhuma diferença considerável entre os dois limitadores.

3.2 Controladores de Alto Nível

3.2.1 GoToPoint

O ponto fundamental de todos os diferentes controladores desenvolvidos, têm como base o controlador goToPoint desenvolvido com o objetivo de ser capaz de guiar o veículo até um determinado ponto dado. Este controlador é constituído por três funções, a primeira calcula o erro de rumo (heading), uma segunda função que calcula o erro do angulo vertical (pitch) e por fim a função que calcula o erro de velocidade entre a velocidade atual e a velocidade desejada.

A função de orientação no eixo horizontal, utiliza os eixos X e Y, de maneira a calcular o erro entre a posição do ponto desejado e a orientação do veículo. Como podemos observar pela Figura 59, este controlador é baseado no cálculo do erro entre o ângulo do veículo (AngleAxis) e o angulo do ponto desejado (AnglePoint) obtendo o valor de erro entre os dois (AngleError).

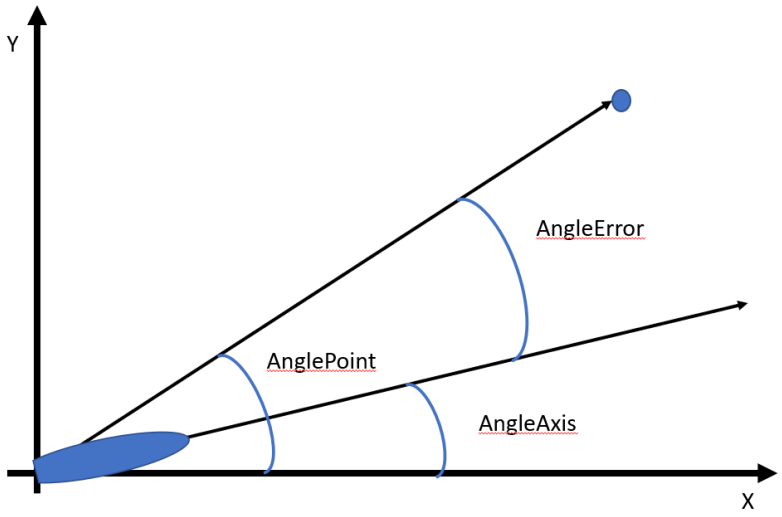

Figura 59: Representação dos respetivos ângulos para o cálculo do erro de orientação.

Depois de ser calculado o erro do ângulo entre o ponto e o veículo, esse valor é usado no controlador proporcional de maneira a ser determinado qual o ângulo necessário aplicar à deflexão da cauda principal para a correção do seu rumo.

Através da utilização do mesmo procedimento, mas utilizando os eixos XY e Z, é possível obter o erro de ângulo entre a profundidade atual e a profundidade pretendida, atuando na deflexão das barbatanas laterais, diminuindo o ângulo de erro.

A terceira função presente no controlador é a função Speed, esta função foi desenvolvida de maneira a ser possível implementar uma velocidade desejada ao veículo. O seu cálculo é baseado no erro entre a diferença da velocidade atual e da velocidade desejada.

Foram executados alguns testes de respostas quando utilizado o controlador goToPoint, representado na Figura 60 é possível observar a resposta do veículo quando iniciando o seu movimento no ponto (0.0, 0.0, 0.0) com a proa no sentido do eixo Ox face ao ponto (50.0, 40.0, 30.0) com uma velocidade de 1 m/s e valore de K=0.8 para o controlo de *heading*, K=0.9 para o controlo de *pitch* e K=0.8 para o controlo de velocidade. Depois de uma análise da resposta obtida, é concluído que a implementação de apenas o termo proporcional e a melhor relação do valor de K apresentam alguma instabilidade na resposta do veículo, sendo necessária a implementação de um termo derivativo e integral com o objetivo de estabilizar as curvas de resposta do veículo.

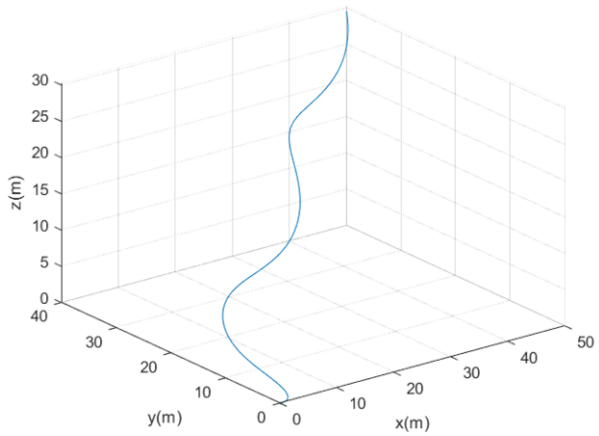

Figura 60: Resposta goToPoint usando um controlador apenas com o termo proporcional para o ponto (50.0, 40.0, 30.0).

Para a estabilização da resposta obtida anteriormente, foram adicionados os termos integrais e derivativos aos controladores já existentes de rumo, profundidade e velocidade.

Para a implementação dos controladores PID, normalmente é utilizado de maneira continua, neste caso foi usada uma versão discreta em sintonia com a periodicidade do simulador (T) tendo como equação:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt}$$

onde,

- K_p representa o ganho proporcional;
- K_i representa o ganho integral;
- K_d representa o ganho derivativo;
- e representa o valor do erro;
- t representa o tempo;
- τ representa o tempo de integração.

O coeficiente integral soma o erro ao longo do tempo, criando um aumento no componente integral, ou seja, a resposta integral irá aumentar ao longo do tempo a menos que o erro seja zero. Componente derivada tem como objetivo diminuir a saída caso o erro diminua demasiado rápido, aumentar o valor de Kd irá aumentar o parâmetro de tempo derivativo fazendo com que o controlador reaja de forma mais antecipada.

Para representar a fórmula apresentada anteriormente em código C++, foi usada a seguinte fórmula:

$$u = K * e + Ki * integral + (e - e\_last) / T * Kd$$

O coeficiente integral neste caso será a soma dos erros ao longo do tempo de T em T, onde T é a periodicidade da simulação. Em cada iteração é guardado o erro em e_last.

Depois de implementado o controlador PID, foram feitos novos testes à resposta do controlador goToPoint, usando o mesmo ponto estudado anteriormente. Os testes concluíram que a implementação do coeficiente integral, no controlo de rumo e profundidade, apresenta maior instabilidade ao resultado final, sendo que apenas a implementação do coeficiente derivativo apresenta melhorias nos resultados obtidos. É possível observar os resultados da implementação do termo proporcional na Figura 61 (exemplificado pela linha azul) e o resultado quando implementados vários valores diferentes de Kd até ser encontrado o melhor, tendo sido o valor de Kd = 6.0 o que apresentou melhores resultados em termos de estabilidade.

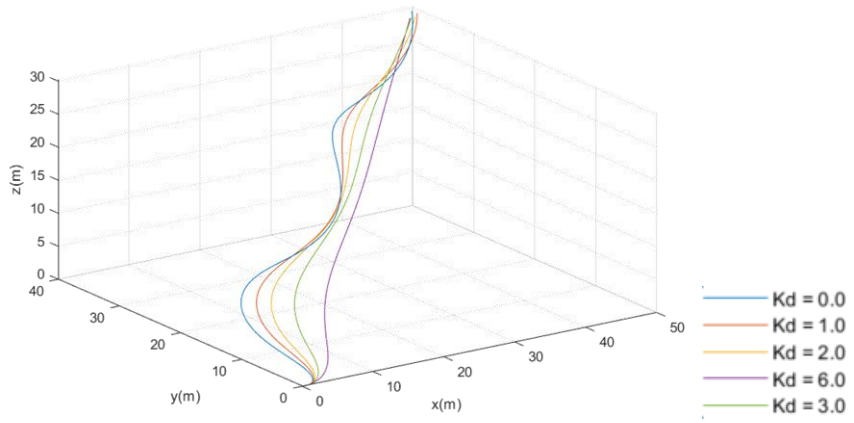

Figura 61: Representação das diferentes respostas quando aplicados diferentes valores para K do termo derivativo.

Na Figura 62 é possível analisar qual a melhoria alcançada entre a melhor resposta utilizando apenas o termo proporcional e a melhor resposta com a implementação do termo proporcional em conjunto com o termo derivativo.

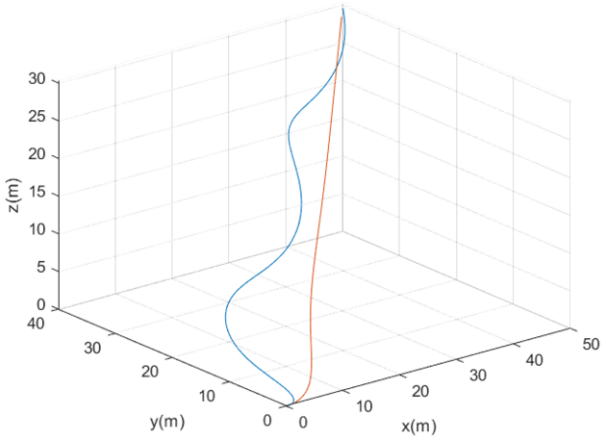

Figura 62: Resposta a vermelho do controlador goToPoint usando apenas um controlador proporcional para o ponto (50.0, 40.0, 30.0) e a azul resposta do controlador para o mesmo ponto com a implementação do termo derivativo ajustado.

Para o controlador de velocidade, foi utilizada a mesma abordagem, calculando o erro entre a velocidade atual e a velocidade desejada através do controlador proporcional. Na Figura 63 encontra-se representado o teste efetuado ao controlador com um valor de velocidade igual a 1.0 m/s, é possível concluir que apesar de apresentar valores próximos, o controlador não é capaz de atingir o valor pretendido quando utilizando apenas o termo proporcional (linha a azul), necessitando da implementação do termo integral e do termo derivativo (linhas vermelha e amarela, respetivamente).

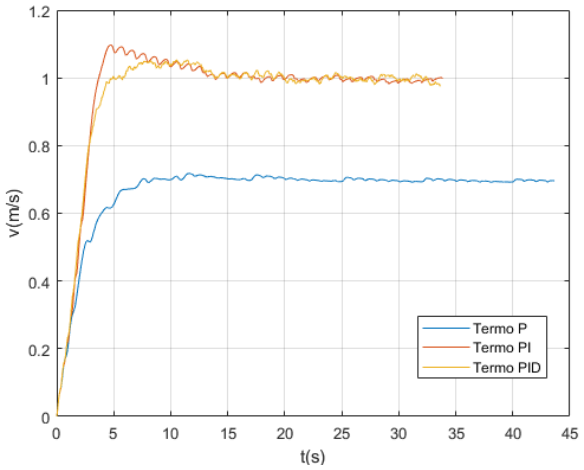

Figura 63: Diferentes respostas com a implementação de cada um dos termos no controlador de velocidade para v=1.0m/s.

Na implementação do controlador PID foram realizados vários testes com o objetivo de encontrar qual o melhor valor para o termo integral, como demonstrado pela Figura 64, apesar de todos os diferentes valores de Ki serem capazes de apresentar o valor pretendido de velocidade é na sua estabilidade inicial que diferem, tendo sido o valor de Ki=0.25 o que apresenta a melhor estabilidade inicial de todos os outros.

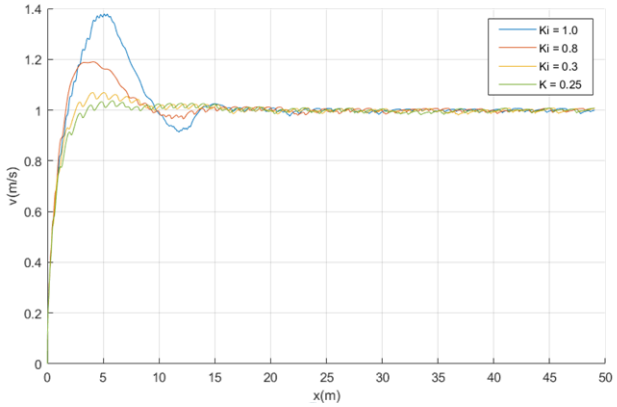

Figura 64: Representação das diferentes respostas do controlador para diferentes valores de Ki.

3.2.2 Follow

O controlador *Follow*, foi criado com o objetivo de proporcionar a capacidade ao veículo de perseguir um alvo em movimento. Este controlador, na sua base, usa o controlador anteriormente desenvolvido *goToPoint*, com a diferença, de em vez de se deslocar para um dado ponto fixo, foi atribuído um objetivo em movimento, chamado de alvo, possível de mudar o seu movimento de acordo com o tempo, neste caso, foi dado o ponto de coordenadas (5.0, 5.0, 5,0) como ponto inicial do alvo com uma velocidade de 1 m/s no sentido do eixo Ox.

Foram executados vários testes de forma a entender qual a resposta do controlador quando seguindo um ponto em movimento, sendo possível observar pelas figuras seguintes quais as diferenças de resposta quando implementado apenas o termo proporcional (Figura 65) e quando adicionado o termo derivativo (Figura 66).

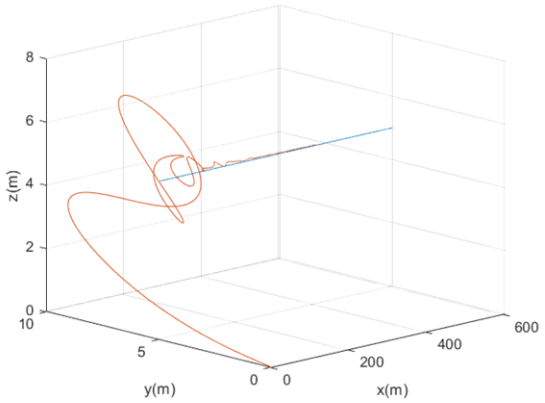

Figura 65: Resultado do controlador *follow* do ponto target inicialmente (5.0 ,5.0 ,5.0) com v=1 m/s no eixo Ox com o termo Proporcional.

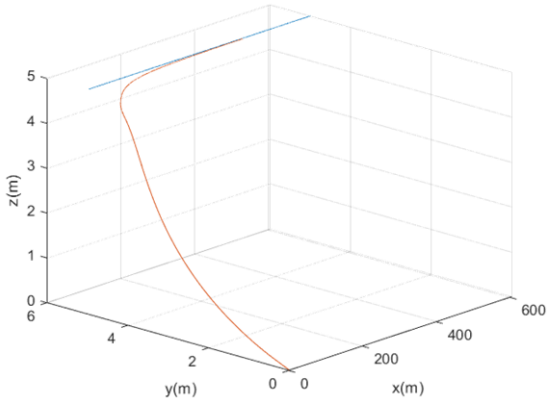

Figura 66: Representação do controlador *follow* do ponto Target inicialmente (5.0 ,5.0 ,5.0) com v=1 m/s no eixo Ox com os termos proporcional e derivativo.

Através da análise da Figura 65, é possível observar que o controlador proporcional apresenta relativamente uma boa resposta sendo capaz de seguir o ponto em movimento, mas demonstra um elevado grau de instabilidade inicial, este problema foi resolvido através da adição do termo derivativo alcançando um valor bastante mais estável para o mesmo ponto em movimento, representado na Figura 66. Outro problema encontrado foi a distância ao alvo ao longo do tempo, como nesta fase apenas foi implementada uma velocidade de cruzeiro a distância ao alvo poderia aumentar ao longo do tempo ou o veículo chegar mesmo a ultrapassar o alvo.

Foram realizados mais testes de maneira a ser possível entender qual o comportamento do controlador em vários casos diferentes, como é possível observar, existe uma perturbação inicial de estabilidade tanto no controlo de rumo como no controlo de profundidade (Figura 67) em todos os casos quando utilizado apenas o termo proporcional, mas todos eles melhorados com a implementação do termo derivativo (Figura 68).

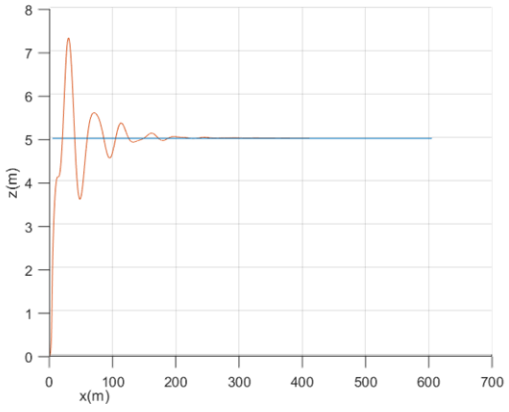

Figura 67: Controlador *follow* com ponto target de origem (5,5,5) e v=1 m/s no eixo Ox (linha azul) sem termo derivativo.

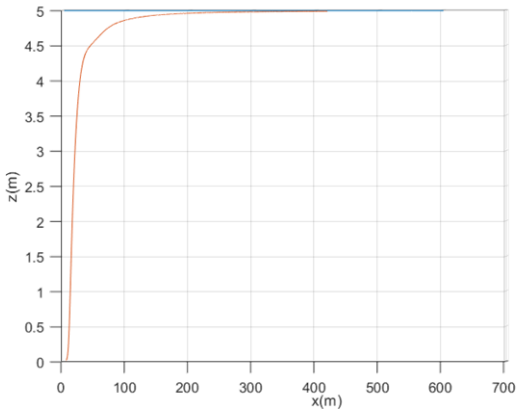

Figura 68: Controlador *follow* com ponto target de origem (5,5,5) e v=1 m/s no eixo Ox (linha azul) com os termos derivativo e proporcional.

Devido ao facto de o *target* poder ser um veículo que realiza vários movimentos de curva apertados, foi criada a possibilidade de implementar uma função que irá executar movimentos sinusoidais, com o objetivo de proporcionar um seguimento de maior dificuldade e com uma velocidade variável ao longo do seu movimento. Para a execução destes movimentos foram necessárias algumas mudanças, nomeadamente no controlador de velocidade, devido ao facto de anteriormente o veículo apenas seguir uma reta, o seu controlador de velocidade apenas necessitava de possuir uma velocidade inferior ao *target*, mas tal não acontece para a implementação de um *target* com um trajeto sinusoidal, como tal, será necessário entender qual a sua velocidade e igualar a mesma à velocidade do veículo quando este se encontra a uma determinada distância.

Posto isto, foi implementado que a velocidade de cruzeiro, ou seja, a velocidade de aproximação ao *target*, fosse uma velocidade 1,05 vezes superior à do *target* até uma distância de 1,5 metros, uma vez atingida uma distância menor que 1,5 metros o veículo irá atingir a velocidade correspondente ao *target* conseguindo assim acompanhar o seu movimento a uma distância constante.

No teste realizado para o movimento sinusoidal, foram utilizadas as seguintes funções:

- $target(x) = 1.0 \times t$;
- $target\ (y) = 5.0 \times sin(0.1 \times t) + 10.0$;
- $target\ (z) = 0.0$;

Os resultados obtidos na Figura 69 demonstram o comportamento do veículo seguindo o *target*, sendo possível observar que o veículo é capaz de seguir o movimento sinusoidal da função dada.

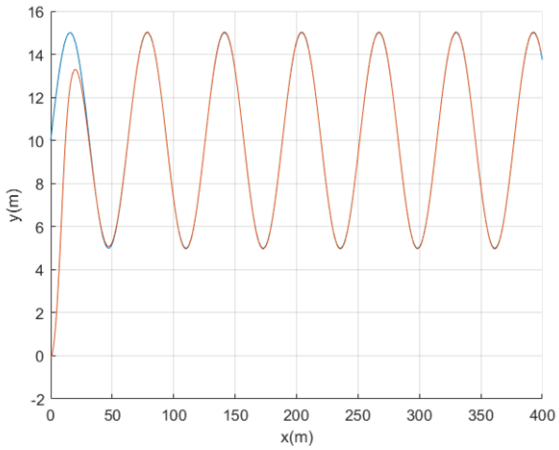

Figura 69: Movimento do veículo quando usado o controlador follow (linha vermelha) seguindo um target com uma função sinusoidal (linha azul).

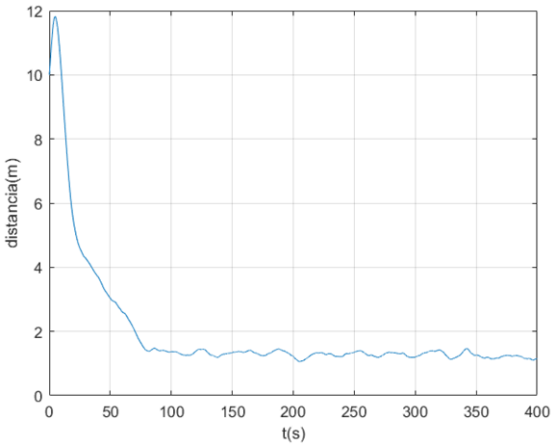

Figura 70: Demonstração da distância ao *target* com v=1.0 ao longo do tempo.

A Figura 70 demonstra a distância ao target ao longo do tempo, sendo que o veiculo foi capaz de manter uma distância constante de 1,5 metros do target, esta distancia foi considerada aceitável devido ao facto de quando se segue um determinado veículo deve sempre existir uma distância ao alvo de maneira a não existir embate.

3.2.1 GoToDepth

Foi desenvolvido um terceiro controlador, baseado no já usado para o controlo de profundidade no *goToPoint*, este controlador, direcionado apenas para a profundidade do veículo, tem apenas em conta qual a profundidade desejada e atua em conformidade com tal, não possuindo qualquer influencia da sua posição em relação aos eixos X e Y.

Este controlador poderá ser usado em várias situações, tais como, ser dada uma profundidade desejada e iniciar uma busca nessa profundidade ou até mesmo para ser dada uma ordem de emergência para regressar à superfície da água o mais rápido possível não sendo necessário um ponto em específico de coordenadas.

Nas seguintes figuras é possível observar os resultados obtidos quando introduzido um valor de profundidade de -40 metros, com a aplicação do termo proporcional (Figura 72) e as melhorias de estabilidade com a aplicação dos termos proporcional e a derivativo (Figura 71). O rumo do veículo não é alterado devido ao facto de a única atuação a ser aplicada ser apenas nas barbatanas laterais e como tal, não existir nenhuma influencia no seu rumo.

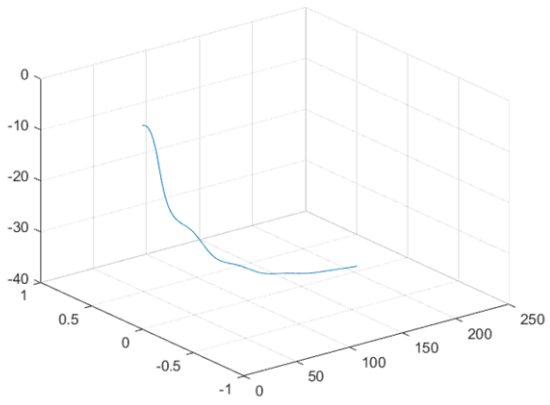

Figura 72: Controlador goToDepth para uma profundidade de -40 metros apenas com o termo proporcional.

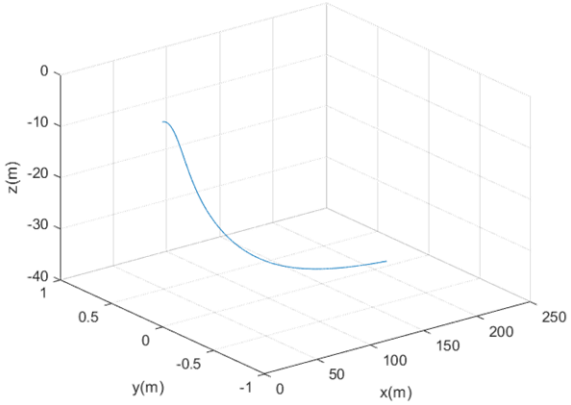

Figura 71: Controlador goToDepth para uma profundidade de -40 metros, utilizando os termos proporcional e derivativo.

4 Implementação dos Controladores e Simulador no Software Dune

De maneira a concluir o objetivo da presente dissertação foi necessário proceder à implementação do simulador e dos respetivos controladores desenvolvidos no software Dune, tendo sido utilizado como guia o simulador já existente para os veículos LAUV.

Para a sua implementação, inicialmente foram adicionadas as bibliotecas do simulador (BUVSimLib) e dos respetivos controladores desenvolvidos (BUVControlLib) às respetivas diretorias.

De seguida foi necessário criar uma tarefa com o objetivo de criar um simulador para o BUV que publique mensagens periódicas simuladas do tipo IMC (IMC::SimulatedState) com informações como a posição, posição angular, velocidade e velocidade angular do veículo. Estas mensagens são recebidas posteriormente pelas tarefas criadas para cada controlador (GoToPoint, GoToDepth e Follow) sendo que cada uma das diferentes tarefas possui um ficheiro de configuração. A tarefa pretendida, ao receber as mensagens IMC::SimulatedState, calcula os valores desejados de atuação para as barbatanas com base nos controladores de alto e baixo nível localizados na biblioteca BUVControlLib. Tendo sido calculados os valores de atuação de cada barbatana, nomeadamente, frequência, amplitude e deflexão, é criada e publicada uma mensagem do tipo IMC::BUVMotorCommand com os respetivos valores. Devido ao facto de esta mensagem não existir, foi necessário recorrer à sua criação. Para criar esta mensagem foi necessário mudar o ficheiro IMC.xml

O simulador estando à espera de que novas mensagens, do tipo IMC::BUVMotorCommand, sejam publicadas, ao receber esta mensagem atualiza os valores de atuação, originando novos valores de posição e velocidade do veículo. É possível observar toda esta interação entre cada task e o simulador, através do uso de mensagens IMC, na Figura 73.

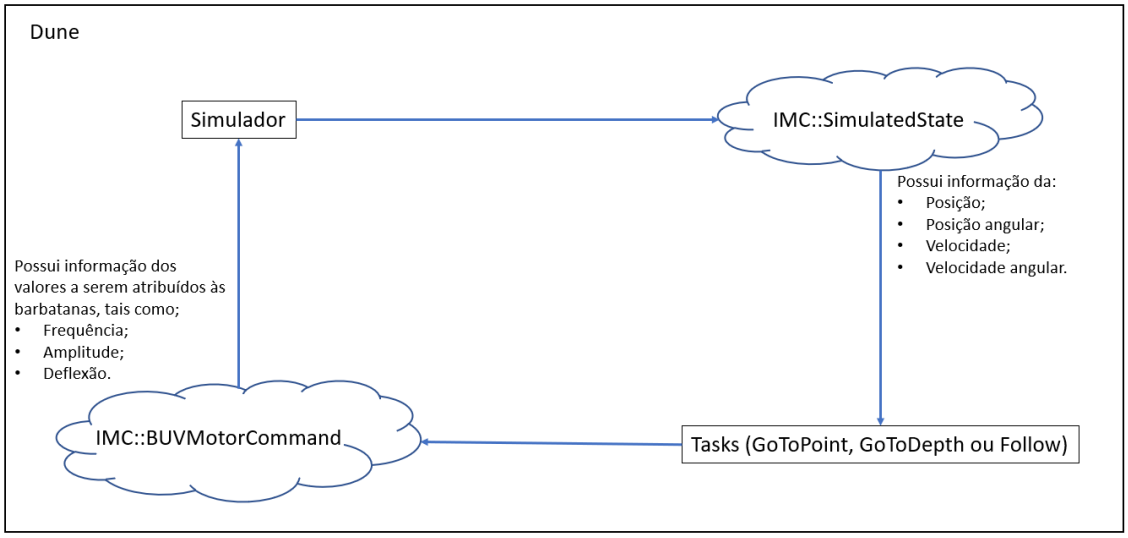

Figura 73: Interação das mensagens IMC.

No link https://github.com/ricardomn23/imc é possivel observar a criação de uma nova mensagem IMC (IMC::BUVMotorCommand) através do ficheiro IMC.xml.

Através do link https://github.com/ricardomn23/dune.git é possível aceder ao código desenvolvido na diretoria dune, onde foi criada uma pasta contendo toda a biblioteca do simulador localizada em src/Simulator/BUVSim. A biblioteca dos controladores foi adicionada à diretoria src/Control/AUV/BUVControlLib. As respetivas pastas com as tarefas de cada controlador (GoToPoint, GoToDepth e Follow) encontram-se localizadas na diretoria src/Control/AUV possuindo os respetivos ficheiros de configuração (.ini) localizados em etc/auv.

5 Conclusões e Trabalho Futuro

O controlo de veículos submarinos, que recorram a hélices, apresentam-se como uma área de relativa complexidade devido ao facto de existirem bastantes variáveis que possam influenciar o seu comportamento e controlo, como tal, para desenvolver um controlador é necessária uma análise complexa de todos os fatores envolventes. O facto de ser pretendido executar um controlador para um veículo biomimético acarreta um nível maior de complexidade, sendo que é necessário entender qual o comportamento face a todas as possíveis atuações e influencias externas.

Primeiramente foi executada uma revisão de literatura de maneira a ser compreendido o movimento dos peixes e quais os avanços recentes sobre esta área científica. A revisão de literatura permitiu entender o seu enquadramento quando comparado com o movimento dos peixes reais e quais as vantagens no seu uso. Entendendo como os peixes se movimentam, foi possível passar para a fase seguinte, do estudo das características do veículo estudado, ou seja, quais os seus componentes, o software implementado e as possíveis atuações disponíveis para o movimento do mesmo.

O objetivo inicial da presente dissertação seria o desenvolvimento de um controlador para um veículo biomimético sendo mais tarde implementado no veículo real e executados testes reais. Devido à situação de pandemia mundial que ocorreu durante o período de desenvolvimento da dissertação, não foi possível implementar o controlador no veículo real e a realização de testes. Como tal, foi utilizado o simulador desenvolvido pela Academia Naval Polaca. Este simulador, apesar de ter sido desenvolvido para um outro veículo, ligeiramente diferente, demonstrou ser uma ferramenta essencial no desenvolvimento do controlador, pois sem o simulador seria impossível compreender qual o comportamento básico do veículo perante as diferentes atuações, como por exemplo, que atuações usar para o veículo virar ou quais as atuações com maior influência na velocidade do veículo.

Através do uso do simulador foram executados inicialmente testes básicos aos atuadores do veículo, definindo quais os movimentos associados a cada atuação, de seguida e com base nos resultados obtidos através do simulador, foram desenvolvidos os controlos básicos de orientação do veículo, tal como, orientação no eixo horizontal e vertical e ainda o controlador de velocidade. Desenvolvidos estes controladores básicos, foi criado o controlador goToPoint que engloba estes três controladores de maneira a que o veículo

seja capaz de se mover até um determinado ponto, mais uma vez, sempre com resultados experimentais através do simulador. Tendo como base o controlador goToPoint, foi possível desenvolver os restantes controladores como o Follow e o goToDepth.

O controlador Follow possui na sua base o controlador goToPoint mas com algumas particularidades, devido ao facto de o seu objetivo não ser um ponto fixo como acontece no goToPoint, foi necessário mudar a sua atuação em velocidade, sendo que neste controlador o veículo varia a sua velocidade consoante a distância ao alvo e não mantém a velocidade fixa como acontece no controlador goToPoint.

O controlador goToDepth quando comparado com o controlador goToPoint, tem apenas a diferença de receber uma determinada profundidade, ou seja, ignora por completo a sua posição no plano horizontal. Este controlador tem como objetivo de ser aplicado caso exista a necessidade de fazer buscas numa determinada profundidade ou em situação de perigo em que seja necessário ao veículo vir à superfície o mais rápido possível.

O produto final da presente dissertação resulta no desenvolvimento de um controlador para o veículo biomimético BUV3 funcional, demonstrado através dos resultados obtidos por simulação e a sua implementação no software Dune. Este desenvolvimento contribui diretamente para o desenvolvimento de BUV's na Marinha Portuguesa fornecendo grandes benefícios associados à utilização de BUV's no mais variado tipo de missões.

A proposta de trabalho futuro, tendo como base o trabalho desenvolvido ao longo da presente dissertação, consiste nas seguintes sugestões:

- Execução de testes reais ao veículo, num ambiente controlado;
- Melhoramento do controlador tendo como base os testes reais executados;
- Implementação de controladores adicionais;
- Desenvolvimento de um simulador mais realista, com os respetivos parâmetros do BUV3 obtidos através de testes experimentais.

Referências Bibliográficas

Al-Mahturi, A., Santoso, F., Garratt, M. A., & Anavatti, S. G. (2019). A Robust Adaptive Interval Type-2 Fuzzy Control for Autonomous Underwater Vehicles. *2019 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*, (August), 19–24. https://doi.org/10.1109/ICIAICT.2019.8784855

Cloitre, A., Arensen, B., Patrikalakis, N. M., Youcef-Toumi, K., & Valdivia Y Alvarado, P. (2014). Propulsive performance of an underwater soft biomimetic batoid robot. *Proceedings of the International Offshore and Polar Engineering Conference*, (June), 326–333.

Colgate, J. E., & Lynch, K. M. (2004). Mechanics and control of swimming: A review. *IEEE Journal of Oceanic Engineering*, *29*(3), 660–673. https://doi.org/10.1109/JOE.2004.833208

Conry, M., Keefe, A., Ober, W., Rufo, M., & Shane, D. (2013). BIOSwimmer: Enabling technology for port security. *2013 IEEE International Conference on Technologies for Homeland Security, HST 2013*, 364–368. https://doi.org/10.1109/THS.2013.6699031

Erstorp, E. S. (2015). *Evaluation of the LSTS Toolchain for Networked Vehicle Systems on KTH Autonomous Maritime Vehicles*. https://doi.org/1651-7660

Ferreira, A. S., Pinto, J., Dias, P., & De Sousa, J. B. (2017). The LSTS software toolchain for persistent maritime operations applied through vehicular ad-hoc networks. *2017 International Conference on Unmanned Aircraft Systems, ICUAS 2017*, 609–616. https://doi.org/10.1109/ICUAS.2017.7991471

Fossen, T. I. (2011). *Handbook of Marine Craft Hydrodynamics and Motion Control*. https://doi.org/10.1002/9781119994138

Holsen, S. A. (2015). *DUNE : Unified Navigation Environment for the REMUS 100 AUV*. Norwegian University of Science and Technology Department of Marine Technology.

Hou, T., Yang, X., Su, H., Jiang, B., Chen, L., Wang, T., & Liang, J. (2019). Design and Experiments of a Squid-Like Aquatic-Aerial Vehicle with Soft Morphing Fins and Arms. *2019 International Conference on Robotics and Automation (ICRA)*, *2019-May*, 4681–4687. https://doi.org/10.1109/ICRA.2019.8793702

Katzschmann, R. K., De Maille, A., Dorhout, D. L., & Rus, D. (2016). Cyclic hydraulic actuation for soft robotic devices. *IEEE International Conference on Intelligent Robots and*

Systems, *2016-Novem*, 3048–3055. https://doi.org/10.1109/IROS.2016.7759472

Kim, B., Kim, D. H., Jung, J., & Park, J. O. (2005). A biomimetic undulatory tadpole robot using ionic polymer-metal composite actuators. *Smart Materials and Structures*, *14*(6), 1579–1585. https://doi.org/10.1088/0964-1726/14/6/051

Listewnik, K. (2013). Sound silencing problem of underwater vehicles. *Solid State Phenomena*, *196*(February 2013), 212–219. https://doi.org/10.4028/www.scientific.net/SSP.196.212

Masoomi, S. F., Gutschmidt, S., Chen, X. Q., & Sellier, M. (2015). The kinematics and dynamics of undulatory motion of a tuna-mimetic robot. *International Journal of Advanced Robotic Systems*, *12*, 1–11. https://doi.org/10.5772/60059

Pinto, José, Calado, P., Braga, J., Dias, P., Martins, R., Marques, E., & Sousa, J. B. (2012). Implementation of a control architecture for networked vehicle systems. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, *3*(PART 1), 100–105. https://doi.org/10.3182/20120410-3-pt-4028.00018

Pinto, Jose, Dias, P. S., Martins, R., Fortuna, J., Marques, E., & Sousa, J. (2013). The LSTS toolchain for networked vehicle systems. *OCEANS 2013 MTS/IEEE Bergen: The Challenges of the Northern Dimension*. https://doi.org/10.1109/OCEANS-Bergen.2013.6608148

Project SABUVIS Technical Report on Milestone No . 5. (2017). Gdynia, Poland.

Project SABUVIS Technical Report on Milestone No . 7. (2017). Gdynia, Poland.

Rufo, M. (2010). *GhostSwimmer ™ : Tactically Relevant , Biomimetically Inspired , Silent , Highly Efficient and Maneuverable Autonomous Underwater Vehicle*. Boston Engineering Corporation, 411.

Salumae, T., Chemori, A., & Kruusmaa, M. (2019). Motion Control of a Hovering Biomimetic Four-Fin Underwater Robot. *IEEE Journal of Oceanic Engineering*, *44*(1), 54–71. https://doi.org/10.1109/JOE.2017.2774318

Szymak, P. (2017). Mathematical Model of Underwater Vehicle with Undulating Propulsion. *2016 Third International Conference on Mathematics and Computers in Sciences and in Industry (MCSI)*, 269–274. https://doi.org/10.1109/mcsi.2016.057

Technical Report on Milestone No . 6 in project No . B-1452-ESM1-GP entitled : ' Swarm of

Biomimetic Underwater Vehicle for Underwater ISR ' SABUVIS Project contracted by : (2017). *3*(5), 1–82.

Wang, Z., Hang, G., Li, J., Wang, Y., & Xiao, K. (2008). A micro-robot fish with embedded SMA wire actuated flexible biomimetic fin. *Sensors and Actuators A: Physical*, *144*(2), 354–360. https://doi.org/10.1016/j.sna.2008.02.013

Yang, G. H., & Ryuh, Y. (2013). Design of high speed robotic fish "ICHTHUS V5.6." *International Conference on Control, Automation and Systems*, (lCCAS), 894–896. https://doi.org/10.1109/ICCAS.2013.6704040

Yu, J., Tan, M., Wang, S., & Chen, E. (2004). Development of a biomimetic robotic fish and its control algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *34*(4), 1798–1810. https://doi.org/10.1109/TSMCB.2004.831151

Apêndice A – Ficheiro de Configuração do Controlador GoToPoint Inserido no Dune

```
[Simulators.BUV_Sim]

Enabled                 = Simulation
Execution Frequency     = 100

Entity Label            = Simulation Engine
T                       = 0.05


[Control.AUV.GoToPoint]
Enabled                 = Simulation
Entity Label            = BUVGoToPoint
msgFrequency            = 100

; Behaviour
T                       = 0.05
duration                = 200.0
goal                    = -10.0, 10.0, 10.0
goTo_K_roll             = 1.0
goTo_K_pitch            = 0.9
goTo_Ki_pitch           = 0.0
goTo_Kd_pitch           = 10.0
goTo_K_heading          = 0.8
```

-Parâmetros de configuração da Task BUV_Sim

-Tipo de Perfil Dune Utilizado

-Frequência em que a Task é executada

-Nome da instância

-Período de simulação

-Parâmetros de configuração da Task GoToPoint

-Tipo de Perfil Dune Utilizado

-Nome da instância

-Tempo de espera por

-Parâmetros de configuração do controlador de alto nível (Behaviour)

```
; Controller
maxTailAmp          = 90.0
maxTailDev          = 90.0
maxTailFreq         = 3.5
cruiseTailAmp       = 60.0
maxSideAmp          = 45.0
maxSideDev          = 60.0
maxSideFreq         = 3.5
cruiseSideAmp       = 20.0
cruiseSideFreq      = 1.0
```

-Parâmetros de configuração do controlador de baixo nível (Controller)

Apêndice B – BUVControlLib/include/BUVControl/BUVControl.hpp

```
1.  #pragma once
2.
3.  #include "BUVSim/BUVSim.hpp"
4.
5.  #include <cmath>
6.  #include <Eigen/Dense>
7.  #include <tuple>
8.
9.  #define TRUNCATE_DEG(x) (std::remainder(x,360.0))
10. #define TRUNCATE_RAD(x) (std::remainder(x,2.0*M_PI))
11. #define SQR(x) (x)*(x)
12. #define LIMIT(x,inf,sup) ( (x < inf) ? (inf) : ((x > sup) ? (sup) : (x)) )
13. #define DEG2RAD(x) ((x)*M_PI/180.0)
14. #define RAD2DEG(x) ((x)*180.0/M_PI)
15.
16.
17. /************ CLASS BEHAVIOUR (HIGH LEVEL) ****************/
18.
19. class Behaviour {
20.     public:
21.         using Float = float;                                  // if double precision
     is needed change this (and typedefs below)
22.         using Actuation = Eigen::Vector4f;                    // change in [roll, pi
    tch, heading, speed]
23.         using Goal = Eigen::Vector3f;                             // [x_des, y_des,
    z_des]
24.         using State = BUV1_Sim::State;      // [x, y, z, roll, pitch, heading]

25.         using DState = BUV1_Sim::DState;  // derivative of State
26.
27.         Behaviour();
28.
29.         // Set Parameters
30.         void setT(Float t) {        // Sample time
31.             T = t;}
32.         void setGoToK_heading(Float K) {
33.             goTo_K_heading = K; }
34.         void setGoToK_pitch(Float K) {
35.             goTo_K_pitch = K; }
36.         void setGoToK_roll(Float K) {
37.             goTo_K_roll = K; }
38.         void setGoToK_speed(Float K) {
39.             goTo_K_speed = K; }
40.         void setCruiseSpeed(Float v) {
41.             cruiseSpeed = v; }
42.         void setReachSpeed(Float v) {
43.             reachSpeed = v; }
44.         void setReachRadius(Float r) {
45.             reachedR_squared = SQR(r); }
46.         void setGoToKi_heading(Float Ki) {
47.             goTo_Ki_heading = Ki; }
48.         void setGoToKd_heading(Float Kd) {
49.             goTo_Kd_heading = Kd; }
50.         void setGoToKd_pitch(Float Kd) {
51.             goTo_Kd_pitch = Kd; }
52.         void setGoToKi_pitch(Float Ki) {
53.             goTo_Ki_pitch = Ki; }
54.         void setGoToKd_speed(Float Kd) {
55.             goTo_Kd_speed = Kd; }
```

```
56.         void setGoToKi_speed(Float Ki) {
57.             goTo_Ki_speed = Ki; }
58.
59.
60.
61.         // Behaviours
62.         Actuation stop();
63.         Actuation goToPoint(Goal const &g, State const &state, DState const &d
    state);
64.         Actuation goToPoint(State const &state, DState const &dstate);
65.         Actuation standStill(Goal const &goal, State const &state, DState cons
    t &dstate);
66.         Actuation follow(Goal const &goal, State const &state, DState const &d
    state, Float targetV);
67.         Actuation goToDepth(Float depth, State const &state, DState const &dst
    ate);
68.         Actuation principalAxis(State const &state, DState const &dstate);
69.
70.
71.         bool hasReachedPoint(State const &state) {
72.             return (goal-
    state.head<3>()).squaredNorm() < reachedR_squared; }
73.
74.     protected:
75.         Float T;
76.         Float goTo_K_roll;
77.         Float goTo_K_pitch;
78.         Float goTo_Ki_pitch;
79.         Float goTo_Kd_pitch;
80.         Float integral_pitchT;
81.         Float goTo_K_heading;
82.         Float goTo_Ki_heading;
83.         Float goTo_Kd_heading;
84.         Float integral_headingT;
85.         Float goTo_K_speed;
86.         Float goTo_Kd_speed;
87.         Float goTo_Ki_speed;
88.         Float cruiseSpeed;
89.         Float reachSpeed;
90.         Float reachedR_squared;
91.         Float err_headingLast;
92.         Float err_pitchLast;
93.         Float err_speedLast;
94.         Float integral_speedT;
95.         Float lastDistance;
96.
97.
98.
99.         Goal goal;
100.       };
101.
102.
103.      /************ CLASS CONTROLLER (LOW LEVEL) ****************/
104.
105.      class Controller {
106.          public:
107.              using Float = float;                              // if double pr
    ecision is needed change this (and typedefs below)
108.              using Deg = Float;
109.              using Hertz = Float;
110.              using FinCommand = Eigen::Vector3f;
111.              using MotorCommand = BUVSimInterface::MotorCommand;
112.
113.              Controller();
```

```
114.
115.                 void setTailMaxAmp(Deg amp) {
116.                     maxTailAmp = amp; }
117.                 void setTailMaxDeviation(Deg dev) {
118.                     maxTailDev = dev; }
119.                 void setTailMaxFreq(Hertz freq) {
120.                     maxTailFreq = freq; }
121.                 void setTailCruiseAmp(Deg amp) {
122.                     cruiseTailAmp = amp; }
123.
124.                 void setSideMaxAmp(Deg amp) {
125.                     maxSideAmp = amp; }
126.                 void setSideMaxDeviation(Deg dev) {
127.                     maxSideDev = dev; }
128.                 void setSideMaxFreq(Hertz freq) {
129.                     maxSideFreq = freq; }
130.                 void setCruiseSideAmp(Deg amp) {
131.                     cruiseSideAmp = amp; }
132.                 void setCruiseSideFreq(Hertz freq) {
133.                     cruiseSideFreq = freq; }
134.
135.                 MotorCommand control(Behaviour::Actuation act);
136.
137.                 // Controllers
138.                 void rollControl( Float dRoll, MotorCommand &mCommand );
139.                 void pitchControl( Float dPitch, MotorCommand &mCommand );
140.                 void headingControl( Float dHeading, MotorCommand &mCommand );
141.                 void speedControl( Float dSpeed, MotorCommand &mCommand );
142.
143.                 std::tuple<Float, Float> physicalLimits( Float deflection, Floa
    t amp );
144.
145.             protected:
146.                 Deg maxTailAmp;
147.                 Deg maxTailDev;
148.                 Hertz maxTailFreq;
149.                 Deg cruiseTailAmp;
150.
151.                 Deg maxSideAmp;
152.                 Deg maxSideDev;
153.                 Hertz maxSideFreq;
154.                 Deg cruiseSideAmp;
155.                 Hertz cruiseSideFreq;
156.
157.
158.         };
```

Apêndice C – Simulador/BUVControlLib/src/BUVControl.cpp

```
1.  #include "BUVControl/BUVControl.hpp"
2.
3.
4.  using namespace std;
5.  using namespace Eigen;
6.
7.
8.
9.  /************ CLASS BEHAVIOUR (HIGH LEVEL) ***************/
10.
11. // Constructor sets default values for variables
12. Behaviour::Behaviour() :
13.         T(0.1),
14.         goTo_K_roll(1.0),
15.         goTo_K_pitch(0.0),
16.         goTo_Ki_pitch(0.0),
17.         goTo_Kd_pitch(1.0),
18.         integral_pitchT(0.0),
19.         err_pitchLast(0.0),
20.         goTo_K_heading(1.0),
21.         goTo_Ki_heading(0.2),
22.         goTo_Kd_heading(0.05),
23.         integral_headingT(0.0),
24.         err_headingLast(0.0),
25.         goTo_K_speed(2.0),
26.         cruiseSpeed(1.0),
27.         reachSpeed(1.0),
28.         reachedR_squared(1.0),
29.         err_speedLast(0.0),
30.         goTo_Kd_speed(4.0),
31.         goTo_Ki_speed(0.01),
32.         integral_speedT(0.0),
33.         lastDistance(0.0)
34.
35. {}
36.
37. Behaviour::Actuation Behaviour::stop() {
38.     return Actuation::Zero();
39. }
40.
41. // O goToPoint com o parâmetro Goal assume que é a primeira vez que é chamado.

42. // Pode-se fazer aqui as inicializações que forem necessárias
43. // (por exemplo, colocar as somas dos termos integrais a 0)
44. // Depois disso chama-se a função de baixo (sem o parâmetro Goal)
45. Behaviour::Actuation Behaviour::goToPoint(Goal const &g, State const &state, D
    State const &dstate) {
46.     goal = g;
47.     return goToPoint(state, dstate);
48. }
49.
50. Behaviour::Actuation Behaviour::principalAxis(State const &state, DState const
     &dstate) {
51.
52.     Actuation act;
53.
54.     // roll control: assuming roll_des = 0.0!
55.     Float err_roll = -TRUNCATE_RAD(state(3));
56.     // Proportional control:
```

```
57.     act[0] = goTo_K_roll * err_roll;
58.
59.     // pitch control (pitch > 0 ----> diving)
60.     Float d_xy = sqrt(SQR(goal(1)-state(1)) + SQR(goal(0)-state(0)));
61.     Float pitch_des = atan2( -(goal(2)-state(2)), d_xy);
62.     Float err_pitch = TRUNCATE_RAD(pitch_des - state(4));
63.     Float integral_pitch = integral_pitchT + err_pitch * T;
64.     integral_pitchT = integral_pitch;
65.     // Proportional control:
66.     //act[1] = goTo_K_pitch * err_pitch;
67.     act[1] =  goTo_K_pitch * err_pitch + (err_pitch -
    err_pitchLast) / T * goTo_Kd_pitch + goTo_Ki_pitch * integral_pitch;
68.     err_pitchLast = err_pitch;
69.
70.     // heading control  (dHeading > 0 ---> counterclockwise movement)
71.     Float heading_des = atan2(goal(1)-state(1),goal(0)-state(0));
72.     Float err_heading = TRUNCATE_RAD(heading_des - state(5));
73.     Float integral = integral_headingT + err_heading * T;
74.     integral_headingT = integral;
75.     // Proportional control:
76.     //act[2] =  goTo_K_heading * err_heading;
77.     act[2] =  goTo_K_heading * err_heading + (err_heading -
    err_headingLast) / T * goTo_Kd_heading + goTo_Ki_heading * integral;
78.     err_headingLast = err_heading;
79.
80.     return act;
81. }
82.
83. Behaviour::Actuation Behaviour::goToPoint(State const &state, DState const &ds
    tate) {
84.     Actuation act = principalAxis (state, dstate);
85.
86.
87.     // speed control (using only cruise speed for now...)
88.     Float curr_speed = sqrt(SQR(dstate(0)) + SQR(dstate(1)) + SQR(dstate(2)));

89.     Float err_speed = cruiseSpeed - curr_speed;
90.     // Proportional control:
91.     //act[3] =  goTo_K_speed * err_speed;
92.     //controlador PID
93.     Float integral_speed = integral_speedT + err_speed * T;
94.     integral_speedT = integral_speed;
95.     act[3] =  goTo_K_speed * err_speed + (err_speed -
    err_speedLast) / T * goTo_Kd_speed + goTo_Ki_speed * integral_speed;
96.     err_speedLast = err_speed;
97.
98.
99.     return act;
100.       }
101.
102.       Behaviour::Actuation Behaviour::standStill(Goal const &goal, State cons
    t &state, DState const &dstate) {
103.           cout << "Behaviour::standStill: NOT IMPLEMENTED YET!" << endl;
104.           return stop();
105.       }
106.
107.       Behaviour::Actuation Behaviour::follow(Goal const &g, State const &stat
    e, DState const &dstate, Float targetV) {
108.           goal = g;
109.
110.           Actuation act = principalAxis (state, dstate);
111.
112.
113.           // speed control (using only cruise speed for now...)
```

```
114.            Float curr_speed = sqrt(SQR(dstate(0)) + SQR(dstate(1)) + SQR(dstat
    e(2)));
115.            //Float distance = sqrt((goal-state.head<3>()).squaredNorm());
116.            Float distance = (goal-state.head<3>()).norm();
117.            Float err_speed;
118.
119.            cout<<"goal "<<goal.transpose()<<endl;
120.            cout<<"state "<<state.head<3>().transpose()<<endl;
121.            cout<<"distance "<<distance<<endl;
122.
123.
124.
125.            if (distance > 1.5){
126.
127.                err_speed = targetV*1.05- curr_speed;
128.
129.            }
130.            else{
131.
132.
133.                err_speed = (targetV) - curr_speed;
134.
135.                //cout<<"targetVF "<<targetV<<endl;
136.                //cout<<"curr_speed "<<curr_speed<<endl;
137.            }
138.
139.
140.            lastDistance = distance;
141.            // Proportional control:
142.            //act[3] =  goTo_K_speed * err_speed;
143.            //controlador PID
144.            Float integral_speed = integral_speedT + err_speed * T;
145.            integral_speedT = integral_speed;
146.            act[3] =  goTo_K_speed * err_speed + (err_speed -
     err_speedLast) / T * goTo_Kd_speed + goTo_Ki_speed * integral_speed;
147.            err_speedLast = err_speed;
148.
149.            //cout<<"act3 "<<act[3]<<endl;
150.
151.            return act;
152.
153.        }
154.
155.        Behaviour::Actuation Behaviour::goToDepth(Float depth, State const &sta
    te, DState const &dstate) {
156.
157.            Goal g;
158.            g << state(0), state(1), depth;
159.            goal = g;
160.
161.            Float Kz = 1.0;
162.            Float e_z = -(depth - state(2));
163.            Float pitch_des = e_z * Kz;
164.            pitch_des = LIMIT(pitch_des,-40.0,40.0) * M_PI/180;
165.
166.            Float err_pitch = TRUNCATE_RAD(pitch_des - state(4));
167.
168.            Float integral_pitch = integral_pitchT + err_pitch * T;
169.            integral_pitchT = integral_pitch;
170.            Actuation act;
171.            act[1] =  goTo_K_pitch * err_pitch + (err_pitch -
     err_pitchLast) / T * goTo_Kd_pitch + goTo_Ki_pitch * integral_pitch;
172.            err_pitchLast = err_pitch;
```

```
173.
174.            //act[1] =  goTo_K_pitch * err_pitch;
175.
176.
177.            act[0] = act[2] = 0;
178.
179.            Float curr_speed = sqrt(SQR(dstate(0)) + SQR(dstate(1)) + SQR(dstat
    e(2)));
180.            Float err_speed = cruiseSpeed - curr_speed;
181.            // Proportional control:
182.            act[3] =  goTo_K_speed * err_speed;
183.
184.
185.            return act;
186.        }
187.
188.
189.        /************ CLASS CONTROLLER (LOW LEVEL) ***************/
190.
191.        // Constructor sets default values for variables
192.        Controller::Controller() :
193.            maxTailAmp(90.0), //maximo valor de amplitude cauda
194.            maxTailDev(90.0), //maximo valor de deflection cauda
195.            maxTailFreq(3.0), //maximo valor de frequencia da cauda
196.            cruiseTailAmp(90.0), //valor standard de amplitude da cauda
197.            maxSideAmp(45.0), //maximo valor de amplitude das barbatanas altera
    is
198.            maxSideDev(60.0),
199.            maxSideFreq(3.0),
200.            cruiseSideAmp(20.0),
201.            cruiseSideFreq(1.0)
202.        {}
203.
204.        Controller::MotorCommand Controller::control(Behaviour::Actuation act)
    {
205.            MotorCommand mCommand;
206.            // Each mCommand column: frequency (Hz), mean value (degrees), ampl
    itude (degrees)
207.            // First column: tail fin
208.            // Second column: left fin
209.            // Third column: right fin
210.
211.            // no need for rollControl
212.            //rollControl( act[0], mCommand );
213.
214.            // pitch control:
215.            pitchControl( act[1], mCommand );
216.
217.            // heading control:
218.            headingControl( act[2], mCommand );
219.
220.            // heading control:
221.            speedControl( act[3], mCommand );
222.
223.            //chamar o physicalLimits para a cauda
224.            std::tuple<Float, Float> deflectionAmp = physicalLimits (mCommand(1
    ,0),mCommand(2,0));
225.
226.            mCommand(1,0) = std::get<0>(deflectionAmp); //deflection
227.            mCommand(2,0) = std::get<1>(deflectionAmp); //amplitude
228.
229.            // If simulator uses frequency in RPM uncomment this:
230.            //mCommand.row(0) *= 60.0 * 12.0;   // Gearbox with 1:12 ratio
231.
```

```
232.            return  mCommand;
233.        }
234.
235.        void Controller::rollControl( Float dRoll, MotorCommand &mCommand ) {
236.            cout << "Controller::rollControl not implemented yet!!" << endl;
237.        }
238.
239.        void Controller::pitchControl( Float dPitch, MotorCommand &mCommand ) {

240.            mCommand(1,1) = mCommand(1,2) = - LIMIT(dPitch,-
    1.0,1.0) * maxSideDev;   // Positive deflection --> negative pitch
241.
242.            mCommand(2,1) = mCommand(2,2) = cruiseSideAmp;
243.            mCommand(0,1) = mCommand(0,2) = cruiseSideFreq;
244.
245.            std::tuple<Float, Float> deflectionAmp = physicalLimits (mCommand(1
    ,2), mCommand(2,2));
246.
247.            mCommand(1,1) = mCommand(1,2) = std::get<0>(deflectionAmp); //defle
    ction
248.            mCommand(2,1) = mCommand(2,2) = std::get<1>(deflectionAmp); //ampli
    tude
249.        }
250.
251.        void Controller::headingControl( Float dHeading, MotorCommand &mCommand
     ) {
252.            mCommand(1,0) = - LIMIT(dHeading,-
    1.0,1.0) * maxTailDev;                  // Positive deflection --
    > negative heading
253.
254.            //cout<<"mCommand(1,0) "<<mCommand(1,0)<<endl;
255.        }
256.
257.        void Controller::speedControl( Float dSpeed, MotorCommand &mCommand ) {

258.            mCommand(2,0) = cruiseTailAmp;
259.            mCommand(0,0) = LIMIT(dSpeed,0.0,1.0) * maxTailFreq;
260.
261.            //cout<<"mCommand(0,0) "<<mCommand(0,0)<<endl;
262.        }
263.        /*
264.        //limitador apenas do angulo de amplitude
265.        std::tuple<Controller::Float, Controller::Float> Controller::physicalLi
    mits( Float deflection, Float amp ) {
266.
267.            Float infLimitFin = deflection - amp;
268.
269.            if (infLimitFin < -85.0){
270.
271.                Float excess = infLimitFin + 85.0;
272.
273.                Float newAmp = amp + excess;
274.
275.                return std::make_tuple (deflection, newAmp);
276.
277.            }
278.
279.            Float supLimitFin = deflection + amp;
280.
281.            if (supLimitFin > 85.0){
282.
283.                Float excess = supLimitFin - 85.0;
284.
```

```
285.                Float newAmp = amp - excess;
286.
287.                return std::make_tuple (deflection, newAmp);
288.
289.            }
290.
291.            return std::make_tuple (deflection, amp);
292.        }
293.        */
294.        //limitador de angulo de amplitude e deflection
295.        std::tuple<Controller::Float, Controller::Float> Controller::physicalLi
    mits( Float deflection, Float amp ) {
296.
297.            Float infLimitFin = deflection - amp;
298.
299.            if (infLimitFin < -85.0){
300.
301.                Float excess = infLimitFin + 85.0;
302.
303.                Float newAmp = amp + excess/2;
304.
305.                Float newDeflec = deflection - excess/2;
306.
307.                return std::make_tuple (newDeflec, newAmp);
308.
309.            }
310.
311.            Float supLimitFin = deflection + amp;
312.
313.            if (supLimitFin > 85.0){
314.
315.                Float excess = supLimitFin - 85.0;
316.
317.                Float newAmp = amp - excess/2;
318.
319.                Float newDeflec = deflection - excess/2;
320.
321.                return std::make_tuple (newDeflec, newAmp);
322.
323.            }
324.
325.            return std::make_tuple (deflection, amp);
326.        }
```

Apêndice D – BUVControlLib/test/BUVControlTest.cpp

```
1.  #include "BUVSim/BUVSim.hpp"
2.  #include "BUVControl/BUVControl.hpp"
3.  #include "Config.hpp"
4.
5.  #include <iostream>
6.  #include <fstream>
7.  #include <cmath>
8.  #include <functional>
9.
10. using namespace std;
11. using namespace Eigen;
12.
13.
14. /**** AUXILIARY FUNCTIONS ******/
15.
16. std::tuple<std::function<float(float)>, std::function<float(float)>> createFun
    ction(Config &config, std::string const& functionName, std::string const& axis
    , float defaultValue) {
17.     if (functionName == "constant") {
18.         float c = config.getFloat(axis+"Const", defaultValue);
19.         std::function<float(float)> f = [c](float t) { return c; };
20.         std::function<float(float)> df = [](float t) { return 0.0; };
21.         return std::make_tuple (f, df);
22.     } else if (functionName == "proportional") {
23.         float a = config.getFloat(axis+"A", defaultValue);
24.         float c = config.getFloat(axis+"C", defaultValue);
25.         std::function<float(float)> f = [a,c](float t) { return a * t + c; };

26.         std::function<float(float)> df = [a](float t) { return a; };
27.         return std::make_tuple (f, df);
28.     } else if (functionName == "sin") {
29.         float b = config.getFloat(axis+"B", defaultValue);
30.         float w = config.getFloat(axis+"w", defaultValue);
31.         float c = config.getFloat(axis+"C", defaultValue);
32.         std::function<float(float)> f = [b,w,c](float t) { return b * sin(w *
    t) + c; };
33.         std::function<float(float)> df = [b, w](float t) { return b*w*cos(w*t)
    ; };
34.         return std::make_tuple (f, df);
35.     } else if (functionName == "cos") {
36.         float b = config.getFloat(axis+"B", defaultValue);
37.         float w = config.getFloat(axis+"w", defaultValue);
38.         float c = config.getFloat(axis+"C", defaultValue);
39.         std::function<float(float)> f = [b,w,c](float t) { return b * cos(w *
    t) + c; };
40.         std::function<float(float)> df = [b, w](float t) { return -
    b*w*sin(w*t); };
41.         return std::make_tuple (f, df);
42.     } else { //default -> constant with defaultValue
43.         std::function<float(float)> f = [defaultValue](float t) { return defau
    ltValue; };
44.         std::function<float(float)> df = [](float t) { return 0.0; };
45.         return std::make_tuple (f, df);
46.     }
47.
48. }
49.
50. std::tuple<std::function<float(float)>, std::function<float(float)>> getFuncti
    on(Config &config, std::string const& name, float defaultValue) {
```

```
51.
52.     return createFunction(config, config.getString(name, ""), name.substr(0,1)
    , defaultValue);
53. }
54.
55. int main()
56. {
57.     cout << "BUV3 Control Demo" << endl;
58.
59.     Config config("configuration.txt");
60.
61.
62.     float T = config.getFloat("T", 0.05);                    // Sampling time
63.     float duration = config.getFloat("duration", 200.0);     // Simulation tim
    e
64.     float N = ceil(duration/T);                              // Simulation step
    s
65.
66.
67.     // Simulator and simulator parameters
68.     BUV1_Sim buv(T, config.getString("logFilename", "BUV1_Sim.log")); //possib
    ilidade de mudar o nome do ficheiro criado com parametros diferentes
69.     buv.setSeaCurr(config.getVector3f("seaCurr", Eigen::Vector3f(0.0,0.0,0.0))
    );
70.     buv.setBuoyancy(config.getFloat("buoyancy", 0.0)); // Flutuabilidade neutr
    a
71.
72.
73.
74.     // Behaviour (High Level Controller)
75.     Behaviour b;
76.     b.setT(T);
77.     b.setGoToK_heading(config.getFloat("goTo_K_heading", 0.8));
78.     b.setGoToK_pitch(config.getFloat("goTo_K_pitch", 0.4));
79.     b.setReachRadius(config.getFloat("reachRadius", 2.0));
80.     b.setGoToK_roll(config.getFloat("goTo_K_roll", 1.0));
81.     b.setGoToK_speed(config.getFloat("goTo_K_speed", 1.0));
82.     b.setCruiseSpeed(config.getFloat("cruiseSpeed", 1.0));
83.     b.setReachSpeed(config.getFloat("reachSpeed", 1.0));
84.     b.setGoToKi_heading(config.getFloat("goTo_Ki_heading", 0.1));
85.     b.setGoToKd_heading(config.getFloat("goTo_Kd_heading", 0.1));
86.     b.setGoToKi_pitch(config.getFloat("goTo_Ki_pitch", 0.1));
87.     b.setGoToKd_pitch(config.getFloat("goTo_Kd_pitch", 0.1));
88.     b.setGoToKd_speed(config.getFloat("goTo_Kd_speed", 0.1));
89.     b.setGoToKi_speed(config.getFloat("goTo_Ki_speed", 0.1));
90.
91.
92.     Controller c;
93.     c.setTailMaxAmp(config.getFloat("maxTailAmp", 45.0));
94.     c.setTailMaxDeviation(config.getFloat("maxTailDev", 60.0));
95.     c.setTailMaxFreq(config.getFloat("maxTailFreq", 3.0));
96.     c.setTailCruiseAmp(config.getFloat("cruiseTailAmp", 20.0));
97.     c.setSideMaxAmp(config.getFloat("maxSideAmp", 45.0));
98.     c.setSideMaxDeviation(config.getFloat("maxSideDev", 60.0));
99.     c.setSideMaxFreq(config.getFloat("maxSideFreq", 3.0));
100.            c.setCruiseSideAmp(config.getFloat("cruiseSideAmp", 20.0));
101.            c.setCruiseSideFreq(config.getFloat("cruiseSideFreq", 1.0));
102.
103.            Behaviour::Actuation act;
104.            Controller::MotorCommand mCommand;
105.
106.
107.            Behaviour::Goal goal = config.getVector3f("goal", Eigen::Vector3f(0
    .0,0.0,0.0));
```

```
108.
109.            Behaviour::Goal target = config.getVector3f("followTarget", Eigen::
    Vector3f(0.0,0.0,0.0));
110.
111.            float depth = config.getFloat("goToDepth", -
    10.0); //seleção da profundidade pretendida
112.
113.            int runnigMethod = (int) config.getFloat("runnigMethod", 1.0);
114.
115.            std::ofstream logfile;
116.
117.            std::string savefile = "target.log";
118.            logfile.open(savefile);
119.            if( !logfile.is_open() )
120.            {
121.                cout << " could not open \"" << savefile << "\" for logging!" <
    < endl;
122.                return false;
123.            }
124.
125.            std::tuple<std::function<float(float)>, std::function<float(float)>
    > xFunctions = getFunction(config, "xTargetFunction", 0.0);
126.            std::function<float(float)> xTargetFunction = std::get<0>(xFunction
    s);
127.            std::function<float(float)> xDTargetFunction = std::get<1>(xFunctio
    ns);
128.            std::tuple<std::function<float(float)>, std::function<float(float)>
    > yFunctions = getFunction(config, "yTargetFunction", 0.0);
129.            std::function<float(float)> yTargetFunction = std::get<0>(yFunction
    s);
130.            std::function<float(float)> yDTargetFunction = std::get<1>(yFunctio
    ns);
131.            std::tuple<std::function<float(float)>, std::function<float(float)>
    > zFunctions = getFunction(config, "zTargetFunction", 0.0);
132.            std::function<float(float)> zTargetFunction = std::get<0>(zFunction
    s);
133.            std::function<float(float)> zDTargetFunction = std::get<1>(zFunctio
    ns);
134.
135.
136.            float t = 0.0;
137.
138.            for(int i=0; i<N; i++)
139.            {
140.                target (0) = xTargetFunction(t);
141.                target (1) = yTargetFunction(t);
142.                target (2) = zTargetFunction(t);
143.
144.                float vx = xDTargetFunction (t);
145.                float vy = yDTargetFunction (t);
146.                float vz = zDTargetFunction (t);
147.                float targetV = sqrt(SQR(vx)+SQR(vy)+SQR(vz));
148.
149.
150.                switch (runnigMethod){ //switch para escolher qual o controlado
    r a ser utilizado no ficheiro de configuração
151.
152.                    case 1 :
153.                        act = b.goToPoint(goal, buv.getState(), buv.getDState()
    ); // veículo ir para um ponto dado
154.                        break;
155.
156.                    case 2 :
```

```
157.                        act = b.follow(target, buv.getState(), buv.getDState(),
     targetV); //função de veículo seguir outro veículo
158.                        break;
159.
160.                    case 3 :
161.                        act = b.goToDepth(depth, buv.getState(), buv.getDState(
    )); //controlador de profundidade
162.                        break;
163.
164.                    default :
165.                        act = b.goToPoint(goal, buv.getState(), buv.getDState()
    ); // veículo ir para um ponto dado
166.                }
167.
168.                mCommand = c.control(act);
169.
170.                logfile << target.transpose() << endl;
171.                // Simulate
172.                buv.setMotorCommands(mCommand);
173.                buv.update();
174.
175.                // Control
176.                if( b.hasReachedPoint( buv.getState()) )
177.                    break;
178.
179.                t += T;
180.
181.
182.            }
183.
184.            logfile.close();
185.            return 0;
186.        }
```

Apêndice E – imc/IMC.xml (Criação de uma nova mensagem IMC)

```
1.  <message id="910" name="BUVMotorCommand" abbrev="BUVMotorCommand" source="">
2.     <description>
3.       BUVMotorCommand for tail and side fins (Freq, Deflection and Amp)
4.     </description>
5.     <field name="tail_frequency" abbrev="tail_frequency" type="fp32_t" unit="Hz
    ">
6.     </field>
7.     <field name="tail_deflection" abbrev="tail_deflection" type="fp32_t" unit="
    degrees">
8.     </field>
9.  <field name="tail_amplitude" abbrev="tail_amplitude" type="fp32_t" unit="degre
    es">
10.    </field>
11. <field name="left_fin_frequency" abbrev="left_fin_frequency" type="fp32_t" uni
    t="Hz">
12.    </field>
13.    <field name="left_fin_deflection" abbrev="left_fin_deflection" type="fp32_t
    " unit="degrees">
14.    </field>
15. <field name="left_fin_amplitude" abbrev="left_fin_amplitude" type="fp32_t" uni
    t="degrees">
16.    </field>
17. <field name="right_fin_frequency" abbrev="right_fin_frequency" type="fp32_t" u
    nit="Hz">
18.    </field>
19.    <field name="right_fin_deflection" abbrev="right_fin_deflection" type="fp32
    _t" unit="degrees">
20.    </field>
21. <field name="right_fin_amplitude" abbrev="right_fin_amplitude" type="fp32_t" u
    nit="degrees">
22.    </field>
23.   </message>
```

Apêndice F – dune/etc/auv/BUVFollow.ini

```
[Simulators.BUV_Sim]
Enabled                                 = Simulation
Execution Frequency                     = 100
Entity Label                            = Simulation Engine
T                                       = 0.05


[Control.AUV.Follow]
Enabled                                 = Simulation
Entity Label                            = BUVFollow
msgFrequency                            = 100

; TargetFunction can be proportional, constant, sin or cos
;for proportional (xA and xC)
;for constant (zConst)
;for sin or cos (yB, yw and yC)
xTargetFunction                         = proportional
yTargetFunction                         = sin
zTargetFunction                         = constant

xA                                      = 1.0
xC                                      = 0.0

yB                                      = 5.0
yw                                      = 0.1
yC                                      = 10.0

zConst                                  = 2.0

; Behaviour
T                                       = 0.05
duration                                = 200.0
goTo_K_roll                             = 1.0
goTo_K_pitch                            = 0.9
goTo_Ki_pitch                           = 0.0
goTo_Kd_pitch                           = 10.0
goTo_K_heading                          = 0.8
goTo_Ki_heading                         = 0.0
goTo_Kd_heading                         = 6.0
goTo_K_speed                            = 0.8
goTo_Kd_speed                           = 0.7
goTo_Ki_speed                           = 0.25
cruiseSpeed                             = 1.0
reachRadius                             = 2.0
reachSpeed                              = 1.0

; Controller
maxTailAmp                              = 90.0
maxTailDev                              = 90.0
maxTailFreq                             = 3.5
cruiseTailAmp                           = 90.0
maxSideAmp                              = 45.0
maxSideDev                              = 60.0
maxSideFreq                             = 3.5
cruiseSideAmp                           = 20.0
cruiseSideFreq                          = 1.0
```

Apêndice G – dune/etc/auv/ BUVGoToDepth.ini

```
[Simulators.BUV_Sim]
Enabled                                 = Simulation
Execution Frequency                     = 100
Entity Label                            = Simulation Engine
T                                       = 0.05


[Control.AUV.GoToDepth]
Enabled                                 = Simulation
Entity Label                            = BUVGoToDepth
msgFrequency                            = 100

depth                                   = -40.0

; Behaviour
T                                       = 0.05
duration                                = 400.0
goTo_K_roll                             = 1.0
goTo_K_pitch                            = 0.9
goTo_Ki_pitch                           = 0.0
goTo_Kd_pitch                           = 10.0
goTo_K_heading                          = 0.8
goTo_Ki_heading                         = 0.0
goTo_Kd_heading                         = 6.0
goTo_K_speed                            = 0.8
goTo_Kd_speed                           = 0.7
goTo_Ki_speed                           = 0.25
cruiseSpeed                             = 1.0
reachRadius                             = 2.0
reachSpeed                              = 1.0

; Controller
maxTailAmp                              = 45.0
maxTailDev                              = 60.0
maxTailFreq                             = 3.0
cruiseTailAmp                           = 20.0
maxSideAmp                              = 45.0
maxSideDev                              = 60.0
maxSideFreq                             = 3.5
cruiseSideAmp                           = 20.0
cruiseSideFreq                          = 1.0
```

Apêndice H – dune/etc/auv/ BUVGoToDepth.ini

```
[Simulators.BUV_Sim]
Enabled                         = Simulation
Execution Frequency             = 100
Entity Label                    = Simulation Engine
T                               = 0.05


[Control.AUV.GoToPoint]
Enabled                         = Simulation
Entity Label                    = BUVGoToPoint
msgFrequency                    = 100

; Behaviour
T                                = 0.05
duration                         = 200.0
goal                             = -10.0, 10.0, 10.0
goTo_K_roll                      = 1.0
goTo_K_pitch                     = 0.9
goTo_Ki_pitch                    = 0.0
goTo_Kd_pitch                    = 10.0
goTo_K_heading                   = 0.8
goTo_Ki_heading                  = 0.0
goTo_Kd_heading                  = 6.0
goTo_K_speed                     = 0.8
goTo_Kd_speed                    = 0.7
goTo_Ki_speed                    = 0.25
cruiseSpeed                      = 1.0
reachRadius                      = 2.0
reachSpeed                       = 1.0

; Controller
maxTailAmp                       = 90.0
maxTailDev                       = 90.0
maxTailFreq                      = 3.5
cruiseTailAmp                    = 60.0
maxSideAmp                       = 45.0
maxSideDev                       = 60.0
maxSideFreq                      = 3.5
cruiseSideAmp                    = 20.0
cruiseSideFreq                   = 1.0
```

Apêndice I – dune/src/Control/AUV/Follow/Task.cpp

```
1.  // DUNE headers.
2.  #include <DUNE/DUNE.hpp>
3.  #include "../BUVControlLib/include/BUVControl/BUVControl.hpp"
4.  #include <vector>
5.  #include <functional>
6.  #include <csignal>
7.
8.  namespace Control
9.  {
10.   //! Insert short task description here.
11.   //!
12.   //! Insert explanation on task behaviour here.
13.   //! @author ricardo
14.   namespace AUV
15.   {
16.     namespace Follow
17.     {
18.       using DUNE_NAMESPACES;
19.
20.       struct Arguments
21.       {
22.           float msgFrequency;
23.           std::string xTargetFunction;
24.           std::string yTargetFunction;
25.           std::string zTargetFunction;
26.           float xA;
27.           float xB;
28.           float xC;
29.           float xw;
30.           float xConst;
31.           float yA;
32.           float yB;
33.           float yC;
34.           float yw;
35.           float yConst;
36.           float zA;
37.           float zB;
38.           float zC;
39.           float zw;
40.           float zConst;
41.
42.
43.
44.           float T;
45.           float duration;
46.           float goTo_K_roll;
47.           float goTo_K_pitch;
48.           float goTo_Ki_pitch;
49.           float goTo_Kd_pitch;
50.           float goTo_K_heading;
51.           float goTo_Ki_heading;
52.           float goTo_Kd_heading;
53.           float goTo_K_speed;
54.           float goTo_Ki_speed;
55.           float goTo_Kd_speed;
56.           float cruiseSpeed;
57.           float reachRadius;
58.           float reachSpeed;
59.
```

```
60.           float maxTailAmp;
61.           float maxTailDev;
62.           float maxTailFreq;
63.           float cruiseTailAmp;
64.           float maxSideAmp;
65.           float maxSideDev;
66.           float maxSideFreq;
67.           float cruiseSideAmp;
68.           float cruiseSideFreq;
69.           float targetV;
70.       };
71.
72.
73.       struct Task: public DUNE::Tasks::Task
74.       {
75.         IMC::BUVMotorCommand m_motorCommand;
76.         Behaviour* behaviour;
77.         Controller* controller;
78.         Arguments m_args;
79.         float currentTime;
80.         float T;
81.         bool hasFinished;
82.         float duration;
83.         std::function<float(float)> xTargetFunction;
84.         std::function<float(float)> yTargetFunction;
85.         std::function<float(float)> zTargetFunction;
86.         std::function<float(float)> xDTargetFunction;
87.         std::function<float(float)> yDTargetFunction;
88.         std::function<float(float)> zDTargetFunction;
89.         std::ofstream targetlogfile;
90.
91.
92.
93.
94.         //! Constructor.
95.         //! @param[in] name task name.
96.         //! @param[in] ctx context.
97.         Task(const std::string& name, Tasks::Context& ctx):
98.           DUNE::Tasks::Task(name, ctx)
99.         {
100.                param("msgFrequency", m_args.msgFrequency)
101.                .defaultValue("100")
102.                .description("msgFrequency");
103.
104.                param("xTargetFunction", m_args.xTargetFunction)
105.                .defaultValue("constant")
106.                .description("função do follow");
107.
108.                param("yTargetFunction", m_args.yTargetFunction)
109.                .defaultValue("constant")
110.                .description("função do follow");
111.
112.                param("zTargetFunction", m_args.zTargetFunction)
113.                .defaultValue("constant")
114.                .description("função do follow");
115.
116.                param("xA", m_args.xA)
117.                .defaultValue("1.0")
118.                .description("valor de xA");
119.
120.                param("yA", m_args.yA)
121.                .defaultValue("1.0")
122.                .description("valor de yA");
123.
```

```
124.                    param("zA", m_args.zA)
125.                    .defaultValue("1.0")
126.                    .description("valor de zA");
127.
128.                    param("xB", m_args.xB)
129.                    .defaultValue("1.0")
130.                    .description("valor de xB");
131.
132.                    param("yB", m_args.yB)
133.                    .defaultValue("1.0")
134.                    .description("valor de yB");
135.
136.                    param("zB", m_args.zB)
137.                    .defaultValue("1.0")
138.                    .description("valor de zB");
139.
140.                    param("xC", m_args.xC)
141.                    .defaultValue("1.0")
142.                    .description("valor de xC");
143.
144.                    param("yC", m_args.yC)
145.                    .defaultValue("1.0")
146.                    .description("valor de yC");
147.
148.                    param("zC", m_args.zC)
149.                    .defaultValue("1.0")
150.                    .description("valor de zC");
151.
152.                    param("xw", m_args.xw)
153.                    .defaultValue("1.0")
154.                    .description("valor de xw");
155.
156.                    param("yw", m_args.yw)
157.                    .defaultValue("1.0")
158.                    .description("valor de yw");
159.
160.                    param("zw", m_args.zw)
161.                    .defaultValue("1.0")
162.                    .description("valor de zw");
163.
164.                    param("xConst", m_args.xConst)
165.                    .defaultValue("1.0")
166.                    .description("valor de xConst");
167.
168.                    param("yConst", m_args.yConst)
169.                    .defaultValue("1.0")
170.                    .description("valor de yConst");
171.
172.                    param("zConst", m_args.zConst)
173.                    .defaultValue("1.0")
174.                    .description("valor de zConst");
175.
176.                    //Behaviour
177.
178.                    param("T", m_args.T)
179.                    .defaultValue("0.05")
180.                    .description("Valor da duração de cada simulação");
181.
182.                    param("duration", m_args.duration)
183.                    .defaultValue("400.0")
184.                    .description("Valor da duração da simulação");
185.
186.                    param("goTo_K_roll", m_args.goTo_K_roll)
```

```
187.                    .defaultValue("1.0")
188.                    .description("Valor de K para o roll");
189.
190.                    param("goTo_K_pitch", m_args.goTo_K_pitch)
191.                    .defaultValue("0.9")
192.                    .description("Valor de K para o pitch");
193.
194.                    param("goTo_Ki_pitch", m_args.goTo_Ki_pitch)
195.                    .defaultValue("0.0")
196.                    .description("Valor de Ki para o pitch");
197.
198.                    param("goTo_Kd_pitch", m_args.goTo_Kd_pitch)
199.                    .defaultValue("10.0")
200.                    .description("Valor de Kd para o pitch");
201.
202.                    param("goTo_K_heading", m_args.goTo_K_heading)
203.                    .defaultValue("0.8")
204.                    .description("Valor de K para o heading");
205.
206.                    param("goTo_Ki_heading", m_args.goTo_Ki_heading)
207.                    .defaultValue("0.0")
208.                    .description("Valor de Ki para o heading");
209.
210.                    param("goTo_Kd_heading", m_args.goTo_Kd_heading)
211.                    .defaultValue("6.0")
212.                    .description("Valor de Kd para o heading");
213.
214.                    param("goTo_K_speed", m_args.goTo_K_speed)
215.                    .defaultValue("0.8")
216.                    .description("Valor de K para a velocidade");
217.
218.                    param("goTo_Ki_speed", m_args.goTo_Ki_speed)
219.                    .defaultValue("0.25")
220.                    .description("Valor de Ki para a velocidade");
221.
222.                    param("goTo_Kd_speed", m_args.goTo_Kd_speed)
223.                    .defaultValue("0.7")
224.                    .description("Valor de Kd para a velocidade");
225.
226.                    param("cruiseSpeed", m_args.cruiseSpeed)
227.                    .defaultValue("1.0")
228.                    .description("Valor da velocidade de cruzeiro");
229.
230.                    //Controller
231.                    param("maxTailAmp", m_args.maxTailAmp)
232.                    .defaultValue("90.0")
233.                    .description("Valor de amplitude maxima da cauda principal");
234.
235.                    param("maxTailDev", m_args.maxTailDev)
236.                    .defaultValue("90.0")
237.                    .description("Valor de deflexão maxima da cauda principal");
238.
239.                    param("maxTailFreq", m_args.maxTailFreq)
240.                    .defaultValue("3.5")
241.                    .description("Valor de frequencia maxima da cauda principal")
    ;
242.
243.                    param("cruiseTailAmp", m_args.cruiseTailAmp)
244.                    .defaultValue("90.0")
245.                    .description("Valor de amplitude de cruzeiro da cauda princip
    al");
246.
```

```
247.                 param("maxSideAmp", m_args.maxSideAmp)
248.                 .defaultValue("45.0")
249.                 .description("Valor de amplitude máxima das barbatanas latera
    is");
250.
251.                 param("maxSideDev", m_args.maxSideDev)
252.                 .defaultValue("60.0")
253.                 .description("Valor de deflexão máxima das barbatanas laterai
    s");
254.
255.                 param("maxSideFreq", m_args.maxSideFreq)
256.                 .defaultValue("3.5")
257.                 .description("Valor de deflexão máxima das barbatanas laterai
    s");
258.
259.                 param("cruiseSideAmp", m_args.cruiseSideAmp)
260.                 .defaultValue("20.0")
261.                 .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
262.
263.                 param("cruiseSideFreq", m_args.cruiseSideFreq)
264.                 .defaultValue("1.0")
265.                 .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
266.
267.                 param("reachRadius", m_args.reachRadius)
268.                 .defaultValue("2.0")
269.                 .description("Valor de distancia ao objetivo em que para a si
    mulação");
270.
271.                 param("reachSpeed", m_args.reachSpeed)
272.                 .defaultValue("1.0")
273.                 .description("Valor de velocidade de chegada ao objetivo");
274.
275.                 behaviour = new Behaviour();
276.                 controller = new Controller();
277.                 currentTime = 0.0;
278.                 hasFinished = false;
279.
280.
281.                 bind<IMC::SimulatedState>(this);
282.             }
283.
284.             std::string getFunctionName (std::string const& axis){
285.
286.                 if (axis == "x") {
287.                     return m_args.xTargetFunction;
288.                 }
289.                 else if (axis == "y") {
290.                     return m_args.yTargetFunction;
291.                 }
292.                 else {
293.                     return m_args.zTargetFunction;
294.                 }
295.
296.             }
297.
298.             float getA (std::string const& axis){
299.
300.                 if (axis == "x")
301.                     return m_args.xA;
302.
303.                 else if (axis == "y")
```

```
304.                         return m_args.yA;
305.
306.                     else
307.                         return m_args.zA;
308.
309.                 }
310.
311.                 float getB (std::string const& axis){
312.
313.                     if (axis == "x")
314.                         return m_args.xB;
315.
316.                     else if (axis == "y")
317.                         return m_args.yB;
318.
319.                     else
320.                         return m_args.zB;
321.
322.                 }
323.
324.                 float getC (std::string const& axis){
325.
326.                     if (axis == "x")
327.                         return m_args.xC;
328.
329.                     else if (axis == "y")
330.                         return m_args.yC;
331.
332.                     else
333.                         return m_args.zC;
334.
335.                 }
336.
337.                 float getw (std::string const& axis){
338.
339.                     if (axis == "x")
340.                         return m_args.xw;
341.
342.                     else if (axis == "y")
343.                         return m_args.yw;
344.
345.                     else
346.                         return m_args.zw;
347.
348.                 }
349.
350.                 float getConst (std::string const& axis){
351.
352.                     if (axis == "x")
353.                         return m_args.xConst;
354.
355.                     else if (axis == "y")
356.                         return m_args.yConst;
357.
358.                     else
359.                         return m_args.zConst;
360.
361.                 }
362.
363.                 std::tuple<std::function<float(float)>, std::function<float(flo
    at)>> createFunction(std::string const& axis) {
364.                     std::string functionName = getFunctionName(axis);
365.                     if (functionName == "constant") {
366.                         float c = getConst (axis);
```

```
367.                         std::function<float(float)> f = [c](float t) { return c
    ; };
368.                         std::function<float(float)> df = [](float t) { return 0
    .0; };
369.                         return std::make_tuple (f, df);
370.                     } else if (functionName == "proportional") {
371.                         float a = getA (axis);
372.                         float c = getC (axis);
373.                         std::function<float(float)> f = [a,c](float t) { return
     a * t + c; };
374.                         std::function<float(float)> df = [a](float t) { return
    a; };
375.                         return std::make_tuple (f, df);
376.                     } else if (functionName == "sin") {
377.                         float b = getB (axis);
378.                         float w = getw (axis);
379.                         float c = getC (axis);
380.                         std::function<float(float)> f = [b,w,c](float t) { retu
    rn b * sin(w * t) + c; };
381.                         std::function<float(float)> df = [b, w](float t) { retu
    rn b*w*cos(w*t); };
382.                         return std::make_tuple (f, df);
383.                     } else if (functionName == "cos") {
384.                         float b = getB (axis);
385.                         float w = getw (axis);
386.                         float c = getC (axis);
387.                         std::function<float(float)> f = [b,w,c](float t) { retu
    rn b * cos(w * t) + c; };
388.                         std::function<float(float)> df = [b, w](float t) { retu
    rn -b*w*sin(w*t); };
389.                         return std::make_tuple (f, df);
390.                     } else { //default -> constant with defaultValue
391.                         float c = getConst (axis);
392.                         std::function<float(float)> f = [c](float t) { return c
    ; };
393.                         std::function<float(float)> df = [](float t) { return 0
    .0; };
394.                         return std::make_tuple (f, df);
395.                     }
396.
397.                 }
398.
399.                 //! Update internal state with new parameter values.
400.                 void
401.                 onUpdateParameters(void)
402.                 {
403.                   T = m_args.T;
404.                   duration = m_args.duration;
405.                   behaviour->setT(m_args.T);
406.                   behaviour->setGoToK_heading(m_args.goTo_K_heading);
407.                   behaviour->setGoToK_pitch(m_args.goTo_K_pitch);
408.                   behaviour->setReachRadius(m_args.reachRadius);
409.                   behaviour->setGoToK_roll(m_args.goTo_K_roll);
410.                   behaviour->setGoToK_speed(m_args.goTo_K_speed);
411.                   behaviour->setCruiseSpeed(m_args.cruiseSpeed);
412.                   behaviour->setReachSpeed(m_args.reachSpeed);
413.                   behaviour->setGoToKi_heading(m_args.goTo_Ki_heading);
414.                   behaviour->setGoToKd_heading(m_args.goTo_Kd_heading);
415.                   behaviour->setGoToKi_pitch(m_args.goTo_Ki_pitch);
416.                   behaviour->setGoToKd_pitch(m_args.goTo_Kd_pitch);
417.                   behaviour->setGoToKd_speed(m_args.goTo_Kd_speed);
418.                   behaviour->setGoToKi_speed(m_args.goTo_Ki_speed);
419.
```

```
420.                 controller->setTailMaxAmp(m_args.maxTailAmp);
421.                 controller->setTailMaxDeviation(m_args.maxTailDev);
422.                 controller->setTailMaxFreq(m_args.maxTailFreq);
423.                 controller->setTailCruiseAmp(m_args.cruiseTailAmp);
424.                 controller->setSideMaxAmp(m_args.maxSideAmp);
425.                 controller->setSideMaxDeviation(m_args.maxSideDev);
426.                 controller->setSideMaxFreq(m_args.maxSideFreq);
427.                 controller->setCruiseSideAmp(m_args.cruiseSideAmp);
428.                 controller->setCruiseSideFreq(m_args.cruiseSideFreq);
429.
430.                 std::tuple<std::function<float(float)>, std::function<float(f
    loat)>> xFunctions = createFunction("x");
431.                 xTargetFunction = std::get<0>(xFunctions);
432.                 xDTargetFunction = std::get<1>(xFunctions);
433.                 std::tuple<std::function<float(float)>, std::function<float(f
    loat)>> yFunctions = createFunction("y");
434.                 yTargetFunction = std::get<0>(yFunctions);
435.                 yDTargetFunction = std::get<1>(yFunctions);
436.                 std::tuple<std::function<float(float)>, std::function<float(f
    loat)>> zFunctions = createFunction("z");
437.                 zTargetFunction = std::get<0>(zFunctions);
438.                 zDTargetFunction = std::get<1>(zFunctions);
439.
440.
441.                   std::string savefile = "target.log";
442.                   targetlogfile.open(savefile);
443.                   if( !targetlogfile.is_open() )
444.                   {
445.                       std::cout << " could not open \"" << savefile << "\" fo
    r logging!" << std::endl;
446.
447.                   }
448.               }
449.
450.               //! Reserve entity identifiers.
451.               void
452.               onEntityReservation(void)
453.               {
454.               }
455.
456.               //! Resolve entity names.
457.               void
458.               onEntityResolution(void)
459.               {
460.               }
461.
462.               //! Acquire resources.
463.               void
464.               onResourceAcquisition(void)
465.               {
466.               }
467.
468.               //! Initialize resources.
469.               void
470.               onResourceInitialization(void)
471.               {
472.               }
473.
474.               //! Release resources.
475.               void
476.               onResourceRelease(void)
477.               {
478.                 //Memory::clear(behaviour);
479.                 //Memory::clear(controller);
```

```
480.                }
481.
482.                void
483.                consume(const IMC::SimulatedState* msg)
484.                {
485.                  BUV1_Sim::State state;
486.                  state(0)= msg->x;
487.                  state(1)= msg->y;
488.                  state(2)= msg->z;
489.                  state(3)= msg->phi;
490.                  state(4)= msg->theta;
491.                  state(5)= msg->psi;
492.
493.                  BUV1_Sim::DState dstate;
494.                  dstate(0)= msg->u;
495.                  dstate(1)= msg->v;
496.                  dstate(2)= msg->w;
497.                  dstate(3)= msg->p;
498.                  dstate(4)= msg->q;
499.                  dstate(5)= msg->r;
500.
501.                  Behaviour::Goal target;
502.
503.                  target (0) = xTargetFunction(currentTime);
504.                  target (1) = yTargetFunction(currentTime);
505.                  target (2) = zTargetFunction(currentTime);
506.
507.                  float vx = xDTargetFunction (currentTime);
508.                  float vy = yDTargetFunction (currentTime);
509.                  float vz = zDTargetFunction (currentTime);
510.                  float targetV = sqrt(SQR(vx)+SQR(vy)+SQR(vz));
511.
512.                  Behaviour::Actuation act = behaviour-
    >follow(target, state, dstate, targetV);
513.                  Controller::MotorCommand mCommand = controller-
    >control(act);
514.                  m_motorCommand.tail_frequency = mCommand(0,0);
515.                  m_motorCommand.tail_deflection = mCommand(1,0);
516.                  m_motorCommand.tail_amplitude = mCommand(2,0);
517.                  m_motorCommand.left_fin_frequency = mCommand(0,1);
518.                  m_motorCommand.left_fin_deflection = mCommand(1,1);
519.                  m_motorCommand.left_fin_amplitude = mCommand(2,1);
520.                  m_motorCommand.right_fin_frequency = mCommand(0,2);
521.                  m_motorCommand.right_fin_deflection = mCommand(1,2);
522.                  m_motorCommand.right_fin_amplitude = mCommand(2,2);
523.
524.                  if(currentTime >= duration){
525.
526.                    if(!hasFinished){
527.
528.                      std::cout <<"Duration Time Out!"<< std::endl;
529.
530.                      raise( SIGINT);
531.                    }
532.
533.                    hasFinished = true;
534.                    return;
535.                  }
536.
537.
538.                  currentTime += T;
539.
540.
```

```
541.                 targetlogfile << target.transpose() << std::endl;
542.
543.                 dispatch(m_motorCommand);
544.               }
545.
546.
547.               //! Main loop.
548.               void
549.               onMain(void)
550.               {
551.                 while (!stopping())
552.                 {
553.                   waitForMessages(1.0/m_args.msgFrequency);
554.                 }
555.               }
556.             };
557.           }
558.         }
559.       }
560.
561.       DUNE_TASK
```

Apêndice J – dune/src/Control/AUV/GoToDepth/Task.cpp

```
1.  // DUNE headers.
2.  #include <DUNE/DUNE.hpp>
3.  #include "../BUVControlLib/include/BUVControl/BUVControl.hpp"
4.  #include <vector>
5.  #include <csignal>
6.
7.
8.  namespace Control
9.  {
10.   //! Insert short task description here.
11.   //!
12.   //! Insert explanation on task behaviour here.
13.   //! @author ricardo
14.   namespace AUV
15.   {
16.     namespace GoToDepth
17.     {
18.       using DUNE_NAMESPACES;
19.
20.       struct Arguments
21.       {
22.           float msgFrequency;
23.           float depth;
24.           float T;
25.           float duration;
26.           float goTo_K_roll;
27.           float goTo_K_pitch;
28.           float goTo_Ki_pitch;
29.           float goTo_Kd_pitch;
30.           float goTo_K_heading;
31.           float goTo_Ki_heading;
32.           float goTo_Kd_heading;
33.           float goTo_K_speed;
34.           float goTo_Ki_speed;
35.           float goTo_Kd_speed;
36.           float cruiseSpeed;
37.           float reachRadius;
38.           float reachSpeed;
39.
40.           float maxTailAmp;
41.           float maxTailDev;
42.           float maxTailFreq;
43.           float cruiseTailAmp;
44.           float maxSideAmp;
45.           float maxSideDev;
46.           float maxSideFreq;
47.           float cruiseSideAmp;
48.           float cruiseSideFreq;
49.           float targetV;
50.       };
51.
52.
53.       struct Task: public DUNE::Tasks::Task
54.       {
55.         IMC::BUVMotorCommand m_motorCommand;
56.         Behaviour* b;
57.         Controller* c;
58.         Arguments m_args;
59.         bool hasReachedPoint;
```

```
60.
61.
62.         //! Constructor.
63.         //! @param[in] name task name.
64.         //! @param[in] ctx context.
65.         Task(const std::string& name, Tasks::Context& ctx):
66.           DUNE::Tasks::Task(name, ctx)
67.         {
68.
69.           param("msgFrequency", m_args.msgFrequency)
70.           .defaultValue("100")
71.           .description("msgFrequency");
72.
73.           //Behaviour
74.
75.            param("depth", m_args.depth)
76.           .defaultValue("-10.0")
77.           .description("Valor da profundidade");
78.
79.           param("T", m_args.T)
80.           .defaultValue("0.05")
81.           .description("Valor da duração de cada simulação");
82.
83.           param("duration", m_args.duration)
84.           .defaultValue("400.0")
85.           .description("Valor da duração da simulação");
86.
87.           param("goTo_K_roll", m_args.goTo_K_roll)
88.           .defaultValue("1.0")
89.           .description("Valor de K para o roll");
90.
91.           param("goTo_K_pitch", m_args.goTo_K_pitch)
92.           .defaultValue("0.9")
93.           .description("Valor de K para o pitch");
94.
95.           param("goTo_Ki_pitch", m_args.goTo_Ki_pitch)
96.           .defaultValue("0.0")
97.           .description("Valor de Ki para o pitch");
98.
99.           param("goTo_Kd_pitch", m_args.goTo_Kd_pitch)
100.                  .defaultValue("10.0")
101.                  .description("Valor de Kd para o pitch");
102.
103.                 param("goTo_K_heading", m_args.goTo_K_heading)
104.                 .defaultValue("0.8")
105.                 .description("Valor de K para o heading");
106.
107.                 param("goTo_Ki_heading", m_args.goTo_Ki_heading)
108.                 .defaultValue("0.0")
109.                 .description("Valor de Ki para o heading");
110.
111.                 param("goTo_Kd_heading", m_args.goTo_Kd_heading)
112.                 .defaultValue("6.0")
113.                 .description("Valor de Kd para o heading");
114.
115.                 param("goTo_K_speed", m_args.goTo_K_speed)
116.                 .defaultValue("0.8")
117.                 .description("Valor de K para a velocidade");
118.
119.                 param("goTo_Ki_speed", m_args.goTo_Ki_speed)
120.                 .defaultValue("0.25")
121.                 .description("Valor de Ki para a velocidade");
122.
123.                 param("goTo_Kd_speed", m_args.goTo_Kd_speed)
```

```
124.                    .defaultValue("0.7")
125.                    .description("Valor de Kd para a velocidade");
126.
127.                    param("cruiseSpeed", m_args.cruiseSpeed)
128.                    .defaultValue("1.0")
129.                    .description("Valor da velocidade de cruzeiro");
130.
131.                    //Controller
132.                    param("maxTailAmp", m_args.maxTailAmp)
133.                    .defaultValue("90.0")
134.                    .description("Valor de amplitude maxima da cauda principal");

135.
136.                    param("maxTailDev", m_args.maxTailDev)
137.                    .defaultValue("90.0")
138.                    .description("Valor de deflexão maxima da cauda principal");

139.
140.                    param("maxTailFreq", m_args.maxTailFreq)
141.                    .defaultValue("3.5")
142.                    .description("Valor de frequencia maxima da cauda principal")
    ;
143.
144.                    param("cruiseTailAmp", m_args.cruiseTailAmp)
145.                    .defaultValue("90.0")
146.                    .description("Valor de amplitude de cruzeiro da cauda princip
    al");
147.
148.                    param("maxSideAmp", m_args.maxSideAmp)
149.                    .defaultValue("45.0")
150.                    .description("Valor de amplitude máxima das barbatanas latera
    is");
151.
152.                    param("maxSideDev", m_args.maxSideDev)
153.                    .defaultValue("60.0")
154.                    .description("Valor de deflexão máxima das barbatanas laterai
    s");
155.
156.                    param("maxSideFreq", m_args.maxSideFreq)
157.                    .defaultValue("3.5")
158.                    .description("Valor de deflexão máxima das barbatanas laterai
    s");
159.
160.                    param("cruiseSideAmp", m_args.cruiseSideAmp)
161.                    .defaultValue("20.0")
162.                    .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
163.
164.                    param("cruiseSideFreq", m_args.cruiseSideFreq)
165.                    .defaultValue("1.0")
166.                    .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
167.
168.                    param("reachRadius", m_args.reachRadius)
169.                    .defaultValue("2.0")
170.                    .description("Valor de distancia ao objetivo em que para a si
    mulação");
171.
172.                    param("reachSpeed", m_args.reachSpeed)
173.                    .defaultValue("1.0")
174.                    .description("Valor de velocidade de chegada ao objetivo");
175.
176.                    b = new Behaviour();
```

```
177.                c = new Controller();
178.
179.                bind<IMC::SimulatedState>(this);
180.              }
181.
182.              //! Update internal state with new parameter values.
183.              void
184.              onUpdateParameters(void)
185.              {
186.                b->setT(m_args.T);
187.                b->setGoToK_heading(m_args.goTo_K_heading);
188.                b->setGoToK_pitch(m_args.goTo_K_pitch);
189.                b->setReachRadius(m_args.reachRadius);
190.                b->setGoToK_roll(m_args.goTo_K_roll);
191.                b->setGoToK_speed(m_args.goTo_K_speed);
192.                b->setCruiseSpeed(m_args.cruiseSpeed);
193.                b->setReachSpeed(m_args.reachSpeed);
194.                b->setGoToKi_heading(m_args.goTo_Ki_heading);
195.                b->setGoToKd_heading(m_args.goTo_Kd_heading);
196.                b->setGoToKi_pitch(m_args.goTo_Ki_pitch);
197.                b->setGoToKd_pitch(m_args.goTo_Kd_pitch);
198.                b->setGoToKd_speed(m_args.goTo_Kd_speed);
199.                b->setGoToKi_speed(m_args.goTo_Ki_speed);
200.
201.                c->setTailMaxAmp(m_args.maxTailAmp);
202.                c->setTailMaxDeviation(m_args.maxTailDev);
203.                c->setTailMaxFreq(m_args.maxTailFreq);
204.                c->setTailCruiseAmp(m_args.cruiseTailAmp);
205.                c->setSideMaxAmp(m_args.maxSideAmp);
206.                c->setSideMaxDeviation(m_args.maxSideDev);
207.                c->setSideMaxFreq(m_args.maxSideFreq);
208.                c->setCruiseSideAmp(m_args.cruiseSideAmp);
209.                c->setCruiseSideFreq(m_args.cruiseSideFreq);
210.              }
211.
212.              //! Reserve entity identifiers.
213.              void
214.              onEntityReservation(void)
215.              {
216.              }
217.
218.              //! Resolve entity names.
219.              void
220.              onEntityResolution(void)
221.              {
222.              }
223.
224.              //! Acquire resources.
225.              void
226.              onResourceAcquisition(void)
227.              {
228.              }
229.
230.              //! Initialize resources.
231.              void
232.              onResourceInitialization(void)
233.              {
234.              }
235.
236.              //! Release resources.
237.              void
238.              onResourceRelease(void)
239.              {
240.                //Memory::clear(b);
```

```
241.                 //Memory::clear(c);
242.               }
243.
244.               void
245.               consume(const IMC::SimulatedState* msg)
246.               {
247.                 BUV1_Sim::State state;
248.                 state(0)= msg->x;
249.                 state(1)= msg->y;
250.                 state(2)= msg->z;
251.                 state(3)= msg->phi;
252.                 state(4)= msg->theta;
253.                 state(5)= msg->psi;
254.
255.                 BUV1_Sim::DState dstate;
256.                 dstate(0)= msg->u;
257.                 dstate(1)= msg->v;
258.                 dstate(2)= msg->w;
259.                 dstate(3)= msg->p;
260.                 dstate(4)= msg->q;
261.                 dstate(5)= msg->r;
262.
263.                 Behaviour::Actuation act = b-
    >goToDepth(m_args.depth, state, dstate);
264.                 Controller::MotorCommand mCommand = c->control(act);
265.                 m_motorCommand.tail_frequency = mCommand(0,0);
266.                 m_motorCommand.tail_deflection = mCommand(1,0);
267.                 m_motorCommand.tail_amplitude = mCommand(2,0);
268.                 m_motorCommand.left_fin_frequency = mCommand(0,1);
269.                 m_motorCommand.left_fin_deflection = mCommand(1,1);
270.                 m_motorCommand.left_fin_amplitude = mCommand(2,1);
271.                 m_motorCommand.right_fin_frequency = mCommand(0,2);
272.                 m_motorCommand.right_fin_deflection = mCommand(1,2);
273.                 m_motorCommand.right_fin_amplitude = mCommand(2,2);
274.
275.                 if( b->hasReachedPoint(state) ){
276.
277.                   if(!hasReachedPoint){
278.
279.                     std::cout <<"BUV has Reached Depth!"<< std::endl;
280.
281.                     raise( SIGINT);
282.                   }
283.                   hasReachedPoint = true;
284.                   return;
285.                 }
286.
287.                 dispatch(m_motorCommand);
288.               }
289.
290.
291.               //! Main loop.
292.               void
293.               onMain(void)
294.               {
295.                 //while (!stopping() && !hasReachedPoint)
296.                 while (!stopping())
297.                 {
298.                   waitForMessages(1.0/m_args.msgFrequency);
299.                 }
300.               }
301.             };
302.           }
```

```
303.          }
304.        }
305.
306.        DUNE_TASK
```

Apêndice K – dune/src/Control/AUV/GoToPoint/Task.cpp

```
1.  // DUNE headers.
2.  #include <DUNE/DUNE.hpp>
3.  #include "../BUVControlLib/include/BUVControl/BUVControl.hpp"
4.  #include <vector>
5.  #include <csignal>
6.
7.  namespace Control
8.  {
9.    //! Insert short task description here.
10.   //!
11.   //! Insert explanation on task behaviour here.
12.   //! @author ricardo
13.   namespace AUV
14.   {
15.     namespace GoToPoint
16.     {
17.       using DUNE_NAMESPACES;
18.
19.       struct Arguments
20.       {
21.         float msgFrequency;
22.         float T;
23.         std::vector<float> goal;
24.         float duration;
25.         float goTo_K_roll;
26.         float goTo_K_pitch;
27.         float goTo_Ki_pitch;
28.         float goTo_Kd_pitch;
29.         float goTo_K_heading;
30.         float goTo_Ki_heading;
31.         float goTo_Kd_heading;
32.         float goTo_K_speed;
33.         float goTo_Ki_speed;
34.         float goTo_Kd_speed;
35.         float cruiseSpeed;
36.         float reachRadius;
37.         float reachSpeed;
38.
39.         float maxTailAmp;
40.         float maxTailDev;
41.         float maxTailFreq;
42.         float cruiseTailAmp;
43.         float maxSideAmp;
44.         float maxSideDev;
45.         float maxSideFreq;
46.         float cruiseSideAmp;
47.         float cruiseSideFreq;
48.       };
49.
50.
51.       struct Task: public DUNE::Tasks::Task
52.       {
53.         IMC::BUVMotorCommand m_motorCommand;
54.         Behaviour* b;
55.         Controller* c;
56.         Behaviour::Goal goal;
57.         Arguments m_args;
58.         bool hasReachedPoint;
59.
```

```
60.
61.         //! Constructor.
62.         //! @param[in] name task name.
63.         //! @param[in] ctx context.
64.         Task(const std::string& name, Tasks::Context& ctx):
65.           DUNE::Tasks::Task(name, ctx)
66.         {
67.
68.           param("msgFrequency", m_args.msgFrequency)
69.           .defaultValue("100")
70.           .description("msgFrequency");
71.
72.           //Behaviour
73.
74.           param("T", m_args.T)
75.           .defaultValue("0.05")
76.           .description("Valor da duração de cada simulação");
77.
78.           param("duration", m_args.duration)
79.           .defaultValue("400.0")
80.           .description("Valor da duração da simulação");
81.
82.           param("goal", m_args.goal)
83.           .defaultValue("10.0, 10.0, 10.0")
84.           .size(3)
85.           .description("Posição do objetivo");
86.
87.           param("goTo_K_roll", m_args.goTo_K_roll)
88.           .defaultValue("1.0")
89.           .description("Valor de K para o roll");
90.
91.           param("goTo_K_pitch", m_args.goTo_K_pitch)
92.           .defaultValue("0.9")
93.           .description("Valor de K para o pitch");
94.
95.           param("goTo_Ki_pitch", m_args.goTo_Ki_pitch)
96.           .defaultValue("0.0")
97.           .description("Valor de Ki para o pitch");
98.
99.           param("goTo_Kd_pitch", m_args.goTo_Kd_pitch)
100.                  .defaultValue("10.0")
101.                  .description("Valor de Kd para o pitch");
102.
103.                 param("goTo_K_heading", m_args.goTo_K_heading)
104.                 .defaultValue("0.8")
105.                 .description("Valor de K para o heading");
106.
107.                 param("goTo_Ki_heading", m_args.goTo_Ki_heading)
108.                 .defaultValue("0.0")
109.                 .description("Valor de Ki para o heading");
110.
111.                 param("goTo_Kd_heading", m_args.goTo_Kd_heading)
112.                 .defaultValue("6.0")
113.                 .description("Valor de Kd para o heading");
114.
115.                 param("goTo_K_speed", m_args.goTo_K_speed)
116.                 .defaultValue("0.8")
117.                 .description("Valor de K para a velocidade");
118.
119.                 param("goTo_Ki_speed", m_args.goTo_Ki_speed)
120.                 .defaultValue("0.25")
121.                 .description("Valor de Ki para a velocidade");
122.
123.                 param("goTo_Kd_speed", m_args.goTo_Kd_speed)
```

```
124.                    .defaultValue("0.7")
125.                    .description("Valor de Kd para a velocidade");
126.
127.                    param("cruiseSpeed", m_args.cruiseSpeed)
128.                    .defaultValue("1.0")
129.                    .description("Valor da velocidade de cruzeiro");
130.
131.                    //Controller
132.                    param("maxTailAmp", m_args.maxTailAmp)
133.                    .defaultValue("90.0")
134.                    .description("Valor de amplitude maxima da cauda principal");

135.
136.                    param("maxTailDev", m_args.maxTailDev)
137.                    .defaultValue("90.0")
138.                    .description("Valor de deflexão maxima da cauda principal");

139.
140.                    param("maxTailFreq", m_args.maxTailFreq)
141.                    .defaultValue("3.5")
142.                    .description("Valor de frequencia maxima da cauda principal")
    ;
143.
144.                    param("cruiseTailAmp", m_args.cruiseTailAmp)
145.                    .defaultValue("90.0")
146.                    .description("Valor de amplitude de cruzeiro da cauda princip
    al");
147.
148.                    param("maxSideAmp", m_args.maxSideAmp)
149.                    .defaultValue("45.0")
150.                    .description("Valor de amplitude máxima das barbatanas latera
    is");
151.
152.                    param("maxSideDev", m_args.maxSideDev)
153.                    .defaultValue("60.0")
154.                    .description("Valor de deflexão máxima das barbatanas laterai
    s");
155.
156.                    param("maxSideFreq", m_args.maxSideFreq)
157.                    .defaultValue("3.5")
158.                    .description("Valor de deflexão máxima das barbatanas laterai
    s");
159.
160.                    param("cruiseSideAmp", m_args.cruiseSideAmp)
161.                    .defaultValue("20.0")
162.                    .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
163.
164.                    param("cruiseSideFreq", m_args.cruiseSideFreq)
165.                    .defaultValue("1.0")
166.                    .description("Valor de amplitude de cruzeiro das barbatanas l
    aterais");
167.
168.                    param("reachRadius", m_args.reachRadius)
169.                    .defaultValue("2.0")
170.                    .description("Valor de distancia ao objetivo em que para a si
    mulação");
171.
172.                    param("reachSpeed", m_args.reachSpeed)
173.                    .defaultValue("1.0")
174.                    .description("Valor de velocidade de chegada ao objetivo");
175.
176.                    b = new Behaviour();
```

```
177.                 c = new Controller();
178.                 hasReachedPoint = false;
179.
180.                 bind<IMC::SimulatedState>(this);
181.               }
182.
183.               //! Update internal state with new parameter values.
184.               void
185.               onUpdateParameters(void)
186.               {
187.                 b->setT(m_args.T);
188.                 b->setGoToK_heading(m_args.goTo_K_heading);
189.                 b->setGoToK_pitch(m_args.goTo_K_pitch);
190.                 b->setReachRadius(m_args.reachRadius);
191.                 b->setGoToK_roll(m_args.goTo_K_roll);
192.                 b->setGoToK_speed(m_args.goTo_K_speed);
193.                 b->setCruiseSpeed(m_args.cruiseSpeed);
194.                 b->setReachSpeed(m_args.reachSpeed);
195.                 b->setGoToKi_heading(m_args.goTo_Ki_heading);
196.                 b->setGoToKd_heading(m_args.goTo_Kd_heading);
197.                 b->setGoToKi_pitch(m_args.goTo_Ki_pitch);
198.                 b->setGoToKd_pitch(m_args.goTo_Kd_pitch);
199.                 b->setGoToKd_speed(m_args.goTo_Kd_speed);
200.                 b->setGoToKi_speed(m_args.goTo_Ki_speed);
201.
202.                 c->setTailMaxAmp(m_args.maxTailAmp);
203.                 c->setTailMaxDeviation(m_args.maxTailDev);
204.                 c->setTailMaxFreq(m_args.maxTailFreq);
205.                 c->setTailCruiseAmp(m_args.cruiseTailAmp);
206.                 c->setSideMaxAmp(m_args.maxSideAmp);
207.                 c->setSideMaxDeviation(m_args.maxSideDev);
208.                 c->setSideMaxFreq(m_args.maxSideFreq);
209.                 c->setCruiseSideAmp(m_args.cruiseSideAmp);
210.                 c->setCruiseSideFreq(m_args.cruiseSideFreq);
211.
212.
213.
214.                 goal(0) = m_args.goal[0];
215.                 goal(1) = m_args.goal[1];
216.                 goal(2) = m_args.goal[2];
217.               }
218.
219.               //! Reserve entity identifiers.
220.               void
221.               onEntityReservation(void)
222.               {
223.               }
224.
225.               //! Resolve entity names.
226.               void
227.               onEntityResolution(void)
228.               {
229.               }
230.
231.               //! Acquire resources.
232.               void
233.               onResourceAcquisition(void)
234.               {
235.               }
236.
237.               //! Initialize resources.
238.               void
239.               onResourceInitialization(void)
240.               {
```

```
241.                }
242.
243.                //! Release resources.
244.                void
245.                onResourceRelease(void)
246.                {
247.                  //Memory::clear(b);
248.                  //Memory::clear(c);
249.                }
250.
251.                void
252.                consume(const IMC::SimulatedState* msg)
253.                {
254.                  BUV1_Sim::State state;
255.                  state(0)= msg->x;
256.                  state(1)= msg->y;
257.                  state(2)= msg->z;
258.                  state(3)= msg->phi;
259.                  state(4)= msg->theta;
260.                  state(5)= msg->psi;
261.
262.                  BUV1_Sim::DState dstate;
263.                  dstate(0)= msg->u;
264.                  dstate(1)= msg->v;
265.                  dstate(2)= msg->w;
266.                  dstate(3)= msg->p;
267.                  dstate(4)= msg->q;
268.                  dstate(5)= msg->r;
269.
270.
271.                  Behaviour::Actuation act = b-
    >goToPoint(goal, state, dstate);
272.                  Controller::MotorCommand mCommand = c->control(act);
273.                  m_motorCommand.tail_frequency = mCommand(0,0);
274.                  m_motorCommand.tail_deflection = mCommand(1,0);
275.                  m_motorCommand.tail_amplitude = mCommand(2,0);
276.                  m_motorCommand.left_fin_frequency = mCommand(0,1);
277.                  m_motorCommand.left_fin_deflection = mCommand(1,1);
278.                  m_motorCommand.left_fin_amplitude = mCommand(2,1);
279.                  m_motorCommand.right_fin_frequency = mCommand(0,2);
280.                  m_motorCommand.right_fin_deflection = mCommand(1,2);
281.                  m_motorCommand.right_fin_amplitude = mCommand(2,2);
282.
283.                  if( b->hasReachedPoint(state) ){
284.
285.                    if(!hasReachedPoint){
286.
287.                      std::cout <<"BUV has Reached Point!"<< std::endl;
288.
289.                      raise( SIGINT);
290.                    }
291.
292.                    hasReachedPoint = true;
293.                    return;
294.                  }
295.
296.                  dispatch(m_motorCommand);
297.                }
298.
299.
300.                //! Main loop.
301.                void
302.                onMain(void)
```

```
303.               {
304.                 //while (!stopping() && !hasReachedPoint)
305.                 while (!stopping())
306.                 {
307.                   waitForMessages(1.0/m_args.msgFrequency);
308.                 }
309.               }
310.             };
311.           }
312.         }
313.       }
314.
315.       DUNE_TASK
```