TRABALHO DE GRADUAÇÃO

APLICAÇÃO DE SISTEMAS AUTOADAPTATIVOS EM ROBÓTICA MÓVEL: UM ESTUDO DE CASO NO ROBÔ PIONEER

João Vitor de Melo Peixoto

Brasília, Dezembro de 2019

ENGENHARIA MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

APLICAÇÃO DE SISTEMAS AUTOADAPTATIVOS EM ROBÓTICA MÓVEL: UM ESTUDO DE CASO NO ROBÔ PIONEER

João Vitor de Melo Peixoto

Relatório submetido como requisito parcial de obtenção de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Carla Maria C. C. Koike, UnB/CIC _____

Prof. Genaína Nunes Rodrigues, UnB/CIC _____

Prof. Marcus Vinícius Lamar, UnB/CIC _____

Braslia, Dezembro de 2019

FICHA CATALOGRÁFICA

JOÃO VITOR, DE MELO PEIXOTO
Aplicação de sistemas autoadaptativos em robótica móvel: um estudo de caso no robô Pioneer

[Distrito Federal] 2019.

xii, 79p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2019). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. Robótica móvel
2. Sistema autoadaptativo
3. Visão computacional
4. ROS
I. Mecatrônica/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

PEIXOTO, J. V. M., (2019). Aplicação de sistemas autoadaptativos em robótica móvel: um estudo de caso no robô Pioneer. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-$n°$07, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 91p.

CESSÃO DE DIREITOS

João Vitor de Melo Peixoto

Colônia Agrícola Samambaia, Ch. 94 - Vicente Pires

72155-000– DF – Brasil.

Agradecimentos

Primeiramente, a mim mesmo por toda dedicação, esforço e persistência em realizar este trabalho, por ter sempre acreditado no meu potencial.

Aos meus pais, por terem sempre me dado as condições necessárias incondicionalmente, pelo apoio e conselhos para chegar até onde cheguei. Eu não seria ninguém na vida se não fosse por todo amor de vocês.

À minha coordenadora, Carla, por todo conhecimento e sabedoria transmitidos, pela paciência e incentivos quando me encontrei em situações difíceis. Agradeço também a co-orientadora Genaína. Com certeza o aprendizado adquirido com vocês será levado para além da minha graduação.

Ao Eric, meu amigo, cujas contribuições foram importantíssimas e agregaram muito ao meu trabalho. Agradeço muito também ao Ricardo, que em várias vezes me ajudou com seu conhecimento.

Aos meus melhores amigos da faculdade, Peg e Comps, por todos momentos descontraídos e risadas. Mesmo sem perceber, serviram de inspiração em diversos momentos ao longo do curso. Agradeço também a todos meus amigos da faculdade.

À minha amiga Laís, que muito me motivou, incentivou e compreendeu. Gratidão eterna pela grande amizade, sorrisos, carinho e pelo coração generoso que carrega dentro de si que me motiva a enxergar tudo de uma maneira melhor. Suas contribuições são imensuráveis.

À Carol, que me deu sempre muito apoio, compreensão e me manteve de cabeça erguida quando achei que não seria possível. Agradeço por sempre ter acreditado em mim e por todo carinho. Obrigado também por todas correções na escrita deste documento.

Aos meus demais amigos, Lucas, Erick, Greg, Jorge, Douglas, Igor, Gabriela, Jéssica, Thaís, Vitória, Dani, Drika e Mari (sem ordem de importância), que de alguma forma puderam elevar meus ânimos, o que contribuiu muito para a conclusão deste trabalho. A amizade de vocês é única e indispensável na minha vida.

Gratidão a todos que influenciaram diretamente ou indiretamente a realização deste trabalho, em especial a família e amigos: as grandes etapas da vida se tornam muito mais fáceis quando estamos cercados de pessoas queridas, não há felicidade maior no mundo do que poder compartilhar momentos bons e ruins com vocês. Muito obrigado.

João Vitor de Melo Peixoto

RESUMO

Para que sistemas robóticos possam executar tarefas complexas e autônomas em ambientes sujeitos a eventos imprevisíveis e, ainda assim, obter os resultados mínimos esperados, é fundamental que o modelo de arquitetura de controle de robô seja autoadaptativo. Um robô autoadaptativo é capaz de identificar as alterações nas características do ambiente e se adaptar-se a elas, de modo a garantir a integridade do sistema em tempo-real.

Nesse contexto, esse trabalho apresenta um estudo sobre a aplicação de uma arquitetura de software autoadaptativa na robótica móvel, em um estudo de caso no robô Pioneer. O conceito de autoadaptatividade será aplicado na tarefa de deslocar-se por um extenso corredor em diferentes períodos do dia e, visão computacional e outros sensores, identificar as portas no corredor e classificá-las como abertas ou fechadas.

Palavras Chave: Sistema Autoadaptativo, Robótica Móvel, Visão Computacional, Robô Pioneer, ROS

ABSTRACT

So robotic systems can perform complex and autonomous tasks in environments subject to unpredictable events and still achieve the minimum expected results, it is fundamental that the robot control architecture model be self-adaptive. A self-adapting system for robotics is able to identify and adapt to changes in environmental characteristics to ensure system integrity in real-time.

In this context, this paper presents a study on the application of a self-adaptive software architecture in mobile robotics, in a case study in the Pioneer robot. The concept of self-adaptation will be applied to the task of moving through an extensive corridor at different times of the day, and computer vision and other sensors, identifying doors in the corridor and classifying them as open or closed.

Keywords: Self-adaptive System, Mobile Robotics, Computer Vision, Pioneer Robot, ROS

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Latinos

v_1	Velocidade linear inferior	$[m/s]$
v_2	Velocidade linear superior	$[m/s]$
D_f	Distância frontal	$[m]$
D_b	Distância traseira	$[m]$
d_i	Distância inicial	$[m]$
d_f	Distância final	$[m]$
Δd	Diferença entre distância final e inicial	$[m]$
D_{med}	Valor médio das leituras de distância na lateral	$[m]$
d_{min}	Distância mínima na lateral	$[m]$
F	Valor médio das leituras de distância à frente	$[m]$
T	Limiar do *Thresholding*	
x_0	Coordenada horizontal do pixel	
y_0	Coordenada vertical do pixel	
X_{min}	Mínimo horizontal da região restrita da imagem	
X_{max}	Máximo horizontal da região restrita da imagem	
Y_{min}	Mínimo vertical da região restrita da imagem	
Y_{max}	Máximo vertical da região restrita da imagem	
c_{min}	Mínimo para contador	
Sn	Sensitividade	
Pr	Precisão	
vp	Verdadeiros positivos	
fp	Falsos positivos	
fn	Falsos negativos	

Siglas

SAA	Sistemas Autoadaptativos
BBC	Controle Baseado em Comportamentos
ROS	Robot Operating System
TI	Tecnologia da Informação
RGB	Red Green Blue
IA	Inteligência Artificial

Capítulo 1

Introdução

1.1 Contextualização

1.1.1 Robótica móvel

Robôs e sistemas autônomos são cada vez mais presentes no cotidiano da humanidade, desde simples tarefas cotidianas como lavar louça, acender luzes, acionar alarmes, até atividades voltadas para o desenvolvimento científico, como robôs espaciais, ou voltadas para a indústria a fim de otimizar as linhas de produção. Formalmente falando, um sistema robótico é uma combinação de elementos de hardware e software como duas camadas distintas e integradas para alcançar um objetivo no espaço físico[1].

A robótica móvel autônoma, ramo que será tratado neste trabalho, define-se por um sistema autônomo com um certo grau de liberdade capaz de perceber características do mundo físico e, principalmente, se locomover nele. O desenvolvimento de sistemas robóticos capazes de se deslocar e tomar decisões no ambiente requer o entendimento de uma série de conceitos, tais como de atuação e percepção, como o tipo de comportamento ou arquitetura de controle – podendo ser reativa, deliberativa, híbrida ou baseada em comportamentos. Para se estabelecer um comportamento para o sistema robótico, devem ser considerados alguns fatores como tempo, modularidade e representação, que serão tratados no decorrer deste trabalho.

Além disso, robôs móveis devem possuir uma unidade de controle, que é o elemento responsável pelo gerenciamento e monitoramento dos parâmetros operacionais requeridos para realizar as tarefas do robô. Os comandos de movimentação enviados aos atuadores são originados de controladores de movimento e baseados em informações obtidas pelos sensores. Dessa forma, é bastante comum que sistemas robóticos tenham sua arquitetura de controle implementada em camadas, onde cada camada encapsula uma responsabilidade específica do sistema inserido em um contexto, seja a nível de software ou de hardware, e a união dessas camadas e as interações ocorrentes entre elas é o que compõe a arquitetura de controle do robô que o fará executar a tarefa desejada.

Percebe-se o rumo da complexidade que o desenvolvimento de um sistema robótico pode tomar, uma vez que uma grande quantidade de parâmetros devem ser considerados, parte deles podem ser

previamente ajustados e outra parte apenas calculada ou inferida em tempo de execução. Robôs devem ser projetados não apenas para serem eficientes, mas também de forma a assegurar sua própria segurança e a segurança do ambiente evitando situações de risco. Uma série de incertezas provocadas pelo ambiente e pelo próprio comportamento interno do robô devem ser consideradas, o que faz surgir a necessidade de um sistema complexo e robusto capaz de lidar com a dinamicidade do sistema interno e externo.

1.1.2 Sistemas autoadaptativos

Com o crescimento contínuo de sistemas tecnológicos, um dos principais obstáculos para futuros progressos na indústria de TI será a complexidade dos softwares. Alguns irão ter milhões de linhas de código bastante complexos, tão complexos que requerem profissionais na área com especialidades bem apuradas capazes de fazer instalações, configurações e manutenções [2]. A dificuldade existente de se administrar os sistemas computacionais de altos níveis de complexidade faz surgir a necessidade de novas tecnologias que tornam os sistemas menos dependentes de agentes humanos e capazes de se autoadministrar.

A solução tecnológica que surge como opção para estes conflitos é a computação autônoma,, que consiste em sistemas computacionais capazes de realizar, de forma autônoma, tarefas como adaptar, reconfigurar, otimizar e executar manutenção de seus próprios elementos de software frente à tarefas enigmáticas impostas pelos administradores. O conceito de computação autônoma define uma hierarquia de sistemas autônomos naturais, muitos dos quais consistem em uma infinidade de componentes autoadaptativos que, por sua vez, compreendem um grande número de componentes autônomos e interativos[2]. Uma analogia interessante a se fazer é que os componentes autônomos são como células biológicas: cada uma desempenha uma função específica, mas comunicam entre si sendo capazes de se autogerenciar em situações específicas em prol do sistema biológico. Define-se então que a essência da computação autônoma é o autogerenciamento.

Dessa forma, pode-se chegar a conclusão que sistemas autoadaptativos (SAA) são sistemas que podem ser gerenciados e reconfigurados por eles mesmos, nos quais tanto seus parâmetros quanto sua estrutura se adaptam a situações específicas em tempo-real[2]. O paradigma SAA aplica-se nos mais diversos ramos da computação e tecnologias, entre eles, a robótica.

1.2 Definição do problema

Para o desenvolvimento de um projeto ou aplicação em robótica móvel, onde está presente uma série de componentes internos de software que interagem e alternam entre si, diversos fatores devem ser considerados quanto ao comportamento do robô frente à tarefa que deve ser desempenhada. É possível perceber a complexidade que o software de um sistema robótico deve ter para que seja capaz de controlar eficientemente tais componentes internos de forma a realizar a tarefa final com o desempenho esperado, tentando reduzir ou evitar riscos de falhas durante a sua execução.

Se faz necessária a busca por uma arquitetura de controle para robótica móvel capaz de se

autoadministrar, se autoadaptar em situações específicas, considerando a série de incertezas e características imprevisíveis do ambiente interno e externo. Dessa forma, o conceito de autoadaptatividade surge como uma solução de estrutura interna reconfigurável para administrar questões resultantes da alta complexidade que ocasionam difícil manutenibilidade dos sistemas.

1.3 Objetivo

Contextualizado o problema relacionado à robótica móvel atuando em ambientes dinâmicos e as finalidades de um sistema autoadaptativo, o objetivo deste trabalho é desenvolver o sistema robô que seja capaz de desempenhar uma tarefa definida implementando uma arquitetura de software autoadaptativa. O sistema autoadaptativo do robô o tornará mais robusto e flexível frente às incertezas do ambiente que só são conhecidas em tempo de execução. A implementação será feita em um estudo de caso com o robô Pioneer e o framework ROS – Robot Operating System. A tarefa programada consiste em fazer o robô navegar em um corredor no prédio de Ciência da Computação, da Universidade de Brasília, identificando as portas nele presentes e classificando-as como abertas ou fechadas. Por fim, este trabalho visa apresentar os resultados da implementação e mostrar as vantagens de se utilizar um sistema capaz de se autoadaptar e se otimizar em tempo de execução. São requeridos conhecimentos acerca de arquitetura de software para robótica, sistemas autoadaptativos, navegação e utilização de sensores em robótica, programação do robô em ambiente ROS e visão computacional.

1.4 Apresentação do manuscrito

Após este capítulo introdutório, este documento irá apresentar no Capítulo 2 todo conteúdo utilizado como embasamento teórico, relacionados principalmente à robótica móvel, autoadaptatividade, arquitetura de controle para robótica, e visão computacional. Em seguida, o Capítulo 3 relatará o desenvolvimento do trabalho, que inclui desde a elaboração da metodologia a partir dos fundamentos teóricos apresentados até a implementação dessa metodologia e de todas as etapas que compõem sistema, para a futura execução da tarefa. Após o sistema autoadaptativo ser implementado e as melhorias serem realizadas, os resultados obtidos serão apresentados no Capítulo 4, com análises e avaliações das metodologias propostas. Por fim, o Capítulo 5 apresenta as conclusões obtidas do trabalho como um todo, dificuldades encontradas, êxito obtido e sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo documento é dedicada à apresentação de todo referencial teórico utilizado para o trabalho, abordando todos os conceitos e ferramentas necessárias que tornam possível seu desenvolvimento. Dentre as referências, encontram-se artigos científicos, livros e páginas da web.

2.1 Robótica Móvel

Um sistema robótico pode ser definido como a combinação de componentes de hardware e software que, integrados, compõem um sistema autônomo que atua no espaço físico, movimentando-se e podendo percebê-lo, tomando decisões para atingir um ou mais objetivos. A robótica possui gama de aplicações, como industrial, bélica, médica, monitoramento exploração, educativa, busca e resgate, podendo atuar nos mais diversos tipos de ambientes, como os aquáticos, terrestres – podendo ser desde domésticos ou industriais, até terrenos montanhosos e inconstantes, – ou aéreos – como atmosfera terrestre ou no espaço.

Sistema autônomo não necessariamente implica em um sistema inteligente. Robôs podem ser programados de forma a desempenhar uma tarefa específica e repetitiva, sem ter de considerar fatores externos imprevisíveis. É o caso de robôs manipuladores para a indústria, onde, por exemplo, um braço robótico tem uma tarefa repetitiva de agarrar um objeto e movê-lo para outro lugar, ou de colocar tampinhas em tubos de pasta de dente. Por outro lado, um robô pode ser projetado para atuar em um ambiente em que podem ocorrer eventos previstos pelo criador ou eventos aleatórios, e o robô deverá percebê-los e saber como agir a uma determinada situação sem a intervenção humana, ou seja, requer um certo nível de inteligência. Esse é o caso da robótica móvel.

Existe uma série de fatores que classificam sistemas robóticos com mobilidade, que podem se diferir quanto à:

1. Atuação: atuadores são elementos da parte de hardware que dão movimento ao sistema, compondo os graus de liberdade do robô. Elementos atuadores podem ser motores elétricos,

hidráulicos ou pneumáticos.

2. Percepção: elementos de percepção são chamados de sensores, que são dispositivos que proveem a capacidade de perceber no ambiente características como luminosidade, temperatura, se há um objeto em algum lugar e a que distância está (2.1.1.2). Ou seja, sentidos que geralmente são atribuídos a seres vivos, como tato, audição e visão.

3. Graus de liberdade: grau de liberdade é definido como 1 (um) sentido de movimento linear ou rotacional de uma das juntas do robô no plano bidimensional ou tridimensional. O somatório de todos os movimentos possíveis de todas as juntas do sistema é definido como o número de graus de liberdade.

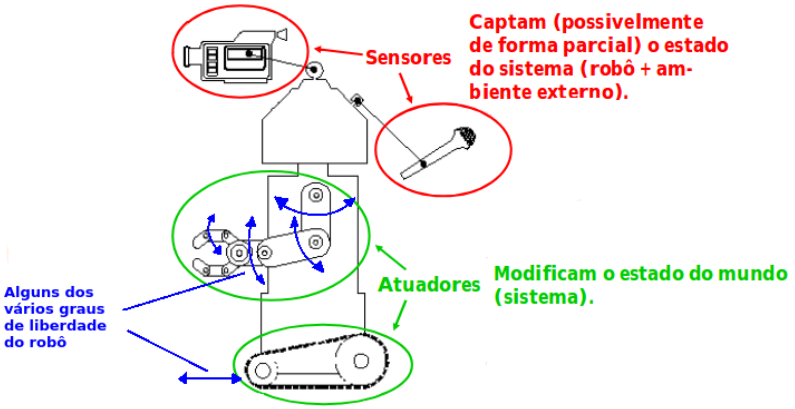


Figura 2.1: Exemplo ilustrativo de robô com atuadores, sensores e vários graus de liberdade. Fonte: o autor.

4. Tipo de controle; o tipo de controle define como o sistema irá se comportar e agir no meio, e se classifica em reativa, deliberativa, híbrida ou baseada em comportamento (esses conceitos serão explicados mais à frente nesta SubSeção).

5. Mapeamento do ambiente; um sistema robótico móvel por ser projetado para atuar em um ambiente previamente mapeado, de forma que ele tenha conhecimento sobre a estrutura do local para poder se deslocar. Por outro lado, pode também estar situado em um ambiente desconhecido, então, dessa forma, o robô poderá mapeá-lo conforme se desloca por ele. Note que essa classificação só é possível para ambientes estáticos, caso contrário, não é possível prever e mapear a estrutura do ambiente, ou no mínimo é inviável.

6. Localização; define-se pela capacidade do robô de ter conhecimento de sua posição em relação ao meio, dado um sistema de coordenadas bidimensional ou tridimensional. Pode ser classificada em absoluta – dado um ambiente conhecido, o robô saberá sua localização em relação à origem do sistema de coordenadas a qualquer instante de tempo de execução – ou relativa – quando o sistema não possui um referencial previamente estabelecido, então, dessa forma, o robô irá considerar sua posição no primeiro instante de tempo de execução como a origem do sistema de coordenadas que terá como referência. Um dos métodos mais utilizados para estimar a posição do robô é utilizando encoders como sensores de posição.

2.1.1 Sensoriamento

Sensores são dispositivos que utilizam princípios físicos e químicos na tentativa de simular os sentidos humanos. Não são utilizados em operações pré-programadas, onde um robô é projetado para realizar tarefas repetitivas por meio de um conjunto de funções programadas, o que predomina no caso de robôs industriais. No entanto, a utilização de sensores agrega ao robô uma inteligência que o permite interagir com o meio, no caso de sensores externos, enquanto os internos fornecem informações sobre seus próprios parâmetros internos[3]. As seções a seguir abordam sobre os principais sensores que serão utilizados para a realização deste trabalho.

2.1.1.1 Sensor de posição: Encoder

Em todos os ramos da robótica é bastante comum que se deseje ter o conhecimento e o controle sobre o posicionamento dos elementos mecânicos, seja um braço robótico, um pistão, ou as rodas. Para essa finalidade existem os encoders, dispositivos eletromecânicos utilizados para o servoposicionamento e estimativa de posição de um eixo a partir de um referencial conhecido. Acoplados aos eixos de movimento, os encoders medem a quantidade e o sentido das rotações de um disco e convertem a informação em deslocamento linear ou angular realizado pelo atuador. Tendo o conhecimento de um referencial inicial em um sistema de coordenadas e os deslocamentos dos eixos, é possível calcular a nova posição de um braço robótico industrial ou o novo local onde robô móvel se encontra no ambiente.

Apesar de serem bastante precisos, é necessário lembrar que sempre existirá uma pequena taxa de erro gerada pela imprecisão do dispositivo real. Dessa forma, quanto mais o robô se deslocar, maior será o erro de deslocamento acumulado durante sua jornada.

2.1.1.2 Sensores de Distância: Laser e Sonar

ara grande parte das aplicações em robótica móvel é fundamental que o robô seja capaz de perceber os objetos no ambiente e a que distância estão, para que possa, por exemplo navegar em um ambiente, evitar uma colisão, identificar o perfil de um objeto, etc. Esta é a finalidade de sensores de distância como o laser e o sonar, dispositivos que tem como princípio a emissão de um sinal a partir de um ponto inicial e uma direção conhecida, onde este sinal vai refletir (ou não) em algum objeto, e então retornar para o receptor. Os sensores de distância a laser e ultrassônico, disponíveis para este projeto, possuem diferenças quanto a técnica utilizada, alcance e resolução.

O sensor laser mede propriedades de luz refletida ou absorvida para obter distâncias entre robô e obstáculos ao seu redor, baseando-se na concentração do feixe de luz refletido a distância pretendida. O laser geralmente possui longo alcance, com um vasto campo de cobertura e alta resolução, isto é, a quantidade de feixes de luz emitidos dentro de um setor circular.

Já o sonar utiliza propriedades de sinais ultrassônicos e tem o mesmo princípio de funcionamento do ouvido humano quando escuta eco: uma onda de ultrassom é emitida a uma certa frequência e capturada após sua reflexão, e dessa informação extrai-se distâncias em uma área coberta. Sensores

ultrassom geralmente possuem baixo alcance e não são tão precisos quanto sensores laser, além de que o sinal sonoro pode ser perdido após a reflexão em uma superfície não perpendicular a direção do feixe. O sonar pode ser observado na Figura 2.2.

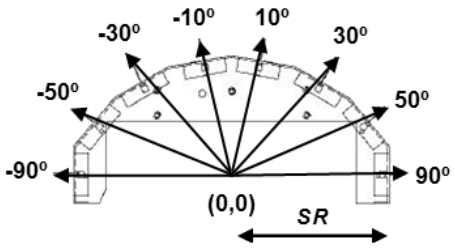


Figura 2.2: Sensores de ultrassom do robô Pioneer emitindo sinais em diferentes direções. Um sensor a laser tem configuração semelhante. Fonte: [4]

2.1.1.3 Sensor Câmera

Câmeras são os sensores que tentam simular a visão humana: a luz refletida pelos objetos do ambiente são captadas por elementos foto-sensitivos do sensor e, após um processamento, gera-se um *frame* de imagem, que é na realidade uma matriz de *pixels*. Determinando-se uma taxa de aquisição de frames por segundo, o robô consegue adquirir as imagens em tempo real do que acontece no ambiente. Como o sistema do robô (logicamente) não sabe por si só analisar uma imagem e identificar Figuras e eventos que são óbvios para nós, seres humanos, cabe ao programador utilizar técnicas de processamento de imagem para extrair as informações relevantes a partir da distribuição dos pixels e seus valores RGB. Estes procedimentos são a essência básica de uma área de estudo da robótica chamada de visão computacional, que será abordada posteriormente na Seção 2.6.

2.1.2 Paradigmas de Controle na Robótica Móvel

Apenas reunir um conjunto de regras e programas não é suficiente para dizer que está definido o comportamento do sistema robótico. Existem inúmeras formas de encontrar uma maneira correta, eficiente e ótima de controlar um robô para uma determinada tarefa. Para isso, arquiteturas comportamentais proveem princípios de orientação e restrições que organizam o sistema de controle do robô, e ajudam ao designer projetar o robô de forma a produzir o comportamento global de saída desejado, sendo importante para realizar tarefas em ambientes não triviais e talvez imprevisíveis. Dessa forma, os principais paradigmas são definidos por[1]:

1. Controle Deliberativo
2. Controle Reativo
3. Controle Híbrido
4. Controle Baseado em Comportamentos

O paradigma de **controle deliberativo** pode ser entendido como aquele que requer um planejamento, onde faz-se a utilização de mapas virtuais – que serão construídos em tempo de execução ou previamente conhecidos – e o planejamento de ações e trajetórias. São sistemas do tipo *Sense, Plan and Act* (SPA), que consiste em um modelo centralizado que coleta informações usando sensores, cria um modelo do ambiente, planeja o próximo movimento e executa a ação[5]. Dependendo do nível de complexidade da aplicação, o controle deliberativo pode ser muito custoso e lento. Aplicações que utilizam planejamento são vantajosas por serem "conscientes", ou seja, o sistema sabe como deverá agir para cada situação, já que trajetórias estão previamente definidas devido ao conhecimento do mapa estático. No entanto, as desvantagens desta abordagem são o custo computacional e por não serem indicadas para ambientes dinâmicos, devido às necessidades de atualizações e replanejamento.

Um dos paradigmas mais utilizados para controles de robôs é o de **controle reativo**. Sistemas puramente reativos não necessitam de representações internas do ambiente e não fazem um planejamento de sua atividade baseado nestes mapas virtuais, mas sim se baseiam em reações à eventos detectados pelos sensores. Ou seja, o controle reativo é utilizado para aplicações que dependem dos sensores para captar os eventos que ocorrem no ambiente e então reagir à eles, como um reflexo, definido pelo programador[1]. Estes sistemas são geralmente decompostos em módulos unitários *sense-react*, cada um responsável por reagir à um evento específico, podendo estar sendo executados paralelamente para compor uma tarefa global. É utilizado, por exemplo, em aplicações de navegação evitando colisões e obstáculos, deslocamento acompanhando uma parede ou corredor, seguir na direção de um objeto identificado pelos sensores, etc. Esse paradigma surgiu a partir das necessidades deixadas pelo controle deliberativo, e costuma ser um conceito utilizado em outros tipos de controle, como o híbrido e o baseado em comportamentos.

No entanto, os sistemas puramente reativos não tinham capacidade de planejamento ou memorização, fazendo com que o robô não fosse capaz de planejar suas trajetória, construir mapas, monitorar seu desempenho, entre outros. A busca por aplicações que aproveitassem as vantagens dos paradigmas deliberativo e reativo deu origem a um novo tipo de controle, a arquitetura de **controle híbrido**, que tenta combinar a capacidade de (re)planejamento e uso de representação virtuais dos ambientes, com o objetivo de prover robustez, resposta em tempo real e flexibilidade dos sistemas puramente reativos. Arquiteturas híbridas podem permitir a reconfiguração de controles reativos baseado no conhecimento do mundo através da sua capacidade de raciocinar, assim como os modelos de mundo podem ser atualizados de acordo com mudanças no ambiente detectadas pela unidade reativa[5]. Os sistemas de controle híbrido são estruturados em níveis de camadas, onde uma camada de baixo nível está relacionada à um controle reativo, enquanto o controle deliberativo, de planejamento, atua em uma camada de alto nível, podendo existir uma ou mais camadas intermediárias entre elas. Esse modelo de organização possibilitou um comportamento consistente e robusto para executar tarefas mais complexas, no entanto, requerem meios complexos de interação entre as camadas, elevando consideravelmente o custo de implementação

em muitos casos.

Por fim, o paradigma de **Controle Baseado em Comportamentos** *(Behavior-Based Control, BBC)* surgiu como uma evolução dos sistemas reativos, que eram inflexíveis, inadaptáveis e incapazes de aprender. Sistemas BBC possuem componentes reativas, como os híbridos possuem, mas não possuem componentes deliberativas. Essencialmente, esse tipo de controle utiliza "comportamentos" como módulos de controle[1]. Comportamentos podem ser definidos como componentes que possuem uma função específica, como por exemplo "seguir paralelo à parede", "rastrear um objeto", "construir mapa", diferentemente dos paradigmas anteriores que decompunham em níveis ainda mais baixos. Os módulos de comportamento recebem como entrada dados dos sensores, regras de reação, ou informações de outros comportamentos, e podem ter como saída comandos para atuadores e informações para outras unidades do mesmo tipo, criando uma rede de comportamentos que interagem entre si. Estas unidades podem ser executadas concorrentemente, em paralelo, ou sequencialmente. Para poder organizar as chamadas aos comportamentos visando conseguir garantir eficiência, é importante que em alguns casos se tenha um agente interno do sistema capaz de selecionar um único ou vários módulos que vão ser colocados em execução em um determinado momento.

2.2 Robô Pioneer

Para a realização deste trabalho é fundamental implementar toda a teoria em uma aplicação real. Os estudos e experimentos acerca de arquitetura autoadaptativa em robótica móvel será implementada no robô Pioneer 3-AT, esta breve seção relata algumas características relevantes do robô.

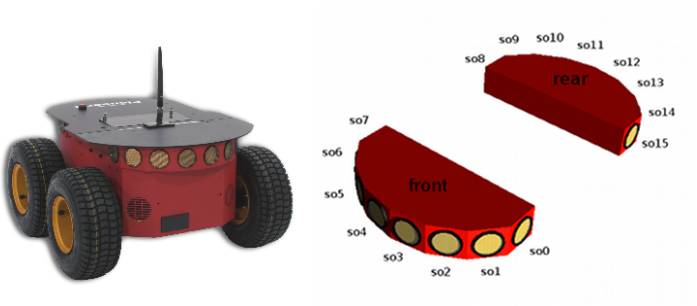


Figura 2.3: Robô móvel Pioneer 3-AT e sensores sonar embarcados. Fonte: `https://cacic-robot.readthedocs.io/en/latest/index.html`

O Pioneer 3-AT é um dos robôs da família Pioneer, possui 4 rodas tracionadas, corpo de alumínio e pode ser operado em diversos tipos de terreno. Possui computador de bordo, software de navegação autônoma, vem de fábrica com dezesseis sensores sonar dispostos de acordo com a Figura 2.3, encoders como sensores de posição, entre outros. Possibilita a instalação de dispositivos

de hardware (teclado, monitor, *joystick*) e de outros sensores (câmera, laser, sonar), além de possuir entradas para comunicação Ethernet e outras. Projetado para pesquisa, este robô móvel é reprogramável e versátil, podendo ser utilizado para uma infinidade de aplicações. O exemplar utilizado[4] pertence ao Departamento de Ciência da Computação da Universidade de Brasília, e tem como framework operacional o *ROS*, abordado na Seção 2.3.

Informações técnicas sobre dispositivos sensores e câmera instaladas no robô podem ser vistas em [4].

2.3 ROS – Robot Operating System

O ROS é um sistema meta-operacional de código aberto para coordenar robôs[6]. Fornece os serviços esperados de um sistema operacional, incluindo abstração de hardware, controle de dispositivo de baixo nível, passagem de mensagens entre processos e gerenciamento de pacotes. Ele também fornece ferramentas e bibliotecas para obter, criar, gravar e executar código em vários computadores.

A arquitetura em tempo de execução do ROS pode ser entendida como uma rede peer-to-peer de processos que são acoplados usando a infraestrutura de comunicação do próprio ROS. Esse framework implementa vários estilos diferentes de comunicação, incluindo comunicação síncrona sobre serviços, fluxo assíncrono de dados de tópicos e armazenamento de dados em um servidor.

O ROS é uma estrutura distribuída de processos (também conhecidos como Nós) que permite que os executáveis sejam individualmente projetados e interativos em tempo de execução. Esses processos podem ser agrupados em Pacotes e Pilhas, que podem ser facilmente compartilhados e distribuídos. O ROS também suporta um sistema federado de Repositórios de códigos que também permite a colaboração ser distribuída.

Para melhor conhecimento da arquitetura ROS, é necessário compreender alguns dos elementos fundamentais do modelo estruturado:

- **Pacote (*package*):** os pacotes são a unidade principal para organizar o software no ROS. Um pacote pode conter processos (nós) a tempo de execução do ROS, bibliotecas, conjuntos de dados, arquivos de configuração ou qualquer outra coisa que seja organizada de maneira útil.

- **Nó (*node*):** nós são processos que compõem o sistema computacional. Por exemplo, um nó controla um localizador de faixa de laser, um nó controla os motores de roda, um nó realiza a localização, um nó executa o planejamento de caminho, um nó fornece uma visão gráfica do sistema e assim por diante. Um nó ROS pode ser programado utilizando bibliotecas como roscpp ou rospy.

- **Mestre (*master*):** o mestre é responsável por gerenciar e encontrar nós quando são requisi-

tados. O primeiro passo para se executar um programa que utilize ROS é iniciar o nó mestre, que representa a rede de conexão na qual os outros nós irão se comunicar. Sem o mestre, os nós não poderiam se encontrar, trocar mensagens ou chamar serviços.

- **Mensagem (*message*):** os nós comunicam entre si através da troca de mensagens. A mensagem é uma estrutura de dados compreendendo campos que possuem tipos diferentes de dados, como inteiros, ponto flutuante, booleanos, vetores, structs (linguagem C).

- **Tópico (*Topic*):** as mensagens são roteadas por meio de um sistema de transporte com semântica de publicação/assinatura. A troca de mensagens entre os nós acontecem por meios dos tópicos, que são como canais próprios para se trocar um único tipo de mensagem. Primeiro, um nó publica no tópico (*publisher node*) a mensagem deixando-a acessível para outros nós, que irão assinar o mesmo tópico (*subscriber node*) para adquiri-la. Podem haver vários publishers e subscribers simultâneos para um único tópico, e um único nó pode publicar e / ou assinar vários tópicos.

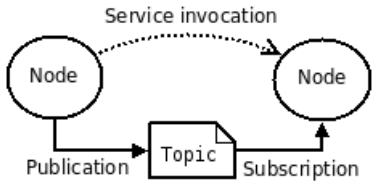


Figura 2.4: Nós do ROS publicando e assinando em um tópico de mensagens. Fonte: `http://library.isr.ist.utl.pt/docs/roswiki/New(2f)Concepts.html`

2.4 Sistemas Autoadaptativos

Como foi contextualizado na Seção 1.1.2, o conceito de autoadaptatividade surge em meio ao cenário de necessidade de aprimoramento dos sistemas de software devido ao constante aumento da dificuldade de se gerenciar esses sistemas, que ficam cada vez mais complexos. Sistemas computacionais autoadaptativos, como diz o próprio nome, são sistemas capazes de adaptar a si próprios – isso significa configurar-se, otimizar-se, manutenir-se – para realizar eficientemente as tarefas impostas pelos desenvolvedores. Em outras palavras, são sistemas que podem ser desenvolvidos e aprendidos por eles mesmos e nos quais tanto seus parâmetros quanto sua estrutura se adaptam a situações específicas durante sua execução.

Um dos grandes desafios atuais, por exemplo, no campo da IA (Inteligência Artificial) é o desenvolvimento de métodos, algoritmos e implementações de sistemas com alto nível de flexibilidade e autonomia. Mas para que isso seja alcançado, esses sistemas devem ser capazes de desenvolver sua estrutura e seu conhecimento no ambiente em que se encontram, desenvolvendo assim sua inteligência. Para lidar com problemas de controle, previsão, classificação e processamento de dados em sistemas localizados em ambientes com constantes mudanças, o sistema utilizado deve ser

capaz de modificar e evoluir tanto sua estrutura quanto os parâmetros que a compõem, isto é, autoadaptar-se. A autoadaptatividade é um tema que pode ser desmembrado em alguns conceitos, dentre eles, autoconfiguração, autocura, autoproteção e auto-otimização[2].

- **Autoconfiguração:** é o processo em que os nós recém-implantados são configurados por procedimentos de instalação automática de modo a obter a configuração básica necessária para cooperar com as demais partes do sistema. Esse conceito surge como solução para a dificuldade que se tem de instalar, configurar e integrar novas componentes ou atualizações em sistemas complexos, que costuma consumir muito tempo e ser propensa a erros.
- **Autocura:** detectar a origem de falhas e *bugs* no sistema, diagnosticá-las e repará-las são as responsabilidades do processo de autocura de sistemas autoadaptativos. Quando uma falha é detectada no sistema, é fundamental que ele tenha conhecimentos sobre suas próprias configurações e de informações em relatórios anteriores, como arquivos de *log*, que tornam possível que o diagnóstico correto seja feito.
- **Autoproteção:** define-se como a capacidade do sistema de detectar em tempo de execução e bloquear ataques de invasores e malwares, aproveitando as informações de dentro do software. Sistemas autoadaptativos irão defender sua integridade contra ataques maliciosos e falhas que remanescem após a autocura.
- **Auto-otimização:** Sistemas autonômicos tendem a continuamente buscarem maneiras de melhorar sua operação, identificando e dimensionando oportunidades para se tornarem mais eficientes em custo, tempo e performance. A auto-otimização trata justamente desse aspecto, é a capacidade do sistema autoadaptativo de identificar parâmetros, funções ou unidades modulares que iriam melhor contribuir para a performance do sistema, decidindo se deve manter a condição de operação corrente ou modificá-la, de forma a buscar constantemente otimizar a realização de uma tarefa.

2.5 Arquitetura de Controle Autoadaptativa para Sistemas Robóticos

É comum que as aplicações em robótica atualmente busquem pela máxima eficiência, isto é, melhor utilização de recursos e de tempo para realizar tarefas. Muitos desses sistemas robóticos estão inseridos em ambientes dinâmicos e possuem tarefas que requerem diferentes metodologias para que possam eficientemente atuar nesses ambientes, e isso significa que é importante que o sistema de gerenciamento dos parâmetros, como dados coletados dos sensores, componentes de software e controle de atuadores seja feito em tempo real, seguindo políticas de adaptação[7]. A dificuldade existente de se suspender a execução para modificações e atualizações faz surgir a necessidade de softwares que sejam capazes de prover adaptações em tempo de execução.

Nesse contexto, com o intuito de buscar soluções para os conflitos descritos, uma equipe de cientistas e engenheiros da americanos realizou um estudo[8] e observou-se o surgimento recente de vários frameworks com arquiteturas adaptativas e com modelos de projeto semelhantes. Após algumas análises, reuniram essas semelhanças e definiram características dos estilos de arquitetura de sistemas robóticos, propondo uma nova arquitetura resultante. Os principais tópicos que levaram ao surgimento dessa nova arquitetura foram:

- Usualmente, sistemas robóticos tem sua arquitetura estruturada em componentes independentes e interativos, responsáveis exclusivamente por uma função específica do todo e que trocam informações com os demais. Assim, em uma aplicação de robótica, as operações de adaptação podem atuar especificamente em cima de um ou mais elementos, o que garante que o resto do sistema permaneça intocado e que, ainda assim, receberá desse os dados esperados desses elementos adaptados.

- A partir disso, é possível se ter uma arquitetura organizada em camadas onde estabelecem-se dois níveis:

  - **Camada de Aplicação**: onde estão implementados os elementos do sistema robótico (citados no tópico anterior) que, juntos, desempenham a tarefa principal. É desta camada que são extraídas as informações necessárias para se caracterizar a necessidade de adaptação.
  - **Camada de Adaptação (ou Camada de Controle)**: nesta camada é onde está implementada a estrutura do sistema que provê a capacidade de realizar adaptações nos componentes da Camada de Aplicação. Este nível também é estruturado em componentes menores, responsáveis por instanciar, configurar, monitorar e implantar componentes na outra camada de acordo com políticas e objetivos específicos da tarefa global.

- Por fim, sabe-se que a maioria dos sistemas robóticos possuem componentes de controle reativo (ver 2.1.2), esta natureza sugere: detectar o ambiente, calcular planos de resposta e executar ações de controle apropriada. Pensando nisso, os pesquisadores propuseram que os elementos da Camada de Adaptação, responsáveis pela adaptação do sistema, pudessem estar organizados em três subelementos sequenciais: coletor, analisador e administrador. A Figura 2.5 mostra um modelo final da arquitetura resultante.

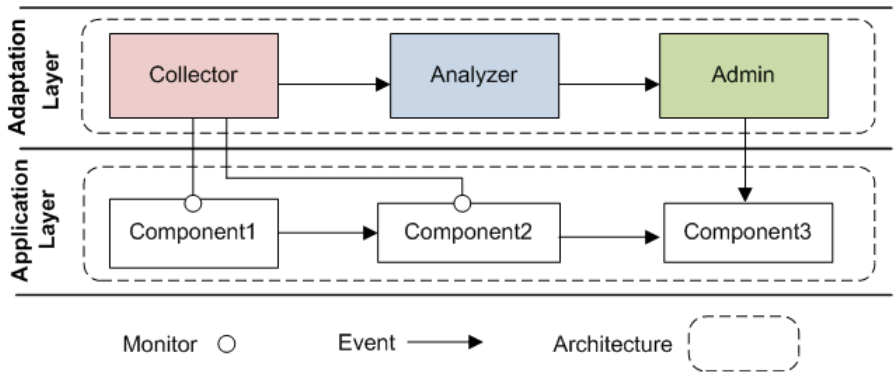


Figura 2.5: Modelo da arquitetura autoadaptativa em camadas proposta. Fonte: `http://seams2009.cs.uvic.ca/talks/20-USC.pdf` (adaptada)

A componente de coleta possui interfaces com as unidades da camada de aplicação que reúnem os dados de interesse relacionados às necessidades do sistema de realizar mudanças, como parâmetros internos dos componentes, dados de sensores, etc. As interfaces funcionam como monitores dos elementos da camada de aplicação, e os dados agregados são transmitidos para o analisador.

A etapa de análise reúne os dados coletados e avalia as políticas de adaptação implementadas, os desenvolvedores definem mecanismos para avaliar as políticas e constroem os planos de adaptação, caso sejam necessários. O conjunto de adaptações necessárias é passado então para a última etapa.

Por fim, o elemento administrador executa os planos de adaptação definidos pelos analisadores. Possuem mecanismos de adaptação capazes de adicionar, remover, conectar e desconectar os componentes da camada de aplicação para prover as adaptações necessárias. Para isso, são invocadas interfaces especializadas providas pelos elementos da camada de aplicação, que então recebem as "ordens" da Camada de Adaptação/Controle de forma que a adaptação identificada pelo analisador seja feita corretamente.

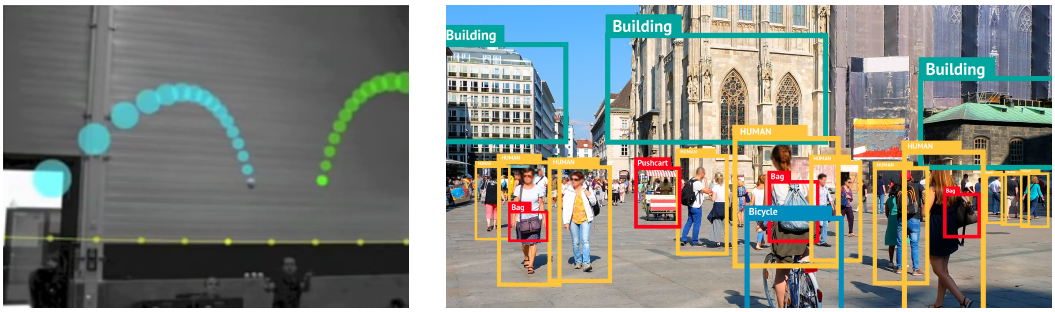
Talvez uma das maiores dificuldades que podem ser encontradas no projeto seja a identificação das políticas adaptativas da componente de análise da Camada de Controle, citada anteriormente, uma vez que as requisições e a tarefa da aplicação do sistema robótico são conhecidas e deve-se identificar os fatores sujeitos a adaptação[8]. A etapa da administração pode ser relativamente simples, se implementada como um elemento que envia para a camada de aplicação os novos valores dos parâmetros adaptados ou que envia comandos para as componentes dessa mesma camada serem ativados ou desativados, de forma que as novas combinações das componentes ativas componham a adaptação requisitada.

Uma das vantagens de se utilizar esse tipo de arquitetura é que as camadas, seus elementos e as interfaces podem ser implementadas como partes individuais do projeto. Este modelo de arquitetura ajuda a garantir qualidades de serviço de maneira flexível, no contexto de um cenário de robótica móvel que requer autoconsciência e autoadaptação.

2.6 Visão Computacional na Robótica

Reproduzir a visão humana em uma máquina é uma tarefa extremamente complicada. Por exemplo, quando alguém arremessa repentinamente uma bola em nossa direção e reagimos a tempo de pegá-la no ar, para nós é algo simples, pois a visão e o cérebro humano altamente sofisticados permitem que tenhamos capacidade de processar e reagir em frações de segundo com facilidade. No entanto, replicar essa tarefa em um robô é algo bastante complicado pois, no mesmo exemplo, requer que a máquina seja capaz de adquirir imagens do ambiente, reconhecer que existe uma bola na imagem e que ela está se aproximando, detectar situação de possível colisão, calcular o caminho aproximado que a bola fará, e o sistema irá utilizar seus mecanismos para agarrar a bola, tudo isso em um breve espaço de tempo, enquanto a bola está no ar. Percebe-se então a dificuldade existente em analisar simples imagens e extrair uma série de informações nada simples para um robô.

A visão computacional na robótica é uma Inteligência Artificial que permite a replicação da visão humana usando hardware e software para que o robô seja capaz de interagir com o mundo físico[9]. É um ramo que estuda como o sistema robótico irá reconstruir e compreender uma cena capturada tridimensional do mundo real, a partir de imagens 2D e converter as informações em comandos para os atuadores do robô e para seu software interno. A partir dos valores e distribuição dos pixels na imagem e utilizando-se uma base de dados, é possível que o software reconheça características na imagem que seres humanos são capazes de reconhecer, como identificar objetos, pessoas, padrões, formas, cálculos de áreas e distâncias, identificação de cores e muito mais são apenas algumas das funcionalidades mais básicas da visão computacional. Com essas técnicas é possível que, por exemplo, um robô rastreador identifique seu alvo nas imagens e calcule os comandos necessários para que os atuadores o coloquem em rota de perseguição.



(a) (b)

Figura 2.6: Exemplos de aplicações na visão computacional. Em 2.6(a), o software calcula a trajetória de objetos lançados no espaço. Já em 2.6(b), o software utiliza algum algoritmo de detecção e demarcação de objetos na imagem. Fonte: `https://bitmovin.com/object-detection/`

Para dar suporte à visão computacional em termos de software e programação, existem bibliotecas que disponibilizam uma série de ferramentas e técnicas para os mais variados tipos de aplicações em processamento de imagens. A popular biblioteca de código aberto *OpenCV* é a

que será utilizada neste trabalho para a parte de reconhecimento dos objetos desejados, as portas, em imagens adquiridas do corredor e para a classificação dessas portas como abertas ou fechadas. Algumas das ferramentas que podem ser utilizadas são a correlação, correspondência de templates, limiarização, segmentação de imagem, gradientes e detecção de bordas, encontro de contornos, transformações geométricas, entre outros. Existem também outras ferramentas mais avançadas que utilizam técnicas de *deep learning* para o reconhecimento de objetos e padrões desejados em imagens. Abaixo, seguem breves descrições e exemplos de aplicação sobre duas das principais ferramentas de processamento utilizadas para este trabalho.

2.6.1 Thresholding

O ***thresholding*** é uma técnica de segmentação de imagem cujo processo consiste em construir uma imagem binária (geralmente em preto e branco) de saída de acordo com os valores dos pixels da imagem de entrada. No processo, define-se um valor limiar T para o qual os pixels da imagem de entrada serão avaliados de acordo com a equação a seguir:

$$O(x,y) = \begin{cases} 0, \text{ se } f(x,y) \leq T \\ 1, \text{ caso contrário} \end{cases} \tag{2.1}$$

Utiliza-se a técnica *thresholding* em casos onde se deseja colocar em destaque os elementos desejados de uma imagem em relação a um plano de fundo a partir da segmentação em planos, onde a divisão destes dois planos é definida baseando-se na luminosidade dos pixels dos elementos da imagem original[10]. Dessa forma, é necessário que o usuário conheça a faixa de valores dos pixels desejados par que se estabeleça o limiar T correto. É possível também que a imagem de saída não seja binária, mas que possua três ou mais faixa de valor, sendo necessário então definir vários limiares. Para este trabalho, a imagem de saída binária em preto e branco será utilizada, como exemplificado na Figura 2.7.

Figura 2.7: Neste exemplo, definiu-se um valor limiar que binarizasse a imagem de forma a destacar a lua e as nuvens do céu. Nota-se que os pássaros acabam ficando em preto também. Fonte: `https://medium.com/@vad710/cv-for-busy-developers-thresholds-and-templates-15d660694c38`

2.6.2 Detecção de Objetos com Deep Learning: o método da Classificação em Cascata

A detecção de objetos é uma tarefa fundamental da visão computacional, uma vez que, assim como na visão humana, o primeiro passo para se compreender uma imagem é localizar e identificar os objetos e elementos nela presentes. Tecnicamente falando, detecção de objetos é o procedimento de determinar a instância da classe a qual o objeto pertence e estimar a localização do objeto ao demarcar um retângulo na imagem (*bounding box*). Muitos desafios precisam ser solucionados quando se trata de uma detecção eficaz dos objetos, tais como oclusão parcial/total, variação de luminosidades, posicionamento, escala, etc. Assim, torna-se difícil de realizar uma detecção perfeita, o que atraiu muita atenção nos últimos anos[11].

O rápido crescimento do *deep learning* (aprendizagem "profunda") e das redes neurais em meados dos anos 80-90, surgiram ferramentas poderosas capazes de aprender semântica em alto nível que foram também introduzidas para solucionar questões na área da visão computacional. Atualmente, tecnologias de *deep learning* são indispensáveis para os excelentes resultados obtidos na área de processamento de imagens devido à poderosas unidades de processamento gráfico e a grande disponibilidade de conjuntos de dados[12]. Motivados pelos resultados da classificação de imagens, diversos modelos de *deep learning* foram desenvolvidos também para a detecção de objetos, como redes neurais convolucionais (*CNN*), *YOLO* e *Classificação em Cascata*.

A **Classificação em Cascata**, ou *Haar Cascade*, revelante para este trabalho, é um método que utiliza a análise de *Haar features*[13], que são pequenos conjuntos de pixels que caracterizam subseções de uma imagem, isto é, cada pequena característica dos elementos e objetos de uma imagem são na verdade padrões de pixels que provavelmente se repetem para o mesmo objeto que apareça em qualquer imagem, como o canto dos olhos, contorno da boca e as entradas das narinas em uma detecção de faces.

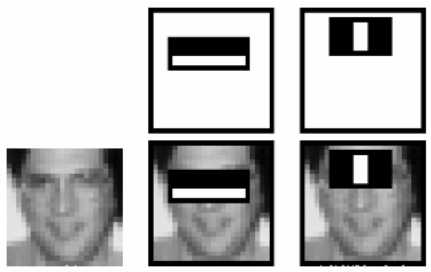


Figura 2.8: Exemplos de Haar features em detecção facial. Fonte: `https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html`

Para treinar um classificador, inicialmente o algoritmo precisa de muitas imagens positivas (imagens de rostos) e negativas (imagens sem rostos) para treinar o classificador, de onde as *features* relevantes serão extraídas. Inicialmente, milhares delas são encontradas, mas então passam por um processo de seleção que mantém apenas as *features* que, juntas, geram a acurácia e taxa de erros requeridas, descartando- então todas as outras *features* restantes.

O classificador em cascata consiste em uma "lista de estágios"dessas *Haar features*, onde o a imagem em que se deseja detectar o objeto obrigatoriamente deve ser aprovada por todos os estágios de *features* do classificador, um a um, ou seja, o objeto avaliado deve conter todos os padrões de pixels do classificador. Avalia-se a imagem por janelas/seções: se uma falhar em um estágio, é descartada e passa-se para a próxima janela. Caso ela passe por todos os estágios mas falhe no último, próxima. A janela que passar por todos os estágios indica que naquela seção da imagem provavelmente existe o objeto buscado. No final, sabe-se se a imagem por completo possui o objeto e em qual região está localizado.

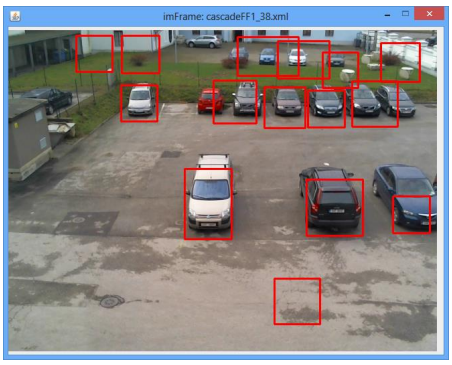


Figura 2.9: Detecção de carros em uma imagem de um estacionamento. Note a existência de falsos positivos e marcações. Fonte: `https://pdfs.semanticscholar.org/0f1e/866c3acb8a10f96b432e86f8a61be5eb6799.pdf`

Logicamente, um classificador está associado a uma taxa de acerto e de erro, podendo gerar falsos positivos e falsos negativos. Essas taxas estão diretamente associadas ao processo de treinamento: a quantidade de imagens positivas e negativas, as variações nos próprios objetos das imagens positivas, e outros parâmetros determinados durante o processo. Ainda assim, este método pode trazer resultados muito bons e satisfatórios.

Capítulo 3

Desenvolvimento

3.1 Introdução

O capítulo de desenvolvimento é dedicado a relatar todo o desenvolvimento deste trabalho, desde a modelagem até a implementação. Todo o conteúdo do projeto proposto foi baseado em conhecimentos adquiridos pelos referenciais teóricos, abordados no capítulo anterior. Os temas serão reunidos gradativamente ao longo do capítulo de forma a compor a arquitetura de controle e os algoritmos do sistema robótico do estudo de caso.

Este capítulo está organizado em três seções principais. Esta seção de introdução anuncia o problema em questão, com o objetivo de pesquisa que resultou no projeto atual. A seção de modelagem primeiramente apresenta o estudo de caso, os motivos de sua escolha como base para este trabalho, introduzindo os principais conceitos relacionados a robótica móvel, arquitetura autoadaptativa e visão computacional. A mesma seção apresenta, em seguida, a modelagem do sistema robótico e da arquitetura de software em ROS que será implementada. Por fim, a seção de implementação descreve detalhadamente os processos e algoritmos que, juntos, compõem a arquitetura de controle autoadaptativa que realiza a tarefa principal do robô móvel.

Conforme situado anteriormente nas seções 1.2 e 1.3 do primeiro capítulo deste documento, existe uma dificuldade de se trabalhar com robôs móveis atuando em ambientes dinâmicos e sujeitos a imprevisibilidades, como o trânsito de objetos e pessoas ou variação das condições climáticas. Estas características dos ambientes fazem com que seja fundamental utilizar um sistema robótico flexível e robusto, capaz de autogerenciar seus componentes de software e de hardware, devido à complexidade existente em se manter um sistema em tempo real de alta performance com grande fluxo de dados entre sensores e atuadores para prover a reatividade.

Ao se projetar a estrutura dos códigos que definem o software do robô é importante, primeiramente, conhecer sobre o framework operacional do robô, que no caso é o ROS - *Robot Operating System*. Como já dito na Seção 2.3, o ROS é um sistema meta-operacional de código aberto feito para coordenação de robôs e sua estrutura é divida em processos (nós) individuais que trocam

mensagens em tempo de execução. A forma com que esse sistema é estruturado facilita a implementação de diversos tipos de arquiteturas de controle para robôs. A Figura 3.1 mostra um exemplo[14] de um processo estruturado em ROS.

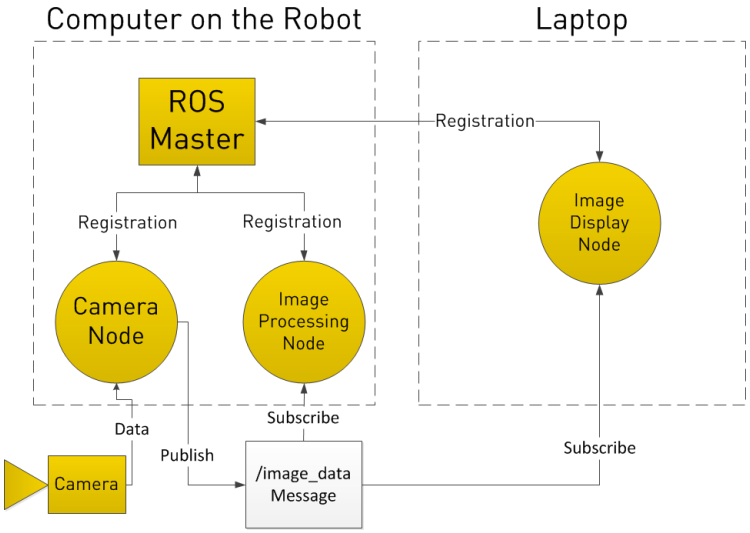


Figura 3.1: Nós do ROS comunicando entre si dentro do robô, que é acessado por um dispositivo externo. Observe que o nó *ROS master* foi iniciado no sistema embar se comunicando cado do robô, e os demais nós internos e externos irão se registrar através dele. O robô ainda contém os nós de aquisição de imagem da câmera e de processamento de imagem (círculos amarelos), enquanto o Laptop possui um nó de exibição das imagens na tela. Os dados de imagem adquiridos são enviados e recebidos pelos nós através do tópico */image_data*. Fonte: `https://www.clearpathrobotics.com/assets/guides/ros/IntrototheRobotOperatingSystem.html`

.

Conhecendo a estrutura do sistema operacional para robôs em questão, é necessário que a tarefa que se deseja executar seja detalhada. O primeiro passo é definir qual o paradigma de controle (Seção 2.1.2) associado à aplicação em questão, uma vez que a forma como o robô se comporta durante a execução encaminhará a escolha da arquitetura de controle. O segundo passo é escolher uma arquitetura própria para prover a características em questão – no caso, a autoadaptatividade – e que dê suporte ao paradigma de controle definido. Por fim, uma vez compreendidos e definidos estes principais conceitos, será possível iniciar a modelagem da arquitetura.

3.2 Modelagem

3.2.1 Definindo a Tarefa do Robô: o caso em estudo

Será utilizado o robô móvel Pioneer 3-AT (ver 2.2) e seus sensores de distância a laser, posicionamento, odometria e a câmera para visão computacional. O framework Operacional é o ROS e a linguagem de programação utilizada será o *python*, que possui as bibliotecas para ROS como

rospy e *roslib*.

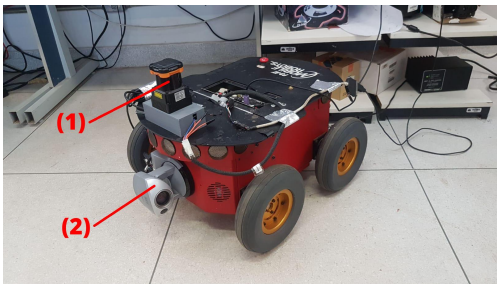


Figura 3.2: Robô Pioneer 3-AT utilizado no estudo de caso instalado com (1) sensor de distância a laser e (2) câmera. Internamente, ainda existe o sensor de posição e velocidade das rodas, encoder, acessível por meio de sua odometria. Fonte: O autor.

A tarefa a ser executada é definida por: o robô autônomo deve inicialmente se posicionar em um ponto de partida no início de um dos corredores do prédio, então deverá prosseguir em linha até o fim do corredor enquanto utiliza seus sensores para identificar as portas (ao lado esquerdo) e classificá-las como abertas e fechadas. Ao final da tarefa, é gerado um mapa unidimensional do corredor com suas portas. A Figura 3.3 mostra uma foto tirada no início de um dos corredores e uma ilustração de um trecho. À esquerda dele estão as portas, enquanto do lado direito existe uma grade/cerca com vista para o pátio do andar debaixo do prédio. Para este trabalho, apenas o lado esquerdo interessa. Inclusive, como mostra a foto na Figura 3.2, percebe-se que, devido a sua instalação no robô, o lado esquerdo possui melhor visualização que o lado direito. Os corredores possuem aproximadamente 80 metros de comprimento por 3 de largura, contendo cerca de 25 ou 26 portas.

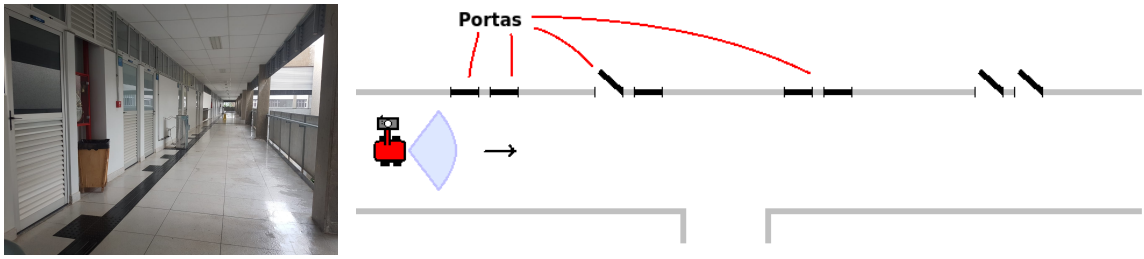
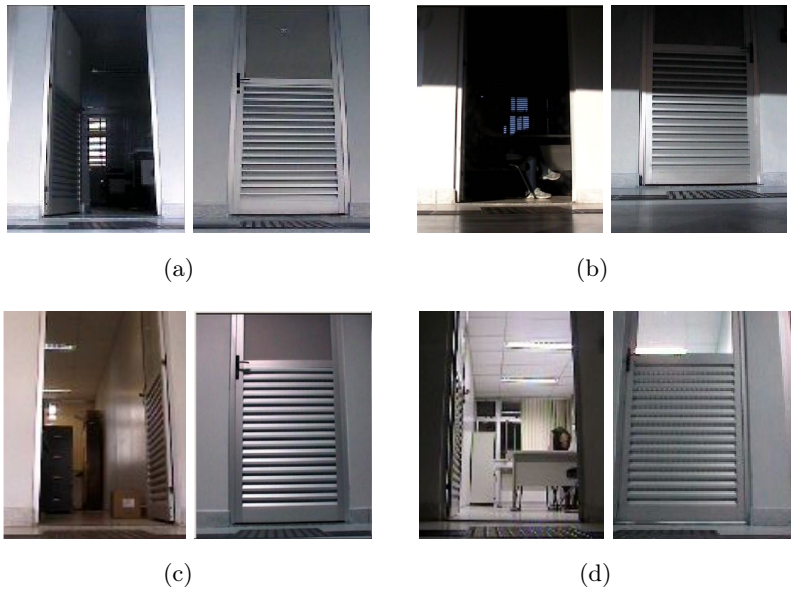


Figura 3.3: O corredor onde será realizado o trabalho. Fonte: o autor.

Na primeira etapa, assume-se que o robô pode ser posicionado pelo usuário, no início do corredor, em uma orientação e distância da parede aleatórias. Antes de poder avançar pelo corredor, é necessário que o robô esteja posicionado a uma distância aproximada de 2,20 metros da parede para que a câmera possa enquadrar bem as portas. Para isso, o robô fará um procedimento inicial de posicionar-se em um ponto ideal e orientar-se de frente para o corredor. Então, é iniciada a execução da rotina principal, que utiliza o sensor laser e a câmera e analisa os dados de distância e de imagem para detectar as portas abertas e portas fechadas.

A principal forma de identificar as portas é a partir do processamento das imagens da câmera. Enquanto que o laser servirá como uma espécie de suporte para confirmação ou rejeição das detecções. Durante a etapa de definir os algoritmos de processamento de imagens para encontrar as portas, percebeu-se que a luminosidade do corredor, dependente de período do dia e fatores naturais/climáticos, tem grande influência nos resultados da análise da imagem. Essas variações das luminosidades do ambiente – combinações entre luz do lado de fora do corredor, luz de dentro das salas, dia ensolarado/nublado, trecho com sol/sombra – afetam significativamente os algoritmos escolhidos com as ferramentas de processamento de imagem para detecção das portas, e verificou-se necessário criar uma **classificação para luminosidade do ambiente**, com algoritmos de detecção de portas específicos para cada um dos tipos identificados. Observe a Figura 3.4.



(a) (b)

(c) (d)

Figura 3.4: Imagens das portas adquiridas pela câmera do robô Pioneer. As fotos foram tiradas em diferentes momentos do dia e em dias diferentes em um mesmo corredor: (a) durante o dia porém em um trecho sem iluminação do sol, (b) trecho com iluminação do sol, (c) no final da tarde e (d) durante a noite. Fonte: o autor.

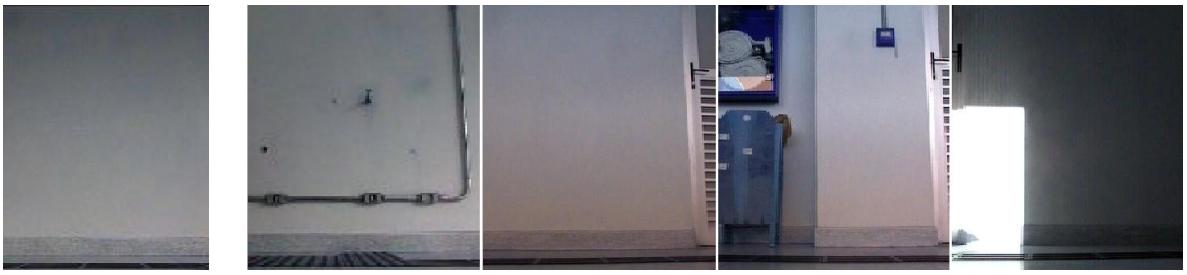
Analisando os casos estudados, foi possível classificá-los em três situações principais, inicialmente, de acordo com a á 3.1:

Tabela 3.1: Tabela com a classificação de luminosidade ambiente estabelecida.

Classe	**Atuação**
Dia com Sombra	Válido desde a manhã até o início do anoitecer. Inclui todos os casos durante o dia onde não ocorre iluminação direta do sol.
Dia com sol	Válido onde o sol ilumina diretamente as paredes do corredor, em dias ensolarados.
Noite	Válido para o período da noite.

Esse caso será usado como exemplo de **autoadaptatividade** do sistema. Ao longo do corredor, o robô está sujeito a se deparar com variações de luz do corredor, como por exemplo estar em um trecho com sol nas paredes e então entrar em um trecho com sombra. Ou então estar passando por um trecho com sol e de repente o céu ficar nublado. Ou mesmo estar andando pelo corredor em um fim de tarde sem sol e algumas salas abertas têm as luzes acesas enquanto outras tem luzes apagadas. Cada uma dessas situações necessita de que um novo algoritmo seja aplicado para que a visão computacional possa identificar todas as portas quando chegar ao fim do corredor. Ou seja, o robô deve ser capaz de identificar essas variações do ambiente a partir das analises dos dados dos sensores e adaptar seus algoritmos de acordo.

Para garantir que essas adaptações na forma de chaveamento entre os algoritmos de detecção de portas aconteçam a tempo de uma porta ser detectada, é necessário impor uma baixa velocidade linear para o robô, a fim de evitar que os objetos nas imagens não sejam perdidos entre um frame de vídeo analisado e outro. No entanto, devido a essa baixa velocidade, o robô levaria um tempo demasiadamente longo para fazer o procedimento em todo o corredor. Pensando nisso e considerando que entre uma porta e outra existirão intervalos de alguns metros de apenas parede, define-se que o robô irá realizar um processamento extra nas imagens para identificar esses trechos uniformes no corredor. Isto é, segmentos do corredor onde há apenas a parede branca sem nenhum objeto, de acordo com as fotos da Figura 3.5. Assim, quando o robô identificar esses trechos uniformes nas imagens, uma velocidade linear maior será estabelecida até que algum elemento que não é parede seja encontrado na imagem, o que fará com que volte a andar em velocidade baixa. Essa alteração na velocidade do robô caracteriza-se também uma adaptação visando otimização de tempo de execução.



(a) (b)

Figura 3.5: Imagens adquiridas do corredor. Em (a), imagem de um trecho do corredor que tem apenas parede. Em (b), imagens que apresentam qualquer outro objeto ou variação de luminosidade brusca. Fonte: o autor.

Por outro lado, conforme o final da Seção 2.6.2, a detecção de objetos na visão computacional está sempre associada a uma taxa de acertos e de erros. Então, dependendo do caso e das técnicas utilizadas, é importante utilizar outras metologias e sensores para aumentar a taxa de acertos na detecção de objetos por processamento das imagens. Pensando nisso, foi definido que o sensor de distância a laser será usado como elemento para confirmação ou rejeição das supostas detecções de portas abertas por parte da visão. Um algoritmo adicional é usado para identificar, nas leituras do

sensor, as situações de variações na parede, indicando a possibilidade de ocorrência de uma porta aberta, como mostrado na Figura 3.6.

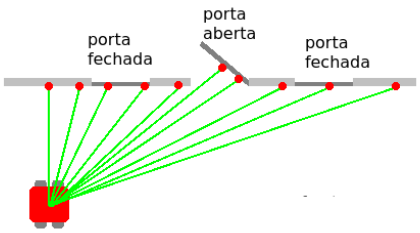


Figura 3.6: Detectando possível porta aberta com o sensor de distância a laser. Fonte: o autor.

No entanto, no robô Pioneer disponível para os nossos experimentos, existe um desbalanceamento dos eixos que o faz levemente curvar para a esquerda. Isso é notado quando se aplica um comando de velocidade linear qualquer: mesmo com velocidade angular igual a zero, o robô começa a virar para a esquerda. Este fato gera um problema, uma vez que é necessário que o robô ande em linha reta pelo corredor, à uma distância fixa da parede. Para que o algoritmo de detecção de portas abertas com o sensor laser funcione, é necessário um procedimento de reajuste de orientação do robô que seja executado periodicamente, usando os dados do sensor de distância a laser. Nesse procedimento, o robô estabelece um ajuste na velocidade angular para que seja possível prosseguir paralelo à parede, conforme ilustrado na Figura 3.7. O algoritmo de reajuste de orientação paralela pode ser também utilizado para outros casos onde se deseja alinhar a direção do robô, como por exemplo no procedimento inicial de ajuste de distância. Detalhes serão explicados na seção de desenvolvimento.

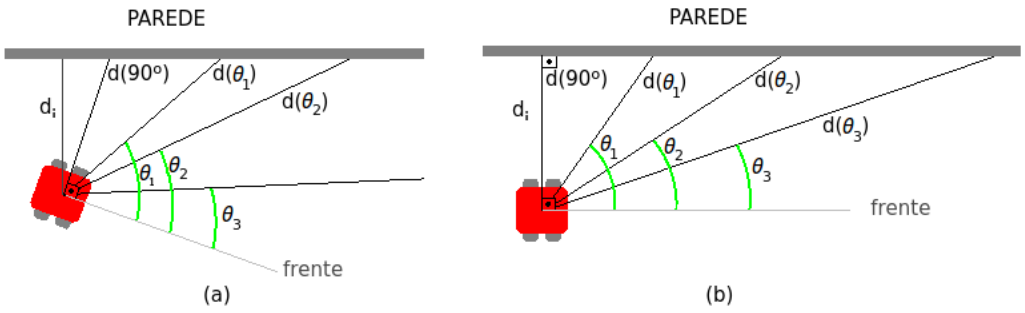


Figura 3.7: A esquerda, o robô encontra-se desalinhado em relação à direção da parede, e a direita, o robô está orientado corretamente após um procedimento de reajuste de orientação utilizando os sensores de distância. Fonte: o autor.

Por fim, vale ressaltar que este trabalho não lidará com a detecção de obstáculos que possam vir a colidir com o robô ou afetar os processamentos das imagens e das leituras do sensor laser. Assume-se que, durante a execução da tarefa, as pessoas não transitam pelo corredor ou, se transitam, contornam o robô pelo lado direito, não afetando os procedimentos realizados.

Após reunir todos esses tópicos discutidos, redefine-se a tarefa: *o robô deve se deslocar pelo corredor em linha reta, ininterruptamente e sendo capaz de se alinhar paralelamente à parede quando necessário, enquanto utiliza visão computacional e o sensor laser para identificar as portas à sua esquerda e classificá-las como abertas ou fechadas. Além disso, o sistema deverá ser capaz de se autoadaptar em duas situações: primeiro ao identificar variação de luminosidade do ambiente, onde deve realizar a troca do algoritmo de detecção de portas; e, segundo, ao identificar trechos do corredor de "somente parede" ou "com objetos", onde deve ajustar sua velocidade linear. As capacidades de autoadaptação e de correção da orientação do robô se caracterizam, respectivamente, como auto-otimização e autocura do sistema.*

3.2.2 Modelagem da Arquitetura Autoadaptativa

Analisando a definição, é possível perceber que a tarefa do robô pode ser decomposta em componentes de comportamento, como "se distanciar da parede" e "seguir em frente mantendo-se paralelo a parede", e componentes de percepção, como "encontrar portas por imagem" e "analisar corredor pelo sensor laser". Nesse contexto, é possível verificar que o sistema pode atuar na tarefa segundo o paradigma de Controle Baseado em Comportamentos (BBC, *Behavior Based Control*), que possui componentes reativas e que os objetivos do robô são compostos por *componentes de comportamento* e *componentes de percepção*, independentes e interativos, segundo apresentado no capítulo de fundamentação na Seção 2.1.2. Dessa forma, além do sistema em questão estar seguindo a abordagem BBC, a autoadaptatividade está diretamente relacionada com o controle reativo: os dados conseguidos pela visão computacional a partir das imagens adquiridas pelo sensor câmera geram uma resposta interna imediata de se adaptar a componente de detecção de portas, ou de alterar a velocidade do robô. Assumindo então que o sistema seguirá o paradigma de Controle Baseado em Comportamentos, definem-se os componentes de comportamento e de percepção que compõem a estrutura do sistema, assim como os nomes pelos quais serão tratados daqui para frente, para posterior implementação:

1. **Distanciar-se da parede** *(set_ distance)*: comportamento responsável pelo procedimento inicial de estabelecer a posição e orientação correta do robô;
2. **Seguir em frente mantendo-se paralelo à parede** *(follow_ line)*: comportamento responsável por aplicar comandos de velocidade nos atuadores do robô, sendo assim responsável por fazê-lo avançar em frente e corrigir a orientação quando necessário, caracterizando a autocura do sistema;
3. **Detectar objetos em imagem** *(find_ doors)*: componente de percepção por processamento de imagem cujo algoritmo desempenha a detecção portas abertas e portas fechadas. Existirão 4 versões desta unidade que serão selecionadas, mas apenas uma é utilizada a cada instante;
4. **Identificar luminosidade do ambiente** *(analyze_ adaptation)*: componente de percepção por processamento de imagem com objetivo de identificar o tipo de luminosidade do ambiente

e por detectar paredes lisas. Perceba que este algoritmo se responsabiliza por verificar as necessidades de autoadaptação do robô (no caso, auto-otimização), relativas à seleção do algoritmo de detecção de portas e à alteração da velocidade linear;

5. **Identificar com sensor laser portas abertas ou fim do corredor**: *(analyze_laser)*: componente de percepção responsável por analisar as leituras do sensor laser para identificar portas abertas, dando suporte as detecções da visão computacional. Além disso, será responsável por detectar o final do corredor, onde há uma cerca, caracterizando o fim da atividade.

6. **Gerar mapa unidimensional do corredor**: *(create_map)*: componente que gera o resultado final da tarefa na forma de um mapa unidimensional com a localidade das portas e se estão abertas ou fechadas.

Definidos os componentes que compõem o sistema BBC, pode-se inserir a arquitetura autoadaptativa de software para robótica móvel proposta, descrita na Seção 2.5. Essa arquitetura se baseia em três princípios: (1) a decomposição do sistema em componentes independentemente implementáveis; (2) a organização em camadas (nível de aplicação e meta-nível); e (3) a separação dos componentes da camada de controle em três estágios: coleta de dados, análise dos dados de acordo com as políticas de adaptação e a administração das adaptações. Com base nesses princípios, o sistema pode ser implementado com a estrutura descrita abaixo.

- as unidades do sistema correspondem as componentes de comportamento e percepção independentemente implementáveis e que interagem entre si;

- os componentes são divididos entre dois níveis de camadas: o nível inferior, que é a **Camada de Aplicação**, e o nível superior, que será denominado a partir de agora de **Camada de Adaptação**. A camada de adaptação é composta unicamente pelo componente de percepção *(analyze_adaptation)*, enquanto todos os demais listados compõem a Camada de Aplicação.

- para tornar o sistema autoadaptativo quanto ao algoritmo de detecção de portas e quanto à velocidade linear do robô, a componente *(analyze_adaptation)* da Camada de Adaptação possui três estágios sequenciais: **coleta** de dados dos comportamentos, **análise** dos dados conforme as regras de adaptação e então a **administração** das mudanças necessárias.

Estabelecidos esses conceitos básicos da arquitetura de software autoadaptativa para robótica, a próxima etapa é iniciar a modelagem das componentes de software que serão implementadas em linguagem ROS.

3.2.2.1 Modelagem em ROS

Ao se implementar uma arquitetura em ROS, é importante primeiramente definir o que serão os nós do sistema, os tópicos e as mensagens. Conhecendo os conceitos apresentados na Seção 2.3, estabelece-se que os nós do código são os comportamentos da arquitetura, os sensores e os

atuadores. A diferença é que, enquanto os nós de sensores e atuadores já estão prontos, os nós de comportamento e percepção serão desenvolvidos.

Os tópicos são canais por onde as mensagens são enviadas e lidas pelos nós. As mensagens são valores calculados dentro dos programas de cada nó, e podem ser variáveis do tipo inteiro, booleano, ponto flutuante, listas, etc. A Figura 3.8 mostra como fica definido o modelo da estrutura em ROS do sistema implementado.

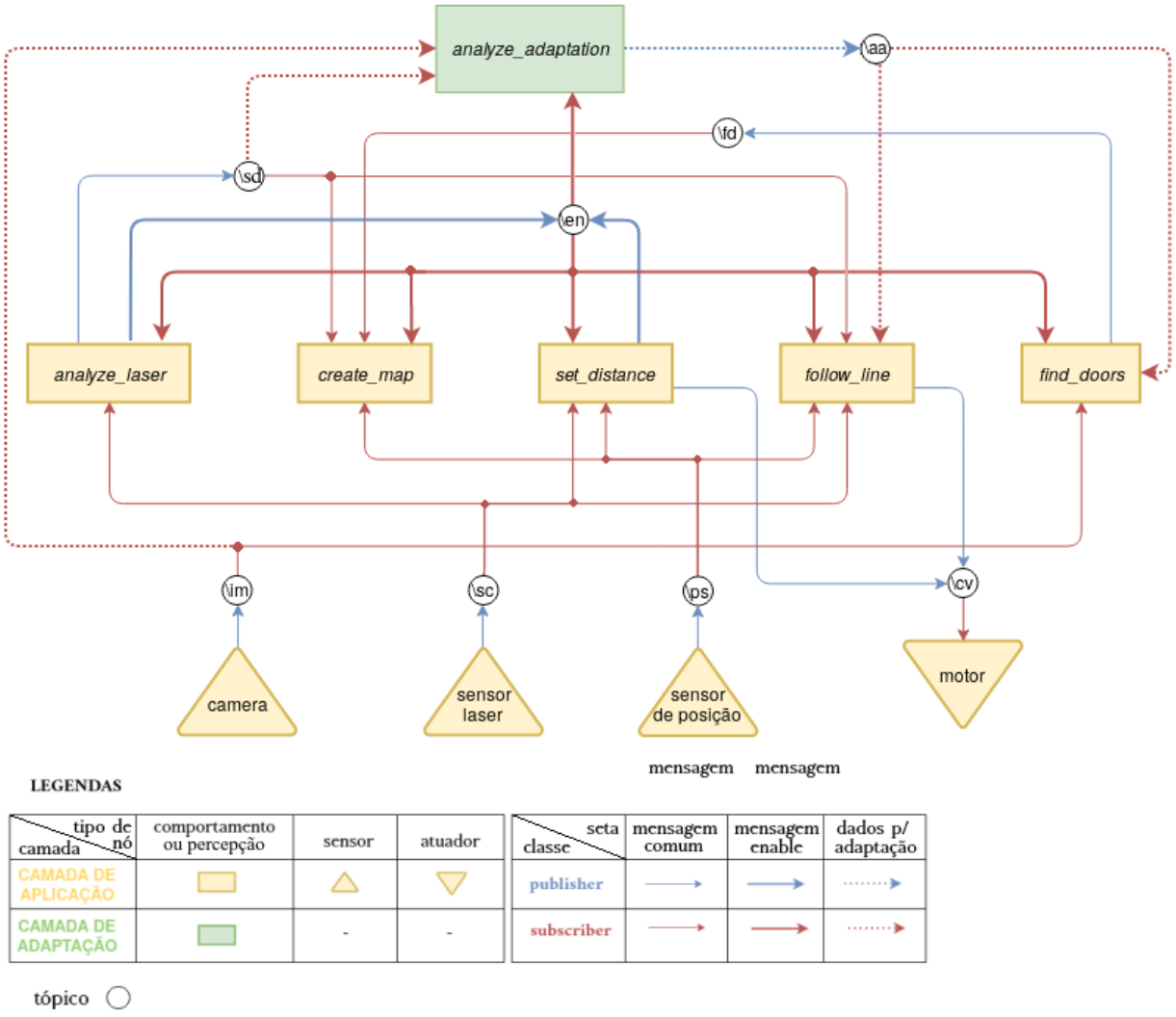


Figura 3.8: Modelo da arquitetura em ROS. Fonte: o autor.

Analisando o diagrama, é possível perceber os nós de comportamentos de ambas as camadas, assim como os nós do sensores. Os nós comunicam-se entre si através dos tópicos com uma semântica de publicação e assinatura, onde um nó envia uma ou mais mensagens relevantes para os tópicos onde os outros nós terão acesso, lembrando que uma mensagem em ROS não necessariamente contém apenas um único campo, podendo ser uma estrutura de dados. As setas na Figura, de acordo com legenda, podem representar três tipos de mensagens:

- *mensagem comum*, que são mensagens trocadas entre os nós durante a execução da tarefa, como por exemplo a detecção de uma porta feita pelo nó *find_ doors* para o nó *create_ map*;

- *mensagem de ativação*, é uma mensagem enviada apenas duas vezes durante todo o processo:

  1. a primeira vez é enviada pelo nó *set_ distance* quando ele finaliza seu procedimento de inicialização, indicando que o robô está pronto para iniciar a rotina principal no início do corredor;
  2. a segunda é enviada pelo nó *analyze_ laser* quando o robô chega no fim do corredor e detecta uma parede/cerca a frente, enviando a mensagem para finalizar a execução dos outros nós do sistema.

  Assim, o primeiro envio dessa mensagem significa que todos os outros nós serão iniciados para a rotina principal, e o segundo envio sinaliza que a tarefa chegou em seu final.

- *mensagens monitoradas para autoadaptatividade*, que contém os dados de imagem que são analisados pelo componente da Camada de Adaptação. Percebe-se que o nó *analyze_ adaptation* acessa o tópico que contém as imagens da câmera, onde os dados do sensor são coletados. Depois, realiza internamente processamento de imagem e análises da regras de adaptação, isto é, identifica o tipo de luminosidade do ambiente ou identifica se existe apenas parede lisa à sua esquerda. Caso adaptações sejam necessárias, a etapa de administração enviará os comandos para os nós *find_ doors* – selecionando o nó específico, dentre os 4, para detectar portas naquela situação de luz ambiente – ou para o nó *follow_ line* – alterando a velocidade do robô.

A Tabela 3.2 mostra como estão organizados os tópicos, em relação aos nós que publicam, os assinantes e o conteúdo das mensagens que por eles transitam.

Tabela 3.2: Tabela dos tópicos da arquitetura ROS.

Tópico	**Publicantes**	**Assinantes**	**Mensagem**
/enable	set_distance analyze_laser	todos nós de comportamentos	-Ativa/Desativa outros nós
/found_doors	find_doors	create_map	-Porta aberta/fechada detectada
/scanned_door	analyze_laser	find_doors follow_line analyze_adaptation	-Porta aberta à esquerda
/analyzed_adapt	analyze_adaptation	create_map find_doors	-Adaptação de luminosidade -Adaptação da velocidade
/image	sensor câmera	analyze_adaptation find_doors	-Dados de imagem
/scan	sensor laser	analyze_laser follow_line set_distance	-Dados de distâncias
/pose	sensor de posição	create_map follow_line set_distance	-Dados de posição
/command_vel	follow_line set_distance	motor	-Comandos de velocidade

3.2.2.2 Modelagem das Componentes Autoadaptativas

A autoadaptatividade do sistema em questão, conforme citado anteriormente, é realizada para adaptar a escolha da componente de software para uma luminosidade específica do ambiente, assim como a velocidade do robô de acordo com o trecho do corredor por onde passa. A partir da estrutura do sistema implementado em ROS de acordo com a Figura 3.8, a autoadaptação ocorre em dois momentos: o primeiro ocorre no chaveamento entre os nós de percepção *find_ doors*, cada um específico para operar em uma situação de luminosidade identificada em tempo-real. O segundo ocorre ao adaptar a velocidade do robô de forma a reduzir o tempo total de execução da tarefa, e isso é feito alterando-se o valor do parâmetro *velocidade linear* de acordo com a existência ou não de portas nas imagens em cada instante, na componente de percepção *find_ doors*. Os dois casos de adaptação existentes estão relacionados à otimização de desempenho, e ambos são detectados pelo componente da Camada de Adaptação *analyze_ adaptation*. Esse componente, como já mencionado, tem três estágios sequenciais, sendo o primeiro a coleta dados, depois a análise da necessidade de adaptação e, por fim, a execução da tal adaptação. No caso deste trabalho, executar a adaptação implica em chavear entre componentes (ativar uma e desativar outras) que desempenham a mesma função ou alterar o parâmetro de uma outra componente, como mostra a Figura 3.9.

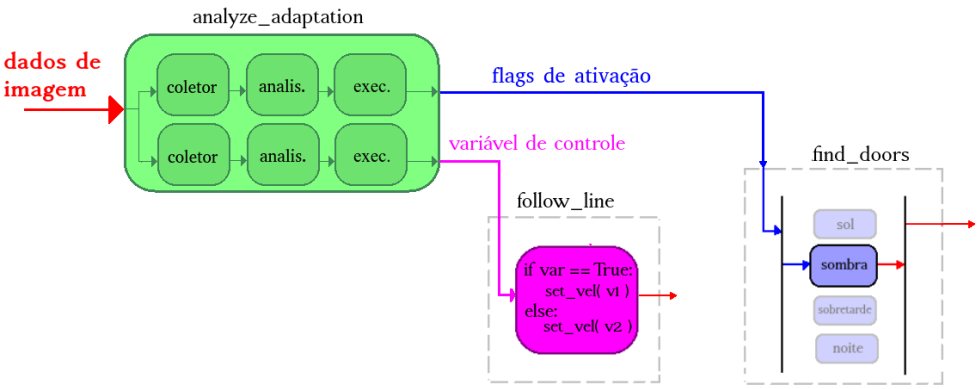


Figura 3.9: Esquemático da parte autoadaptativa do sistema. Fonte: o autor.

Tendo o modelo completo de como funciona o sistema autoadaptativo do robô para executar a tarefa, a próxima seção apresenta a implementação dos nós no framework ROS.

3.3 Implementação dos Componentes de Arquitetura em ROS

A implementação dos nós que utilizam os atuadores do robô foi possível inicialmente com o auxílio de uma simulação fornecida pelo próprio ROS, então posteriormente estes códigos foram testados e adaptados para o Pioneer. Os demais nós, principalmente os relacionados à visão juntamente com sensor laser, foram desenvolvidos com o auxílio da biblioteca *rosbag* do ROS, que possui ferramentas capazes de gravar os dados dos sensores de posição, distância e câmera para

um acesso remoto posterior.

3.3.1 Algoritmo de Orientação Paralela à Parede

Este procedimento é utilizado em dois nós do sistema: o nó de estabelecer distância em relação à parede (*set_ distance*), que é o procedimento inicial da tarefa, e o nó de seguir em linha reta (*follow_ line*), responsável por movimentar o robô ao longo do corredor. O algoritmo desenvolvido é o mesmo para ambos, com diferentes velocidades lineares.

A principal razão para a necessidade deste procedimento é que o robô possui um desalinhamento nas rodas, e mesmo estando sob comando de velocidade angular nula a velocidade real é diferente de zero. Com isso, é importante que periodicamente ocorra essa correção da orientação ao longo da trajetória do robô.

Durante o desenvolvimento deste trabalho, dois algoritmos diferentes foram elaborados. O primeiro deles calcula o ângulo da parede a partir das leituras do sensor de distância. O segundo algoritmo [11] não calcula o ângulo da parede de forma explícita, mas procura localizar o robô de forma paralela a parede. A Figura 3.10 mostra as propriedades do sensor de distância laser utilizado para os algoritmos implementados.

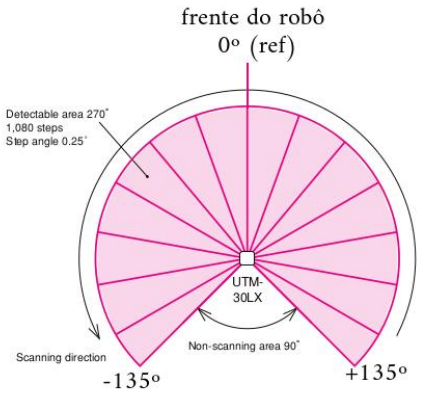


Figura 3.10: Esquemático do sensor laser UTM-30LX instalado no Pioneer, retirado do próprio manual e adaptado. Fonte: manual do Pioneer 3-AT, adaptado.

O primeiro algoritmo implementado utiliza as leituras e propriedades do sensor de distância e informações sobre posição do robô para calcular o ângulo do vetor da parede, para que o robô possa então girar até que seu vetor de orientação coincida com o ângulo da parede calculado. Como mostra a Figura 3.11, a partir das distâncias até a parede coletadas pelo sensor, da diferença angular entre os feixes do laser e da posição e do ângulo de orientação do robô em relação ao sistema de coordenadas global 2D, é possível encontrar os pontos (x,y) no sistema de coordenadas onde os feixes de laser coincidiam com a parede. Fazendo uma regressão linear desses pontos, o ângulo do vetor da parede é calculado, então, por fim, o robô gira no sentido correto até que sua orientação

se torne paralela à da parede calculada.

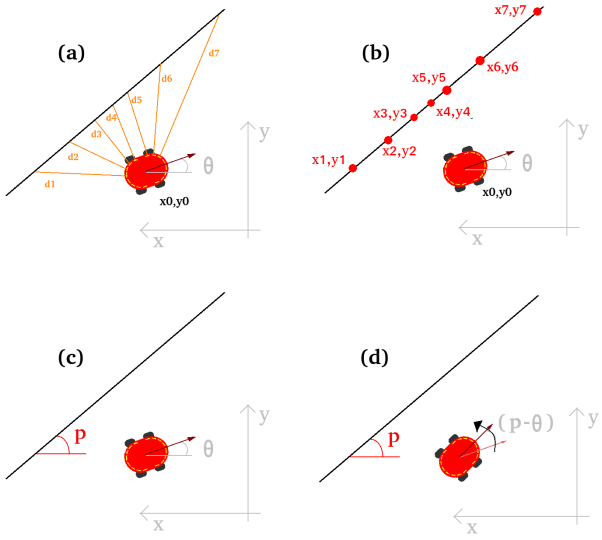


Figura 3.11: O primeiro algoritmo para posicionar o robô em relação a parede. Em (a), as leituras do sensor laser. Em (b), cálculo dos pontos (x, y) da parede. Em (c), é calculado o ângulo da parede. Em (d), o robô gira até seu ângulo coincidir com o da parede. Fonte: o autor.

O grande problema desse algoritmo é que o ângulo da parede é calculado matematicamente à partir do ângulo de orientação de robô. O ângulo de orientação do robô não será sempre o mesmo uma vez que o eixo das rodas é desbalanceado. Dessa forma, mesmo que seja dado um comando para o robô navegar por 10 metros em com velocidade angular nula, no final desse percurso a referência de orientação angular não será a mesma do início. Então, o ângulo da parede calculado pelo algoritmo estará incorreto, e esse erro será maior quanto mais o robô tiver andado desde o início da tarefa. Por esses motivos, esse algoritmo teve de ser descartado, e se fez necessário um procedimento que não dependesse do ângulo da orientação do robô.

O segundo algoritmo desenvolvido depende unicamente das medidas de distância, e só é possível devido a alta precisão das leituras do sensor laser. O robô utilizaria o que foram denominados por "braços virtuais", ambos do lado esquerdo do robô, um mais à frente e outro mais atrás, simetricamente espaçados em relação à sua lateral esquerda. Como mostra a Figura 3.12, cada um desses braços são, na realidade, a média de 5 medidas adjacentes de distância do sensor, e o resultado são as variáveis D_f (distância frontal) e D_t (distância traseira). Quando o robô estiver quase paralelo à parede, a razão D_f/D_t será aproximadamente 1. Se o robô girar no sentido horário em relação à orientação paralela, o braço frontal será maior que o traseiro, então a razão D_f/D_t será maior que 1. Caso gire no sentido anti-horário, a razão será menor que 1, conforme mostra a Figura. Assim, para que o robô fique paralelo à parede, basta girar no sentido correto até que a razão entre o tamanho dos braços virtuais fique próximo de 1. Ou seja, que esteja dentro do intervalo $1,00 \pm tol$ onde tol é uma tolerância suficientemente pequena, definida empiricamente, de forma que a orientação do robô ainda possa ser considerada paralela à da parede.

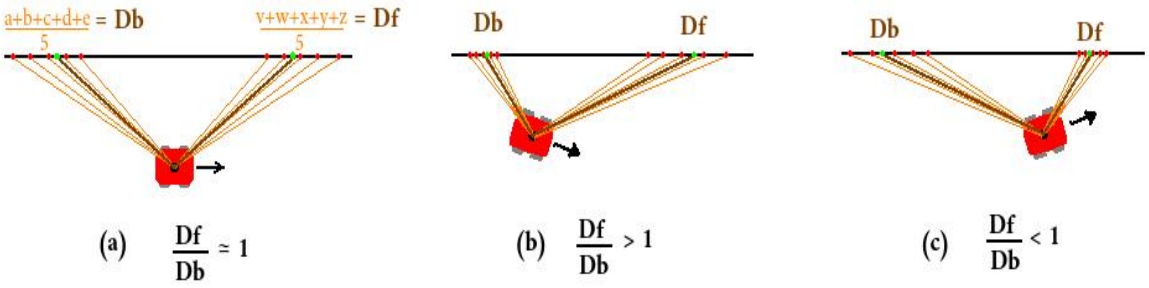


Figura 3.12: O algoritmo de posicionamento implementado. Em (a), com $D_f/D_t = 1,00$, o robô está orientado paralelamente à parede. Em (b) e (c), com $D_f/D_t \neq 1,00$, o robô não está alinhado. Fonte: o autor.

Importante ressaltar que este algoritmo só é válido para ser utilizado em superfícies planas, ou seja, paredes. Então, para todos os casos em que for utilizado certificar-se que o robô esteja ao lado de uma parede lisa, sem objetos, portas, tubulações ou extintores de incêndio. Em experimentos foi constatado que este algoritmo traz resultados satisfatórios.

Para que este algoritmo seja utilizado no nó de seguir em frente, foi necessário realizar uma pequena alteração no algoritmo. Embora o ajuste de orientação ocorra periodicamente buscando pela razão $D_f/D_t = 1$, o robô não mantém sempre a mesma distância desejada até a parede, e devido a característica assimétrica do seu desalinhamento, a distância até a parede será cada vez menor a cada ajuste.

A solução implementada foi alterar o algoritmo para buscar uma razão para D_f/D_t ligeiramente maior que 1, indicando que o robô estará levemente voltado para à sua direita em relação a orientação paralela. Escolheu-se empiricamente o valor 1.018 ± 0.004 pelo seguinte motivo: com ele, o robô ainda fará a curva indesejada devido ao desalinhamento dos eixos, porém após um certo deslocamento sua distância da parede permanecerá próxima da que estava quando fez o procedimento de ajuste. Observe a Figura 3.13, que ilustra os efeitos dessa alteração na trajetória.

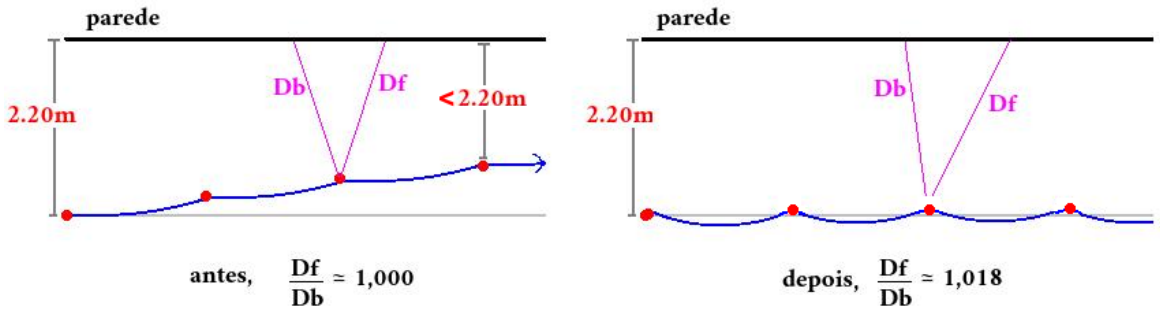


Figura 3.13: Trajetória do robô para valores diferentes de D_f/D_t. Fonte: o autor.

Após este ajuste, o algoritmo implementado está pronto para ser utilizado pelos nós de comportamento do sistema. Para referências posteriores neste documento, este algoritmo será denominado

por *parallelize()*.

3.3.2 Comportamento de Estabelecer Distância da Parede: a rotina de inicialização

Este comportamento corresponde ao nó *set_ distance* do sistema implementado em ROS. No início da execução da tarefa, a premissa é que o robô esteja posicionado no começo do corredor obedecendo à duas condições: a primeira é que esteja localizado no extremo início do corredor, de forma que esteja ao lado de um trecho totalmente liso da parede (sem objetos). A segunda é que a orientação inicial deve ser tal que o lado esquerdo do sensor laser do robô consiga escanear a parede lisa, ou seja, o robô deve estar virado de frente para o corredor, mesmo que sua orientação não seja paralela à da parede.

Quando a tarefa é iniciada, este comportamento do sistema do robô tem como primeiro objetivo alinhar-se com a parede à sua esquerda, utilizando o algoritmo descrito na Seção 3.3.1. Sabendo que o robô está localizado a uma distância inicial aleatória d_i da parede, o próximo passo consiste em se mover até a distância d_f definida pelo programador, tipicamente $d_f = 2,20m$, de onde o robô irá começar sua trajetória principal. Para alcançar d_f e se posicionar paralelo à parede, o robô utiliza o algoritmo descrito em *Algoritmo* 1, ilustrado na Figura 3.14.

Quando os objetivos forem alcançados, o nó *set_ distance* irá publicar no tópico */enable* uma mensagem booleana de valor *verdadeiro*, que será lida por todos outros nós sinalizando que a rotina principal deve ser iniciada, como mostrado na Tabela 3.2.

O algoritmo do nó *set_ distance* se resume em:

Algoritmo 1: Algoritmo do nó *set_ distance*.

```
1  início
2      Se alinhar paralelo à parede (algoritmo parallelize());
3      Girar 90º horário;
4      Calcular Δd = d_f − d_i;
5      if Δd > 0 then
6          andar |Δd| metros para frente;
7      else
8          andar |Δd| metros para trás;
9      end
10     Girar 90º anti-horário;
11     Publicar True no tópico /enable;
12 fim
```

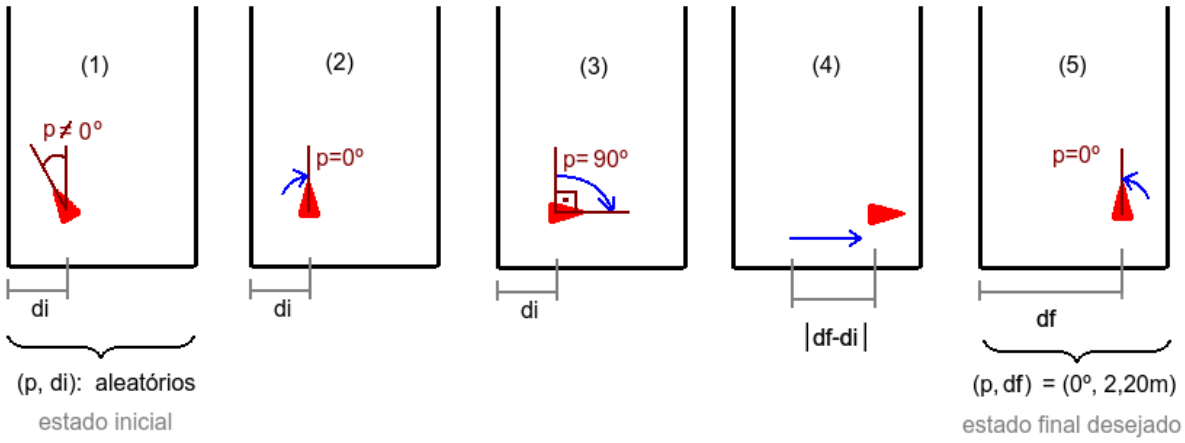


Figura 3.14: Ilustração do algoritmo descrito. Fonte: o autor.

3.3.3 Comportamento de Navegação em Linha Reta: a rotina principal

Este comportamento da Camada de Aplicação é implementado no nó *follow_line* do sistema. É responsável por gerenciar a navegação do robô em sua trajetória pelo corredor, seguindo paralelo à parede a uma distância média igual a que o robô estava quando foi posicionado pelo comportamento anterior, de distanciamento da parede. Esse comportamento é chamado, em robótica reativa, de *wall following*, ou *seguir a parede*. Enquanto anda pelo corredor, os outros componentes de percepção do sistema se encarregam da detecção de portas e de gerenciar a autoadaptatividade.

Este comportamento é composto por dois algoritmos independentes: um deles aplica comandos de velocidade linear para os atuadores do robô, para controlar sua velocidade de avanço. Este procedimento não é interrompido até que o robô conclua a tarefa por inteiro. O segundo procedimento ajusta a orientação, e é chamado periodicamente quando o robô verifica alguma condições, conforme descrito na Seção 3.3.1. A Figura 3.13 ilustra a execução de ambos algoritmos.

3.3.3.1 Propriedade autoadaptativa de velocidade

Para a etapa de seguir em frente, assim que o nó *follow_line* recebe pelo tópico */enable* a mensagem booleana de ativação enviada pelo nó *set_distance*, os comandos de velocidade linear são enviados para os motores do robô. Para definir o valor dessa velocidade, é necessário levar em consideração as restrições em tempo-real do sistema robótico.

Essas restrições se devem à capacidade de processamento do sistema embarcado do robô e do laptop e da grande quantidade de dados sendo transmitidos a uma alta frequência, dados esses de imagem, de sensor de distância, de posicionamento, além dos dados de todos outros processos que compõem o sistema. Além disso, o fato de se terem vários nós ROS sendo executados simultaneamente pode sobrecarregar o escalonamento de tarefas dos processadores afim de controlar os parâmetros do robô, como a velocidade linear. Considerando essas dificuldades existentes para se

cumprir as restrições de tempo-real, associadas aos procedimentos periódicos dos algoritmos dos nós do sistema e ao grande volume de dados processados, é necessário definir um valor de velocidade linear v_1 baixo o suficiente para aumentar o período necessário entre envios de velocidade. A uma baixa velocidade de avanço, possibilita-se, por exemplo, que haja tempo suficiente para que os algoritmos de detecção de portas não as percam entre um frame de imagem e outro analisados, uma vez que o robô não irá ultrapassar as portas antes que possam ser detectadas. Assim, em experimentos, foi verificado que a baixa velocidade de $v_1 = 0,1\ m/s$ satisfazia as condições impostas pelas restrições em tempo-real do sistema do robô.

Quando a visão computacional detectar que, nas imagens, não há nenhum objeto à esquerda do robô que possa vir a ser classificado como uma porta, não há necessidade de utilizar o algoritmo de detecção de portas. Ao se ter menos processos para serem escalonados, menor pode ser o período dos processos temporários (como a velocidade linear). Dessa forma, afim de otimizar a tarefa quanto ao tempo total de execução, definiu-se que a velocidade de avanço do robô poderia ser aumentada nesses casos, tornando o sistema autoadaptativo quanto ao parâmetro velocidade linear. Assim, pode-se afirmar que a autoadaptatividade deste comportamento é dependente da visão computacional, que será feita pelo nó *analyze_ adaptation* da Camada de Adaptação. Este nó estará constantemente verificando nas imagens se o robô está passando por um trecho onde não existem objetos na parede branca, e a resposta estará sendo publicada no tópico */analyzed_ adapt* através de uma mensagem booleana: verdadeiro para "objeto detectado" e falso para "parede sem objetos", como mostrado na Figura 3.5(a). Assim, o comportamento de navegação pode alterar a velocidade linear com base na mensagem recebida, adaptando-se à situação detectada. Define-se que essa velocidade linear máxima será de $v_2 = 0,2\ m/s$.

Quando o robô desloca-se com a velocidade linear máxima v_2, a trepidação resultante deixa as imagens trêmulas, mas como essa velocidade é alcançada somente quanto o robô encontra-se frente a parede sem portas, não há redução na qualidade do reconhecimento.

No capítulo de resultados, serão apresentados casos em que ocorrem essa adaptação da velocidade do robô, de acordo com a resposta obtida pela visão computacional para detecção de paredes.

3.3.3.2 Chamado recursivo ao algoritmo de ajuste orientação

Para que o robô consiga manter uma distância média da parede durante todo o percurso, é necessário definir quando que o algoritmo de ajuste de orientação será escalonado pelo sistema em tempo-real, assim como as condições necessárias para que ocorra.

A primeira condição é clara: o robô só poderá utilizar as leituras de distância à parede do sensor laser no algoritmo caso exista parede disponível para este procedimento. Será impossível o robô se alinhar à parede caso exista algum objeto como bebedouro, lixeira ou uma porta aberta no local onde as leituras estão sendo feitas. Assim, é necessário que o sistema do robô seja capaz de identificar que a superfície à sua esquerda é lisa, a partir dos mesmos dados lidos no tópico */analyzed_ adapt*, utilizada para a finalidade descrita na Seção 3.3.3.1. Assim, para fazer o ajuste

de orientação, é necessário que o robô detecte a existência de superfície lisa à sua esquerda.

Na Seção 3.3.1, foi definido que o robô estabelece seu ângulo baseando-se no valor da razão D_f/D_t do algoritmo *parallelize()*, que representa o quão paralelo o vetor de orientação está em relação à parede, onde o valor desta razão sendo igual a 1 significa alinhamento perfeito. Porém, o algoritmo irá buscar a razão $D_f/D_t = 1,018 \pm 0,04$, para compensar o desalinhamento das suas rodas. Essa escolha garante com que o robô mantenha a distância de 2,20m da parede entre os instantes de ajuste. É possível definir a distância mínima a ser percorrida pelo robô antes que seja necessário chamar o algoritmo por $P_{min} = 3,0$ metros. Esta é a segunda condição imposta.

Devido O algoritmo do nó de navegação seguindo a parede pode ser definido:

Algoritmo 2: Algoritmo do nó *follow_ line.*

```
1  início
2      Verificar ativação pelo tópico /enable;
3      Contador de deslocamento c_d = 0;
4      while não chegar ao fim do corredor do
5          if somente parede à esquerda then
6              Avançar com velocidade linear = 0,2 m/s;
7          else
8              Avançar com velocidade linear = 0,1 m/s;
9          end
10         if c_d > P_min e somente parede à esquerda then
11             parallelize() ;
12             c_d = 0;
13     end
14 fim
```

Como pode ser observado na descrição do algoritmo acima, a execução desse nó será encerrada quando for detectado pelo sensor laser que existe um obstáculo à frente do robô, indicando a cerca ou parede localizada no final do corredor. Esta funcionalidade é responsabilidade do nó de analise do sensor laser, descrito na Seção 3.3.4.

3.3.4 Componente de Percepção de Analisar Corredor com Sensor Laser

Este nó, nomeado de *analyze_ laser* e pertencente à Camada de Aplicação, é responsável por utilizar o sensor de distância a laser para analisar no corredor possíveis portas abertas e também identificar o fim do corredor para finalizar todos os outros processos. Este processo é assinante e publicante do tópico */enable*, uma vez que a mensagem de ativação enviada pelo nó *set_ distance* o ativa e também é responsável por enviar uma mensagem de desativação para todos outros nós. Também é publicante do tópico */scanned_ door*, para onde constantemente envia mensagens boo-

leanas sobre escaneamento de portas abertas.

O funcionamento destas detecções é similar para os dois casos. São selecionadas seções circulares da área escaneada pelo sensor laser, de acordo com a Figura 3.15.

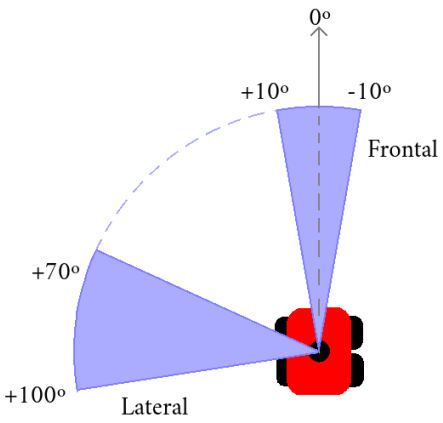


Figura 3.15: Seções circulares da área varrida pelo sensor de distância a laser para análises do algoritmo. A seção lateral para analisar paredes e portas abertas do corredor, e a seção frontal para identificar o fim do corredor. Fonte: o autor.

A funcionalidade de detectar portas abertas serve como um suporte para a visão computacional que também faz a mesma detecção. Devido as falsas detecções positivas de portas abertas, a maneira encontrada para validar essas indicações foi através dos dados do sensor. Para realizar essas detecções, escolhe-se uma seção circular na lateral esquerda do robô, que vai de $+70°$ até $+100°$ em relação ao vetor correspondente à orientação do Pioneer, equivalente à 120 leituras do sensor (resolução de 4 $leituras/°$). A partir dessas leituras, o valor médio delas D_{med} será analisado. Sabe-se que a parede está afastada à 2,20m do robô com uma tolerância de $\pm 0,30m$ (valor experimentalmente definido porque, com 2,50m, o robô fica muito perto da cerca a direita, e com menos 1,90 as portas não cabem na imagem da câmera), então espera-se que quando o robô passar em frente a uma porta aberta, D_{med} seja maior que uma distância mínima d_{min}, tal que $d_{min} > 2,20m$.

Para encontrar o valor ideal de d_{min}, que indica uma porta aberta, utilizou-se o método da tentativa e erro para valores maiores que $2,20m$ que apresentassem os melhores resultados. Com o robô localizado ao lado das salas abertas, as leituras do sensor na direção do interior da sala indicavam valores mínimos de, em média, 1,0 metro a mais que a distância da parede. Assim, incluindo uma margem de segurança, verificou-se que a escolha de $d_{min} = 2,80m$ era satisfatória para realizar as detecções. Dessa forma, resume-se: Caso verifique-se que $D_{med} > 2,80m$, pode-se afirmar que há uma porta aberta ao lado. Caso contrário, afirmativa falsa. Essa resposta ocorrerá na forma de uma mensagem booleana constantemente publicada no tópico */scanned_ door*, acessada pelos demais nós.

No procedimento para detectar o final do corredor, a seção circular analisada tem 20°, equivalente a 80 leituras, e é localizada simetricamente à frente do robô. Faz-se uma média dessas 80

leituras e armazena-se numa variável F que indica a distância até o fim do corredor. Define-se também que o fim da tarefa ocorrerá quando o robô estiver de frente para um obstáculo (cerca ou parede) a 1 metro de distância ($F \leqslant 1,0m$), o que resultará na publicação de uma mensagem booleana de valor falso no tópico */enable*.

Por fim, o algoritmo completo deste nó *analyze_ laser* é mostrado no *Algoritmo* 3:

Algoritmo 3: Algoritmo do nó *analyze_ laser*.

```
1  início
2      Verificar ativação pelo tópico /enable;
3      SecaoLateral = vetor da seleção de 120 leituras à esquerda do sensor;
4      D_med = média(SecaoLateral); SecaoFrontal = vetor da seleção de 80 leituras à frente
         do sensor;
5      F = média(SecaoFrontal); while F > 1,0m do
6          if D_med > 2,80m then
7              Publicar (Verdadeiro) em /scanned_ door;
8          else
9              Publicar (Falso) em /scanned_ door;
10         end
11     end
12     if F ⩽ 1,0 then
13         Publicar (Falso) em /enable;
14 fim
```

3.3.5 Componente de Percepção para de Detecção de Portas em Visão Computacional

Este componente da Camada de Aplicação é realizado por 4 nós, conforme explicado na Seção 3.2.1. Para relembrar, uma das propriedades autoadaptativas do sistema tem origem no fato de que a luminosidade do ambiente tem influencia sobre as técnicas usadas no processamento das imagens para detectar as portas abertas e fechadas. Assim, foi estabelecida uma classificação para cada luminosidade ambiente, onde as 3 classes são nomeadas por **sol**, **sombra** e **noite** (observe a Tabela 3.1), cujos nós são respectivamente denominados por *find_ doors_ sun*, *find_ doors_ shadow*, *find_ doors_ night_ default* e *find_ doors_ night_ equalized*. Lembre-se que foi determinado que, apesar de serem 3 classes, serão necessários dois nós para a classe noite. Dentre os quatro, apenas um dos nós é ativo por vez, e este é escolhido pelo componente da Camada de Adaptação, seguindo as regras de adaptação explicadas na Seção 3.3.6.

Duas metodologias diferentes foram utilizadas para detectar portas fechadas e abertas nas imagens. Para encontrar as portas fechadas, foi utilizado o método de Classificação em Cascata disponibilizado pelo *OpenCV* (Seção 2.6.2), capaz de realizar detecção de objetos em imagens a partir de um classificador treinado com *deep learning*. Já para as portas abertas, implementou-

se um algoritmo baseado na aplicação da técnica *Thresholding* (Seção 2.6.1) e segmentação nas imagens.

Esta seção está dividida da seguinte forma: as duas primeiras subseções apresentam, ainda de uma forma geral, os algoritmos e as técnicas utilizadas para a detecção de portas fechadas e de portas abertas. As seções seguintes relatam particularidades dos algoritmos implementados. A seção final trata com brevidade a união dos dois algoritmos de detecção de portas nos nós.

3.3.5.1 Técnicas Utilizadas para Detecção de Portas Fechadas

Um Classificador em Cascata é usado para detectar portas fechadas porque o objeto buscado não possui grande variedade: as portas do corredor são sempre metálicas, possuem uma janela de vidro na parte superior e venezianas na parte inferior, como é possível observar na Figura 3.16. Uma das vantagens encontradas em utilizar este método é que a detecção das portas não depende da luminosidade do ambiente, desde que tenham sido utilizadas várias imagens com todos os tipos de luminosidade para o treinamento do classificador. Sendo assim, a maior dificuldade encontrada para a implementação deste método encontra-se justamente no treinamento deste classificador, uma vez que uma alguns milhares de imagens são necessários para se ter resultados mais precisos. Após obter o classificador treinado, basta configurar alguns parâmetros utilizados no código da execução.

Figura 3.16: Algumas fotos tiradas de portas fechadas. Fonte: o autor.

Para o treinamento, foram utilizadas mais de 2000 imagens positivas, como as acima, e cerca de 4000 imagens negativas, que são as que não possuem portas (detalhes de como realizar o treinamento pode ser visto em [15]). Com o classificador pronto, foram realizados os primeiros testes no programa em *python* e verificou-se que, ao utilizar os valores *default* dos argumentos da função de detecção[16] do OpenCV, eram gerados falsos negativos e falsos positivos indesejados. Algumas alterações nos parâmetros elevaram consideravelmente a taxa de acerto, porém não resolveram totalmente a questão dos falsos positivos (detecção de outros objetos que não são portas), como

mostra a Figura 3.17.

Figura 3.17: Exemplos de falsos positivos detectados. Fonte: o autor.

Para contornar essas falsas detecções, foram impostas as seguintes condições:

1. Como as portas estão localizadas sempre nas mesmas posições nas imagens capturadas, delimitou-se a região em que uma porta pode ser detectada. Essa delimitação é imposta ao criar a condição de que o canto superior esquerdo da região retangular que marca a região da porta detectada na imagem esteja dentro de limites mínimo e máximo. Ou seja, a detecção do Classificador só poderia ser uma porta fechada se o canto superior esquerdo do *boundbox* tivesse (x_0, y_0) tal que $X_{min} \leq x_0 \leq X_{max}$ e $Y_{min} \leq y_0 \leq Y_{max}$, de acordo com a Figura 3.18(a).

2. Foi observado também que os falsos positivos, na grande maioria das vezes, não aconteciam em frames consecutivos como as detecções das portas fechadas, e sim em frames intercalados, de forma que a *boundbox* apareça piscando nas imagens. Assim, definiu-se que era necessário um valor mínimo de frames consecutivos para caracterizar a detecção de uma porta, numa variável c_{min}. Para definir esse valor mínimo, foi necessário considerar a quantidade de processos rodando simultaneamente no sistema, o que irá diminuir a frequência de análise de cada frame. Sabendo que o robô está em movimento e que, de acordo com o item 1 acima, existe uma região delimitada na horizontal para validar essa detecção, foi necessário escolher um valor para c_{min} pequeno o suficiente para garantir que as detecções consecutivas ocorram e grande o suficiente para filtrar os falsos positivos. Dessa forma, em todos experimentos foi verificado que $c_{min} = 10$ frames satisfaria as necessidades, independente da condição de luminosidade.

3. Por último, pensando nos casos onde as duas condições acima fossem satisfeitas e ainda assim ocorresse um falso positivo, tornou-se necessária uma última etapa de verificação. Assim que a contagem de detecções consecutivas em frames alcançar $c_{min} = 10$, o último segmento de imagem contido dentro do *boundbox* é levado para uma análise: aplica-se um *threshold* neste segmento de imagem evidenciando as venezianas da porta, que serão mostradas na forma de faixas horizontais pretas e brancas intercaladas, como mostra a Figura 3.18(a). Assim, a terceira condição imposta é que esse padrão seja verificado na imagem.

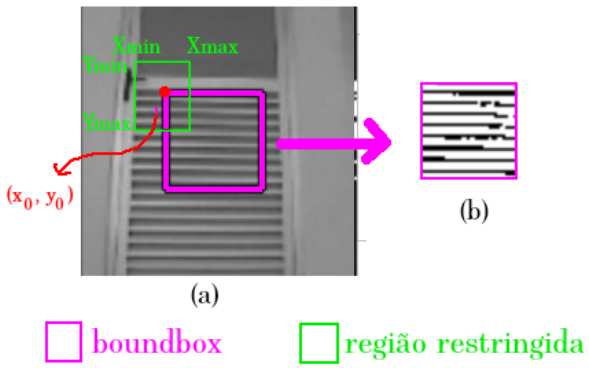


Figura 3.18: Método complementar para eliminação de falsos positivos das detecções de portas fechadas. Em (a), a condição do item 1. Em (b), a condição do item 3. Fonte: o autor.

Foi verificado que as detecções corretas das portas cumpriam essas condições na grande maioria das vezes, quase em 100%, e praticamente todos os falsos positivos detectados também eram filtrados. Os resultados obtidos na execução deste algoritmo podem ser vistos no próximo capítulo, de Análise de Resultados.

Por fim, assim que uma detecção é realizada, o nó programado envia no tópico */found_ doors* um sinal de "porta fechada detectada".

Este método do Classificador de objetos não sofre influência da luminosidade, o que não é o caso para o algoritmo de detecção de portas abertas.

3.3.5.2 Técnicas Utilizadas para Detecção de Portas Abertas

Para a implementação desse algoritmo, não foi considerado válido utilizar um novo Classificador em Cascata para encontrar as portas abertas nas imagens pelo motivo de que o treinamento seria muito mais custoso devido à este objeto "porta aberta" ter inúmeras variações possíveis: as salas no interior das portas são diferentes umas das outras, luz desligada ou apagada, pessoas em pé próximas a entrada, etc. É necessário um algoritmo que utilize outras técnicas e ferramentas de processamento de imagem disponibilizadas pelo *OpenCV*.

O algoritmo consiste em primeiramente aplicar um *threshold* na imagem com o objetivo de separar a porta aberta do restante dos elementos do corredor (parede, chão, etc) baseando-se na diferença de luminosidade entre estes elementos. Por exemplo, em uma imagem adquirida de uma porta aberta durante o dia onde a luz da sala está apagada, é esperado que, ao binarizar a imagem, o interior da sala fique majoritariamente em cor preta e a parede do lado de fora fique de cor branca. Quando a porta estiver centralizada no meio da imagem, o normal é que se tenha uma larga faixa preta na vertical, cercada pelos lados com o branco da parede. Para constatar esse padrão em termos de programação, basta então selecionar os segmentos de imagem, ou melhor, retângulos que correspondem à porta e as paredes na esquerda e na direita, e então verificar a quantidade de pixels pretos e brancos nesses segmentos. A Figura 3.19 ilustra o procedimento.

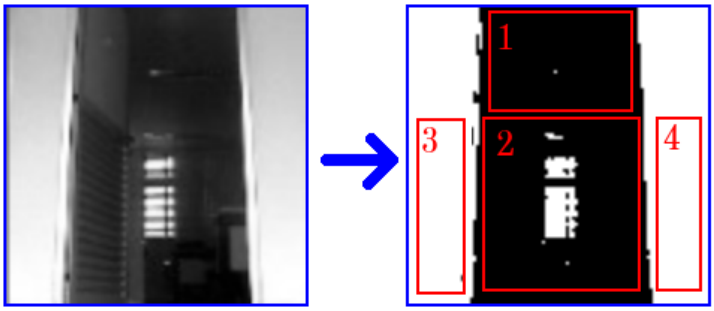


Figura 3.19: Método geral para detectar portas abertas em imagem]. Em (a), a imagem em tons de cinza. Após aplicar o *threshold*, em (b), seleciona-se os segmentos de imagem da porta, 1 e 2, e das paredes, 3 e 4. Para a porta, é esperado que os pixels de 1 e 2 sejam majoritariamente de cor preta, enquanto 3 e 4 espera-se que sejam brancos. Fonte: o autor.

Dessa forma, seguindo este algoritmo, para que ocorra a detecção de uma porta aberta, é necessário definir:

1. o valor do limiar T do *threshold*;
2. os limites dos segmentos retangulares na imagem;
3. e os valores das porcentagens máximas/mínimas de pixels de uma cor para cada um dos retângulos para validar os segmentos retangulares.

Para cada retângulo em cada frame de imagem processado, calcula-se então as porcentagens de cor preta ou branca pela equação

$$\%\text{ de pixels pretos (ou brancos)} = \frac{\text{quantidade de pixels pretos (ou brancos)}}{\text{quantidade total de pixels}} \times 100\% \qquad (3.1)$$

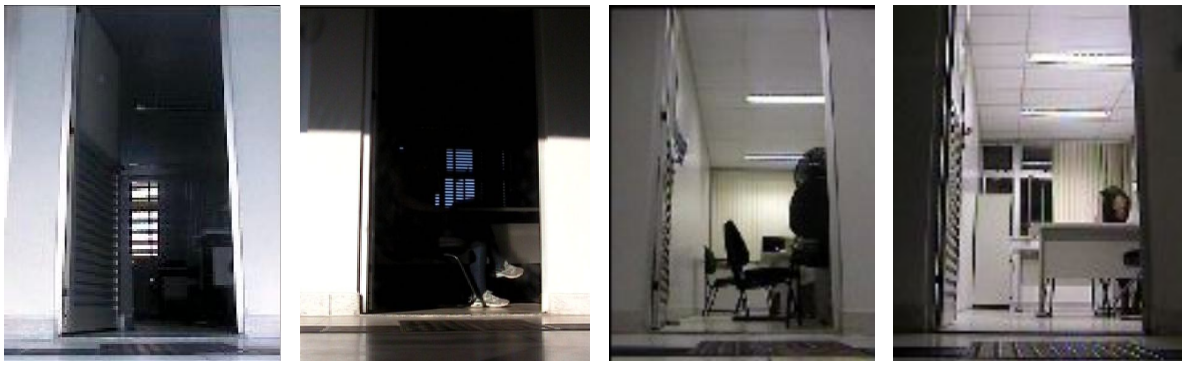
e, por fim, compara-se esses valores com os limites definidos no item (2).

Uma consequência problemática deste procedimento é que todos os valores definidos acima são dependentes da luminosidade do ambiente. Quando a sala está escura, é fácil diferenciar da parede. Porém existem situações em que o sol ilumina metade da imagem, situações onde é dia e a luz da sala está acesa, deixando-a com nível de luminosidade semelhante ao exterior da sala, ou mesmo situações em que é noite e as salas estão tão claras quanto o corredor ou mesmo mais claras. Todas essas situações fazem com que seja necessário ter um novo valor limiar para o *threshold*, definir novos limites mínimos/máximos de cor predominante nos segmentos retangulares selecionados, ou redefinir as áreas e quantidades desses segmentos.

Sabendo de todas essas variações para cada caso de luminosidade (e considerando que serão necessários procedimentos exclusivos e outras peculiaridades em cada um deles, a serem descritos posteriormente), percebeu-se a necessidade de ter um algoritmo específico para cada caso. Enquanto o robô avança, ele pode se deparar com situações diferentes de luminosidade, o que define

a necessidade de adaptação do sistema. Por exemplo, estar andando em um trecho claro e iluminado do corredor onde as salas abertas estão com luzes apagadas, e logo depois entrar em outro trecho menos iluminado do corredor, onde as salas já estão com luzes acesas. Dessa forma, pode-se afirmar então que a real origem da requisição de um sistema autoadaptativo vem da necessidade de adaptação dos algoritmos de detecção de portas abertas devido aos diferentes tipos de ambiente e suas respectivas luminosidades.

As variações possíveis de ambientes foram divididas em grupos: *dia com sol, dia com sombra* e *noite* (a princípio). Todos eles, mesmo com suas diferenças, seguirão o mesmo procedimento de aplicar-se a técnica *thresholding* e depois analisar os segmentos retangulares individualmente. A classe *dia com sombra* reúne as situações em que é dia, sem as paredes estarem sob iluminação solar. A classe *dia com sol* vale para trechos do corredor onde o sol reflete em parte da parede, dividindo a imagem horizontalmente à uma certa altura. A classe *noite* é válida para quando não há mais luz do dia, toda iluminação vem de luz elétrica. Observe a Figura 3.20 para efeitos de comparação.



(a) (b) (c) (d)

Figura 3.20: Portas abertas em (a) dia com sombra, (b) dia com sol e (c, d) noite. Fonte: o autor.

Para implementar cada um dos algoritmos, será necessário definir, nesta ordem:

1. O melhor valor para o limiar T do *threshold*, de forma a conseguir deixar a região das paredes toda branca e a região da porta com a maior quantidade de pixels pretos possível;
2. Quantos segmentos retangulares da imagem, correspondentes à porta aberta e as paredes, e a área desses retângulos;
3. Os limites mínimos/máximos de porcentagem de pixels brancos ou pretos contido nos segmentos;
4. Outras particularidades quaisquer.

Assim como na detecção de portas fechadas, nestes algoritmos é possível que ocorram também os falsos positivos e falsos negativos. Pensando nisso, o valor do limiar T do *threshold* deve ser escolhido de maneira a se eliminar ao máximo os falsos negativos, ou seja, que o maior número

possível de portas abertas sejam detectadas por este método. Isso significa que as portas nas imagens, em geral, estejam o máximo possível em cor preta sem que a parede deixe de estar toda branca, na imagem binária. Isso pode ser conseguido ao escolher um alto valor de T, que fará com que mais pixels pretos apareçam na imagem, porém isso tem um lado negativo: apesar de conseguir capturar as portas abertas, uma quantidade maior de objetos podem ser erroneamente detectados. Ou seja, por um lado está se eliminando a quantidade de falsos negativos porém aumentando os falsos positivos.

Pensando nisso, surgiu a ideia de utilizar o sensor laser como suporte para essas detecções, o que justifica a existência do nó *analyze_laser*. Uma detecção em imagem de uma porta aberta sempre estará acompanhada de uma detecção pelo sensor laser. As duas indicações juntas praticamente eliminam falsas portas abertas detectadas. Dessa forma, por exemplo, uma lixeira ou um bebedouro que seja aprovado pelo processamento de imagem, será reprovado pela análise do sensor de distância.

Por fim, assim que o nó *find_doors* encontrar uma porta, um sinal de "porta aberta detectada" será enviado para o tópico */found_doors*. Essa mensagem será recebida então pelo nó de mapeamento, juntamente com a identificação do laser através de outro tópico. Assim, será possível gerar a resposta correta para as detecções.

As subseções a seguir descreverão as peculiaridades de cada um dos tipo de nós implementados.

3.3.5.3 Nó de detecção para: Dia com Sombra

Este nó, *find_doors_shadow*, será utilizado quando a visão do robô detectar que for dia e que não há sol refletindo nas paredes. Não depende se o dia está nublado ou não, se chove, ou se as luzes das salas estão acesas ou apagadas.

Para detectar portas fechadas, basta deixar o Classificador trabalhar junto com as condições determinadas na subseção anterior 3.3.5.1. Já para as portas abertas, ao se aplicar o *threshold* nas imagens, espera-se que as paredes fiquem brancas e a parte da sala seja composta, em sua maioria, por pixels pretos, sendo que os brancos são causados por luzes acesas, janela aberta no interior da sala, ou objetos mais claros. Observe os exemplos da Figura 3.21.

Figura 3.21: Portas abertas na situação de dia com sombra. Fonte: o autor.

Após analisar vários casos, definiu-se os valores ideais para os parâmetros:

- Limiar do *threshold*: $T = 110 \rightarrow$ este valor foi o que conseguiu apresentar o padrão desejado, onde as paredes do corredor ficam brancas e o interior da sala fica em preto com algumas partes brancas.
- 4 segmentos retangulares: parede à esquerda, parede à direita, parte de cima da porta e a parte de baixo.
- Para cada um dos 4:
  - Segmentos de parede: limite mínimo de pixels brancos = 90%
  - Segmento da parte de cima da porta: limite mínimo de pixels pretos = 60%
  - Segmento da parte de cima da porta: limite mínimo de pixels pretos = 40%

Estas definições ao serem aplicadas nos exemplos das imagens da Figura 3.21, geram as imagens da Figura 3.22:

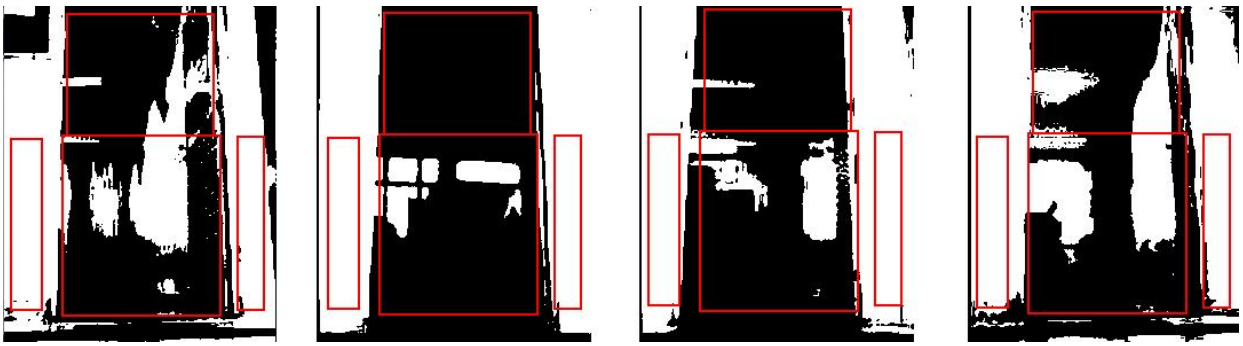


Figura 3.22: Thresholding e segmentos retangulares no algoritmo de detecção de portas abertas na sombra. Observe que os retângulos nas laterais capturam a parede em branco, enquanto os retângulos das portas possuem sua maior parte em preto, obedecendo as porcentagens mínimas definidas. Fonte: o autor.

3.3.5.4 Nó de detecção para: Dia com Sol

O nó *find_ doors_ sun* foi implementado para funcionar em trechos do corredor onde há sol, a luz do sol divide a parede na horizontal de forma que a metade de cima esteja com sombra e a debaixo fica iluminada. Percebeu-se que ter sombra em metade da parede afeta negativamente a detecção de portas fechadas, uma vez que o Classificador não foi treinado com imagens de portas com metade sombra e metade sol. Assim, determinou-se que o primeiro passo a ser feito em uma imagem de sol seria recortá-la no meio horizontalmente, removendo a metade superior (os 40% de cima) e utilizando apenas a inferior para análise (os 60% debaixo da imagem original). Observe o procedimento na Figura 3.23.

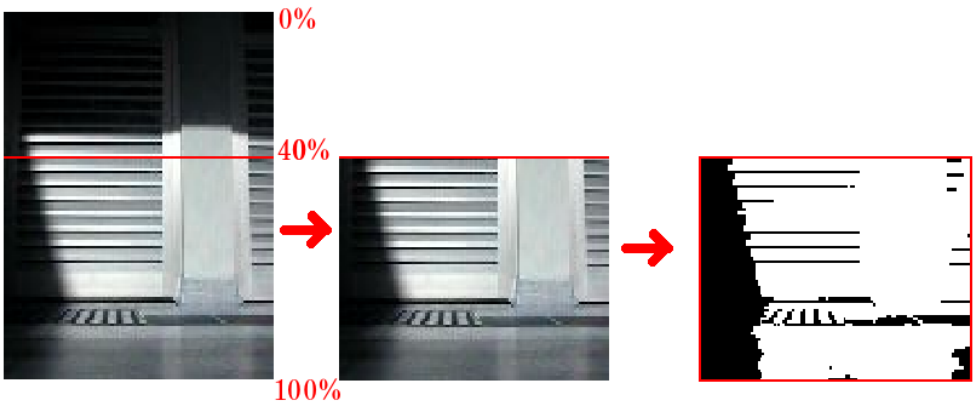


Figura 3.23: Recorte da imagem para eliminar a parte com sol. Fonte: o autor.

Após fazer o recorte na imagem, aplica-se o *threshold* e os segmentos retangulares definidos da seguinte forma:

- Limiar do *threshold*: $T = 100$
- 3 segmentos retangulares: parede à esquerda, parede à direita, e porta.
- Para cada um dos 3:
  - Segmentos de parede: limite mínimo de pixels brancos = 90%
  - Segmento da porta: limite mínimo de pixels pretos = 65%

Um problema encontrado neste algoritmo é justamente causado pelo sol. Em um momento de um dos corredores existe uma pilastra que é iluminada pela sol gerando uma sombra na parede, que pode fazer o algoritmo a confundir com uma porta aberta. Porém, caso essa falsa detecção venha a acontecer, existe a detecção também feita pelo sensor laser, que será analisada em conjunto com a detecção de imagem no nó *create_ map*, reprovando a falsa detecção de porta aberta. Um outro problema encontrado é quando a transição sol/sombra (na vertical) acontece justamente em cima de uma porta, o que pode fazer com que elas passem despercebidas. A Figura 3.24 traz exemplos destes dois casos.

Figura 3.24: Possíveis falsos positivos e falsos negativos no algoritmo. Na imagens da esquerda, a sombra de uma pilastra semelhante a uma porta aberta na imagem binária. Nas imagens da direita, a transição sol-sombra ocorrendo sobre uma porta, influenciando na detecção do algoritmo. Fonte: o autor.

3.3.5.5 Nó de detecção para: Noite

Para o período da noite, este algoritmo não precisa de alterações quanto à detecção de portas fechadas, se comparado com o algoritmo para funcionar durante o dia pela sombra. O objeto "porta fechada" é facilmente encontrado pelo mesmo Classificador. Porém já para detectar portas abertas, o fato das salas estarem com as luzes acesas já faz com que o padrão de pixels pretos e brancos nas imagens binarizadas pelo *thresholding* seja invertido: as paredes ficam escuras e os interiores das salas tendem a ficar mais claros, como mostrado na Figura 3.25.

Figura 3.25: Portas abertas no período da noite e aplicação do *thresholding*. Fonte: o autor.

Durante as análises para se definir o valor dos parâmetros de segmentação de imagem, houve uma dificuldade em encontrar um valor para o limiar T que satisfizesse todos os casos de segmentação das portas abertas durante a noite. Isso ocorria devido a grande possibilidade de combinações de luminosidade devido a quantidade e intensidade das lâmpadas no interior das salas e combinadas com iluminação variável dos corredores, com trechos mais claros e outros mais escuros.

Para a imagem binária mostrar as paredes em preto e o interior das salas de porta aberta majoritariamente em branco, era preciso definir um limiar T mediano que seja: alto suficiente para deixar as paredes iluminadas do corredor em cor preta, e baixo suficiente para que a sala e os objetos no interior fiquem brancos (lembrando que valores altos de T implicam em aumentar a quantidade de pixels brancos e valores menores implicam em aumentar a quantidade de pixels pretos). Encontrou-se $T = 150$. No entanto, aconteciam situações em que a sala não estava iluminada o suficiente para atender a requisição de ficar branca na imagem binária.

Pensou-se então em utilizar uma técnica chamada de Equalização de Histograma[17], comumente utilizada (na imagem em níveis de cinza, antes do *thresholding*) quando se deseja aumentar o contraste e evidenciar detalhes pouco visíveis a partir da distribuição dos valores de ocorrência dos níveis de cinza da imagem. Aplicando este método e depois o *thresholding*, percebeu-se que o aumento do contraste resultava em um maior clareamento das regiões mais iluminadas, fazendo com que aparecessem mais a cor branca na imagem binária. Assim, as portas que antes não apareciam bem passaram a aparecer, porém a parede do corredor também ficou parcialmente branca (devido a iluminação externa), o que ainda não resolvia o problema uma vez que necessita-se que as paredes estejam em preto para classificar uma porta aberta na imagem. A solução encontrada foi aumentar o valor de limiar da binarização novamente, para $T = 220$ e o resultado obtido foi

satisfatório: as paredes apareceram mais em preto, sem que as portas deixassem de ficar brancas. A Figura 3.26 a seguir mostra o resultado desta técnica para estes casos.

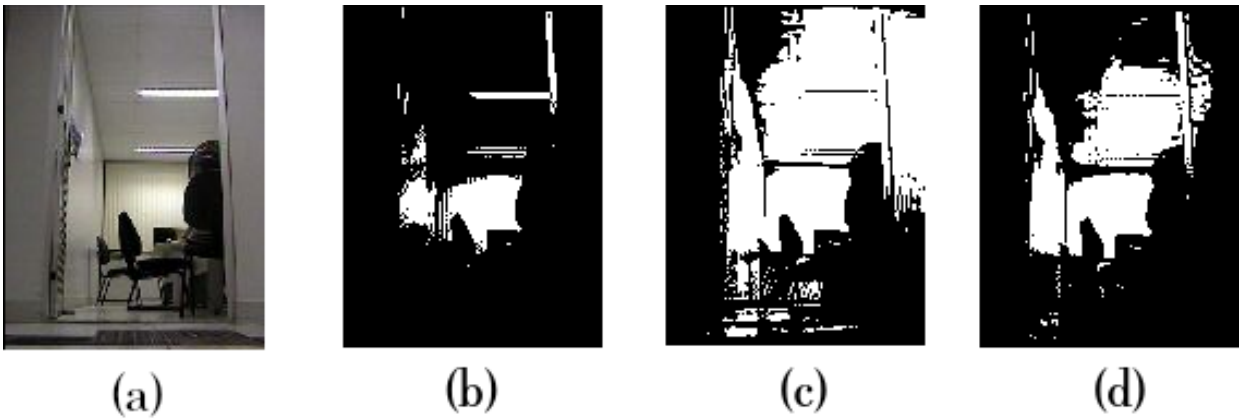


Figura 3.26: Implementação do segundo algoritmo para detectar portas à noite. Em (a), a imagem original. Em (b), aplica-se o *thresholding* com $T = 150$, porém a porta ficou praticamente imperceptível. Em (c), utilizou-se a equalização de histograma antes de aplicar o mesmo *threshold* na imagem, o que clareou o interior da sala, porém a parede do corredor também. Para melhorar a detecção, em (d) mudou-se o limiar do *thresholding* para $T = 210$, tornando possível a percepção da porta pelo algoritmo. Fonte: o autor.

Essa nova técnica resolve os problemas desses casos encontrados, porém, para os demais casos, como já dito, deixava a parede muito clara (como mostrado na imagem da Figura 3.26(c)). A solução encontrada foi implementar **dois algoritmos diferentes para a noite**, onde um dos dois será chaveado para realizar as detecções, uma vez que cada um dos dois procedimentos apresenta vantagens e desvantagens:

- Para $T = 150$ e sem equalização de histograma: detecta boa parte das portas, porém algumas passam despercebidas. Não deixa paredes em branco, evitando falsas detecções. $\downarrow$ Falsos positivos e $\uparrow$ falsos negativos.
- Para $T = 210$ e com equalização de histograma: detecta praticamente todas as portas, porém o fato de as paredes aparecerem com manchas brancas faz o algoritmo realizar várias detecções incorretas. $\downarrow$ Falsos negativos e $\uparrow$ falsos positivos.

Tendo os dois procedimentos definidos para os nós de detecções noturnas, é possível estabelecer os parâmetros para realizar as identificações de portas abertas:

- Limiar do *threshold*: $T = 150$ (normal) e $T = 210$ (equalizado)
- 3 segmentos retangulares: parede à esquerda, parede à direita, e porta (região central).
- Para cada um dos 3:
  - Segmentos de parede: limite mínimo de pixels brancos = 90%
  - Segmento da porta: limite mínimo de pixels pretos = 65%

A Figura 3.27 mostra que os retângulos nas laterais pretendem capturar a parede em cor preta, e o retângulo do meio capturar o interior da sala majoritariamente em cor branca, com partes em preElse

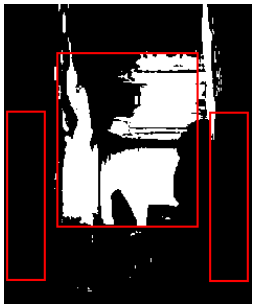


Figura 3.27: Segmentos retangulares para os dois nós de detecção noturna. Fonte: o autor.

3.3.5.6 Visão Geral

Conforme mostrado na subseção anterior, dois nós foram implementados para trabalhar durante a noite, totalizando assim 4 nós que correspondem ao Componente de Percepção para Detecção de Portas: um para o dia em trechos do corredor com sombra, outro para trechos de sol, e dois para a noite. Cada um deles, mesmo implementado de formas diferentes, devem entregar o mesmo resultado para o sistema: detecções de portas fechadas e de portas abertas. Como o processamento das imagens é feito frame a frame, os dois tipos de porta serão buscados em cada um desses frames e as que forem encontradas vão gerar uma publicação instantânea no tópico */found_ doors*.

Os quatro nós implementados compõem a parte autoadaptativa do sistema: a componente responsável por gerenciar as adaptações (descrita na Seção 3.3.6 a seguir) irá selecionar um dentre os quatro nós para executar a funcionalidade de detecção das portas em visão. Dessa forma, cada um dos quatro nós foi implementado com uma *flag* de ativação, que é o valor da mensagem publicada no tópico */analyzed_ adapt* pelo nó responsável por gerenciar a autoadaptatividade do sistema. A situação que for identificada resultará em uma mensagem de valor *verdadeiro* para o algoritmo escolhido, enquanto os outros três receberão *falso*.

O algoritmo resultante destes nós fica da seguinte forma:

Algoritmo 4: Algoritmo dos nós *find_ doors*.

```
1  início
2      Verificar ativação pelo tópico /enable;
3      while não chegar no fim do corredor do
4          if flag == Verdadeiro then
5              Receber frame;
6              Buscar porta fechada;
7              if detectou porta fechada then
8                  Publicar ("Fechada") em /found_ doors;
9              else
10                 Buscar porta aberta;
11                 if detectou porta aberta then
12                     Publicar ("Aberta") em /found_ doors;
13             end
14     end
15 fim
```

3.3.6 Componente de Gerenciamento da Autoadaptatividade

O componente de percepção responsável por atribuir ao sistema do robô a capacidade de autoadaptação em tempo de execução é implementado no nó *analyze_ adaptation*. Esse único componente da Camada de Adaptação tem como principais atribuições de, enquanto o robô avança, identificar a luminosidade do ambiente para adaptar o Componente de Detecção de Portas em Imagem e identificar os trechos do corredor onde o robô pode navegar a uma velocidade menor ou maior, como mostrado na Figura 3.9.

Essa gerenciamento é resultante de um processo dividido em três etapas: a primeira é a etapa de coleta, responsável por monitorar as informações de interesse na Camada de Aplicação e extrair as *features* relevantes para serem então analisadas. A segunda etapa é a de análise das políticas de adaptação, que são um conjunto de regras e lógicas que avaliam os dados coletados para se verificar o requerimento de adaptação. Caso a necessidade de adaptação seja identificada, o último estágio é o responsável por de fato executar essa adaptação no sistema, por meio de alterações nas variáveis de controle.

Como o sistema possui dois componentes da Camada de Aplicação passíveis de adaptação (nó de navegação e nó de detecção de portas), o ideal é que se tenha dois processos rodando simultaneamente para cada um deles, e cada processo sendo composto pelos três estágios descritos anteriormente. Para a adaptatividade do componente de detecção de portas, os dados monitorados são as imagens da câmera, onde serão feitos processamentos para se identificar as situações de luminosidade. Da mesma forma, para a adaptatividade do parâmetro velocidade linear no comportamento de navegação, as informações relevantes são também extraídas das imagens, onde um processamento irá identificar trechos no corredor onde se vê apenas paredes, ou melhor, onde

possivelmente não há portas. Como os dois componentes adaptativos serão avaliados a partir de dados extraídos de imagem, os dois processos de gerenciamento de adaptação terão um único estágio de coleta, com uma entrada, que é o frame de imagem, e duas saídas: "situações de luminosidade" e "parede com ou sem objetos"). A Figura 3.28 mostra como está estruturado o nó *analyze_ adaptation* e como se comunica com os componentes da Camada de Aplicação.

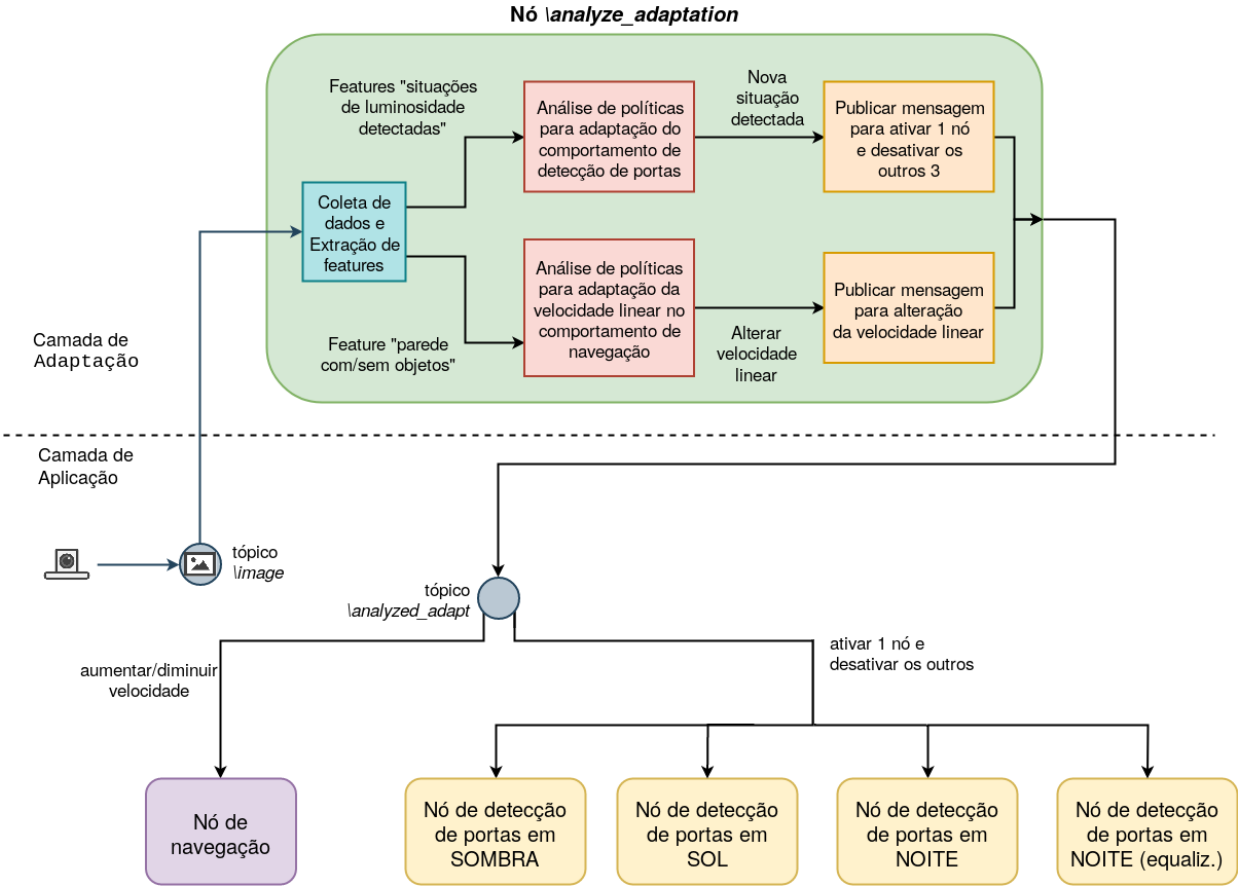


Figura 3.28: Estrutura do nó responsável pela autoadaptatividade do sistema na Camada de Adaptação e comunicação via tópicos com os demais nós da Camada de Aplicação. Fonte: o autor.

Nas subseções a seguir, serão apresentados detalhes de cada um dos dois segmentos adaptativos no nó *analyze_ adaptation*, seguindo uma abordagem de etapa por etapa.

3.3.6.1 Adaptatividade do componente de detecção de portas

Estágio 1: coleta e extração de dados.

O objetivo aqui é identificar uma ou mais possíveis situações de luz do ambiente. Foram criadas 4 variáveis booleanas para cada tipo de ambiente, onde *verdadeiro* significa "ambiente indicado" e *falso* significa "não indicado". Os métodos de processamento de imagem escolhidos se baseiam em utilizar imagens binárias, então o passo inicial é converter o frame para tons de cinza e então

aplicar o método de *thresholding*.

Ao serem feitas análises nas imagens adquiridas do robô antes da implementação, percebeu-se que, exceto pelo caso de identificação do ambiente "sol" (através da percepção da sombra na parte superior da parede), só era possível de fato identificar a luminosidade do ambiente que o robô se encontrava quando o mesmo passava em frente à uma porta aberta, pelo seguinte motivo: durante o dia, as salas abertas estão com luzes apagadas, enquanto, durante a noite, estão com as luzes acesas, então a diferença entre as intensidades da luz no corredor e da luz no interior da sala, em um determinado período do dia, fornecia os maiores indícios de qual é a luminosidade instantânea do ambiente. Todos os outros elementos do ambiente (parede, bebedouro, lixeira, portas fechadas) não apresentavam diferença nítida verificável por processamento de imagem para qualquer condição de iluminação e período do dia. A partir dessa constatação, definiu-se que o chaveamento entre os algoritmos de detecção de portas seria feito a partir da análise de luminosidade geradas pelas salas abertas no corredor.

Para identificar um ambiente iluminado pelo sol, basta verificar na imagem binária se a parede está dividida horizontalmente, onde a metade de cima é completamente preta e a de baixo está em branco. Para identificar as situações "dia com sombra" ou "noite", foram definidos *boundboxes* retangulares fixos, verticais e posicionados lado a lado, até que o padrão "branco, preto, branco" ou "preto, branco, preto" fosse encontrado, o que representa, respectivamente, uma porta aberta durante o dia ou durante a noite (é válido ressaltar que o intuito de usar aqui o mesmo método para detecção das portas abertas nos 4 nós *find_ doors*, não é de fato identificar esses objetos, e sim o ambiente em que o robô se encontra). Além disso, o padrão "preto, branco, preto" seria identificado para duas situações semelhantes, diferenciadas pelas porcentagens de cor preta ou branca neles, e cada um correspondente à um dos dois algoritmos de detecção de portas à noite (sem e com equalização de histograma, definidos na Seção 3.3.5.6).

Outro ponto levado em consideração é: por ser necessário que o robô esteja passando em frente a uma porta aberta para identificar a luminosidade do ambiente e selecionar o algoritmo correto para realizar a detecção do objeto, é necessário garantir que a mesma não seja ultrapassada pelo robô entre os instantes do chaveamento do algoritmo e da real detecção da porta, sendo perdida entre os frames de imagem analisados. Isso pode ocorrer devido ao tempo mínimo existente entre a indicação da adaptação pela imagem até a efetivação dessa adaptação no sistema, e esse tempo talvez possa ser suficiente para que o robô ultrapasse a porta sem que ela seja identificada nas imagens pelos nós de detecção.

A solução encontrada para esta problemática está ilustrada na Figura 3.29: definiu-se que os segmentos retangulares citados no parágrafo anterior (para identificar os ambientes a partir da quantidade de cor preta ou branca no interior dos retângulos) estariam localizados na imagem ao lado mais à direita possível, enquanto que os segmentos retangulares nos algoritmos de detecção de portas estariam localizados o mais a esquerda possível, visando assim permitir que haja tempo suficiente entre a indicação de adaptação e o chaveamento do algoritmo de detecção, sem que a porta passe despercebida.

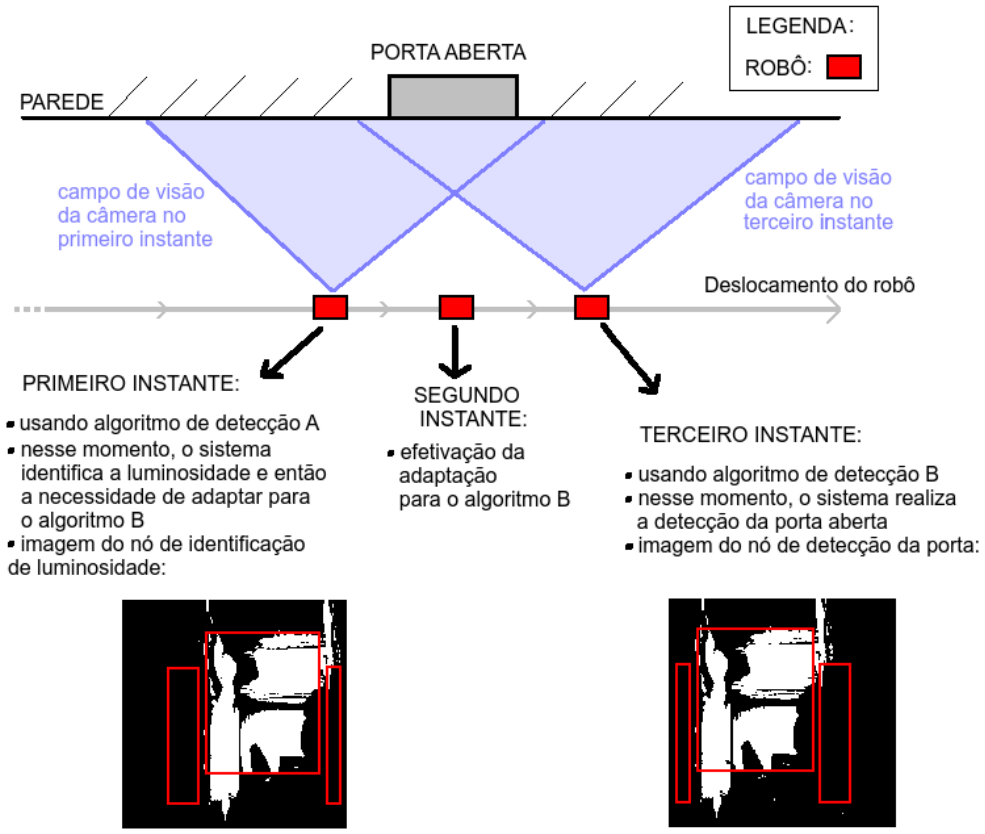


Figura 3.29: Ilustração do robô se deslocando pelo corredor, passando em frente a uma porta aberta. A Figura mostra a solução encontrada para garantir que haja tempo entre para adaptar o sistema (selecionar o nó de detecção de portas) e realizar a detecção da porta com o novo algoritmo. Observe onde estão localizados os retângulos vermelho nas imagens em preto e branco, a imagem da esquerda é do nó autoadaptativo (retângulos deslocados para a direita) e a imagem da direita é do nó de detecção de portas (retângulos deslocados para a esquerda). Fonte: o autor.

Pode acontecer de este estágio de coleta do nó autoadaptativo identificar mais de um tipo de luminosidade ao mesmo tempo, porém o próximo estágio, de análise, irá selecionar o algoritmo de detecção correto com base nas políticas de adaptação. Como são 4 algoritmos, então existem 4 variáveis booleanas, cada uma correspondente a identificação de um algoritmo de detecção, e elas são utilizadas no próximo estágio.

Estágio 2: análise das políticas de adaptação.

Para definir a lógica que seleciona uma dentre as 4 situações existentes, é necessário considerar não apenas um único grupo das 4 variáveis booleanas que chegam do estágio anterior, mas sim vários desses grupos que chegam ciclo após ciclo, pois as sequências e repetições de cada um desses valores compõem os termos das políticas de adaptação.

Outro fator importante levado em consideração é que ter apenas uma simples indicação pontual de adaptação não é suficiente, uma vez que essas detecções podem ser resultantes de falhas do algoritmo. Por exemplo, suponha que o robô está executando a tarefa e utilizando o algoritmo A para identificar portas abertas e fechadas, até que em um determinado momento a componente de verificação de adaptação, que está analisando imagem após imagem, em ciclos, indica em um único frame que o sistema deve se adaptar para usar o algoritmo B, e então nos frames seguintes volta a indicar o A. Essa ocorrência possui grandes chances de ser apenas uma falha do componente de verificação de adaptação, o que é comum de acontecer. Pensando nisso, foi estabelecido que, para se efetivar uma adaptação, será necessário um número mínimo de indicações consecutivas. Cada tipo de situação, correspondente a um algoritmo de detecção de porta, deverá cumprir uma quantidade mínima de indicações consecutivas, entre uma imagem e outra analisadas pelo nó de verificação de adaptação, para que essa adaptação seja considerada de fato verdadeira.

Na seção sobre o estágio anterior, definiu-se que o chaveamento entre os algoritmos de detecção de portas será realizado através da análise da diferença de luminosidade na imagem binária gerada por salas abertas em contraste com a parede do corredor. Assim, para evitar que ambientes não sejam detectados sem que o robô esteja próximo a porta de uma sala, definiu-se que o algoritmo não irá sequer tentar identificar a luminosidade do ambiente enquanto o robô não se aproximar de uma porta aberta, o que é possível detectar a partir das análises das leituras do sensor de distância a laser, pelo nó *analyze_ laser*. Observe que, na arquitetura da Figura 3.8, o nó *analyze_ adaptation* é assinante do tópico */scanned_ door* justamente por esse motivo.

Após estabelecer essas definições para a etapa de análise, foi possível então definir a lógica do algoritmo que de fato identifica a necessidade de adaptação a partir das 4 variáveis booleanas (uma para cada algoritmo) que chegam em ciclos, vindo do estágio anterior, onde *verdadeiro* é "ambiente identificado"e *falso* é "não identificado".

Afim de facilitar as referências aos algoritmos, a partir deste ponto os 4 nós serão chamadas pelas seguintes nomenclaturas: $Nó_{sombra}$, $Nó_{sol}$, $Nó_{noite-1}$ e $Nó_{noite-2}$ (onde $Nó_{noite-2}$ é o que utiliza equalização de histograma nas imagens). A política de adaptação fica definida da seguinte forma:

- Primeiramente, escolheu-se um dos nós para estar ativado quando o robô iniciar a execução da rotina principal da tarefa, no início do corredor, enquanto os outros três nós se iniciam desativados. Definiu-se o que $Nó_{sombra}$ seria o tal nó, e como a detecção de portas fechadas pelo Classificador em Cascata não depende da luminosidade, essa função do sistema não é afetada.
- Os quatro nós podem ser separados em dois grupos: *dia* e *noite*. Assim que ocorrer o primeiro

chaveamento para um nó de um dos dois períodos, elimina-se a possibilidade do sistema chavear posteriormente para um nó do outro período. Portanto, a primeira adaptação irá determinar o período definitivo, *dia* ou *noite*. Caso identifique-se *dia*, um nó do grupo *noite* não poderá depois ser selecionado, e vice-versa.

- Mínimo de indicações consecutivas para cada nó, determinadas experimentalmente:
  - $Nó_{sol}$: 0, porque a identificação de $Nó_{sol}$ é inconfundível e precisa. Basta identificar a sombra formada pelo sol cobrindo a parte superior da parede, como mostra a Figura 3.23.
  - $Nó_{sombra}$: 5, para caso seja a primeira adaptação do sistema, ou 0, se o período já tiver sido classificado com *dia* (porque o $Nó_{sombra}$ será selecionado quando não houver identificação do $Nó_{sol}$).
  - $Nó_{noite-1}$: 5, valor padrão igual ao do $Nó_{sombra}$.
  - $Nó_{noite-2}$: 9, um número maior porque o algoritmo pode detectar falsos positivos mas que não ocorrem tantas vezes consecutivas, então 9 foi a quantidade encontrada experimentalmente para garantir a escolha correta deste nó.

- Quando o sistema requirir a adaptação para o $Nó_{noite-2}$, foi determinado que sua permanência deve ser temporária devido à grande quantidade de falsas detecções de portas abertas que ele pode gerar. O chamado para este nó ocorre apenas para que possa detectar as portas abertas que o $Nó_{noite-1}$ não é capaz de identificar. Assim, determina-se que este nó permanece ativo por um tempo fixo, e então ocorre a readaptação para o $Nó_{noite-1}$.

- A identificação de novos ambientes (exceto no caso de $Nó_{sol}$) só acontecerá quando a mensagem recebida do tópico */analyze_ laser* for verdadeira, o que significa que o robô está ao lado de uma sala aberta, que é onde os ambientes podem ser identificados na imagem. Para o $Nó_{sol}$ essa condição não é necessária, a identificação do ambiente com luz solar direta pode ocorrer a qualquer momento.

- quando ocorrer a confirmação da detecção de um ambiente, uma mensagem será repassada para o próximo estágio, que irá executar a adaptação.

Estágio 3: administração da adaptação.

O último estágio funciona de forma simples: dentre os 4 ambientes, o que foi detectado pelo estágio de análise será convertido numa mensagem numérica que pode possuir valor de 1 a 4, que são números de identificação de cada um dos 4 algoritmos. Essa mensagem será publicada no tópico */analyzed_ adapt*, que então será recebida pelos quatro nós. O nó cujo número de identificação for igual ao da mensagem recebida pelo do tópico, irá se tornar o nó ativo para realizar a funcionalidade de detectar portas abertas e fechadas, enquanto os outros três irão se desativar, aguardando por serem ativos novamente. É dessa forma que ocorre o chaveamento entre os nós, tornando o sistema autoadaptativo.

3.3.6.2 Adaptatividade do componente de detecção de portas

Estágio 1: coleta e extração de dados.

Na primeira etapa o objetivo é analisar imagens binárias para identificar se ao lado esquerdo do robô há apenas parede, sem elementos que possam ser identificados como portas quando se enquadrarem na imagem, conforme o robô avança. O passo inicial é converter o frame para tons de cinza e então aplicar o método de *thresholding*. Depois, estabelece-se um segmento retangular fixo na horizontal, na parte debaixo da imagem, e a porcentagem de cor branca no interior do retângulo será calculada. Para classificar como "apenas parede", foi estabelecido que o mínimo é de $99,0\%$ de pixels brancos, caso contrário, classifica-se como "parede e objetos". Veja alguns exemplos na Figura 3.30

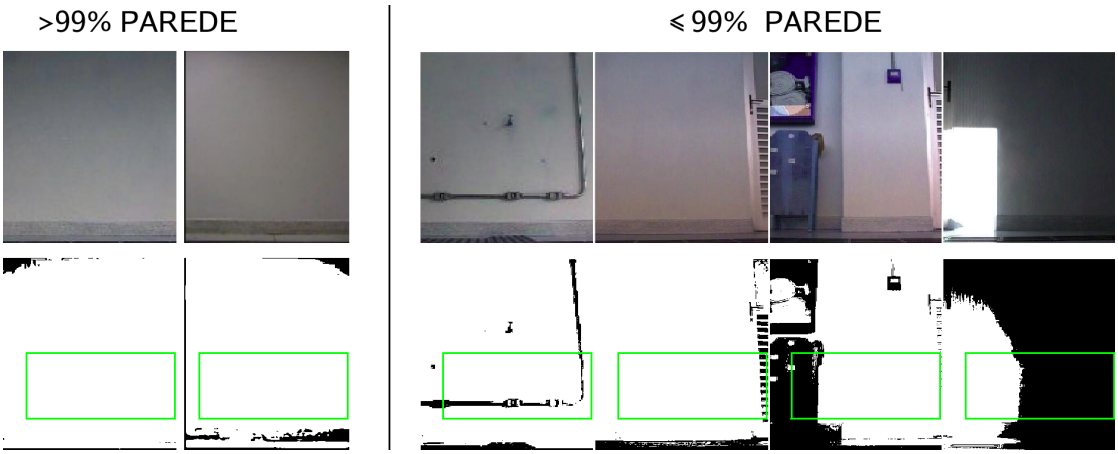


Figura 3.30: Método para identificação de paredes lisas. Fonte: o autor.

A resposta dessa identificação é enviada para o próximo estágio por meio de uma variável booleana *parede*.

Estágio 2: análise das políticas de adaptação.

O segundo estágio tem como objetivo converter a detecção de parede lisa em adaptação de velocidade para o robô. Sabe-se que o robô é iniciado em uma velocidade menor $v_1 = 0,1\ m/s$, podendo alterar para uma velocidade superior $v_2 = 0,2\ m/s$.

Assim, política de adaptação é definida com os seguintes termos:

- analisa-se as sequências recebidas da variável *parede*, onde *verdadeiro* significa "apenas parede" e *falso* significa "parede e objetos".
- Caso o algoritmo detecte uma transição $falso \rightarrow verdadeiro$ e, necessariamente, uma sequência de 5 *verdadeiros* consecutivos (F, V, V, V, V, V), será repassado o comando de adaptação de velocidade linear para $v_2 = 0,2\ m/s$ para o estágio seguinte. Esta sequência mínima é estabelecida porque é preciso se ter certeza de que o robô entrou em um trecho de "apenas

parede", para que assim possa aumentar sua velocidade linear.

- Caso o algoritmo detecte uma transição $verdadeiro \rightarrow falso$, no mesmo instante será repassado o comando de adaptação de velocidade para $v_1 = 0,1\ m/s$ para o estágio seguinte. Não é necessário uma sequência mínima (como no outro caso) por dois motivos: primeiro porque considera-se urgente que o robô diminua sua velocidade, para poder analisar melhor os elementos na imagem, e, segundo, para que haja tempo da imagem da câmera, que está trêmula, se recuperar da trepidação causada pelo robô em uma velocidade maior.

Estágio 3: administração da adaptação.

O último estágio tem a função de implementar a alteração de velocidade linear do robô no nó de navegação. Essa mensagem será publicada no tópico */analyzed_adapt*, que então será recebida pelo nó *follow_line*. Assim, ocorre a alteração da velocidade do robô para otimização do tempo total da execução da tarefa, tornando o sistema autoadaptativo.

3.3.7 Componente de Gerar Mapa Unidimensional

Este nó é o responsável por gerar a resposta final do sistema, que mostra todas as portas que foram encontradas pelo corredor no decorrer da execução da tarefa. O nó implementado se chama *create_map* e é assinante dos tópicos */found_doors* e */scanned_door*, por onde são recebidas dos outros nós do sistema as detecções das portas abertas e fechadas por visão computacional e pelo sensor de distância a laser.

Como explicado nas seções anteriores, é comum que aconteçam falsas detecções de portas abertas pelas imagens, devido à uma pequena taxa de erro associada ao algoritmo implementado. No entanto, esse problema é corrigido com o auxílio do sensor laser, e isso explica o motivo do nó *create_map* ser assinante do tópico */scanned_door*: as informações adquiridas nele servem para validar ou invalidar as detecções de portas abertas por visão computacional.

Ao final da tarefa, quando o final do corredor for identificado, este nó gera um mapa unidimensional do corredor com as posições em que as portas foram detectadas, onde o início do corredor é o marco zero. Para que isso seja possível é necessário ser assinante do tópico */pose*, onde se encontram dados sobre a localização absoluta do robô em relação ao sistema de coordenadas global.

O modelo dos mapas resultantes obtidos serão apresentados no capítulo seguinte.

Capítulo 4

Análises de Resultados

4.1 Introdução

Este capítulo tem como propósito apresentar os principais resultados obtidos pelo modelo implementado, em ROS, de sistema autoadaptativo baseado em comportamentos para executar a tarefa do robô móvel, cujo objetivo de detectar portas por meio da visão computacional é influenciado pela característica de luminosidade dinâmica do ambiente. A análise de resultados visa mostrar a capacidade de autoadaptar de um sistema robótico reativo, assim como mostrar que a arquitetura desenvolvida é capaz promover as adaptações requisitadas.

Uma abordagem *bottom-up* é utilizada para apresentar individualmente os principais componentes arquiteturais (subsistemas) e depois então o sistema como um todo, onde esses componentes interagem para cumprir os objetivos da tarefa. Primeiro, na Seção 4.2, é feita uma avaliação dos resultados obtidos nos quatro procedimentos de detecção de portas abertas e fechadas por processamento de imagens, individualmente, mostrando a importância de usar algoritmos diferentes para situações diferentes de luminosidade no corredor, geradas pelas combinações de luz natural e elétrica. Em seguida, na Seção 4.3, são apresentados resultados da componente de arquitetura responsável por identificar as necessidades de adaptação do sistema e efetivá-las, tanto para o chaveamento dos algoritmos de detecção de portas quanto para a alteração de velocidade linear na navegação.

Quando todos os componentes de software apresentados neste documento se reúnem para compor o sistema global, finalmente obtém-se a arquitetura de controle baseada em comportamentos para robótica móvel, capaz de se autoadaptar às características de ambiente dinâmico para cumprir o objetivo de detectar portas no corredor. Na Seção 4.4, os resultados finais são apresentados e avaliados em relação a eficácia e importância do modelo de sistema autoadaptativo com visão computacional para robótica. A Seção 4.5, apresenta os mapas unidimensionais gerados pelo nó *create_ map*. Por fim, na Seção 4.6, são feitas avaliações gerais dos resultados obtidos.

4.2 Avaliação dos Algoritmos de Detecção de Portas

Em visão computacional, a detecção de objetos nas imagens está sempre vinculada à uma taxa de acertos e alarmes falsos nas identificações. Com o intuito de testar os quatro algoritmos implementados e calcular a eficiência de cada um deles, foram feitas várias gravações das imagens da câmera com o auxílio da ferramenta *rosbag* do ROS, imagens essas coletadas do corredor e das portas em diferentes momentos do dia e em dias diferentes. Além disso, é apresentada aqui uma análise de cada um desses algoritmos (um para cada ambiente) atuando em cada uma das quatro situações identificadas com o intuito de mostrar o motivo de criar essa classificação de luminosidade e um algoritmo de detecção de portas específico para cada um deles.

É válido lembrar que são quatro algoritmos, porém existem três períodos diferentes: dia com sol, dia sem sol e noite (dois algoritmos para o período da noite). No entanto, devido às grandes variações de luminosidade do corredor no período da noite (combinações entre luzes acesas das salas, do corredor e luzes apagadas) foi necessário implementar dois procedimentos para este mesmo período, conforme apresentado na Seção 3.3.5.6. Assim, foram coletadas imagens do corredor de dia com sol, de dia com sombra e de noite. A Tabela 4.1 mostra o total de portas presentes nas imagens dos vídeos gravados nas 3 situações:

Tabela 4.1: Tabela com as quantidades de portas presentes nas imagens adquiridas dos corredores, que são usadas para análises dos algoritmos implementados.

Tipo de luminosidade	Total de portas	Portas abertas	Portas fechadas
Dia, sombra	225	66	159
Dia, sol	43	10	33
Noite	93	13	80

Para avaliar os quatro algoritmos de detecção das portas atuando em cada um dos três períodos, serão utilizados os indicadores de desempenho[18]: *sensitividade* (Sn), que indica a eficácia do algoritmo em reconhecer todas as portas do corredor, e *precisão* (Pr), que é o percentual de detecções corretas dentre todas as que foram realizadas. A métrica de avaliação final será através da *F-measure*, que é calculada pela harmônica entre *sensitividade* e *precisão*. Os indicadores são obtidos pelas seguinte equações:

$$Sn = \frac{vp}{vp + fn} \quad , \tag{4.1}$$

$$Pr = \frac{vp}{vp + fp} \quad e \tag{4.2}$$

$$\text{F-measure} = \frac{2}{\frac{1}{Sn} + \frac{1}{Pr}} \quad ; \tag{4.3}$$

onde vp é o número de verdadeiros positivos, fn são falsos negativos e fp são falsos positivos.

A *sensitividade*, que é a taxa de acertos, é o parâmetro de maior interesse para avaliar os classificadores. No entanto, ter uma alta taxa de acertos e também apresentar vários falsos alarmes,

torna o classificador ineficaz.

A seguir, são apresentados os resultados obtidos ao testar os quatro algoritmos em imagens com diferente tipos de luminosidade. Para cada um, são feitas análises dos indicadores e relacionando com as peculiaridades de cada nó de detecção de portas por processamento de imagens apresentados na Seção 3.3.5, do capítulo anterior.

4.2.1 Ambiente: corredor durante o dia em trechos sombreados ou dias sem sol

Observando a Tabela 4.2, os resultados obtidos pelo $Nó_{sombra}$ ao atuar em um ambiente com sombra foram extremamente satisfatórios. Foi observado que a única dificuldade era para detectar salas de porta aberta cujas luzes estavam acesas, fazendo com que o requisito mínimo de cor preta na imagem em preto e branco não fosse atendido.

Pela Tabela 4.3, percebe-se que o $Nó_{sol}$ teve uma boa taxa de acertos para portas abertas mas acompanhada de uma baixa precisão, uma vez que 81.08% significa que praticamente 1 a cada 5 detecções são falsos positivos. Para detectar portas fechadas o desempenho foi baixo.

Quanto aos dois algoritmos feitos para operar à noite, $Nó_{noite-1}$ e $Nó_{noite-2}$, observa-se nas Tabelas 4.4 e 4.5 desempenhos semelhantes: praticamente nenhuma das portas abertas foi identificada, uma vez que os valores alto do *threshold* $T = 210$ e $T = 150$ fez com que os requisitos mínimos de cor branca na imagem binária não fossem atendidos. Já para detecção das portas fechadas, a sensibilidade e a precisão foram altas, uma vez que o Classificador em Cascata foi treinado de forma a não depender da luminosidade do ambiente.

Tabela 4.2: Desempenho do $Nó_{sombra}$ atuando durante o dia em trechos com sombra

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	40	34	6	5	85%	87.18%	86.08%
Fechadas	101	97	4	0	96.04%	100%	97.98%
Total	141	131	10	5	92.91%	96.32%	94.58%

Tabela 4.3: Desempenho do $Nó_{sol}$ atuando durante o dia em trechos com sombra

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	40	30	10	7	75%	81.08%	77.92%
Fechadas	101	9	92	1	8.91%	90%	16.21%
Total	141	39	102	8	27.66%	82.98%	41.49%

Tabela 4.4: Desempenho do $Nó_{noite-1}$ atuando durante o dia em trechos com sombra

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	40	2	38	28	5%	6.67%	5.72%
Fechadas	101	87	14	5	86.14%	94.57%	90.16%
Total	141	89	52	33	63.12%	72.95%	67.68%

Tabela 4.5: Desempenho do $Nó_{noite-2}$ atuando durante o dia em trechos com sombra

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	40	1	39	6	2.5%	14.29%	4.26%
Fechadas	101	87	14	5	86.14%	94.57%	90.16%
Total	141	88	53	11	62.41%	88.89%	73.33%

Em suma, observando os valores dos *F-measure*, os nós para detecções noturnas não podem ser utilizados para operar de dia devido à baixíssima capacidade de detectar portas abertas. O $Nó_{sol}$ também não apresentou bom desempenho nesta situação. Assim, é possível perceber que o único método para conseguir bons resultados operando durante o dia em situações de sombra no corredor é utilizar um algoritmo específico para este caso. O $Nó_{sombra}$ implementado apresentou desempenho geral bem satisfatório, com *F-measure*$= 94,58\%$.

4.2.2 Ambiente: corredor durante o dia em trechos de sol

O $Nó_{sol}$ apresentou bons índices nas detecções, de acordo com a Tabela 4.6. Todas as portas abertas foram identificadas, porém ocorreram alguns falsos positivos ocasionados por sombras de pilares largos, que formavam regiões escuras nas imagens binárias semelhantes confundidas com salas abertas. Houveram também casos em que a transição sol-sombra ocorria no local onde uma porta fechada estava localizada, dificultando o trabalho do Classificador de objetos treinado, e essas situações reduziram a sensitividade resultante do algoritmo. Ainda assim, os resultados finais foram satisfatórios se comparados com os outros algoritmos.

Na Tabela 4.7, observa-se que o $Nó_{sombra}$ conseguiu boas taxas de acerto para identificar portas abertas atuando em situações de sol. Porém, a baixa precisão ocorre porque, nas imagens binárias após o *thresholding*, a parte de cima da parede coberta pela sombra em conjunto com objetos escuros que aparecem na parte inferior fizeram o algoritmo confundir o formato com portas abertas. Além disso, o fator sensitividade nas detecções de portas fechadas foi quase zero devido ao fato da sombra causada pelo sol na parede "camuflar" os alvos, impedindo que Classificador de objetos implementado pudesse identificá-los.

Os nós de detecção noturnos não conseguiram identificar as portas abertas e nem gerar falsos alarmes, uma vez que as paredes iluminadas pelo sol nas imagens continuam brancas mesmo com os valores altos de *threshold* nos algoritmos, que buscam tornar a imagem escura para identificar as portas. A sensitividade do Classificador de portas fechadas foi bem mediana, em torno de 56% apenas.

Tabela 4.6: Desempenho do $Nó_{sol}$ atuando no durante o dia em trechos iluminados pelo sol

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	11	11	0	3	100%	78.57%	88%
Fechadas	32	25	7	1	78.12%	96.15%	86.2%
Total	43	36	7	4	83.72%	90%	86.75%

Tabela 4.7: Desempenho do $Nó_{sombra}$ atuando durante o dia em trechos iluminados pelo sol

Portas	Real	vp	fn	fp	Sn	Pr	$F\text{-}measure$
Abertas	11	10	1	15	90.91%	40%	55.56%
Fechadas	32	1	31	1	3.12%	50%	5.87%
Total	43	11	32	16	25.58%	40.74%	31.43%

Tabela 4.8: Desempenho do $Nó_{noite-1}$ atuando durante o dia em trechos iluminados pelo sol

Portas	Real	vp	fn	fp	Sn	Pr	$F\text{-}measure$
Abertas	11	0	11	4	0%	0%	0%
Fechadas	32	18	14	3	56.25%	85.71%	67.92%
Total	43	18	25	7	41.86%	72%	52.94%

Tabela 4.9: Desempenho do $Nó_{noite-2}$ atuando durante o dia em trechos iluminados pelo sol

Portas	Real	vp	fn	fp	Sn	Pr	$F\text{-}measure$
Abertas	11	0	11	2	0%	0%	0%
Fechadas	32	18	14	3	56.25%	85.71%	67.92%
Total	43	18	25	5	41.86%	78.26%	54.54%

Devido ao reduzido número de oportunidades de dias ensolarados e do fato do corredor ter apenas alguns curtos trechos iluminados pelo sol, foram coletadas poucas amostras de imagens para esta situação de luminosidade. Ainda assim os resultados obtidos condizem com a expectativa. O $Nó_{sol}$ implementado para atuar em trechos sob a luz solar foi o único que obteve o desempenho desejado, bem acima dos demais, o que afirma a importância de uma metodologia específica para atuar nessa situação.

Quanto aos falsos negativos de portas fechadas gerados pelo $Nó_{sol}$ (resultantes das transições sol-sombra que ocorrem na mesma localização dos alvos, atrapalhando o Classificador em Cascata), uma rápida transição entre os nós $Nó_{sol}$ e $Nó_{sombra}$ supera essa limitação. Os efeitos dessa solução são apresentados adiante neste capítulo, na Seção 4.4.

4.2.3 Ambiente: corredor durante a noite

Antes de apresentar os resultados, as diferenças entre os dois procedimentos dos nós implementados para operar à noite são resumidos a seguir.

No primeiro nó, o $Nó_{noite-1}$, o *thresholding* com $T = 150$ torna a maioria dos elementos que aparecem na imagem binária na cor preta, enquanto alguns outros elementos, incluindo partes das salas de porta aberta (que estão de luzes acesas), apareçam geralmente em branco. Quando a luminosidade não é suficiente para que a porta aberta atenda o requisito mínimo de pixels brancos para ser classificada como tal, esse algoritmo pode apresentar falsos negativos de portas abertas.

Para lidar com essas situações, foi implementado o segundo nó, $Nó_{noite-2}$, cujo procedimento consiste em primeiro aplicar uma equalização de histograma na imagem em níveis de cinza e então um *thresholding* de $T = 210$. A equalização de histograma faz com que muitos dos elementos que eram pretos na imagem binária do algoritmo anterior apareçam em branco, inclusive e principal-

mente as paredes. Este procedimento é capaz de detectar as portas abertas não detectadas pelo primeiro nó, porém a quantidade de falsas detecções tende a ser bem superior. Ou seja, resolve a questão dos falsos negativos de portas abertas, porém piora quanto aos falsos positivos.

Os resultados apresentados nas tabelas a seguir comprovam essas informações.

Tabela 4.10: Desempenho do $Nó_{noite-1}$ atuando à noite

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	13	8	5	2	61.54%	80%	69.57%
Fechadas	79	78	1	1	98.73%	98.73%	98.73%
Total	92	86	6	3	93.48%	96.63%	95.03%

Tabela 4.11: Desempenho do $Nó_{noite-2}$ atuando à noite

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	13	11	2	41	84.62%	21.15%	33.84%
Fechadas	79	78	1	2	98.73%	97.5%	98.11%
Total	92	89	3	43	96.74%	67.42%	79.46%

Tabela 4.12: Desempenho do $Nó_{sombra}$ atuando à noite

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	13	0	13	16	0%	0%	0%
Fechadas	79	24	55	4	30.38%	85.71%	44.86%
Total	92	24	68	20	26.09%	54.55%	35.3%

Tabela 4.13: Desempenho do $Nó_{sol}$ atuando à noite

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	13	0	13	2	0%	0%	0%
Fechadas	79	27	52	4	34.18%	87.1%	49.09%
Total	92	27	65	6	29.35%	81.82%	43.2%

Pelas Tabelas 4.10 e 4.11, observa-se que os dois algoritmos para a noite apresentaram altíssima sensitividade e precisão para identificar portas fechadas: quanto ao $Nó_{noite-1}$, 98.73% e 98.73%, respectivamente, e quanto ao $Nó_{noite-2}$, 98.73% e 97.5%, na mesma ordem.

Note como o $Nó_{noite-1}$ apresenta baixa taxa de acertos para portas abertas, devido aos falsos negativos mencionados anteriormente. Perceba também como o $Nó_{noite-2}$ corrige esse indicador, porém apresenta péssima precisão devido a enorme quantidade de falsos positivos.

Quanto aos dois algoritmos que operam durante o dia, $Nó_{sombra}$ e $Nó_{sol}$, nenhuma das 13 portas abertas foram detectadas, uma vez que a claridade das salas abertas na imagem binária com valor de *threshold* baixos não atendia o requisito mínimo de pixels pretos. Nem mesmo para as portas fechadas os resultados foram positivos, uma vez que uma precisão alta para uma baixa sensitividade não é desejada, como mostram os *F-measure*.

É possível concluir que os resultados obtidos pelo algoritmos noturnos, mesmo seus pontos negativos, são bem superiores aos diurnos. No entanto, os dois algoritmos devem ser tido como

complementares quando ocorrer o chaveamento entre ambos, com a finalidade de que os falsos positivos ocorrentes no $Nó_{noite-2}$ sejam melhorados quando combinado com $Nó_{noite-1}$, assim como a quantidade de falsos negativos deste sejam solucionados em conjunto com a melhor sensitividade do primeiro.

4.3 Avaliação da Metodologia para Autoadaptatividade

Após serem apresentados os algoritmos que compõem o elemento de percepção do sistema robótico para identificar portas, observou-se que individualmente eles conseguem entregar bons resultados quando operam em situações de ambientes que os favorecem. Como o ambiente é dinâmico e imprevisível, o sistema reativo do robô deve ser capaz de identificar o ambiente e então adaptar o componente de detecção de portas, chaveando entre os algoritmos apresentados, afim de se obter na saída todas as portas do corredor independente da luminosidade do corredor. Além disso, visando a otimização do tempo total de execução, a adaptação também ocorre na velocidade linear do robô, que pode ser aumentada ou diminuída conforme as necessidades do sistema.

Assim, essa seção trata de analisar os resultados do nó *analyze_adaptation* para gerenciar a autoadaptatividade do robô a partir da visão computacional, apresentada na Seção 3.3.6.2 do capítulo de Desenvolvimento. Dois conjuntos de imagens gravadas são utilizadas nos testes dos procedimentos implementados, e os resultados são analisados e verificados em relação as políticas de adaptação apresentadas na mesma seção.

Os gráficos a seguir apresentam os dados coletados dos tópicos */analyzed_adapt*, */cmd_vel* e */analyzed_laser*, assim como o *ground truth*, com as verdadeiras informações sobre o corredor e seus elementos. Os resultados apresentados são relacionados somente ao funcionamento do detector de condições, sem levar em consideração os algoritmos de detecção detalhados anteriormente. O gráfico da Figura 4.1 mostra o resultado dos parâmetros do sistema coletados durante a execução, as imagens foram adquiridas durante um dia de sol, onde apenas 4 salas estavam abertas, de um total de 25.

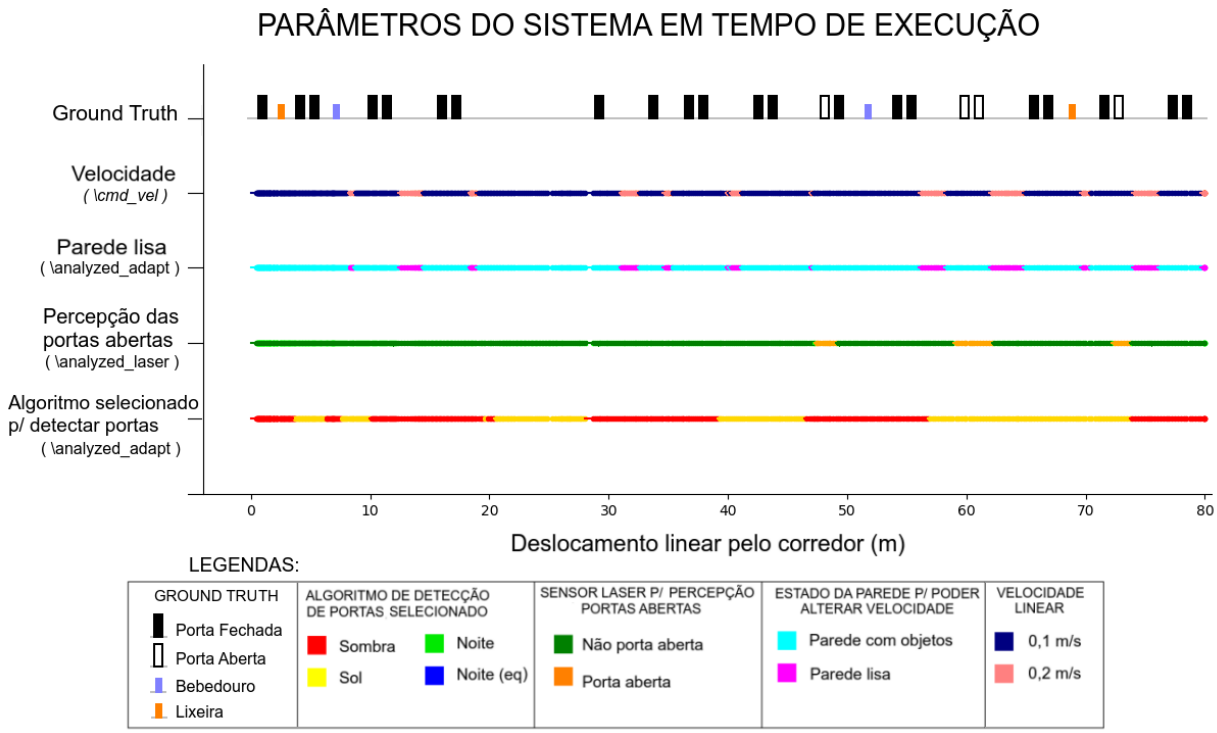


Figura 4.1: Gráfico obtido com mensagens publicadas nos tópicos ROS ao longo da execução da tarefa em um corredor durante o dia com trechos de sol e de sombra. Fonte: o autor.

No gráfico da Figura 4.1, a linha inferior (em vermelho e amarelo) corresponde à resposta gerada pelo nó *analyze_ adaptation*, publicada no tópico */analyzed_ adapt*, sobre qual o algoritmo de detecção de portas é selecionado em um determinado instante no corredor. As transições entre sol e sombra foram perfeitamente captadas pelo algoritmo nos momentos corretos, e essas identificações resultarão no chaveamento dos nós *find_ doors* para encontrar as portas. Além disso, percebe-se que o sistema foi capaz de encontrar os pequenos trechos no corredor onde havia apenas parede lisa à esquerda, ou seja, trechos em que não haviam portas para serem identificadas, possibilitando assim o robô de aumentar sua velocidade de $0,1m/s$ para $0,2m/s$, afim de economizar tempo de execução, e então depois reduzir novamente quando novos objetos que pudessem vir a ser identificados como portas.

Observe pelo gráfico que a componente de percepção das portas abertas, utilizando do sensor de distância a laser, conseguiu identificar os alvos conforme mostra a linha verde e laranja, onde os trechos em laranja representam as portas abertas existentes.

No gráfico da 4.2, os resultados foram obtidos de processamentos feitos durante a a noite, em um corredor de 26 portas onde 5 estavam abertas.

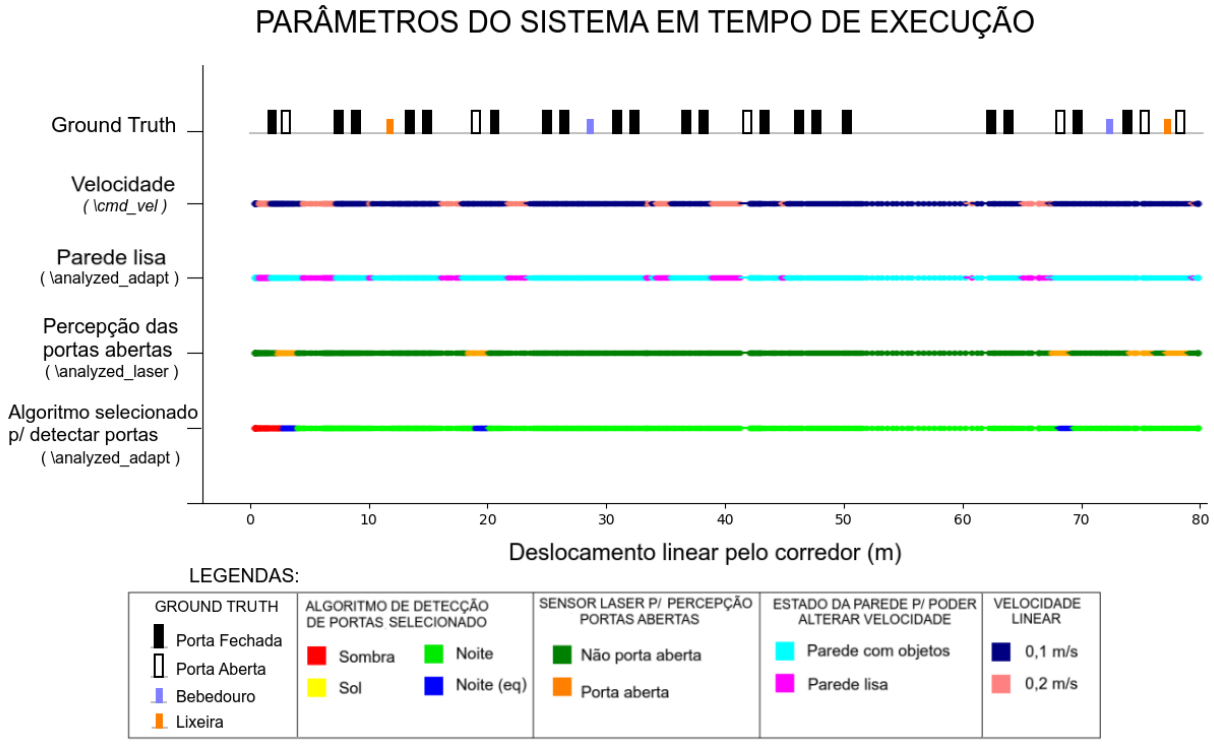


Figura 4.2: Gráfico obtido com mensagens publicadas nos tópicos ROS ao longo da execução da tarefa no corredor durante a noite. Fonte: o autor.

Observando a linha inferior do gráfico referente ao chaveamento dos algoritmos de detecção de portas pelo *analyze_ adaptation*, nota-se que o comportamento está de acordo com o que foi definido nos termos da política adaptativa, apresentada na Seção 3.3.6.2. No início da execução da tarefa, onde o robô ainda não sabe qual a luminosidade do ambiente e, por consequência, qual o algoritmo correto para se ativar, observa-se que o $Nó_{sombra}$ aparece ativo no início, em vermelho, conforme a política definida. Mesmo estando a noite, essa escolha não interfere na identificação das portas fechadas. Quando o robô estiver se aproximando da primeira porta aberta, a adaptação para o algoritmo correto será realizada.

Havia sido definido que a identificação das situações de ambiente nas imagens ocorre por meio da diferença de claridade entre o interior das salas abertas e a parede do corredor. Percebe-se que, assim que a primeira porta aberta é identificada pelo sensor de distância, o algoritmo do nó *analyze_ adaptation*identifica que deve ser feita a adaptação para o $Nó_{noite-2}$, representado em azul, e então efetiva o chaveamento.

Porém, de acordo com os dados da Tabela 4.11, a metodologia utilizada no $Nó_{noite-2}$ apresenta baixa precisão para detecção de portas abertas, o que significa que são gerados falsos positivos. Para solucionar isso, foi definido na política adaptativa que, uma vez que fosse feito o chaveamento para o $Nó_{noite-2}$ (cor azul), ele só permaneceria ativo por um tempo fixo pré-determinado, e então seria alternado novamente para o $Nó_{noite-1}$ (cor verde limão), afim de evitar as falsas detecções. Por sua vez, o algoritmo do $Nó_{noite-1}$ não é capaz de reconhecer algumas portas abertas, como

mostra o valor de Sn na Tabela 4.10, o que faz com que eventualmente seja necessário a readaptação temporária para o outro algoritmo. Contudo, o $Nó_{noite-1}$ ainda é capaz de reconhecer boa parte das portas abertas, implicando que não há necessidade de chaveamento para o outro algoritmo.

Em suma, a partir das análises dos gráficos, é possível observar que os procedimentos implementados obedecem os termos das políticas de adaptação proposta. As situações de luminosidade do ambiente podem ser identificadas e o chaveamento pode ser realizado pelo componente do sistema responsável pela autoadaptatividade. A seção a seguir mostra o sistema global, composto pela componente de adaptação operando juntamente com os algoritmos de detecção de portas selecionados.

4.4 Avaliação do Sistema Autoadaptativo Final

Avaliando individualmente os componentes do sistema, nas duas seções anteriores, notou-se que os diferentes algoritmos de identificação de portas nas imagens, pertencentes a Camada de Aplicação da arquitetura autoadaptativa proposta, apresentaram resultados satisfatórios enquanto operavam nos ambientes em que foram projetados para funcionar, assim como resultados ruins nos demais ambientes, o que é esperado. A componente responsável por identificar, analisar e aplicar as adaptações no sistema mostrou-se eficaz quantos às percepções das necessidades de adaptação, seja para o algoritmo de detecção de portas, seja para alteração do parâmetro velocidade linear.

A etapa final consiste em unir todos os componentes implementados de forma a produzirem a resposta final, chaveando corretamente entre os algoritmos para identificação de portas. Relembrando os resultados obtidos nos indicadores de desempenho das Tabelas 4.2 até a 4.13, na Seção 4.2, o objetivo na presente seção é mostrar como o sistema autoadaptativo contribui para gerar desempenhos melhores, reduzindo a taxa de falsos positivos e falsos negativos. Inclusive, considerando apenas os nós projetados para operar à noite, $Nó_{noite-1}$ e $Nó_{noite-2}$, o objetivo da autoadaptatividade em tempo de execução é combinar os dois de forma que se comportem como um único componente do sistema com índices de desempenho satisfatórios.

Os gráficos a seguir apresentam os resultados obtidos pelo sistema autoadaptativo completo, após a execução da tarefa nos corredores. Foram acrescentadas as seguintes informações: o *ground truth*, representando as verdadeiras informações do corredor (portas, lixeira, bebedouro), e mais 4 linhas no final, cada uma representando um algoritmo de detecção de portas por visão computacional. É possível perceber, pelo gráfico, como as detecções das portas são feitas pelos nós que são selecionados pela componente responsável por essa seleção.

Os gráficos das Figuras 4.3 e 4.4 apresentam os casos A e B, ambos ocorrem durante um dia ensolarado, onde os corredores possuem trechos de sombra e trechos com sol, e um total de 25 portas cada um. Observe como as cores da linha "Algoritmo selecionado para detectar portas" indicam qual é o nó de detecção de portas que está atuando no mesmo instante, representados nas quatro linhas inferiores. Quando a linha está vermelha, o algoritmo para sombra que detecta as portas, e quando a linha está amarela, quem o faz é o algoritmo para sol. A diferença entre os dois

casos é que o caso B foi registrado em outra época do ano, onde acontecem algumas transições sol-sombra em cima das portas.

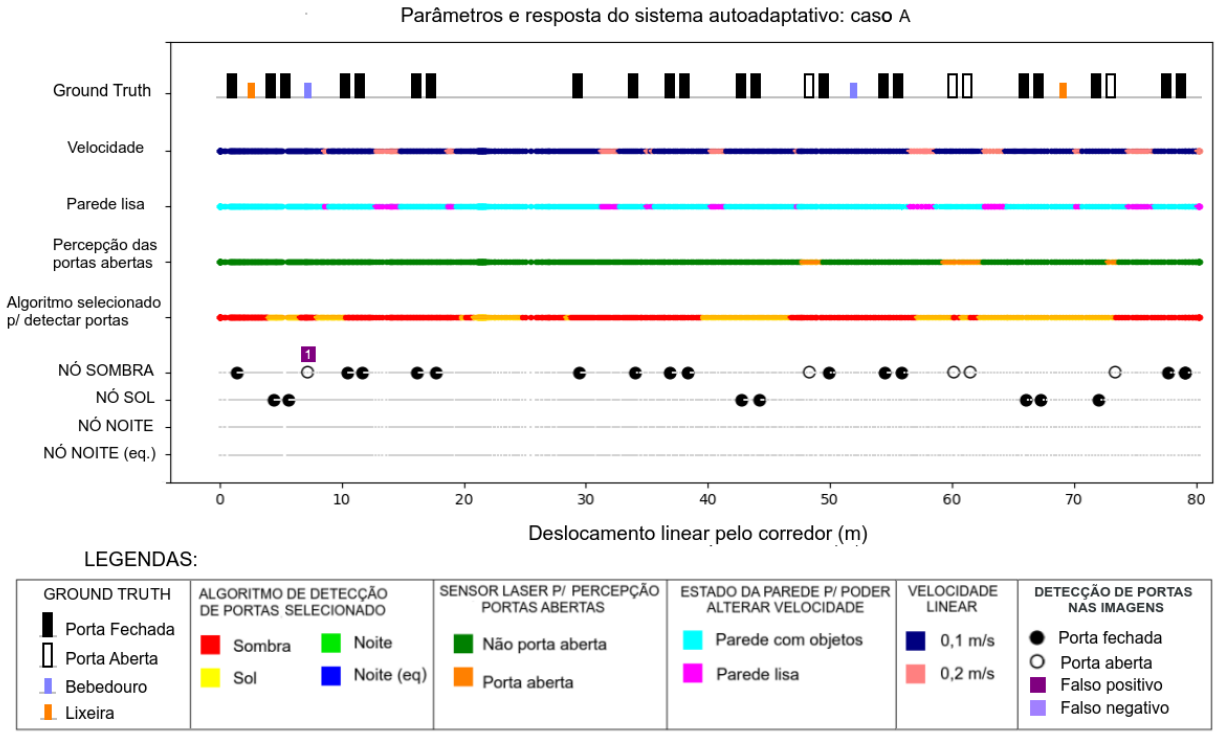


Figura 4.3: Caso A: o mesmo exemplo da Figura 4.1, que ocorre em um dia de sol, porém agora com o sistema autoadaptativo completo. Total de 25 portas, 4 abertas e 21 fechadas. 1 Falso positivo. Fonte: o autor.

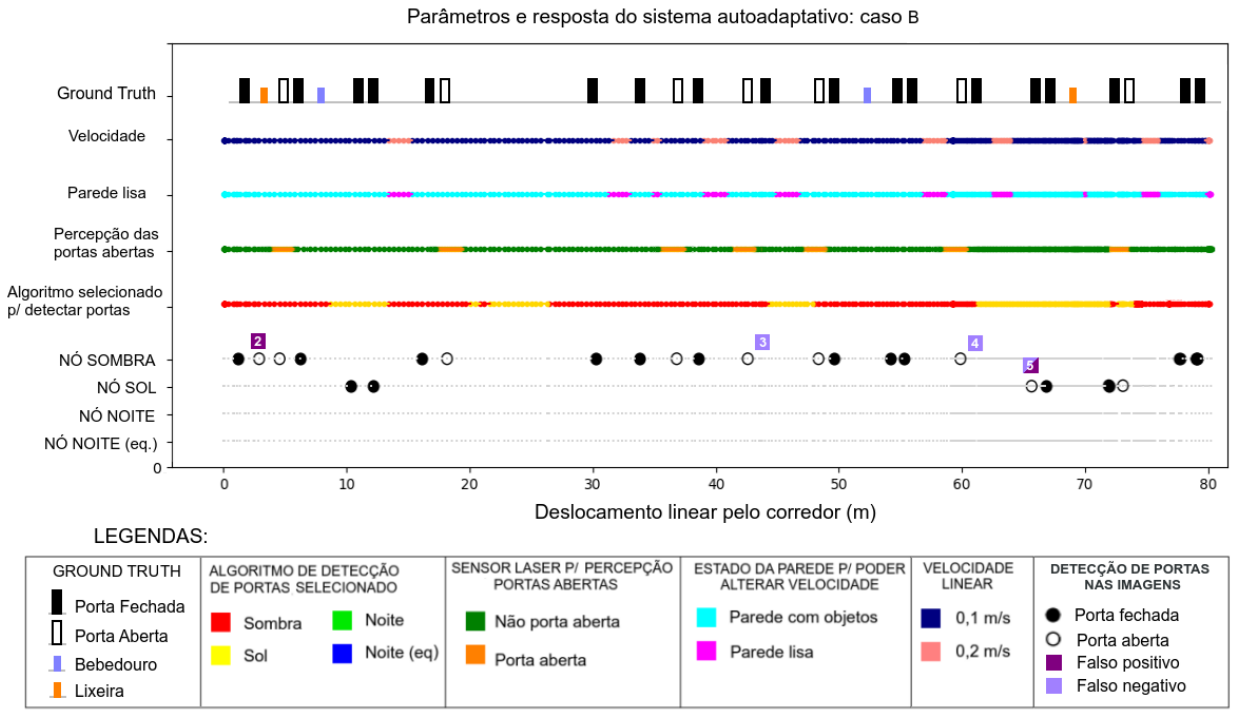


Figura 4.4: Caso B: ocorre em um dia de sol, existem transições sombra-sol em cima de portas fechadas. Total de 25 portas, 7 abertas e 18 fechadas. 2 Falsos positivos e 3 falsos negativos. Fonte: o autor.

No caso A, ocorreu um falso positivo que era, na realidade, um bebedouro que foi classificado como porta aberta, indicado pelo quadradinho roxo com número (1). No caso B, o quadradinho roxo com número (2) é outro falso positivo causado por uma lixeira. Ainda no caso B, duas portas fechadas passaram despercebidas, (marcadores (3) e (4)), porque a transição sombra-sol aconteceu em cima delas, atrapalhando o Classificador treinado para reconhecê-las, como mostrado nas Figuras 4.5(a) e 4.5(b). Já na indicação (5), o robô estava em um trecho de sol onde havia a sombra de uma pilastra em cima de uma porta fechada, como pode ver na Figura 4.5(c), isso fez com que o algoritmo reconhecesse a sombra como uma porta aberta e impediu que o verdadeiro alvo fosse reconhecido, ou seja, um falso positivo que implicou em um falso negativo.

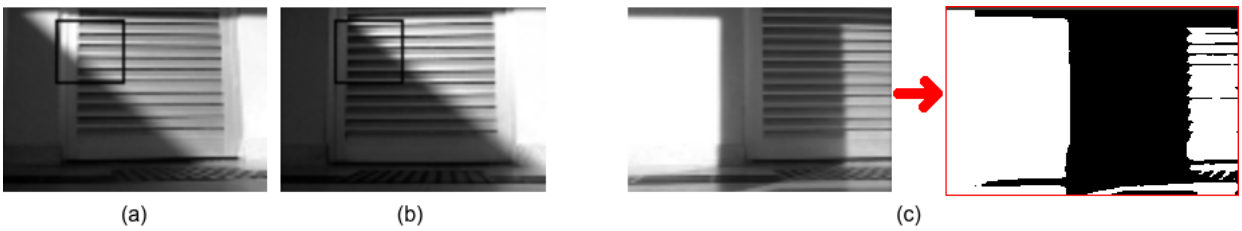


Figura 4.5: Falsos negativos e falso positivo do caso B. Fonte: o autor.

No entanto, os falsos positivos de portas abertas (causados por bebedouros, lixeiras e sombras de pilares) encontrados pelos nós de detecção de portas por processamento de imagens serão verificados posteriormente quando o sistema for gerar a resposta final em um mapa unidimensional,

implementado no nó *create_map*, uma vez que o procedimento utiliza o escaneamento do sensor laser para identificar se realmente ali havia uma porta aberta. Nos gráficos acima, observa-se que, nos momentos em que ocorreram essas falsas detecções de salas abertas por imagem, o sensor laser não realizou a mesma identificação, como pode ser percebido pela ausência de segmento em cor laranja na linha "Percepção das portas abertas". Já quanto às portas fechadas, percebe-se que não ocorreram falsas detecções, sendo assim, a maior falha a ser considerada são as essas portas fechadas que passaram despercebidas devido à transições sol-sombra, que as escondem do Classificador de objetos.

Os casos C e D, a seguir, foram registrados durante a noite e os parâmetros do sistema estão representados nos gráficos das Figuras 4.6 e 4.7.

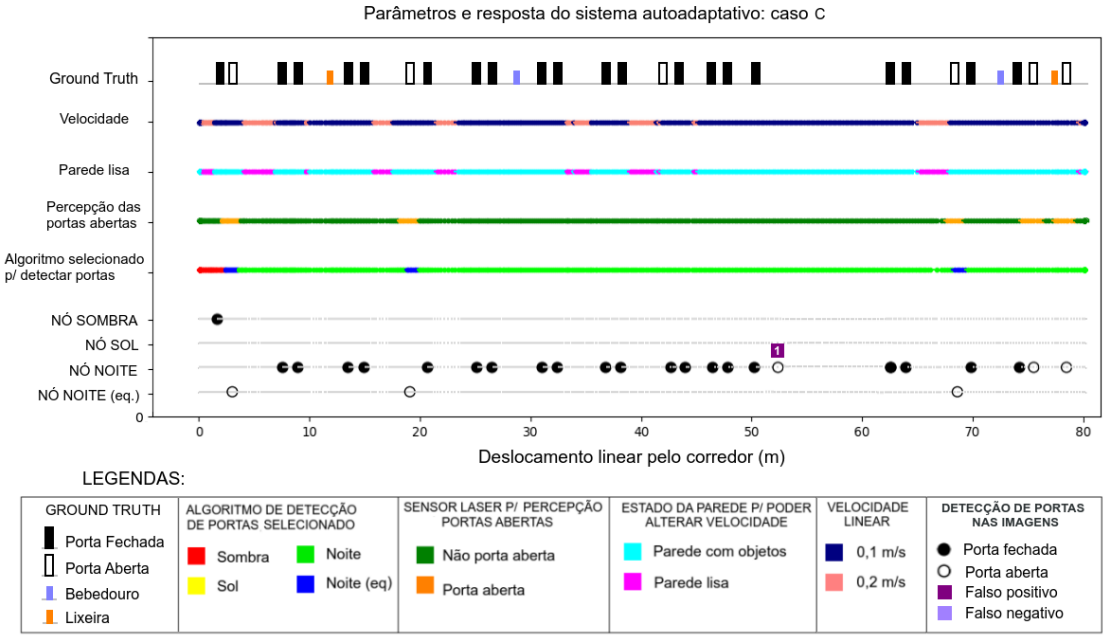


Figura 4.6: Caso C: o mesmo exemplo da Figura 4.2, que ocorre durante à noite, porém agora com o sistema autoadaptativo completo. Total de 26 portas, 5 abertas e 21 fechadas. 1 Falso positivo. Fonte: o autor.

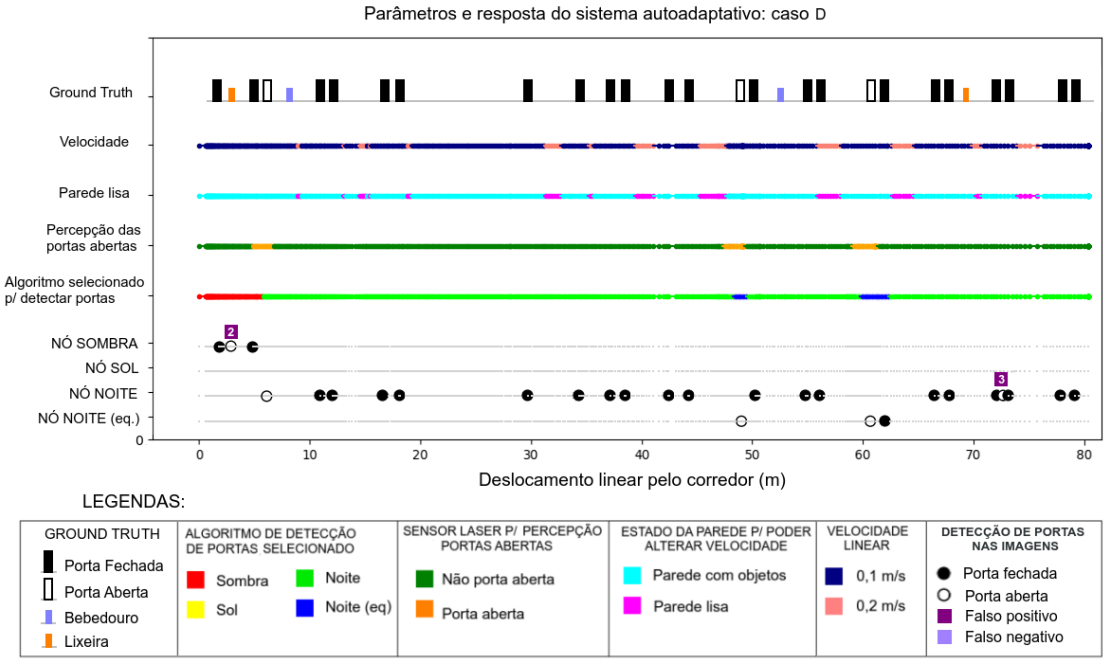


Figura 4.7: Caso *D*: ocorre também durante a noite. Total de 25 portas, 3 abertas e 22 fechadas. 2 Falsos positivos. Fonte: o autor.

Nestes casos, o nó *analyze_adaptation* conseguiu identificar o ambiente noturno assim que o robô passou pela primeira porta aberta, como mostrado nos gráficos. A partir de então, o chaveamento entre os nós de detecção de portas passou a ser entre os nós $Nó_{noite-1}$ e $Nó_{noite-2}$. Ao comparar os resultados obtidos à noite por este sistema autoadaptativo (ver Tabela 4.15) com os desempenhos individuais de cada um dos dois nós noturnos de detecções das portas (mostrados nas Tabelas 4.10 e 4.11 da Seção 4.2), percebe-se que os índices de desempenho obtidos pelo sistema autoadaptativo foram superiores. Todas as portas fechadas foram encontradas e não houve falsas detecções. Para as portas abertas, ocorreram apenas 3 falsos positivos, e um deles (no caso *B*) foi devido enquanto o nó $Nó_{sombra}$ estava em ativo no início do corredor, que confundiu uma lixeira como porta aberta na imagem binária, o que é compreensível uma vez que este algoritmo não foi projetado para operar durante a noite. Ainda assim, essas falsas identificações de portas abertas serão posteriormente descartadas com o auxílio do sensor laser, uma vez que não faz a mesma identificação.

Assim como esperado, percebe-se que, enquanto o $Nó_{noite-1}$ estiver ativo (cor verde limão no gráfico), eventualmente o robô pode se deparar com uma porta aberta que este nó não é capaz de detectar. Neste caso, o sistema se adapta chaveando para o $Nó_{noite-2}$ temporariamente, que identifica as portas necessárias.

Considerando todos os casos apresentados, é possível perceber que é o sistema autoadaptativo é capaz de identificar as variações de luminosidade dos ambientes e de efetivar a tempo o chaveamento para que os nós de detecções de portas entrem em ação. Isso se deve principalmente à duas características definidas na implementação: a primeira é a baixa do velocidade do robô enquanto os algoritmos estão detectando portas pela visão computacional, atendendo aos requisitos do sistema

em tempo-real; a segunda é em relação ao que se refere a Figura 3.29 na Seção 3.3.6.1, que ilustra o método utilizado para garantir que as portas não serão ultrapassadas nas imagens do robô, ao se considerar que o mesmo está em movimento e que existe um tempo computacional entre a identificação de uma adaptação e o chaveamento dos algoritmos de detecção.

As Tabelas a seguir mostram o desempenho do sistema autoadaptativo nos quatro casos analisados:

Tabela 4.14: Indicadores de desempenho para os casos A e B, que ocorrem durante o dia.

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	11	11	0	3	100%	78.57%	88%
Fechadas	39	36	3	0	92.31%	100%	96%
Total	50	47	3	3	94%	94%	94%

Tabela 4.15: Indicadores de desempenho para os casos C e D, que ocorrem durante a noite.

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	8	8	0	3	100%	72.73%	84.21%
Fechadas	43	43	0	0	100%	100%	100%
Total	51	51	0	3	100%	94.44%	97.14%

Tabela 4.16: Indicadores de desempenho para os 4 casos juntos, que avaliam o desempenham do sistema em geral.

Portas	Real	vp	fn	fp	Sn	Pr	*F-measure*
Abertas	19	19	0	6	100%	76%	86.36%
Fechadas	82	79	3	0	96.34%	100%	98.14%
Total	101	98	3	6	97.03%	94.23%	95.61%

Na Tabela 4.17, o único índice que ainda pode ser melhorado é a precisão para detecção de portas abertas. O valor de $76,00\%$ é resultado dos falsos positivos encontrados, como bebedouros, lixeiras, sombras de pilares durante o dia, ou reflexo da luz do corredor nos objetos da parede durante a noite. No entanto, como é apresentado na Seção 4.5, ao utilizar o escaneamento do sensor de distância a laser como auxílio para verificação das portas abertas, espera-se que o valor do índice precisão seja tão satisfatório quanto foi para portas fechadas.

4.5 Avaliação da Saída Global: as portas finalmente identificadas

O nó *create_ map* corresponde ao componente do sistema responsável por combinar as detecções de portas por visão computacional com as detecções via escaneamento do sensor de distância a laser. Observando os gráficos apresentados na seção anterior, para se ter o resultado almejado basta combinar as identificações de portas por imagens (nas 4 linhas inferiores) com as leituras do sensor laser (segmentos em cor laranja na linha "Percepção das portas abertas"). A Figura 4.8 mostra os mapas unidimensionais gerados pelo componente *create_ map*nos casos A, B, C e D apresentados na seção anterior. É possível perceber as portas abertas e fechadas que foram localizadas ao longo do corredor, com suas respectivas localizações.

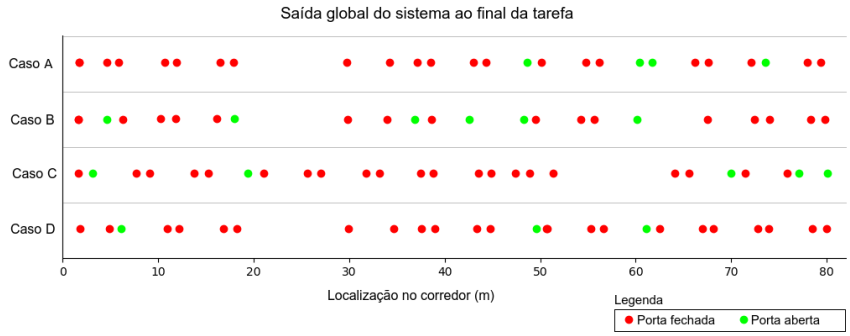


Figura 4.8: Saída global com as portas encontradas e suas localizações ao longo do corredor, para os casos A, B, C e D apresentados na Seção 4.4. Fonte: o autor.

Tabela 4.17: Indicadores de desempenho do sistema autoadaptativo após filtragem de falsas detecções pelo sensor laser, na saída global.

Portas	Real	vp	fn	fp	Sn	Pr	$F\text{-}measure$
Abertas	19	19	0	0	100%	100%	100%
Fechadas	82	79	3	0	96.34%	100%	98.14%
Total	101	98	3	0	97.03%	100%	98.49%

Observando a Tabela 4.17, pode-se afirmar que os resultados obtidos foram excelentes: 100% das portas abertas foram detectadas, juntamente com $98,14\%$ das fechadas. Estes índices resultam no *F-measure* total de $98,49\%$. De todas as 101 portas fechadas, apenas 3 não foram identificadas, correspondentes aos marcadores 3, 4 e 5 na Figura 4.4, na Seção 4.4, onde os motivos para tais ocorrências são descritos.

4.6 Comparativos Gerais em uma Abordagem Bottom-Up

Primeiramente, foram apresentados os resultados dos desempenhos de cada um dos 4 nós implementados para detectar portas operando em diferentes condições de luminosidades do ambiente, as informações obtidas estão representadas no gráfico de barras da Figura 4.9. Condizendo com a expectativa, cada algoritmo apresentou bons resultados ao atuar no ambiente em que foi projetado para atuar, e resultados indesejados nos demais ambientes. Comparando os desempenhos no gráfico da Figura 4.9, é possível perceber a importância de se ter um procedimento próprio para operar em um ambiente com uma iluminação específica. Sabendo que a luminosidade dos corredores é dinâmica e considerando que o sistema do robô não tem conhecimento das características do ambiente ao iniciar a tarefa no início da execução, torna-se necessário que o sistema seja capaz de se autoadaptar, isto é, selecionar o algoritmo correto para operar na condição de luminosidade do ambiente identificada pelo próprio sistema, afim de produzir o melhor resultado possível.

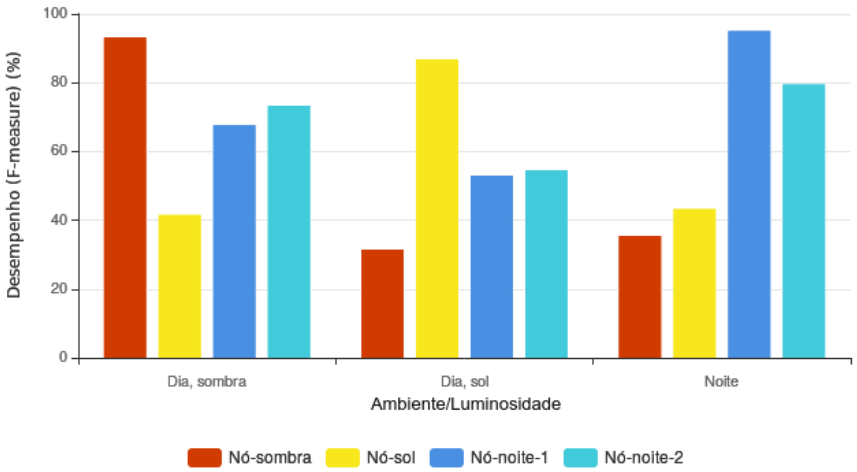


Figura 4.9: Desempenhos individuais (índice F-measure) dos nós de detecção de portas em imagens para cada ambiente/luminosidade. Em dias de sombra, o $Nó_{sombra}$ apresentou melhor desempenho, assim como o $Nó_{sol}$ para trechos sob luz solar direta. Durante a noite, como esperado, os nós $Nó_{noite-1}$ e $Nó_{noite-2}$ obtiveram os melhores resultados. Fonte: o autor.

Analisando os índices nas Tabelas 4.2 até 4.13, nos casos em que um atua sob uma condição de luminosidade em que não foi projetado para atuar (exemplo, $Nó_{sol}$ operando de noite), foram obtidos desempenhos melhores nas detecções de portas fechadas do que abertas: o método de *thresholding* utilizado nas imagens para identificação de portas abertas é totalmente dependente das variações de luz do ambiente, enquanto que o método utilizado para detecções de portas fechadas, através do Classificador em Cascata treinado com *deep learning*, não depende tanto da luminosidade.

Foi verificado que a componente responsável por identificar e implantar essas adaptações do sistema é capaz de promover eficientemente o chaveamento correto dos algoritmos de detecção de portas, de forma que, combinados, os nós puderam identificar tanto portas abertas quanto fechadas com uma boa taxa de acertos e com poucos falsos alarmes, refletidos no índices de Sensitividade e Precisão nas Tabelas 4.14, 4.15 e 4.16. O índice *F-measure* global, na Tabela 4.16, foi de 95.61%.

Afim de elevar ainda mais o desempenho do sistema autoadaptativo do robô para encontrar portas por processamento de imagem, o sistema autoadaptativo na versão final utilizou o auxílio do sensor laser para validar ou rejeitar as identificações de portas abertas. Assim, o índice *F-measure* do sistema final passou de 95.61% para 98.49%, como mostrou a Tabela 4.17.

Os gráficos a seguir, na Figura 4.10, mostram os desempenhos para detecções de portas abertas e fechadas, separadamente, obtidos pelos seguintes nós e sistemas, nessa ordem: os quatro nós de detecções operando do início ao fim da tarefa, sem serem alternados (representados pelas 4 primeiras barras dos gráficos), o sistema autoadaptativo sem a componente de escaneamento de portas abertas pelo sensor laser (barra em cor verde) e o sistema autoadaptativo do robô na versão final (barra em cor roxa).

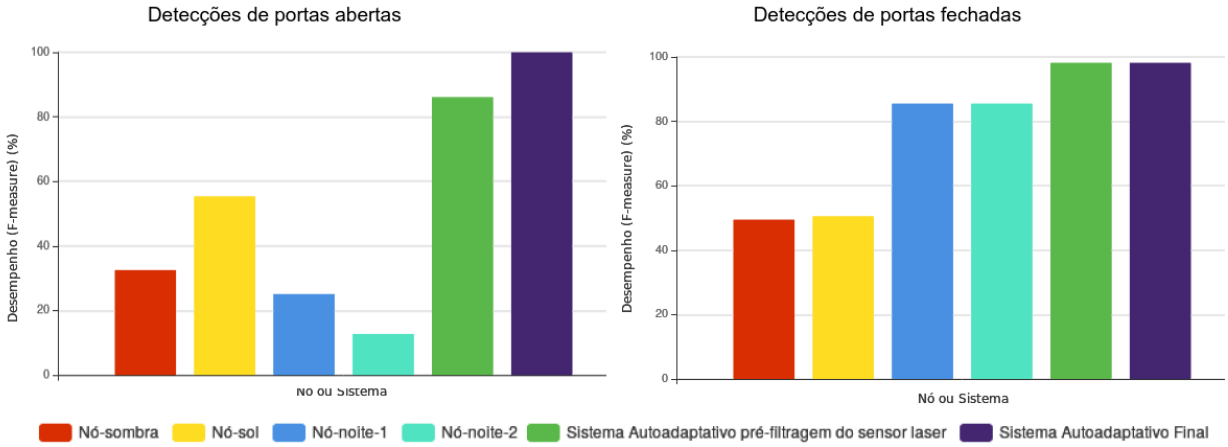


Figura 4.10: Gráfico da evolução do sistema, considerando desde os nós operando individualmente até o sistema autoadaptativo na versão final. As 4 primeiras barras representam o que seria o desempenho dos nós caso fossem os únicos existentes para realizar a tarefa de detectar portas em qualquer situação de luminosidade. As duas últimas representam, respectivamente, o sistema autoadaptativo considerando apenas as detecções por visão computacional, e o sistema na versão final, onde o componente de análise do sensor laser é integrado. Fonte: o autor.

De forma geral, é possível perceber a influência que o sistema autoadaptativo causa nos resultados se comparados com os desempenhos individuais dos quatro nós, uma vez que a autoadaptatividade do sistema robótico torna possível selecionar o nó correto para atuar em um determinado momento, como se o sistema tivesse um único componente responsável por detectar as portas eficientemente, independendo da luminosidade do ambiente. Nota-se também o fato do sistema autoadaptativo final apresentar melhores resultados para identificar portas abertas, uma vez que utiliza a percepção do sensor de distância para invalidar as falsas detecções da visão computacional.

Capítulo 5

Conclusões

Desenvolver aplicações em robótica móvel para atuar em ambientes dinâmicos e que requerem adaptações em tempo-real pode ser uma tarefa difícil, ainda mais em casos onde o comportamento reativo do robô é dependente de processamentos de imagens, o que pode vir a trazer complicações para as requisições em tempo-real uma vez que os componentes internos e atuadores dependem de respostas rápidas. Sistemas autoadaptativos surgem como uma solução para aplicações onde há necessidade de autogerenciamento e auto-otimização. Este trabalho apresentou uma arquitetura autoadaptativa para coordenar um robô móvel em uma tarefa onde o ambiente é dinâmico e imprevisível em termos do parâmetro luminosidade do corredor.

A tarefa de identificar portas e seus estados pôde ser implementada segundo o paradigma de Controle Baseado em Comportamentos, e verificou-se que a Arquitetura de Controle para robótica escolhida[8], organizada em Camadas de Aplicação e Adaptação, possibilitou o desenvolvimento das componentes de comportamento e de percepção do sistema como nós independentes, que interagem entre si e, principalmente, adaptáveis em tempo de execução. Concluiu-se que o desenvolvimento dessa arquitetura autoadaptativa foi facilitada pela flexibilidade que o framework *ROS* ofereceu para implementação dos nós independentes, garantindo, assim, a capacidade de adaptação do sistema em tempo-real.

Os métodos de processamento de imagem escolhidos mostraram resultados satisfatórios para identificar as portas abertas e fechadas no corredor, uma vez que os algoritmos de cada situação de luz puderam ser alternados pelo sistema autoadaptativo conforme a luminosidade do ambiente identificada pelo robô, afim de oferecer o desempenho desejado para identificação das portas. Embora os métodos de processamento de imagens para detecção dos objetos tenham apresentado resultados satisfatórios, é válido dizer que não necessariamente implica em serem os mais eficientes possível para executar a tarefa em questão. No entanto, o próprio fato de a técnica de *thresholding* utilizada ser dependente dos níveis de cinza na imagem, resultantes das variações de iluminação dos corredores, foi que fez surgir a requisição de o sistema ser capaz de se adaptar, uma vez que a luminosidade é imprevisível e dinâmica.

Por fim, considerando-se a quantidade de dados de imagem e de sensores sendo trocados ininterruptamente e simultaneamente entre os nós implementados, mostrou-se que o sistema foi bem

sucedido em entregar uma resposta em tempo-real rápida suficiente para identificar e implementar as adaptações, garantindo a integridade da saída global.

5.1 Trabalhos Futuros

Afim de dar continuidade a este trabalho, seguem algumas sugestões de abordagens implementáveis para aprimorar o sistema autoadaptativo e a aplicação em robótica móvel:

- Adotar mais estados de porta além da classificação binária, incluindo níveis de portas semi-abertas.
- Focando na visão computacional do robô, implementar métodos mais eficazes para as detecções de portas e identificar mais situações de luminosidades do ambiente, além das quatro feitas neste trabalho.
- Encontrar novas métricas para adaptação, relacionadas à, por exemplo, navegação, identificação e tratamento de obstáculos móveis, parâmetros internos do robô (como consumo de bateria) ou mesmo quantos aos sensores de distância laser e ultrassom, empregando uso de filtros e análises estatísticas.
- Implementar soluções para tornar a navegação do robô completamente autônoma, podendo navegar por outros ambientes além dos abordados neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] MATARIĆ, M. J. *The Robotics Primer*. [S.l.]: Massachussetts: MIT Press, 2007.

[2] KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, v. 36, n. 1, p. 41–50, 2003. Institute of Electrical and Electronics Engineers (IEEE).

[3] OLIVEIRA, B. Q. de et al. Tipos e aplicações de sensores na robótica. *Cadernos de Graduação. Ciências Exatas e Tecnológicas*, v. 4, n. 1, p. 223–230, 2017.

[4] CACIC: O Pioneer 3-AT do Departamento de Ciência da Computação da Universidade de Brasília. Disponível em: <https://cacic-robot.readthedocs.io/en/latest/index.html>, acessado às 20:27 de 26/11/2019.

[5] FREITAS, R. S. de. Arquitetura híbrida e controle de missão de robôs autônomos. *Cadernos de Graduação. Ciências Exatas e Tecnológicas*, v. 4, n. 1, p. 223–230, 2017.

[6] ROS WIKI. Disponível em: <http://wiki.ros.org/>, acessado às 16:35 de 01/11/2019.

[7] BROOKS, R. A. A robust layered control system for a mobile robot. *Roboticsand Automation, IEEE Journal of*, v. 2, n. 1, p. 14–23, 1986.

[8] EDWARDS, G. et al. Architecture-driven self-adaptation and self-management in robotics systems. *IEEE Journal of Robotics and Automation*, 2009. SEAMS'09.

[9] THOTAKURI, A. et al. Survey on robot vision: Techniques, tools and methodologies. v. 12, 2009.

[10] KULKARNI, N. Color thresholding method for image segmentation of natural images. *International Journal Of Image, Graphics And Signal Processing*, v. 4, n. 1, p. 28–34, 3 fev. 2012. MECS Publisher. http://dx.doi.org/10.5815/ijigsp.2012.01.04.

[11] ZHA, Z.-Q. et al. Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, august 2017.

[12] PATHAK, A. R.; PANDEY, M.; RAUTARAY, S. Application of deep learning for object detection. *Procedia Computer Science*, v. 132, p. 1706–1717, 2018. Elsevier BV. http://dx.doi.org/10.1016/j.procs.2018.05.144.

[13] OPENCV: Introduction to Cascade Classifier. Disponível em: <`https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html`>, acessado às 12:41 de 02/11/2019.

[14] ROS Tutorials by CLEARPATH. Disponível em: <https://www.clearpathrobotics.com/assets/guides/ros/index.html>, acessado às 23:39 de 26/11/2019.

[15] OPENCV: Cascade Classifier Training. Disponível em: <https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html>, acessado às 12:41 de 02/11/2019.

[16] OPENCV: Cascade Classifier Functions. Disponível em: <https://docs.opencv.org/3.4/d1/de5/classcv_1_1CascadeClassifier.html>, acessado às 16:11 de 07/11/2019.

[17] DOROTHY, R. et al. Image enhancement by histogram equalization. 2015.

[18] POWERS, D. M. W. Evaluation: From precision, recall and f-measure to roc, informedness, markedness correlation. *Journal of Machine Learning Technologies*, v. 2, p. 37–63, 2011.