



Universidade de Brasília

**UNIVERSIDADE DE BRASÍLIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
FUNDAMENTOS COMPUTACIONAIS DE ROBÓTICA – 0/2017
TURMA A**

RELATÓRIO DO QUINTO TRABALHO

**PROFESSORA:
CARLA MARIA CHAGAS E CAVALCANTE KOIKE**

**ALUNO:
THÚLIO NOSLEN SILVA SANTOS**

14/0164090

**TÍTULO:
MAPEAMENTO DO
PRÉDIO DO CIC COM O
ROBÔ PIONEER 3-AT**

Sumário

OBJETIVOS	3
INTRODUÇÃO	4
ESTRUTURA DO PACOTE	5
DETECÇÃO E DESVIO DE OBSTÁCULOS.....	6
PROCESSAMENTO DE DADOS DO SENSOR.....	6
ALGORITMO PARA O DESVIO DE OBSTÁCULOS.....	7
PLANEJAMENTO E EXECUÇÃO DE TRAJETÓRIA	8
PLANEJAMENTO DE TRAJETÓRIA PARA O MAPEAMENTO.....	8
EXECUÇÃO DA TRAJETÓRIA E EXPLORAÇÃO DO MAPA	9
CONSTRUÇÃO DA GRADE DE OCUPAÇÃO.....	10
DESLOCAMENTO ATÉ A POSIÇÃO FINAL	11
ADAPTAÇÕES PARA O ROBÔ REAL	12
LISTAGEM DOS ARQUIVOS ENTREGUES	13
VÍDEOS E RESULTADOS	14
NÓ 2.....	15
NÓ 3.....	16
NÓ 4.....	17
NÓ 5.....	19
BIBLIOGRAFIA	20

1. Objetivos

O robô Pioneer real deve se deslocar no primeiro andar do prédio do CIC, explorando todas as posições acessíveis. Para cada região no grafo do mapa topológico, o robô deve construir uma grade de ocupação, não tocar nenhum objeto que consiga detectar e certificar-se de que explorou todas as posições acessíveis. Em seguida, o operador do robô deve especificar uma configuração qualquer $\langle \mathbf{x}, \mathbf{y}, \theta \rangle$ nos corredores mapeados e o robô deve se deslocar do nó atual até esse ponto pelo caminho mais curto.

2. Introdução

O algoritmo de detecção de obstáculos usado foi baseado no artigo [1] sobre VFH de Johann Borenstein. O método, chamado de Vector Field Histogram (VFH), consiste em montar um histograma polar baseado numa malha de ocupação e a partir da densidade de objetos nos setores do histograma escolher uma direção apropriada para o movimento. O algoritmo foi adaptado para não gerar o histograma a partir da grade de ocupação, e sim diretamente dos valores do sensor laser.

O algoritmo para o planejamento de caminhos para a construção do mapa envolve a construção de um grafo do prédio do CIC a partir de um arquivo texto preparado com antecedência. O arquivo contém informações sobre os nós, como quantos nós tem o mapa, como se descrevem as regiões dos nós e a que outros nós se liga um nó. A partir deste arquivo uma representação interna do mapa é construída e, a partir de um outro arquivo, uma sequência de nós é lida que permite a navegação do robô por qualquer nó do grafo.

Depois do mapeamento, o algoritmo de planejamento para guiar o robô até uma configuração final especificada pelo usuário é similar ao citado acima. A partir do grafo lido e localizando-se o nó a que a configuração final pertence, a rota mais curta que leva do nó inicial à configuração final é determinada usando-se o algoritmo de Dijkstra [2].

O algoritmo para a execução do caminho é similar ao método *follow-the-carrot*, mas com a inclusão do desvio de obstáculos descrito acima. A ideia é que o robô tente sempre ir em direção a um ponto, seu alvo atual, e conforme ele se aproxime de algum obstáculo, o controle da velocidade é transferido, de forma linear, para a função que trata do desvio de obstáculos.

O algoritmo para o mapeamento não foi baseado em nenhum outro método em especial, além das aulas da professora. Para cada célula da grade ocupação, é determinada qual medida do sensor passa pela célula e o valor da medida informa ao algoritmo se a célula está ocupada ou desocupada. Para tratar os dados, um filtro foi aplicado aos dados na forma de uma média ponderada móvel [3]. Este filtro não é tão aplicado quanto os filtros de Kalman ou de Bayes, mas fornece uma boa filtragem e é de mais fácil implementação. Além disso, como forma de representar as grades de ocupação de forma mais amigável, foi criada uma função que cria arquivos no formato bitmap que representam as grades feitas pelo robô, e uma função que mostra a grade no terminal com *color coding* para os valores das células.

Algumas adaptações tiveram de ser feitas para controlar o robô real, como adicionar uma checagem no funcionamento do laser, permitir que o robô se inicie de qualquer posição no mapa e diminuir sua velocidade.

Uma descrição da implementação dos algoritmos com alguns trechos de pseudocódigo pode ser conferida no README do pacote. De fato, várias das descrições feitas aqui sobre a implementação dos algoritmos e descrição do pacote podem ser encontradas com mais detalhe nos arquivos fonte, cabeçalhos e no próprio README.

3. Estrutura do pacote

O pacote foi construído a partir do pacote **fcr2017** disponibilizado e dos trabalhos anteriores. Na pasta **include** se encontram alguns arquivos de cabeçalho do trabalho. O arquivo **obstacle_avoidance.h** contém a classe que implementa as funções do ROS e cria um framework para implementar os outros algoritmos. O arquivo **my_classes.h** contém as definições das classes usadas no desenvolvimento do trabalho. O arquivo **my_constants.h** contém as definições de todas as constantes usadas no código, como o número de graus por setor, a velocidade máxima do robô e o tamanho de uma célula da grade de ocupação. Por fim, o arquivo **my_functions.h** contém todas as funções usadas no trabalho. Em **launch** se encontram arquivos como o que inicia o Rviz e o que permite usar a tele operação com robô. Em **maps**, **meshes**, **models**, **world** e **robots** se encontram os arquivos que modelam o prédio do CIC e o Pioneer. Na pasta **src** se encontram os códigos-fonte do pacote, como **obstacle_avoidance.cpp**, **print.cpp**, **user.cpp** e **obstacle_avoidance_node.cpp**, e alguns arquivos texto. O arquivo **graph.txt** contém a descrição do grafo que representa o mapa do prédio. O arquivo **nodes.txt** contém uma descrição da região dos nós do grafo de uma outra forma. O arquivo **sequence.txt** contém uma sequência de nós que será lida pelo programa e executada pelo Pioneer. Por fim, existem duas pastas em **src**: **bitmaps** e **grids**. Em **bitmaps** se encontram os arquivos das imagens que representam as grades de ocupação e em **grids** se encontram os arquivos texto que representam as grades. Na pasta raiz estão os arquivos essenciais do pacote, como **CMakeLists**, **package** e **README**.

Atenção para um detalhe importante: para executar o programa corretamente, é preciso que o comando **roslaunch fcr2017 obstacle_avoidance_node** seja executado a partir da pasta **src** do pacote **fcr2017**.

4. Detecção e desvio de obstáculos

i) Processamento de dados do sensor

O algoritmo de processamento de dados do sensor foi baseado no método VFH. Uma simplificação foi feita na construção do histograma, em que ao invés de construir uma malha de ocupação com valores probabilísticos para depois converter a malha num histograma, os valores de densidade polar de objetos foram calculados de forma mais simples e o histograma foi construído diretamente a partir das medidas do sensor laser.

Primeiro, varre-se cada medida dos dados do laser com a função **constructPod()** e os valores de “convicção” de cada medida **i** são calculados considerando-se as medidas **i-1**, **i** e **i+1** do sensor. Com isto, se obtém a densidade de objetos de cada medida e a densidade de objetos de cada setor é dada pela soma das densidades de cada medida num setor. Em seguida, baseando-se na função de suavização utilizada em [1], a função **smoothPod()** é usada para se obter um vetor de densidade polar de objetos suavizado para cada setor.

Com isto, a função **findValley()** varre o vetor de densidades suavizado à procura de vales, que são setores contíguos com densidade de objetos menores que um determinado limite, chamado de **valleyThr**. Depois que um vale de tamanho mínimo (**smax**) é determinado, a direção média do vale é determinada e comparada com a direção sendo atualmente seguida pelo robô. O vale mais alinhado com esta direção é tido como o vale alvo. A direção sendo seguida pelo robô é determinada pela reta que liga o robô ao seu alvo atual. Se não houver nenhum vale disponível, o robô gira no sentido anti-horário até encontrar um vale.

ii) Algoritmo para desvio de obstáculos

Antes da execução do algoritmo, alguns limites são estabelecidos, como a já mencionada densidade máxima para que um setor seja considerado vazio (**valleyThr**) e dois limites de proximidade ao obstáculo mais próximo (**proxLimit1** e **proxLimit2**). O obstáculo mais próximo detectado pelos sensores é determinado pela função **findClosestPoint()**.

Estes dois limites de distância regem o comportamento do controle da velocidade. Um deles, **proxLimit1**, determina a distância a partir da qual o controle da velocidade começa a passar da função que faz o robô seguir um alvo, **followGoal()**, para a função que desvia o robô de um obstáculo, **evadeObject()**. A função **followGoal()** será detalhada mais à frente. A função **evadeObject()** determina valores de velocidade angular e linear a fim de desviar o robô para a direção determinada por **findValley()**. O segundo limite, **proxLimit2**, que determina uma distância menor, faz com que o robô reduza sua velocidade linear para zero enquanto manobra para se alinhar com o vale selecionado, e também determina que todo o controle de velocidade será feito por **evadeObject()**.

Este algoritmo possui algumas limitações. Ele não conseguirá desviar de objetos menores que a resolução dos setores. O robô também poderá não passar por alguns vales apertados. Veja que o método VFH, conforme descrito por [1], pode realmente oferecer limitações em ambientes muito apertados. Isso faz com que seja difícil que o robô consiga navegar ao redor de um objeto e voltar ao caminho original se não houver espaço o suficiente. O que acontece é nestes casos é que o robô dá meia-volta mas tenta retomar sua rota, se mantendo preso na mesma região enquanto o obstáculo estiver atrapalhando.

5. Planejamento e execução de trajetória

i) Planejamento de trajetória para o mapeamento

Uma das primeiras tarefas do algoritmo é o planejamento da trajetória. Conforme descrito acima, informações sobre o grafo podem ser obtidas do arquivo **graph.txt**. Para ler estas informações e montar uma representação interna do grafo, o arquivo é lido pela função **constructGraph()**, que monta um vetor do tipo **Node**, em que **Node** é uma estrutura criada para implementar os nós do grafo. Outra estrutura criada para este fim foi a classe **Edge**, que representa um arco no grafo.

A estrutura **Node** contém membros que definem sua região e seus vizinhos, e métodos que permitem verificar se um ponto está dentro de um nó e auxiliam na construção do grafo. A estrutura **Edge** possui um membro que indica o peso do arco e outro que armazena um ponteiro para um nó. As particularidades da implementação e leitura do arquivo podem ser conferidas em **my_classes.h** e na definição da função **constructGraph()** em **my_functions.h**.

Depois de criado o vetor de nós, a função **constructPathFromFile()** lê o arquivo texto **sequence.txt**, que contém uma sequência de nós previamente determinada. O uso de sequência pré-determinada ao invés de um algoritmo foi uma escolha feita para simplificar o programa, já que o CIC é um ambiente bem estruturado e é fácil para um ser humano determinar um caminho que visite todos os nós desejados manualmente. Além disso, isto permite mais flexibilidade para o robô começar de qualquer lugar do mapa. Conforme os valores são lidos de **sequence.txt**, os nós do grafo são acessados e seus centros são estabelecidos como alvos num vetor do tipo **Point**, que foi uma estrutura criada para representar coordenadas (x, y).

Depois da construção do caminho, alguns indicadores, como **step_**, **finishedMapping_** e **finishedMoving_** são inicializados e o algoritmo parte para a etapa de execução do caminho. As funções dos indicadores podem ser conferidas nos arquivos de código-fonte do pacote, e a maioria dos nomes é autoexplicativo.

O mapa do CIC foi dividido da seguinte maneira para este trabalho final:

.
.		.		4		.
.	.	1	2	3	5	.

ii) Execução da trajetória e exploração do mapa

A função **algorithm()** é constantemente chamada num laço até que o nó seja encerrado. Nela, existem três situações gerais possíveis: o robô está mapeando o ambiente, o robô terminou de mapear o ambiente e está esperando um comando de posição ou o robô está se deslocando para a posição final.

A função **followGoal()** determina um comando de velocidade para que o robô se mantenha em direção ao alvo atual da sequência de nós. O robô mantém um contador (**step_**) que determina em que etapa do caminho ele está. Quando o robô chega suficiente perto do alvo, o contador é incrementado e o indicador **spin_** é configurado para 1 para que a função **lookAround()** tome controle da velocidade e faça com que o robô gire em torno de si mesmo e dê uma olhada ao seu redor. Depois que o robô gira 165°, **spin_** se torna zero e o robô dá continuidade à trajetória. Isto permite o acesso a pontos da grade de ocupação que antes não eram acessíveis, como o bordo do nó fica atrás do robô quando ele entra num nó. Esta não é uma abordagem perfeita, porque não garante que o robô vai girar no sentido que permitirá a visualização de mais células e nem garante que tais células serão observadas por tempo suficiente, mas é uma boa aproximação para eliminar algumas incertezas no mapa.

Quando o robô chega ao seu último alvo, o algoritmo configura o indicador **finishedMapping_** e outros indicadores como **step_** e **takeCommand_**, e o algoritmo zera as velocidades e não permite mais o controle da velocidade por **lookAround()**, **evadeObject()** ou **followGoal()**. O robô fica, então, à espera de um comando de posição vindo do nó **user**.

O controle da velocidade, quando **spin_** é zero, define de forma linear quem tem mais controle sobre a velocidade, **evadeObject()** ou **followGoal()**. Baseando-se na distância do objeto mais próximo e dos limites **proxLimit1** e **proxLimit2**, um cálculo é feito em que pesos são atribuídos para os comandos dados por **evadeObject()** e **followGoal()**. Para distâncias acima de **proxLimit1**, **followGoal()** tem total controle sobre o robô. Para distâncias abaixo de **proxLimit2**, **evadeObject()** tem total controle sobre o robô. Entre **proxLimit1** e **proxLimit2**, o controle dá pesos que variam linearmente com a distância para cada função. Isto permite que não haja alterações muito bruscas na velocidade do robô assim que ele encontra um obstáculo.

iii) Construção da grade de ocupação

Uma das tarefas na inicialização do código, além de montar o grafo do CIC, é criar e preencher todas as grades de ocupação com 0.5. A função **constructMap()** é que cuida dessas tarefas, em que o arquivo **nodes.txt** é lido e todas as matrizes são alocadas e preenchidas. Quando o programa é inicializado, se houverem arquivos texto na pasta **grids** de execuções passadas, estes serão lidos e seus valores reutilizados. Se não houverem arquivos presentes, todas as grades são criadas do zero.

A estrutura **Grid** armazena as informações da grade, como a matriz dos valores, a altura e a largura da grade e o seu centro. Todas essas informações são lidas de **nodes.txt** no momento da criação do mapa. O mapa é representado internamente como um vetor do tipo **Grid**.

O preenchimento da grade de ocupação se dá pela função **constructOccupancyGrid()**, a partir de cálculos feitos para cada célula. Primeiro determina-se qual é o índice **k** do vetor de medidas do laser que passa mais próximo do centro da célula. Isto é feito determinando-se a posição relativa da célula com relação ao sensor e recuperando o ângulo em coordenadas polares. Depois disso, o índice **k** e alguns dos seus vizinhos são checados no vetor e se a medida de algum deles, em coordenadas retangulares, caracteriza um ponto dentro da célula, o valor da célula se aproxima de um. Se a medida do índice central **k** não for um ponto dentro da célula e a distância aferida for maior do que a distância do centro da célula ao sensor, então o algoritmo entende que a célula está desocupada e aproxima o valor da célula de zero.

O valor de uma célula, conforme já mencionado, é calculado processando os dados por meio de um filtro, implementado na forma de uma média ponderada móvel. Para este fim, uma contagem do número de iterações de uma grade é mantida no membro **iterNum** de **Grid**. A implementação do filtro pode ser conferida no arquivo fonte e no README do pacote, e é feita de forma iterativa, armazenando somente os dados da iteração anterior.

A todo instante, a grade de ocupação é impressa no terminal. Sempre que há uma transição de nós ou o programa marca **finishedMapping_** como 1, o algoritmo escreve a grade que estava sendo processada na memória, tanto na forma de um arquivo texto quanto na forma de um bitmap.

Para uma recordação do preenchimento do mapa conforme o robô vai se movendo, foi criado um nó **print** que tem a função de criar uma imagem da grade atual a cada dois segundos. Isto se difere da criação de imagens pelo nó principal porque as imagens criadas por **print** não são sobrescritas e representam um histórico do mapa no tempo. A cada vez que uma grade é atualizada, ela é publicada no tópico **/grids** pelo programa principal. O nó **print**, quando é executado numa pasta que contém duas pastas **bitmaps** e **grids**, lê estas matrizes e a cada dois segundos gera uma imagem e um arquivo texto da grade.

A renderização de um bitmap a partir de uma matriz foi implementada seguindo-se as instruções no artigo da Wikipedia sobre o formato Bitmap [4]. Isto envolve a criação e a impressão de cabeçalhos de identificação do arquivo e do formato e de pixels num arquivo binário. A impressão de texto colorido no terminal seguiu instruções do artigo da Wikipedia sobre os caracteres de escape ANSI [5].

iv) Deslocamento até a posição final

Assim que o Pioneer termina seu mapeamento, ele fica à espera de um comando de posição vindo do nó **user**. Este nó pode ser executado em qualquer pasta. O nó lê uma entrada do usuário, publica a entrada no tópico **/user_pos** e a callback deste tópico no programa principal faz uma breve verificação dos dados e, se a mensagem for aceita, o programa faz uma nova trajetória que leve o robô de seu nó atual até a posição final.

Uma limitação é que se a configuração final do robô determinar um ponto muito próximo de uma parede, pode ser que o robô não consiga chegar a sua posição. Isto acontece porque seus sensores indicariam um obstáculo próximo e não deixam o robô se aproximar demais da parede sem ele tentar desviar. Uma solução envolveria reformular o algoritmo de detecção de obstáculos para que o robô tivesse **proxLimit1** e **proxLimit2** menores ou desativar o desvio de obstáculos quando o robô chega perto do seu alvo.

Depois de lidos os dados, o algoritmo chama a função **retrieveNode()** para determinar a que nó pertence a configuração indicada. Com isto, a função **searchGraph()**, que usa o algoritmo de Dijkstra, procura o menor caminho entre o nó atual e o nó determinado por **retrieveNode()**. Esta função gera um vetor de nós ordenados conforme o caminho encontrado. A função **constructPathFromPosition()** é então chamada e constrói um novo caminho que leva o robô até a posição final.

Com o novo caminho, o algoritmo se comporta como no caso da exploração, e o controle para levar o robô até a posição final é o mesmo. A diferença é que agora a variável **spin_** serve para indicar que o robô chegou a sua posição final e que a função **finalAdjustment()** deve reorientar o Pioneer para que ele fique com a orientação indicada pelo usuário.

6. Adaptações para o robô real

Antes de tudo, para rodar o programa no robô real, o trabalho teve de ser adaptado para ler as medidas do rangefinder no tópico **/scan** ao invés de no tópico **/hokuyo_scan**, e o *link* **hokuyo_link** teve de ser modificado para **laser**.

Outra adaptação foi permitir que o robô começasse de qualquer lugar do mapa, e para este fim foi criada uma constante **posOffset** que armazena a posição inicial do robô. Bastou somar esta posição às leituras da odometria para que o programa entendesse que o robô estava em outra posição inicial não sendo a origem.

Por fim, uma última adaptação foi a implementação de uma medida de segurança contra problemas de conexão com o sensor laser. O sensor já vinha apresentando alguns problemas de conexão desde que começaram os testes para o trabalho final, mas nada muito grave e que impedisse completamente o mapeamento. Conforme a semana foi passando, os problemas de conexão intermitentes (perda de conexão de no máximo 1 segundo) se tornaram problemas mais sérios, em que o laser não publicava mais no tópico **/scan**, e só voltava a publicar se alguns dos fios na conexão USB fossem ajustados. Com isto, uma verificação do tempo de envio da última mensagem no tópico foi criada no programa. Dentro da callback do laser, o programa guarda o momento no tempo em que a mensagem foi recebida. Quando o nó vai processar as medidas dos laser, o delay desde que a mensagem foi recebida é calculado e, se for maior do que **0.5s**, o nó considera que o laser não está publicando corretamente em **/scan** e para completamente o robô, já que, pelo menos no meu trabalho, sem o laser o desvio de obstáculos não funciona.

7. Listagem dos arquivos entregues

Vários vídeos [6] foram gravados para mostrar os resultados da execução do trabalho. Os resultados dos mapeamentos de cada vídeo se encontram na pasta **resultados** [7]. Na pasta **inter** se encontram os resultados de cada execução do programa no robô. Em **final** se encontram os resultados mais importantes e serão comentados mais adiante.

Os resultados nas pastas **p1**, **p2** e **p3** foram obtidos antes da criação do nó **print**, então apenas estão presentes os resultados finais dos mapas. Nas pastas **temp***, como o nó já havia sido criado, pode ser conferido um histórico do mapeamento feito pelo robô. Além disso, nas pastas **temp*** também podem ser encontrados arquivos **.gif** que foram criados a partir das imagens geradas, conforme explicado no **vídeo 14**.

Veja que todas as imagens da pasta **inter** foram incluídas no trabalho a fim de registrar todas as execuções do robô, mas não necessariamente os arquivos precisam ser levados em conta. De fato, a nomeação dos arquivos e pastas dentro de **inter** não é muito intuitiva, a não ser para mim, que estabeleci os nomes. Você pode ignorar estes arquivos e considerar somente os resultados mais importantes do mapeamento, incluídos na próxima seção e presentes na pasta **final**.

Como em cada dia uma versão diferente do algoritmo foi carregada no Pioneer, as diferentes versões foram guardadas e todas estão disponíveis na pasta **old** [8]. A seguinte tabela relaciona a execução de cada vídeo com uma pasta em **inter** e uma versão do trabalho, caso os arquivos queiram ser conferidos. Note que a **versão 5** só rodou em simulação e apresenta a mudança de alguns comentários e nos nomes de algumas variáveis e funções com relação à **versão 4**, além de uma pequena correção na escolha de vales válidos na função de desvio de obstáculos. A **versão 5** foi a versão entregue no Moodle. Note que infelizmente o vídeo da execução que gerou a pasta **p2** foi perdido.

Versão do código	v1	v2	v3	v3.1	v3.2
Pasta de imagens	p1, p2	p3	temp3, temp4	temp5, temp6	temp2
Vídeo de execução	2 (p1), ? (p2)	6 (p3)	13 (temp3), 18 (temp4)	20 (temp5), 21 (temp6)	Nenhum

A seguinte tabela relaciona cada imagem em **final** a uma execução do programa, identificada pelo número do vídeo. As primeiras imagens são resultado da simulação e usadas somente para fins de comparação. Os nomes originais das imagens foram mantidos para facilitar sua localização em **inter**. Os nomes das imagens são da forma “image” + nº do nó + “-” + total de iterações na grade do nó + “.bmp”.

Nó 2		Nó 3		Nó 4		Nó 5	
Imagem / Vídeo		Imagem / Vídeo		Imagem / Vídeo		Imagem / Vídeo	
image2-3660	X	image3-5572	X	image4-4720	X	image5-4564	X
image2a	2	image3-33	20	image4-52	21	image5-91	20
image2b	?	image3-4031	20	image4-1952	21	image5-426	18
image2c	6	image3-311	21	image4-4653	21	image5-4593	20

Não é preciso memorizar tudo isto, já que mais à frente os vídeos serão explicados e as imagens geradas e versões do programa usadas serão novamente indicadas.

8. Vídeos e resultados

A execução da **versão 1** do trabalho no sábado, dia 04/02, pode ser conferida no **vídeo 2**, e esta execução gerou as imagens presentes na pasta **p1**. Nesta execução foi possível notar que a odometria não era muito precisa, e o robô tende a se curvar para a esquerda conforme tenta seguir uma linha reta. Neste mesmo dia houve outra execução do algoritmo, que gerou a pasta **p2**, mas o vídeo foi perdido. Somente o **nó 2** foi mapeado. Como o robô tende para a esquerda conforme se movimenta, seu sistema de eixos rotaciona no sentido anti-horário e, ainda, tende a deslocar sua origem a depender da direção do movimento do robô. Isto pode explicar muitos dos deslocamentos nos mapas observados nas imagens geradas mais à frente.

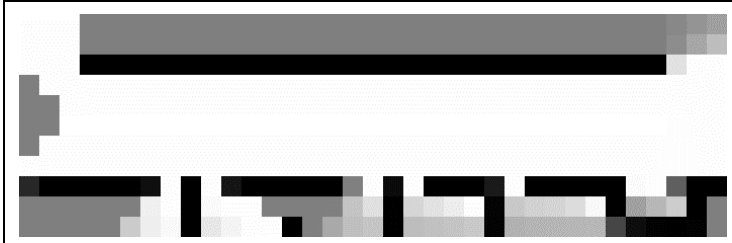
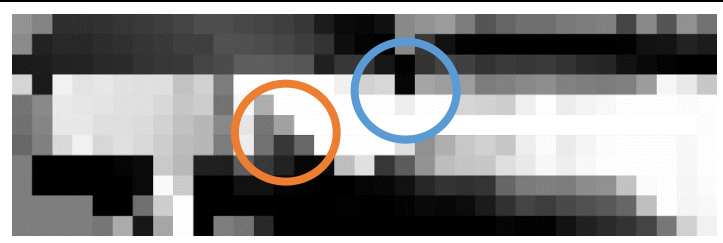
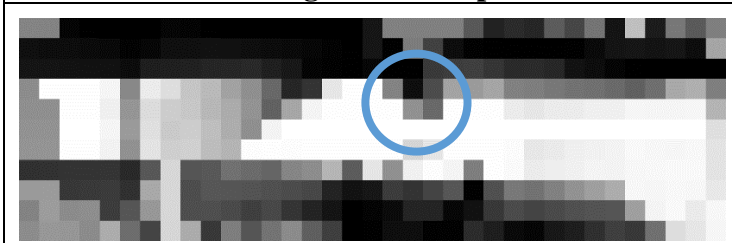
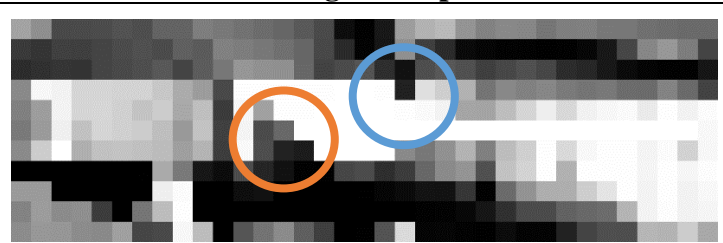
Na segunda-feira, dia 06/02, a **versão 2** do trabalho foi executada no robô obtendo a pasta **p3**, e a execução pode ser conferida no **vídeo 6**. Mais uma vez, o robô fez uma curva para a esquerda e a odometria fez o programa entender que chegou no alvo antes da hora. Neste mesmo dia outra execução foi feita partindo do **nó 3** ao invés do **nó 1**, mas o desvio de obstáculos não estava funcionando. A primeira ideia era de que o robô não estava identificando corretamente as pernas das cadeiras como obstáculos, mas é possível que o sensor laser estivesse começando a apresentar problemas. Neste dia, depois desta tentativa falha, o trabalho começou a apresentar falhas de segmentação constantemente, então só esta pasta foi gerada. Na terça-feira à noite, dia 07/02, descobri que as falhas foram geradas por um vetor de medidas vazio no código, que estava vazio porque o laser não estava mais publicando no tópico `/scan`. Somente o **nó 2** foi mapeado. Na terça-feira esta versão foi novamente carregada no robô, mas o problema na conexão persistiu e nós ainda não havíamos descoberto que este problema existia.

Na quarta-feira, dia 08/02, eu havia descoberto que eu não estava verificando se o vetor de medidas do laser estava vazio antes de operar os dados. Com esta correção feita, a **versão 3** do trabalho foi executada no robô, que pode ser conferida no **vídeo 13**. Esta execução mapeou os **nós 3 e 4** gerando a pasta **temp3**, mas os arquivos de uma simulação anterior não foram deletados antes de executar o algoritmo, então os mapas gerados não foram válidos. Neste dia outra execução foi feita e pode ser conferida no **vídeo 18**, em que o robô mapeia os **nós 3 e 5**, gerando a pasta **temp4**. Para esta execução, os mapas foram gerados corretamente do zero. No final do **vídeo 18** o Pioneer recebe um comando de posição final e, apesar do mal funcionamento do laser em alguns momentos, consegue se dirigir ao ponto final e se orientar conforme o ângulo informado pelo usuário. Neste dia, o sensor laser apresentou muitos problemas de conexão. Portanto, à tarde neste mesmo dia, foi implementada uma pequena medida de segurança no algoritmo, que verifica quando a última mensagem foi publicada em `/scan` e para o robô completamente se a mensagem for muito antiga, indicando que o sensor parou de funcionar. O **vídeo 19** mostra o funcionamento desta medida. Note que somente verificar se o vetor de medidas está vazio não é proteção suficiente contra problemas na conexão, e esta outra verificação teve de ser criada para lidar com isso. Na quinta-feira, dia 09/02, a conexão com o sensor laser estava muito deteriorada e uma substituição na conexão USB teve de ser feita.

Na sexta-feira, dia 10/02, uma **versão 4** do trabalho, com esta medida de segurança, foi executada no robô, apesar de a conexão USB ter sido consertada. O **vídeo 20** mostra o mapeamento dos **nós 3 e 5**, gerando a pasta **temp5**. O **vídeo 21** mostra o mapeamento dos **nós 3 e 4**, gerando a pasta **temp6**. O **vídeo 22** mostra o robô tentando se deslocar para a posição final, mas não conseguindo. Como a odometria não foi reiniciada entre a gravação dos vídeos, o acúmulo dos erros fez com que o robô estivesse com seus eixos cartesianos muito rotacionados, fazendo com que o alvo, que deveria estar no corredor, estar dentro de uma das salas.

i) Nó 2

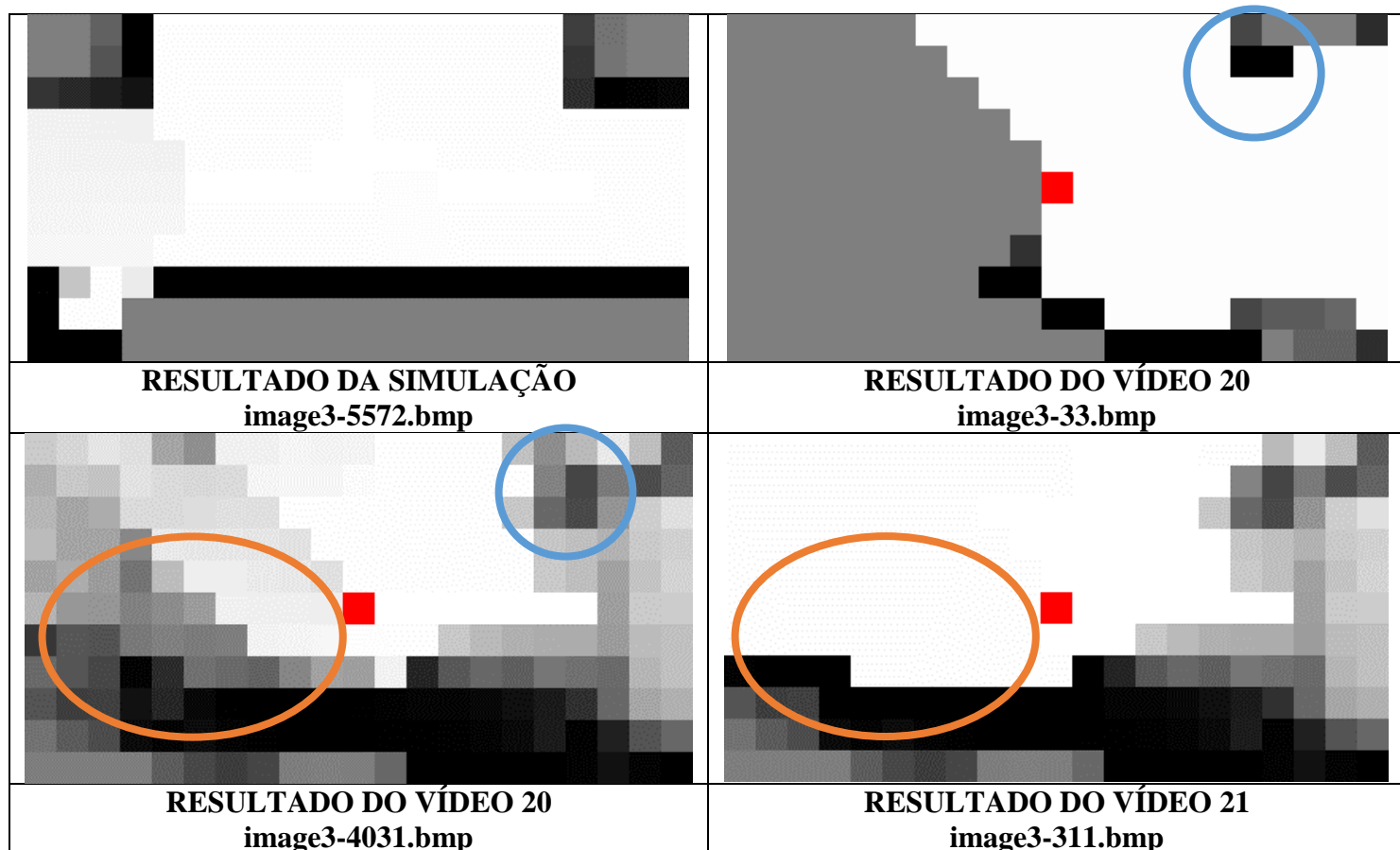
Para o **nó 2**, o mapeamento feito pelo robô gerou os seguintes resultados, percorrendo o nó da esquerda para a direita. Veja que as três imagens reais foram geradas pelo programa principal, já que o nó **print** ainda não havia sido implementado.

	
RESULTADO DA SIMULAÇÃO image2-3660.bmp	RESULTADO DO VÍDEO 2 image2a.bmp
	
RESULTADO DO VÍDEO PERDIDO image2b.bmp	RESULTADO DO VÍDEO 6 image2c.bmp

Nos três mapas, é possível ver a pilastra e a lixeira no mapa, circuladas em azul. Além disso, em **image2a** e **image2c** também é possível ver uma área livre na entrada da sala da professora Carla, já que a porta estava aberta. Note também ao final do corredor que o robô começa a detectar as grades vazadas à direita, que por não serem sólidas, fazem o robô achar que a área está parcialmente livre. Por fim, é possível também ver as cadeiras que estavam no corredor em **image2a** e **image2c**, circuladas em laranja.

ii) Nó 3

Para o **nó 3**, o mapeamento feito pelo robô gerou vários resultados, mas poucos foram satisfatórios. Abaixo estão alguns dos mais precisos. As três imagens reais foram geradas pelo nó **print** ao invés do programa principal. A célula vermelha indica a posição do robô no mapa.



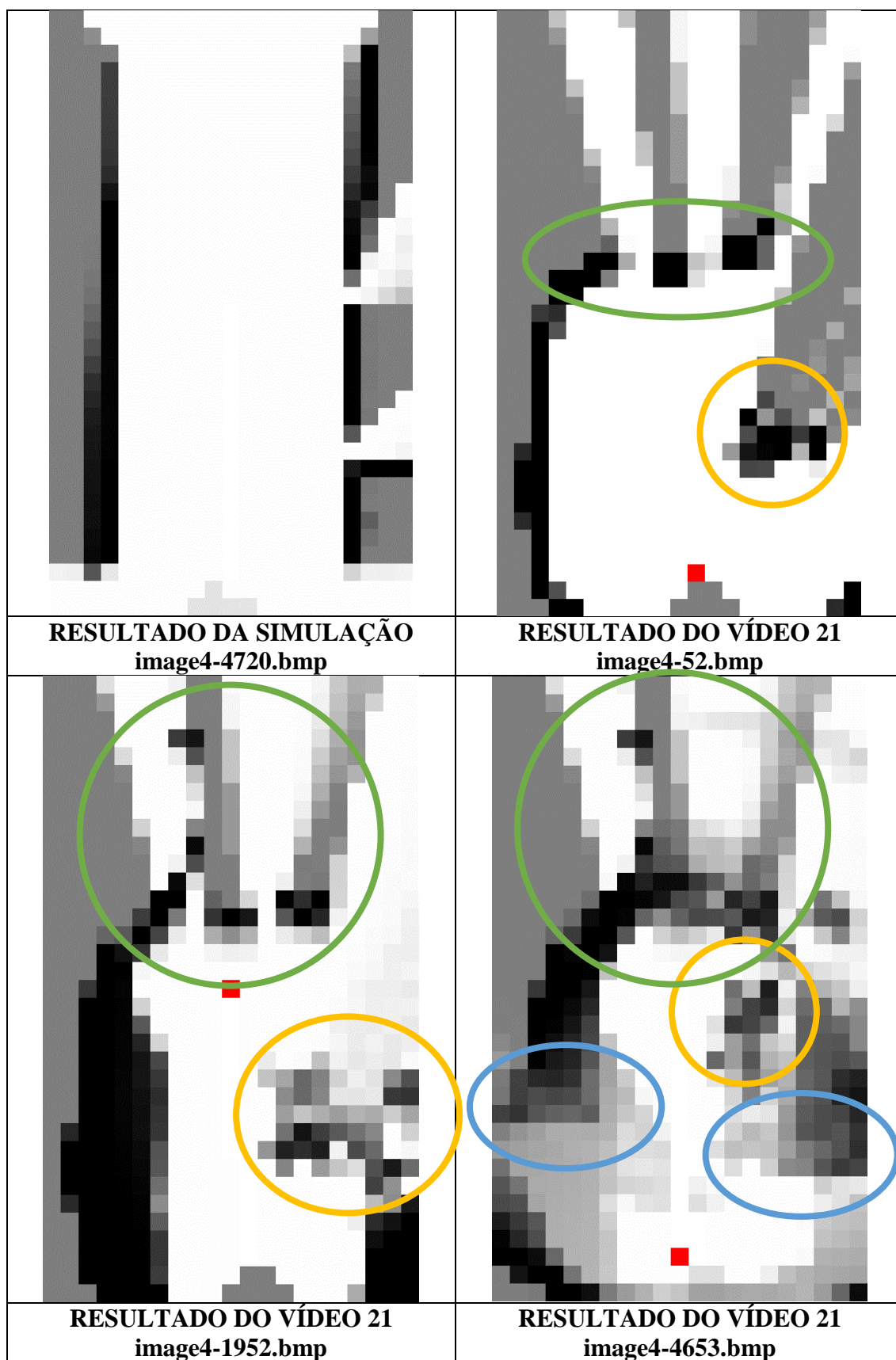
Para este nó, os resultados foram poucos satisfatórios no geral, então somente algumas imagens intermediárias foram tomadas como resultado. Na primeira imagem, **image3-33**, o robô estava parado e é possível ver uma pilastra, circulado de azul. Se você observar o arquivo **temp5-50.gif**, verá que conforme o robô gira para ver atrás de si, muitas “manchas” aparecem no mapa, e elas seguem um certo padrão. Uma primeira teoria era de estas manchas eram causadas pelas falhas de conexão do laser, que estariam fazendo o robô mapear a região com um conjunto de medidas inválido. Como na sexta-feira a conexão USB já havia sido reparada, foi possível ver que esta não era a causa das manchas. Uma outra teoria era de que isto era causado simplesmente porque o movimento do robô afeta as medidas do sensor laser, mas isto não foi analisado.

Algumas iterações depois, na imagem **image3-4031**, quando o robô havia mapeado o **nó 5** e voltado, é possível ver que as paredes inferiores na imagem subiram. Isto possivelmente se deve aos erros na odometria, que deslocaram o ponto alvo para baixo com relação ao ponto real. Isto possivelmente também explica o sumiço da pilastra, cuja região inicialmente ocupada foi esbranquiçada.

Por fim, em **image3-311**, com o robô parado, sem resetar a odometria e aproveitando o mapa antigo, o robô consegue definir mais claramente as paredes do nó. Isto é mais um indício de que as manchas são causadas pelo movimento do robô.

iii) Nó 4

Para o nó 4, os mapas gerados foram mais satisfatórios.



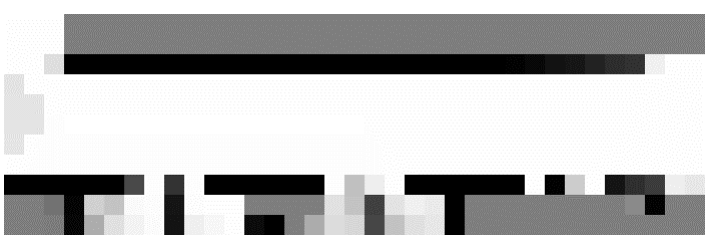
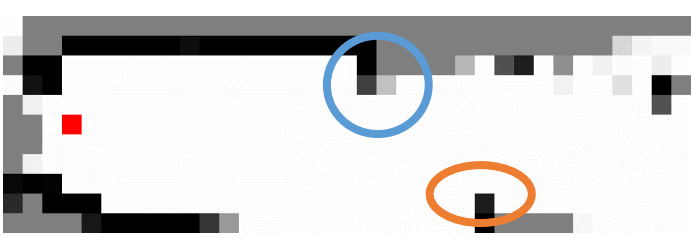
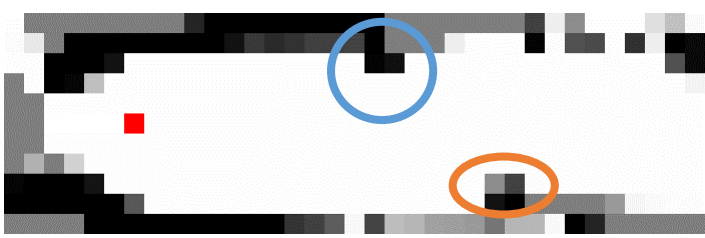
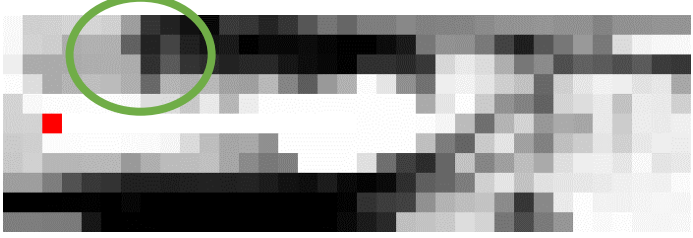
Todos os três mapas gerados são de uma mesma execução, então a comparação é mais fácil. Lembre-se ainda que o **nó 5** foi mapeado antes deste nó e a odometria não foi reiniciada, então havia erro acumulado. O primeiro indício deste erro é que o programa não detectou as paredes à direita no mapa, mas isto pode também ter sido causado pelo fato de que as grades são vazadas.

Uma primeira observação é que é possível ver a mudança nas “sombras” ao longo das três imagens. No arquivo **temp6-50.gif** é possível acompanhar estas mudanças. Além disto, é possível ver que o robô detectou as cadeiras, poltronas, mesas e sofás da região, circulados de verde e amarelo.

Uma outra observação importante é o deslocamento dos móveis circulados de amarelo no mapa. Em **image4-4653**, os móveis estão mais para cima em relação aos móveis em **image4-52**. Isto ilustra outro problema da odometria, que indicava que o robô andou mais do que ele realmente andou. Quando ele chega no que seria o bordo inferior do **nó 4**, o mapa está tão deslocado para cima que parte do **nó 3** é erroneamente mapeada no **nó 4**, incluindo o que parece ser a pilastra que sumiu de **image3-4031** e parte da parede superior do **nó 2**, ambas circuladas de azul.

iv)Nó 5

Para o nó 5, os resultados foram razoavelmente bons.

	
RESULTADO DA SIMULAÇÃO image5-4564.bmp	RESULTADO DO VÍDEO 20 image5-91.bmp
	
RESULTADO DO VÍDEO 18 image5-426.bmp	RESULTADO DO VÍDEO 20 image5-4593.bmp

Em **image5-91** e em **image5-426**, é possível ver que o robô, assim que entra no nó, consegue mapear corretamente a pilastra no corredor e o bebedouro, circulos de azul e laranja, respectivamente. Note ainda que estas foram duas execuções diferentes do trabalho. Em **image5-4593**, que é uma imagem gerada na mesma execução de **image5-91**, o robô está saindo do nó, e no arquivo **temp5-50.gif** é possível acompanhar como o robô consegue inicialmente mapear o nó mas acaba sofrendo com as “manchas” ao dar meia-volta e se dirigir ao nó 3. É plausível ainda que a região circulada de verde corresponda à pilastra que sumiu do nó 3.

9. Bibliografia

- [1] Borenstein, J. e Koren, Y., “The Vector Field Histogram – Fast Obstacle Avoidance For Mobile Robots”, IEEE Journal of Robotics and Automation Vol 7, No 3, Junho 1991, pp. 278-288.
- [2] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs”, Numerische Mathematik 1, 269-271 (1959).
- [3] https://en.wikipedia.org/wiki/Moving_average#Weighted_moving_average
- [4] https://en.wikipedia.org/wiki/BMP_file_format
- [5] https://en.wikipedia.org/wiki/ANSI_escape_code#Colors
- [6] Link para os vídeos:
<https://www.dropbox.com/sh/n34wfe586jw8875/AACqAKvjApyfKbOID7EjZ700a?dl=0>
- [7] Link para as imagens:
https://www.dropbox.com/sh/c9iai341m2mslcc/AADQwkeUgE6DdIH9OGNI_vXMa?dl=0
- [8] Link para as versões:
<https://www.dropbox.com/sh/04in6a37qzv3yf5/AABlrK1TJCUHddwVXFPyeUJaa?dl=0>