UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Rafael Elias Pedretti

PREVISÃO TEMPORAL DE NAVEGAÇÃO PARA ROBÔS MÓVEIS AUTÔNOMOS

Florianópolis

2014

Rafael Elias Pedretti

PREVISÃO TEMPORAL DE NAVEGAÇÃO PARA ROBÔS MÓVEIS AUTÔNOMOS

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação como parte dos requisitos para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientadora: Prof.ª Patrícia Della Mea Plentz

Florianópolis

2014

Rafael Elias Pedretti

PREVISÃO TEMPORAL DE NAVEGAÇÃO PARA ROBÔS MÓVEIS AUTÔNOMOS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Ciências da Computação.

Florianópolis, 15 de Agosto 2014.

Prof.ª Patrícia Della Mea Plentz
Coordenadora

Banca Examinadora:

Prof.º Dr. Mauro Roisenberg
Presidente

Prof.ª Patrícia Della Mea Plentz
Orientadora

Msc. Douglas Wildgrub Bertol

Dedico este trabalho a minha família que acreditou em mim e me deu educação e caráter para enfrentar as dificuldades da vida.

AGRADECIMENTOS

Agradeço a Profª Patrícia Plentz pela ajuda e oportunidade de trabalhar neste projeto. Também agradeço meus amigos cujo suporte foi essencial para o término desse trabalho.

“1 - A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2 - A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.
3- A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.”

Isaac Asimov

RESUMO

A robótica móvel é uma área muito abrangente que está presente em todos os tipos de ambientes. Existem robôs de patrulhamento, reconhecimento, entretenimento e muitas outras funções, todos podendo ser de ambientes terrestres, subaquáticos, aéreos e cada um com sua função específica, com seu tipo de navegação, comunicação e interação com o mundo externo.
Justamente por haver tantas técnicas e tecnologias diferentes é difícil criar mecanismos de previsão de cumprimento dessas tarefas quando há restrições de tempo e/ou autonomia. O objetivo deste documento, através de análises qualitativas dos fatores já mencionados, é expor os fatores conceituais e tecnológicos que estão envolvidos nas decisões de projeto para a criação de tais mecanismos, enfatizando a diferença destes gargalos em relação ao tipo de navegação, comunicação e ambiente no qual o robô será inserido.

Palavras-chave: Robótica. Móvel. Tarefa. Previsão. Mecanismos. Navegação

ABSTRACT

The mobile robotic is a very broad area which is present in all types of environments. There are robots for patrolling, reconnaissance, entertainment and many other functions, all of which can be terrestrial, aquatic, aerial and each with its specific function, with its type of navigation, communication and interaction with the outside world.
Precisely because there are so many different techniques and technologies it's difficult to create prediction mechanisms for these tasks when there is time and/or autonomy constraints. The purpose of this document, through qualitative analysis of the factors already mentioned, is to expose the conceptual and technological factors that are involved in design decisions for the creation of such mechanisms, emphasizing the difference of these bottlenecks in relation to the type of navigation, communication and environment in which the robot will be inserted.

Keywords: Robotics. Mobile. Task. Prediction. Mechanisms. Navigation

LISTA DE FIGURAS

Figura 2.1 Sistema de Tempo Real 30
Figura 2.2 Robô Seekur Jr 36
Figura 2.3 Comparativo da execução dos algoritmos da família bug (NG; BRäUNL, 2007) 38
Figura 3.1 Mapa do terceiro andar do INE 42
Figura 4.1 Pioneer 3-DX 51
Figura 4.2 Sensor laser Sick LMS-200 52
Figura 4.3 Sensor laser Sick LMS-200 instalado no robô 53
Figura 4.4 Interface do aplicativo Mapper3 54
Figura 4.5 Interface do simulador MobileSim 55
Figura 4.6 Aplicativo MobileEyes exibindo o planejamento de rota 56
Figura 6.1 Taxas de acerto e erro com histórico de 25 entradas 68
Figura 6.2 Taxas de acerto e erro com histórico de 50 entradas 70
Figura 6.3 Taxas de acerto e erro com histórico de 100 entradas 72
Figura 6.4 Taxas de acerto e erro com histórico de 200 entradas 73
Figura 6.5 Taxas de acertos das tarefas 74

LISTA DE TABELAS

Tabela 3.1 Exemplo de histórico 47
Tabela 6.1 Relação do tamanho do histórico com as taxas de acerto e falsos positivos 69

LISTA DE ABREVIATURAS E SIGLAS

STR	Sistemas de Tempo Real	29
GPS	*Global Positioning System*	33
SLAM	*Simultanious Localization and Mapping*	34
PID	Proportional-Integral-Derivative	39
INE	Departamento de Informática e Estatística	41
API	Application Programming Interface	41
SGBD	Software de Gerência de Banco de Dados	52
ARIA	*Advanced Robotics Interface for Applications*	53
ARNL	*Advanced Robotics Interface for Navigation and Localization*	53
BD	Banco de Dados	56

LISTA DE SÍMBOLOS

d_e	Distância estimada até o objetivo	43
f_d	$\frac{1}{5}$ do *deadline*	45
d_{q_i}	Distância percorrida até o marco q_i	46
$\overline{d_{q_i}}$	Média dos d_{q_i}'s	46
Δd_{q_i}	Diferença entre d_{q_i} e $\overline{d_{q_i}}$	46
$\overline{v_{q_j}}$	Velocidade média do marco q_j	46
$\overline{v_{m_i}}$	Média das velocidades médias dos trechos subsequentes a q_i	48
t_e	Tempo estimado para completar a tarefa	48

SUMÁRIO

1 INTRODUÇÃO 25
1.1 JUSTIFICATIVA 25
1.2 OBJETIVOS 26
1.2.1 Objetivos Gerais 26
1.2.2 Objetivos Específicos 26
1.2.2.1 Objetivos de base 26
1.2.2.2 Objetivos avançados 27
1.3 METODOLOGIA 27
1.4 ESTRUTURA DO TRABALHO 27
2 FUNDAMENTAÇÃO TEÓRICA 29
2.1 TEMPO REAL 29
2.1.1 Criticalidade 30
2.2 ROBÓTICA MÓVEL 31
2.2.1 Classificação 31
2.2.2 Sensores 32
2.2.3 Localização 33
2.2.3.1 Localização baseada em pontos de referência (*landmarks*) 34
2.2.3.2 Localização e mapeamento simultâneos 34
2.2.4 Atuadores 35
2.3 NAVEGAÇÃO 35
2.3.1 Ambientes não conhecidos 36
2.3.2 Ambientes conhecidos 37
2.4 FONTE DE ENERGIA 39
2.5 PREVISÕES 39
3 PROPOSTA 41
3.1 DEFINIÇÃO DO CENÁRIO 41
3.2 ALGORITMO DE BASE 42
3.3 DEFINIÇÃO DO ALGORITMO PROPOSTO 44
3.3.1 Descrição Conceitual 45
3.3.2 Descrição Matemática 46
4 IMPLEMENTAÇÃO 51
4.1 HARDWARE UTILIZADO 51
4.1.1 Robô Pioneer 3-DX 51
4.1.2 Laser Sick LMS-200 52
4.2 SOFTWARE UTILIZADO 52
4.2.1 ARIA 53
4.2.2 ARNL 53

4.2.3 Mapper3 54
4.2.4 MobileSim 55
4.2.5 MobileEyes 56
4.2.6 SQlite3 56
4.2.7 Kompex 57
4.3 DEFINIÇÃO DO BANCO DE DADOS 57
4.4 IMPLEMENTAÇÃO DO BD 58
4.5 INTERFACES 59
4.6 ALGORITMOS 59
5 SIMULAÇÃO 63
5.1 PREPARAÇÃO DO AMBIENTE DE EXECUÇÃO 63
5.2 COLETA DE DADOS 64
5.3 EXECUÇÃO 64
6 RESULTADOS 67
7 CONCLUSÃO 75
7.1 TRABALHOS FUTUROS 75
7.1.1 Melhorias no banco de dados 75
7.1.2 Comparação com técnicas de Inteligência Artificial 75
Referências Bibliográficas 77
ANEXO A – Descrição do Banco de Dados 83
ANEXO B – Código Fonte 87
ANEXO C – Documentação 151
ANEXO D – Artigo 155

1 INTRODUÇÃO

Não é de hoje que a humanidade sonha com robôs para os auxiliar com tarefas diversas. A palavra *robótica* foi muito popularizada na década de 50 pelo autor de obras de ficção científica Isaac Asimov com seu conto Eu, Robô, do inglês *I, Robot* (ASIMOV, 1950) e tantas outras obras. Desde então a televisão e o cinema também fizeram uso de robôs como a aclamada franquia *Guerra nas Estrelas* com o robô diplomata C-3PO e a série *Battlestar Galactica* com sua raça de robôs inteligentes, os *Cylons*.

Apesar de serem obras de ficção o conceito ficou muito presente na indústria com seus braços robóticos que executam tarefas mais complexas com mais precisão que um ser humano e a partir desse ponto houve uma demanda muito grande desse tipo de equipamento. É possível observar que a robótica está inserida em quase todos os meios: na medicina auxiliando cirurgias, exploração de ambientes inóspitos, na guerra com os VANT's (Veículos Aéreos Não Tripulados), entre outros.

Este trabalho é voltado para a robótica móvel, onde os robôs não são fixados em local, como os robôs de uma montadora de carros, por exemplo. Atualmente existem vários tipos de robôs móveis (SIEGWART; NOURBAKHSH, 2011), sendo os mais conhecidos os com pernas, rodas e aéreos, cada um com um objetivo e modos de operação.

Os robôs móveis podem ser segmentados em duas grandes classes: os tele-operados e os autônomos. Os tele-operados são robôs que precisam de um operador humano para controlá-lo remotamente, enquanto os autônomos são robôs que conseguem se locomover e executar suas funções sem intervenções humanas.

Para um robô autônomo ter uma utilidade, ele deve ser capaz de executar as tarefas que lhe foram passadas de forma eficiente e eficaz. As atividades que um robô deve executar variam de simplesmente "passear" por um mapa sem esbarrar em nada até inspecionar tubulações (NASSIRAEI et al., 2007).

A proposta desse trabalho é fornecer à esses robôs móveis um mecanismo que os possibilitem de dizer se irão conseguir cumprir suas tarefas dentro do prazo estipulado quando houver uma restrição de tempo de execução (um *deadline*).

1.1 JUSTIFICATIVA

Em ambientes no qual se vê necessário a utilização de robôs para realização de tarefas, onde o tempo de execução deve ser considerado, deve-se

também levar em consideração as consequências dessa tarefa exceder o tempo limite ou até mesmo não ser completada. Tal mecanismo pode ser utilizado para contemplar esses casos e poder tomar ações necessárias.

Por exemplo, em um ambiente totalmente automatizado onde robôs levam materiais de um lugar a outro dentro de um determinado espaço um desses robôs começa a se comportar fora do padrão e a atrapalhar os outros robôs, que por sua vez têm que alterar suas rotas para poderem completar suas tarefas. Muitas vezes a perda de um *deadline* pode gerar custos adicionais ou ter consequências mais graves como perda de vidas sendo assim necessário meios de mitigar ou mesmo evitar por completo essas situações.

Nesses casos o mecanismo teria a função de dizer se o *deadline* será perdido e pode gerar ações corretivas a fim de diminuir o impacto dessa perda.

1.2 OBJETIVOS

Abaixo estão descritos os objetivos gerais e específicos deste trabalho.

1.2.1 Objetivos Gerais

A proposta deste trabalho está em desenvolver um mecanismo de previsão de cumprimento de tarefas em robótica móvel. Pretende-se estudar técnicas de previsão que possam ser executadas em tempo de execução e que tenham um baixo custo de processamento para evitar alocação de recursos que prejudiquem a atuação do robô.

Pretende-se também implementar o mecanismo e testá-lo a fim de verificar sua eficácia, tanto em um simulador como em um robô físico em um ambiente conhecido e com obstáculos desconhecidos.

1.2.2 Objetivos Específicos

Planeja-se modelar e implementar o mecanismo de previsão que forneça respostas coerentes e em tempo hábil onde o robô possa tomar as ações necessárias para corrigir ou mitigar os efeitos da perda do *deadline*.

1.2.2.1 Objetivos de base

Abaixo serão listados os objetivos de base deste trabalho.

- Estudar técnicas de previsão de cumprimento temporal de tarefas em robótica;
- Modelar o mecanismo com base nas técnicas estudadas;
- Definir cenários de aplicação do mecanismo
- Implementar o mecanismo definido no modelo;
- Testar o mecanismo no simulador;

1.2.2.2 Objetivos avançados

Abaixo serão listados os objetivos avançados deste trabalho.

- Mapear o ambiente de testes;
- Implementar um mecanismo mais simples para comparação com o proposto;
- Fazer uma corelação entre os resultados dos dois mecanismos;

1.3 METODOLOGIA

Este trabalho foi desenvolvido com o suporte do Laboratório de Pesquisas em Sistemas Distribuídos (LAPESD), situado no prédio do Departamento de Informática e Estatística (INE), que forneceu o robô utilizado durante o desenvolvimento.

Primeiramente foram levantados trabalhos relacionados com o tema, voltados para técnicas de movimentação, navegação e localização, seguido da modelagem do mecanismo de base e do proposto. Após essa etapa foi decidido a plataforma na qual seriam implementados os mecanismos e a linguagem de programação utilizada.

Decidida a linguagem foi então implementado o mecanismo de base e posteriormente o proposto. Em seguida foi mapeado o ambiente no qual foram testados os previsores. Por fim foram feitas análises dos dados para verificar a eficácia dessas implementações.

1.4 ESTRUTURA DO TRABALHO

No capítulo 2 está apresentados conceitos básicos para o entendimento deste documento, sendo estes relacionados a tempo real e robótica.

No capítulo **??** é apresentada a revisão bibliográfica contendo resultados de outros autores sobre o assunto. No capítulo 3 é apresentada a proposta do algoritmo de previsão juntamente com sua implementação.

No capítulo 4 são apresentadas características específicas da implementação da proposta apresentada. No capítulo 5 está apresentado como foi executada a coleta de dados e simulações sobre o mecanismo proposto.

No capítulo 6 e 7 são apresentados respectivamente os resultados obtidos e as considerações finais juntamente com sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo estão descritos os conceitos básicos sobre computação em tempo real e robótica móvel para dar o embasamento teórico necessário para o entendimento deste documento. Para não fugir do escopo do trabalho, este capítulo fica restrito apenas aos conceitos fundamentais da extensa teoria existente.

2.1 TEMPO REAL

Sistemas de tempo real (STR), de acordo com Farines, Fraga e Oliveira (2000), são sistemas computacionais que dever reagir a estímulos vindos do ambiente externos e produzirem resultados respeitando restrições temporais, ou seja, devem produzir um resultado válido dentro de um determinado intervalo de tempo. Diferentes tarefas podem ter diferentes limites de tempo de execução até produzirem os resultados desejados e também podem haver múltiplas políticas de escalonamento para garantir que essas tarefas sejam executadas dentro do prazo definido. Sendo assim o bom funcionamento do sistema de tempo real não só depende dos valores mas também do momento em que eles foram produzidos. A extrapolação do tempo de execução definido para uma tarefa caracteriza uma falha no sistema, podendo essa falha indicar uma ameaça.

A Figura 2.1 mostra um modelo simples de um sitema real, que é composto por um ambiente cujo as informações são captadas pelos sensores do sistema controlado que repassa as informações para um sistema controlador e com essas informações esse sistema, através de atuadores, se locomove pelo ambiente. Esses sensores podem capturar estímulos variados, como intensidade de luz, quantidade de gás carbônico no ar dentre outros enquanto atuadores são componentes que exercem alguma ação mecânica no ambiente como rodas, válvulas e braços. Um exemplo de STR seria o controlador de uma válvula de uma tubulação, que deve controlar o fluxo de gás para manter uma taxa constante. Se o sistema demorar para se ajustar ao fluxo pode haver muita pressão na tubulação e ocorrer um acidente.

Para gerenciar os recursos utilizados de modo que as tarefas consigam ser executadas respeitando suas restrições temporais é preciso que haja um software específico para isso, chamado de Sistema Operacional de Tempo Real, RTOS (*Real Time Operational System*). Ao contrário de um sistema operacional de um computador pessoal um RTOS deve ser determinístico e suportar implementações de tolerância a falha por replicação ativa e, quando

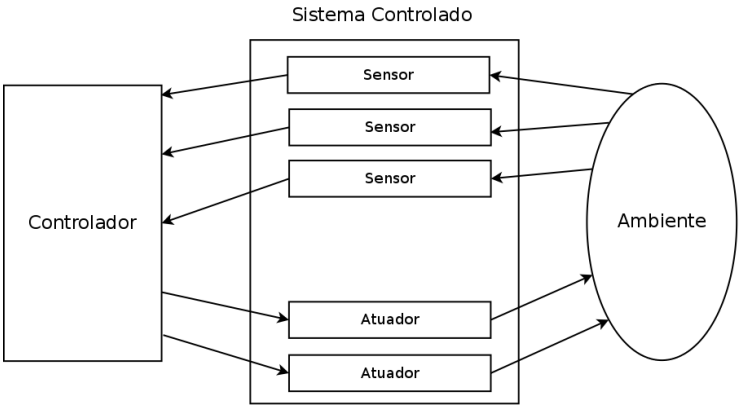


Figura 2.1: Sistema de Tempo Real

a aplicação for de segurança crítica, o RTOS deve ser certificado (KOPETZ, 2011).

Uma tarefa em um sistema de tempo real é executada várias vezes e pode ser classificada como periódica e aperiódica, levando em conta a periodicidade de ativação da mesma. Uma tarefa pode ser considerada periódica se a ca da período P ela sempre é ativada e aperiódica se a tarefa é ativada aleatoriamente. Uma tarefa ainda pode ser considerada esporádica quando existe um período mínimo de ativação.

2.1.1 Criticalidade

Um sistema de tempo real pode ser classificado em relação ao impacto que uma perda de *deadline* causa. Essas classificações são:

> **Críticos (*Hard*):** Em sistemas *hard real-time* a perda de um *deadline* gera uma falha irreversível e irrecuperável, pode causar sérios danos em equipamentos e colocar vidas em risco. Controles de freios ABS são um exemplo dessa categoria.
>
> **Não-Críticos (*Soft*):** A perda de um *deadline* não significa que o sistema falhou completamente, mas haverá uma degradação da performance. Nesse caso, as marcas temporais indicam o funcionamento desejado do sistema. Televsões digitais são exemplos dessa categoria, onde a perda ocasional de *deadline* pode degradar a imagem e/ou o som, mas não causa uma falha completa.

Firmes (*Firm*): A perda de um único *deadline* não causa uma falha no sistema, mas várias perdas podem causar uma falha completa. Quando ocorre uma perda o dado que seria produzido perde seu valor e deve ser descartado. Sistemas multimídia são exemplos essa categoria, onde a perda ocasional de *deadline* degrada a mídia, mas não causa uma falha completa.

2.2 ROBÓTICA MÓVEL

A robótica é uma área jovem, com raízes nas engenharias, computação ciências cognitivas e sociais dentre outras. A robótica alcançou grande prestígio na área de fabricação industrial com os braços robóticos, ou manipuladores, capazes de rapidamente executar tarefas repetitivas como solda ponto e pintura (SIEGWART; NOURBAKHSH, 2011).

Apesar da sua grande utilidade os manipuladores ainda tem uma desvantagem: a falta de mobilidade. Com essa demanda foram criados os robôs móveis, com diferentes capacidades de locomoção para atingir o maior número de áreas possíveis, criando novas possibilidades como por exemplo a inspeção interna de tubulações, onde para um ser humano seria muito perigoso ou de difícil acesso. Sendo assim, a robótica móvel melhorou as condições de trabalho e segurança dos envolvidos nas tarefas.

Um robô móvel é um exemplo de um STR, uma vez que ele faz uso de sensores para analisar o ambiente e se locomover pelo mesmo sem esbarrar em obstáculos.

2.2.1 Classificação

Os robôs móveis podem ser classificados em 3 grandes grupos de acordo com o tipo de locomoção utilizado e 2 grupos considerando o tipo de operação, sendo eles teleoperado e autônomos. Na locomoção então temos os grupos dos robôs terrestres, aquáticos e aéreos.

Terrestres: São robôs utilizados em terra, com os mais variados tipos de meios de locomoção. Robôs com rodas tem boa mobilidade mas podem não ser muito bons em terrenos irregulares. Esteiras resolvem esse problema tendo uma boa estabilidade em terrenos irregulares como arenosos ou com pedras, mas perde-se um pouco de mobilidade. Robôs com pernas tem vantagem em terrenos acidentados, com degraus ou outros tipos de obstáculos que podem ser passados por cima.

Aquáticos: São robôs que se locomovem na água e podem submergir permanecendo submersos por um tempo indeterminado utilizando um cabo de alimentação ou enquanto durar sua fonte de energia. São muito utilizados para exploração e reconhecimento (FAPESP, 2007).

Aéreos: Os robôs aéreos, muito conhecidos por causa dos VANT (Veículo Aéreo Não Tripulado) e *drones* são equipamentos geralmente teleoperados e são muito utilizados pelos militares. Também são utilizados por empresas para fazer entrega de encomendas.

Analisando o tipo de operação utilizado no robô temos os grupos dos teleoperados e dos autônomos. Os teleoperados são robôs que precisam de um operador humano para realizar tarefas como se locomover por um ambiente. De acordo com Siegwart e Nourbakhsh (2011) um robô autonômo deve ser capaz de se locomover e realizar suas tarefas sem nenhum tipo de interferência de um operador humano. Para isso ser capaz é preciso que haja um conjunto de equipamentos que forneçam dados ao robô de modo que ele consiga interpretar o ambiente.

2.2.2 Sensores

Para poderem executar suas tarefas os robôs móveis dispoem de uma gama muito variada de sensores, o que os permitem estarem em vários tipos de ambientes nos mais diversos tipos de configuração. Existem sensores para monitorar o robô em si, chamados de sensores internos, como sensores de temperatura e acelerômetros, que podem ser utilizados para verificar o funcionamento dos sistemas internos garantindo que todos estejam operando dentro dos padrões especificados.

Além dos sensores internos os robôs móveis contam com sensores capazes de monitorar os ambientes nos quais eles estão localizados. Com esses equipamentos os robôs tem a possibilidade de reagir aos estímulos do ambiente e tomar decisões baseadas nas interpretações dessas leituras .Desses sensores os mais comuns estão relacionados à visão e localização, sendo eles:

Odômetros: São dispositivos conectados aos motores ou Eeixos das rodas. Estimam a distância percorrida através da quantidade de voltas que a roda completou. Esse equipamento tem uma boa precisão para distâncias não muito grandes, uma vez que o acumulo de erro é inevitável (DUDEK; JENKIN, 2010).

Sonares: São utilizados em muitos campos, principalmente na área naval. Os sonares emitem um pulso de som em uma frequência es-

pecífica e calculam a distância até objetos a partir do tempo que esse pulso de áudio refletdo demora para retornar ao emissor.

Lasers: Seguindo o mesmo princípio dos sonares, o funcionamento dos sensores lasers consiste em medir o tempo de retorno de um feixe de luz emitido previamente. A precisão e distância de operação são consideravelmente maiores que as dos sonares.

Contato: é um circuito aberto simples, como um interruptor de uma lâmpada, que quando pressionados contra algum objeto fecham esse circuito e geram um sinal elétrico que indica ao robô que houve uma colisão.

Câmeras: As câmeras são utilizadas para se ter uma maior interpretação do ambiente, uma vez que é possível extrair várias informações de uma imagem. Utilizando duas câmeras é possível gerar imagens com perspectiva, do mesmo modo que a visão humana. As técnicas de tratamento de imagens são computacionalmente mais custosas que o tratamento dos outros sensores mencionados.

GPS: Do mesmo modo que em um carro a função do GPS (*Global Positioning System*) é fornecer ao robô uma posição em relação ao sistema de posição global utilizando dados enviados de satélites. A precisão desse equipamento é variável e depende da qualidade do sinal dos satélites e consomem uma quantidade considerável de energia para tratar o sinal analógico.

2.2.3 Localização

A navegação é um dos componentes mais complexos na robótica móvel. Para essa tarefa ser executada sem falhas é preciso ter sucesso em 4 subdivisões: percepção; localização; cognição e movimentação. Dessas subdivisões a localização foi o campo que teve mais atenção nas últimas décadas e consequentemente maiores avanços (SIEGWART; NOURBAKHSH, 2011). O maior desafio dessa frente é obter uma posição de alta precisão com os equipamentos presentes no robô.

Os problemas da navegação poderiam ser praticamente ignorados se a tecnologia de GPS não tivesse limitações de resolução e se a precisão fosse numa escala menor que o metro. Além disso saber a posição é necessário mas não suficiente para o robô poder se localizar no espaço. É preciso também que ele saiba a direção que ele está utilizando, por exemplo, uma bússola.

Outro problema é a resolução dos sensores, que para cenários diferentes podem produzir um mesmo resultado de leitura por falta de resolução para representar a diferença entre esses cenários, o que dificulta a interpretação dos dados. Para tentar minimizar esse problema alguns métodos são propostos:

2.2.3.1 Localização baseada em pontos de referência (*landmarks*)

Quando um robô se locomove por um cenário é possível que ele consiga identificar marcações nesse ambiente, considerando que ele tenha um equipamento que o possibilite fazer isso como por exemplo uma câmera. A relação entre a imagem e a marcação indica ao robô a posição e a orientação dele no mapa, que é conhecido previamente. Esses marcos podem ser algum elemento de uma paisagem como uma montanha, uma árvore, uma pedra ou elementos introduzidos artificialmente como algum símbolo pintado em uma superfície.

Para se obter uma boa precisão com esse método, de acordo com Betke e Gurvits (1997), é preciso que pelo menos 3 marcos esteja visíveis para o robô para ser possível uma triangulação e assim relacionar corretamente a posição entre esses marcos. Esse método, pela sua construção, necessita de $O(m^3)$ inversões de matrizes para ser computado, onde m é a quantidade de marcos analisados, o que inviabiliza o uso em tempo real para navegação.

O autor propõe então um método que não precisa da triangulação, mas sim o uso de uma representação numérica dos marcos que torna possível o mapeamento destes para um conjunto de equações lineares cujo os resultado é um conjunto de estimativas de posições. Para resolver essas equações a complexidade é $O(m)$, o que significa que o algoritmo executa em tempo linear em relação à quantidade de marcos analisados.

2.2.3.2 Localização e mapeamento simultâneos

Uma outra técnica aplicada para facilitar a localização de um robô no espaço é a Localização e Mapeamento Simultâneo, SLAM, do inglês *Simultanious Localization and Mapping* onde o robô, a partir de uma posição desconhecida em um ambiente desconhecido, consegue simultaneamente se localizar e construir um mapa do ambiente que pode ser utilizado para navegação. Em Dissanayake et al. (2001) os autores descrevem esse problema e apresentam uma solução

Sensores dão ao robô a percepção do ambiente e suas rotinas dão a capacidade de tomada de decisão, baseadas nos dados coletados pelos sen-

sores. Com esse conjunto o robô pode navegar seguramente pelo ambiente desviando de obstáculos planejando rotas alternativas quando necessário.

2.2.4 Atuadores

Os atuadores, como mencionado anteriormente são equipamentos utilizados para o robô se locomover no ambiente no qual ele está localizado. Esses atuadores são controlados pelo sistema de controle do robô, que pode ou não ser autônomo. Os atuadores mais comuns encontrados na robótica móvel são:

Rodas: Meio de locomoção muito frequente em robôs móveis mas não é indicado para terrenos acidentados, salvo casos onde a roda não seja fixa em relação ao corpo do robô.

Lagartas: São compostas por uma esteira ligada às rodas do robô para aumentar a superfície de contato e consequentemente a aderência no solo. Altamente indicada para terrenos acidentados e/ou instáveis, como areia ou terrenos rochosos.

Pernas: São interessantes para ambientes que requerem uma interação específica como subir degraus ou se locomover por terrenos onde se necessita de vários pontos de apoio. O sistema de controle desse tipo é mais complexo que os outros mencionados.

Existem atuadores para interagir com o ambiente de uma forma mais específica e podem ser utilizados para realizar as mais diversas tarefas, como pegar amostras de pedras do chão, cortar uma árvore, perfurar o solo. A pinça é um exemplo e possui uma grande versatilidade de operação de objetos. A figura 2.2 mostra um robô(MOBILEROBOTICS, 2014) com uma pinça como manipulador e rodas como atuadores.

2.3 NAVEGAÇÃO

Neste capítulo estão descritos alguns mecanismo e conceitos de navegação para robótica móvel levando em consideração o tipo de ambiente no qual o mesmo está inserido..

Figura 2.2: Robô Seekur Jr

2.3.1 Ambientes não conhecidos

Para ambientes não conhecidos a localização não tem muita utilidade a não ser se aplicadas técnicas de SLAM e por isso é necessário apenas meios de se locomover pelo ambiente e algum modo de desviar de obstáculos. Com essa finalidade foram desenvolvidos vários algoritmos, principalmente os da família *Bug* e por serem de fácil implementação eles são muito utilizados Siegwart e Nourbakhsh (2011). Essa família composta por mais de 10 algoritmos, com alguns muito similares entre si (NG; BRäUNL, 2007).

Começando pelo *Bug1* temos que o robô se locomove em direção ao objetivo e ao encontrar um obstáculo ele o circula completamente antes de tomar a decisão de que direção seguir para alcançar o objetivo. O *Bug2* é uma evolução natural do *Bug1* onde ao invés de circular por completo o obstáculo o robô o circula até reencontrar uma linha imaginária traçada do ponto inicial até o objetivo. Uma grande vantagem desses dois algoritmos é que eles não precisam de sensores complexos para funcionar, sendo necessário apenas sonares como sensores de distância.

O *TangentBug*, um dos algoritmos mais complexos da família, tem como princípio utilizar os dados dos sensores de distância para calcular a menor alteração no percurso planejado para desviar de um obstáculo tangenciando o mesmo (FERREIRA et al., 2008). Sensores laser são necessários

para esse algoritmo uma vez que eles possuem uma boa resolução e grande alcance, fazendo com que em ambientes simples o algoritmo consiga chegar próximo ao caminho ótimo global (KAMON; RIVLIN; RIMON, 1996) fazendo dele o algoritmo mais eficiente da família *Bug* (NG; BRäUNL, 2007). A Figura 2.3 mosta um comparativo entre alguns algorítmos da família *bug*

2.3.2 Ambientes conhecidos

Sendo um pouco diferente do SLAM é possível mapear ambientes previamente para depois ser possível o robô planejar sua rota até os objetivos especificados para ele. Para isso são utilizados as leituras dos sensores de distância, como sonares e laser (esses mais utilizados) e odometria. Com esses dados o robô consegue identificar obstáculos e colocar em um arquivo a leitura de distância dos sensores a cada instante juntamente com a distância percorrida e angulo em relação a posição inicial. Com esses dados em mão é possível que softwares, como o descrito na Subseção 4.2.3, construam um mapa de alta fidelidade.

Com um mapa a sua disposição o robô então pode computar uma rota ótima até o seu objetivo utilizando as técniacas de planejamento de rota. O tipo de abordagem varia muito mas podem ser divididos em dois grupos genéricos: busca em grafos e campos potenciais (SIEGWART; NOURBAKHSH, 2011).

O primeiro constrói um grafo, normalmente em modo *offline*, a partir das leituras do mapa e faz uma busca pelo caminho ótimo até o objetivo. O método de construção do grafo varia muito como o Grafo de Visibilidade, Diagrama de Varonoi, Decomposição Exata de Células, Decomposição Aproximada de Células, Grafo de Látices, dentre outros. Para achar o melhor caminho também existem vários algoritmos como a busca em largura, busca em profundidade e algoritmo de Dijkstra por exemplo.

A abordagem dos campos potenciais é completamente diferente e não utiliza grafos para calcular o melhor caminho até seu objetivo. Ao invés disso são utilizadas equações sobre o mapa para montar vetores gradientes do ponto de origem do robô até o seu destino. Os vetores são montados levando em consideração equações que descrevem “campos” no mapa que atraem e repelem o robô onde a atração é gerada pelo objetivo e a repulsão gerada pelos obstáculos. Uma abstração seria uma partícula eletricamente positiva sendo atraída por uma partícula eletricamente negativa enquanto atravessa um campo com outras partículas positivas fixas no mapa.

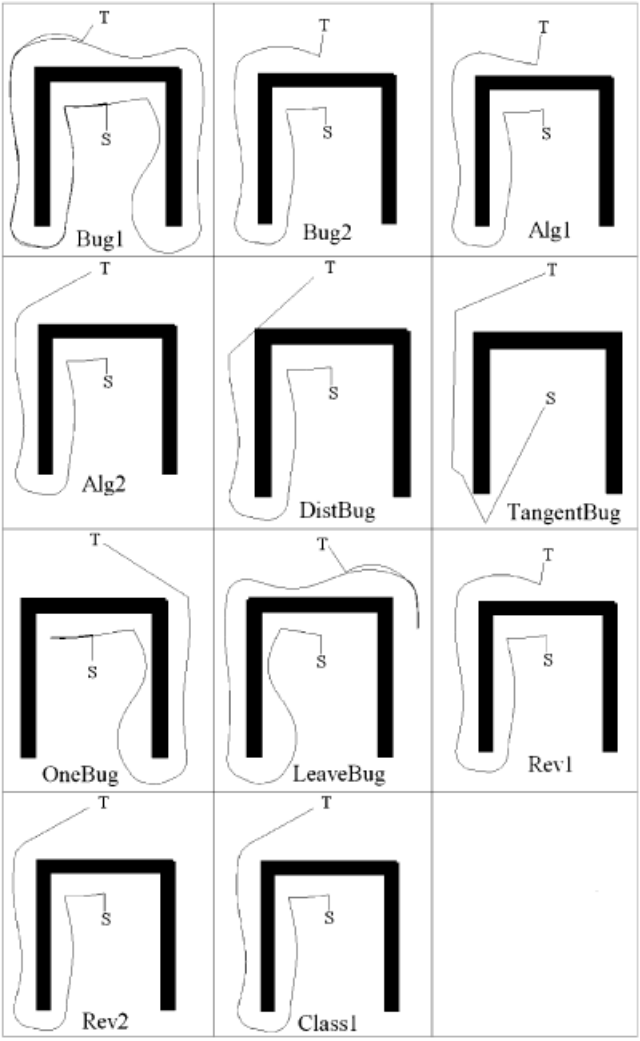


Figura 2.3: Comparativo da execução dos algoritmos da família bug (NG; BRäUNL, 2007)

2.4 FONTE DE ENERGIA

Sempre que se pensa em robôs autônomos logo vem a questão da autonomia que esses equipamentos possuem. É possível que haja uma fonte de energia externa alimentando esses robôs como placas de energia solar ou mesmo cabos, esses muito utilizados em robôs submarinos mas também há uma boa parcela dos robôs que utilizam baterias como fonte de energia primária.

Um estudo realizado por Mei et al. (2005) sobre o consumo de energia dos componentes de um robô muito similar ao utilizado neste trabalho mostra que grande parte do consumo de energia é gerado pelo computador embutido, ficando pouco acima da locomoção. O sensoriamento e os microcontroladores quase não consomem energia quando comparados com os outros dois componentes.

O autor, depois de apresentar esses dados fornece um modelo de um escalonador que utiliza métodos de gerência dinâmica de energia para controlar os gastos de componentes, como diminuir a tensão em componentes ociosos. Também existem técnicas para economizar energia em um nível mais abaixo, mesmo antes das execuções das tarefas. Um robô móvel não tem um gasto constante de energia e por causa disso a vida útil da bateria pode sofrer uma perda grande já que a relação entre consumo e carga não é linear (PEDRAM; WU, 1999).

Existem algoritmos de escalonamento que levam em conta o gasto de energia e tentam minimizar esse problema alterando a fila de prioridade de escalonamento ponderando entre gasto e a escalonabilidade das tarefas. Um controlador PID, do inglês *Proportional-Integral-Derivative* , é utilizado para fornecer um *feedback* para o escalonador alterar a fila de escalonamento a fim de diminuir a flutuação de energia, aumentando assim a vida útil da mesma (WANG et al., 2008).

2.5 PREVISÕES

A área de previsão de perda de *deadlines* possui muitos estudos, desde tarefas de baixo nível até as de mais alto nível. Em (PLENTZ; MONTEZ; OLIVEIRA, 2008) é proposto um mecanismo de previsão para tarefas com *deadlines soft* e *firm* em uma aplicação com um sistema de *threads* distribuídas, ou seja, onde a aplicação é executada em nós remotos e/ou locais. Com esse mecanismo é possível prever se um *deadline* será perdido baseado no itinerário que as chamadas podem assumir.

Em um nível mais alto, já no planejamento de rotas, existem técnicas

para melhorar a rota planejada durante sua execução quando o ambiente é dinâmico ((TSUBOUCHI; ARIMOTO, 1994); (MIURA; UOZUMI; SHIRAI, 1999)). Nesses trabalhos são modeladas equações matemáticas juntamente com um histórico e análises estatísticas sobre esses dados. Com esse tipo de abordagem é possível tomar melhores decisões de qual caminho escolher dentre os vários possíveis para se desviar de um obstáculo. O modelo apresentado por Miura, Uozumi e Shirai (1999) utiliza um método de busca em uma árvore de decisões com pesos nas arestas para calcular a rota até o destino final com o provável menor tempo de execução.

Com técnicas um pouco mais avançadas é possível que o robô aprenda a escolher melhores caminhos do que o atual baseado na relação entre tarefas executadas e ações corretivas tomadas e o tempo gasto em execuções anteriores para avaliar a performance da execução atual(MICHAUD; MATARIć, 1998). Mantendo um histórico de ações executadas e utilizando uma métrica de comparação é possível que o robô decida quando ele deve tomar um caminho diferente de ações para completar suas tarefas de um modo mais eficiente que o atual.

3 PROPOSTA

Este capítulo apresenta a proposta deste trabalho, descrevendo detalhadamente o cenário onde o mecanismo será aplicado, a descrição conceitual e matemática deste mecanismo e a descrição de um outro mecanismo mais simples com qual o mecanismo proposto será comparado.

3.1 DEFINIÇÃO DO CENÁRIO

Antes de definir o previsor proposto primeiro será detalhado o ambiente sobre o qual o mecanismo será aplicado. Mas pelo escopo de tarefas de alto nível ser muito grande ele precisa ser reduzido para ser possível definir tarefas específicas para, se possível, expandir o escopo de atuação. Conforme foram discutidas as tarefas percebeu-se que a grande parte delas podia ser resumida a uma grande tarefa (claro que essa *macro tarefa* pode ser quebrada em tarefas menores) que é se deslocar de um ponto A ao ponto B.

Peguemos a tarefa de verificar se um cano possui falhas. A princípio o robô irá sair de um ponto de início, A, e irá verificar seções desse cano até chegar ao final do segmento ou uma posição especificada, B. Transportar objetos também se enquadram nessa categoria, uma vez que é preciso sair de um ponto inicial e levar o objeto desejado a um ponto final.

Introduzindo o *deadline* no cenário nós temos então um deslocamento com um tempo limite de execução, lembrando que o tempo das ações executadas dentro desse deslocamento já faz parte desse *deadline* e se o programador quiser ele pode codificar o previsor para essas ações, mas nesse trabalho será utilizado apenas um *deadline* para a tarefa.

Decidido a tarefa agora temos que definir onde ela será executada, ou seja, o ambiente que o robô irá se encontrar. Ffoi utilizado um robô real para mapear o terceiro andar do Departamento de Informática e Estatística (INE) e ganhar experiência na API (*Application Programming Interface*) que será utilizada para programar o previsor. Esse local foi escolhido pela fato do laboratório onde esse trabalho foi desenvolvido, LAPESD, se encontrar no mesmo andar.

O mapeamento foi executado do seguinte modo: Foi executado um programa específico para gerar um arquivo de log da leitura do laser SICK que veio junto com a biblioteca ARIA, onde o robô deve ser teleguiado pelo ambiente em questão. Uma vez gerado, esse log é então carregado no programa Mapper3 que se encarrega de transformar as leituras em um mapa e nesse mesmo programa foram adicionados os objetivos “LAPESD” e “Ele-

vador", como mostra a Figura 3.1, cuja as utilizações serão explicadas no capítulo 5.

Tendo já a tarefa e o ambiente definidos falta apenas decidir a dinâmica desse ambiente para completar o cenário. Como já descrito a ideia de aplicação do mecanismo proposto seria em um ambiente autônomo e controlado, ou seja, com robôs completamente autônomos com tarefas predefinidas em um ambiente sem incidentes. Porém, se ocorre algo inesperado no trajeto o previsor tem que ser capaz de prever corretamente se a tarefa será cumprida.

Tomemos a seguinte situação: um robô, por um motivo qualquer, desvia da sua rota e começa a atrapalhar outros robôs no ambiente sem bloquear totalmente o seu caminho, que por sua vez têm que tomar ações para se manterem nas suas respectivas tarefas. Nessa situação o previsor deve retornar uma resposta em tempo hábil para serem tomadas as devidas ações corretivas.

Figura 3.1: Mapa do terceiro andar do INE

3.2 ALGORITMO DE BASE

Partindo então deste cenário foram pensadas algumas abordagens para o problema da previsão. Dessas abordagens duas se destacaram, sendo elas um cálculo a partir da quantidade de obstáculos encontrados durante a exe-

cução da tarefa e outra abordagem levando em consideração o tempo e a distância total percorrida.

A abordagem com obstáculos foi logo descartada pelo fato da complexidade de interpretação desses obstáculos, principalmente decidir quando um obstáculo termina e outro começa e o tempo de contorno dos mesmos. Com essas complicações foi necessário então uma alternativa, que foi justamente a ideia da relação entre a distância percorrida em um dado intervalo de tempo.

A abordagem temporal consiste em um certo marco no *deadline* para executar o cálculo da previsão. Para manter essa abordagem simples foi definido que esse marco seria sempre na metade do *deadline* e quando esse tempo fosse alcançado seria então executado o cálculo da previsão. Esse marco foi escolhido para que quando uma resposta for dada, o robô ou o operador humano tenham tempo de tomar as ações necessárias. Decompondo a linha de tempo de uma tarefa podemos então definir três marcos, sendo eles o início, 50% do *deadline* e o fim da tarefa. Para futuras referências chamemos esses marcos de q_0, q_1 *e* q_2, respectivamente.

Em q_0 os dados do robô que serão utilizados, sendo eles a distância estimada d_e até o objetivo, a distância percorrida d na tarefa e o tempo decorrido t desde o inicio da tarefa, são reinicializados para começar a nova execução. Nesse ponto então temos que o robô percorreu uma distância estimada d_e qualquer, uma distância percorrida d de 0 mm e um tempo decorrido t de 0 segundos e então o previsor "dorme" até q_1 ser alcançado.

Quando esse marco for atingindo é executada a previsão da seguinte maneira: Se em q_1 a distância percorrida d for maior ou igual à distância estimada d_e assume-se que o robô irá chegar ao seu destino dentro do *deadline*, caso contrário a previsão é de não concluir a tempo a tarefa. Definindo esses termos matematicamente chegamos na função 3.1.

$$\text{Vai concluir} = \begin{cases} \text{Sim} & d \geq \frac{de}{2} \\ \text{Não} & d < \frac{de}{2} \end{cases} \tag{3.1}$$

Diferente dos marcos anteriores q_2 não tem relação direta com o *deadline* uma vez que ele é um marco de evento de término da tarefa, ou seja, para cada execução da tarefa q_0 será sempre em 0% do *deadline*, q_1 será sempre em 50% do *deadline* mas q_2 pode assumir qualquer valor de tempo pois ele indica quando a tarefa terminou. E é nesse marco que é verificado se a resposta do previsor foi condizente com a realidade.

Executar uma simulação manualmente deve deixar mais claro o funcionamento do mecanismo. Começando uma tarefa nova, digamos partindo do LAPESD em direção ao elevador, nos temos as seguintes informações fictícias:

- Início = 2014-07-10 05:19:55
- *deadline* = 2014-07-10 05:21:25 (90 segundos)
- d_e = 30000 milímetros
- d = 0 milímetros
- t = 0 segundos

O mecanismo então espera até 50% do *deadline* (q_1) para executar a verificação com as seguintes novas informações:

- d = 19000 milímetros
- t = 45 segundos

Desse modo, de acordo com a função 3.1, temos:

$$\begin{aligned} \text{Vai concluir} &= 19000 \geq \frac{30000}{2} \\ &= 19000 \geq 15000 \\ &= \textit{Sim} \end{aligned}$$

Digamos então que após 70 segundos do início da tarefa o robô chegue ao seu objetivo com os seguintes dados:

- d = 31000 milímetros
- t = 70 segundos

Sendo assim o previsor acertou na resposta pois a previsão foi de conclusão dentro do *deadline* e o término da tarefa foi 70 segundos após o início da tarefa, sendo esse tempo abaixo do *deadline*.

3.3 DEFINIÇÃO DO ALGORITMO PROPOSTO

Nesta seção é apresentado em detalhes o mecanismo proposto. A concepção deste previsor foi baseada no algoritmo de base, como uma evolução natural do mesmo. As duas subseções seguintes descrevem o conceito e a definição matemática do mecanismo, nessa mesma ordem.

3.3.1 Descrição Conceitual

O algoritmo de base é bem simples e justamente por isso surgiram algumas questões sobre ele. A primeira questão foi a existência de um único ponto de execução do cálculo do previsor. Por que não executar esse cálculo em mais pontos para poder detectar mais cedo uma possível perda do *deadline* e consequentemente tomar as ações necessárias para diminuir o impacto dessa perda?

Essa é a primeira alteração incorporada no mecanismo proposto. Ao invés de executar a previsão aos 50% do *deadline* essa verificação é executada 3 vezes ao longo do percurso e do mesmo modo que no algoritmo de base vamos dar um nome para cada marco temporal, de q_0 a q_4, onde q_0 é o início da tarefa, q_1, q_2, e q_3 são os marcos onde serão executados o cálculo de previsão, sendo eles respectivamente em $\frac{1}{5}$, $\frac{2}{5}$ e $\frac{3}{5}$ do *deadline*, e q_4 o fim da tarefa. Chamemos essa fração de $\frac{1}{5}$ do *deadline* de f_d para futuras referências

Desse modo o previsor pode avisar o robô que a tarefa não será cumprida dentro do prazo e pode iniciar suas ações corretivas e evitar a cada execução do cálculo que perdas maiores ocorram. Mas o que verificar nesses pontos é uma questão que não pode ser negligenciada e muito menos ignorada. Seria realmente a simples verificação de quanto o robô andou naquela tarefa suficiente para dar uma boa previsão? Pode ser que até o primeiro marco a velocidade tenha sido mais baixa, porém ela pode ser compensada nos próximos trechos.

Pensando dessa forma foi adotado um sistema de histórico de tarefas para armazenar informações relevantes ao trajeto. Essas informações então serão consultadas e compiladas para fornecer as informações necessárias para o previsor dar uma resposta condizente com aquela tarefa no estado em que o ambiente se encontra. Para esse trabalho o histórico vai possuir informações essenciais para o funcionamento do previsor e algumas informações que serão utilizadas apenas para métricas de comparação entre o mecanismo de base e o proposto. As informações essenciais são as seguintes:

- Um identificador da tarefa;
- A data de início e fim da tarefa;
- A distância percorrida acumulada em cada marco temporal (q_1, q_2, q_3 e q_4);

A Tabela 3.1 é um exemplo de um pequeno histórico contendo valores para as informações acima descritas. Com essas informações pode-se derivar algumas outras que são utilizadas nos cálculos, como a velocidade média

em cada marco temporal, o tempo médio de execução da tarefa e a distância média por marco.

Nota-se que na linha 6 a coluna "Distância em q_3" não possui um valor, o que indica que no marco q_2 o previsor decidiu que a tarefa não seria concluída e não executou a próxima verificação. Com um histórico formado podemos então inferir as condições de cada trecho do trajeto percorrido, como sua taxa de acerto de previsão, o quanto o ambiente muda naquele pedaço do trajeto dentre outros.

Na seção 6 serão apresentados e explicados outros dados guardados no histórico e suas utilizações.

3.3.2 Descrição Matemática

Agora que foram identificados os pontos chaves do mecanismo podemos definir formalmente o seu funcionamento, ou seja, como ele irá se comportar matematicamente, definindo as fórmulas e equações utilizadas nos cálculos que serão executados. Primeiramente vamos começar definindo alguns símbolos.

Comecemos definindo d_{q_i} como a distância percorrida até o marco q_i, com i variando de 1 a 3. Assim podemos também definir $\overline{d_{q_i}}$ como a média dos d_{q_i}'s. Similarmente ao mecanismo de base então podemos verificar se no marco em questão a distância percorrida pela tarefa é maior ou igual à média daquele marco, ou seja, a diferença Δd_{q_i} é maior ou igual a 0 de modo que:

$$\text{Previsão em } q_i = \begin{cases} \text{Vai Concluir} & \text{se } \Delta d_{q_i} \geq 0 \\ \text{Não vai concluir} & \text{se } \Delta d_{q_i} < 0 \end{cases} \tag{3.2}$$

Mas somente essa verificação precisa ser aprimorada para ficar mais robusta e por isso iremos introduzir uma segunda parte na equação 3.2. Vamos pegar como exemplo a linha 6 da tabela 3.1. Ela é uma tarefa do tipo 0, logo devemos pegar apenas o histórico de tarefas desse tipo, no caso as linhas 2 e 4. Utilizando essas linhas nós temos um $\overline{d_{q_1}}$ de 7162,5915 que é maior que o d_{q_1} que tem um valor de 4918.8774.

Pela equação atual nó teríamos que o previsor iria dar uma resposta de perda de *deadline* e terminaria, mas o que de fato aconteceu é que ele previu que iria concluir e posteriormente executou uma segunda verificação. Isso se deve ao fato de ele ter utilizado essa segunda parte da equação.

O que aconteceu é que após verificar que d_{q_i} estava abaixo de $\overline{d_{q_i}}$ ele executou um cálculo que leva em consideração a média das velocidades $\overline{v_{q_j}}$ em cada marco subsequente ao seu. Podemos escrever essa média das velo-

ID da tarefa	Inicio	Fim	*deadline*	Distância em q_1	Distância em q_2	Distância em q_3	Distância em q_4
1	2014-07-10 05:19:55	2014-07-10 05:21:23	2014-07-10 05:21:25	2838.2553	11302.8259	18260.6848	33409.9898
0	2014-07-10 05:21:23	2014-07-10 05:22:59	2014-07-10 05:22:53	5640.2559	12067.7841	22677.8197	33369.1388
1	2014-07-10 05:23:00	2014-07-10 05:24:29	2014-07-10 05:24:30	3556.0058	11001.5208	17253.6237	33897.2974
0	2014-07-10 05:24:29	2014-07-10 05:25:54	2014-07-10 05:25:59	8684.9271	18662.1855	23067.9796	33188.8590
1	2014-07-10 05:25:54	2014-07-10 05:27:19	2014-07-10 05:27:24	3687.1631	11094.9886	18274.8158	33365.9451
0	2014-07-10 05:27:20	2014-07-10 05:28:51	2014-07-10 05:28:50	4918.8774	9975.0822		33504.2667
1	2014-07-10 05:28:51	2014-07-10 05:30:23	2014-07-10 05:30:21	3541.9212	10148.6125	15984.4803	33243.3832
0	2014-07-10 05:30:23	2014-07-10 05:31:46	2014-07-10 05:31:53	8706.2217	18124.9877	24386.2648	33141.6421
1	2014-07-10 05:31:47	2014-07-10 05:33:10	2014-07-10 05:33:17	3743.0888	11811.4899	20208.2445	33356.3476

Tabela 3.1: Exemplo de histórico

cidades médias v_{m_i}:

$$v_{m_i} = \left(\sum_{j=i}^{4} \frac{\overline{d}_{q_{j+1}} - \overline{d}_{q_j}}{\Delta t} \right) / (4-i) \tag{3.3}$$

onde Δt representa o tempo gasto desde a última verificação e tem como valor a seguinte função:

$$\Delta t = \begin{cases} f_d & \text{se } i \in [1,2] \\ (\overline{fim} - \overline{inicio}) - 3 * f_d & \text{se } i = 3 \end{cases} \tag{3.4}$$

A função 3.4 se dá desse modo pelo fato de que até q_3 os intervalos de tempo são bem definidos mas o tempo gasto entre q_3 e q_4 é variável, logo temos que fazer o cálculo da média do tempo total gasto na tarefa menos 3 frações do *deadline*, que representam o tempo gasto de q_0 a q_3.

Tendo em mãos $\overline{v_{m_i}}$ podemos calcular o tempo estimado t_e necessário para o robô completar a tarefa utilizando a distância estimada até o objetivo d_e resolvendo a seguinte equação:

$$t_e = \left\lceil \frac{-\Delta d_{q_i}}{\left(\frac{\overline{d}_{q_i}}{f_d} \right)} + \frac{d_e + \Delta d_{q_i}}{v_{m_i}} \right\rceil \tag{3.5}$$

Agora podemos incorporar esse cálculo na função 3.2 chegando a um resultado final de:

$$\text{Previsão em } q_i = \begin{cases} \text{Sim} & \text{se } \Delta d_{q_i} \geq 0 \wedge t_e \leq \textit{deadline} \\ \text{Não} & \text{se } \Delta d_{q_i} < 0 \vee t_e > \textit{deadline} \end{cases} \tag{3.6}$$

Com a função 3.6 podemos terminar o exemplo da execução da linha 6 da tabela. Sabendo então que a distância percorrida está abaixo da média a segunda parte da verificação será executada. Compilando os dados das linhas 2 e 4 temos então em q_1:

- inicio = 2014-07-10 05:27:20
- *deadline* = 2014-07-10 05:28:50
- $d_{q_1} = 4918,8774$
- $\overline{d}_{q_1} = 7162,5915$

- $\Delta d_{q_1} = -2243,7141$
- $d_{q_2} = 9975,0822$
- $\overline{d}_{q_2} = 15364,9848$
- $\Delta d_{q_2} = -5389,9026$
- $\overline{d}_{q_2} - d_{q_1} = 8202,3933$
- $\overline{d}_{q_3} - d_{q_2} = 7507,9149$
- $\overline{d}_{q_4} - d_{q_3} = 10406,0993$

Expandindo e substituindo os valores na função 3.3 obtemos:

$$\begin{aligned} v_{m_1} &= \left(\frac{8202.3933 + 7507.9149 + 10406.0993}{18} \right) /3 \\ &= 483.6372 \text{ mm}/s \end{aligned}$$

e supondo uma distância até o objetivo d_e é 23000 temos de 3.5:

$$\begin{aligned} t_e &= \left\lceil \frac{2243.7141}{\left(\frac{7162.5915}{18}\right)} + \frac{23000 + (-2243.7141)}{483.6372} \right\rceil \\ &= 49s \end{aligned}$$

Podemos então verificar que adicionando t_e ao início da tarefa obtemos uma data esperada de término de 2014-07-10 05:28:09, que é menor que o *deadline* e prosseguimos com a tarefa. Ao chegar em q_2 verificamos que novamente temos um Δq negativo, de -5389,9026, e verificamos novamente se ainda há tempo para completar a tarefa.

Assim vamos obter:

$$\begin{aligned} v_m(2) &= \left(\frac{7507.9149 + 10406.0993}{18} \right) /2 \\ &= 497.6115 \text{ mm}/s \end{aligned}$$

e com um d_e de 18500 temos: 3.5:

$$t_e = \left\lceil \frac{5389.9026}{\left(\frac{8202.3933}{18}\right)} + \frac{18500 + (-5389.9026)}{497.6115} \right\rceil$$
$$= 39s$$

A verificação em q_2 ocorreu em 2014-07-10 05:27:45 e o previsor, que estimou que demoraria ainda 39 segundos para terminar de percorrer a distância restante, estimou que a chegada ao objetivo se daria em 2014-07-10 05:28:24, o que não condiz com a tabela. Com essa diferença de resultados podemos agora introduzir mais uma questão do previsor: o tamanho do histórico.

Como o mecanismo se baseia muito em médias a quantidade de referências faz diferença no seu funcionamento. Se tivermos poucas amostras uma pequena variação nos dados gera uma diferença muito grande nos valores de médias e se tivermos uma quantidade muito grande de amostras uma variação nos dados terá pouco impacto. Como decidir então qual o tamanho ideal do tamanho desse histórico? Para responder essa pergunta foi pensando em comparar a execução do mecanismo com diferentes tamanhos de histórico.

Por hora vamos assumir que existirão 4 execuções com diferentes tamanhos de histórico, sendo eles 25, 50, 100 e 200 entradas, mas para fins de estatística o banco de dados irá conter mais entradas do que essas quantidade. Na seção 4.4 será descrito como serão manipuladas essas entradas e o tamanho do histórico e no capítulo 5 será descrito como o banco foi populado e preparado para o previsor.

Sendo assim, um novo registro sempre irá sobrescrever o registro mais antigo fazendo com que os dados representem sempre o estado mais recente do ambiente. Isso faz com que o mecanismo tenha um certo nível de adaptabilidade ao ambiente.

Por fim ficou estabelecido que as características principais do mecanismo são:

- A utilização de vários pontos de verificação
- A utilização de um histórico para utilização nos cálculos
- Um tamanho fixo de entradas no histórico

4 IMPLEMENTAÇÃO

Nesse capítulo será descrita a implementação dos mecanismos acima descritos juntamente com os *softwares* e *hardwares* utilizados no desenvolvimento. Vale ressaltar que todo o código fonte e documentação do mesmo se encontram em inglês para divulgação no meio acadêmico.

4.1 HARDWARE UTILIZADO

Nesta sessão serão apresentados os *hardwares* utilizados no desenvolvimento deste trabalho. O modelo de testes do mecanismo é o Pioneer 3-DX com o laser Sick LMS-500 acoplado.

4.1.1 Robô Pioneer 3-DX

Para a execução e testes do previsor foi escolhido o robô Pioneer 3-DX, ou simplesmente P3-DX(Figura 4.1). Este robô contém uma roda livre e duas tracionadas, estas contendo um encoder de 500 ticks de precisão e podendo alcançar uma velocidade de 1,6 metros por segundo. O corpo do robô também possui 16 sonares, 8 na parte da frente e 8 na parte de trás, sendo capazes de detectar objetos no intervalo de 15 cm a 7 metros de distância. Um computador contendo um sistema operacional Linux também está presente nas configurações além dos microcontroladores de subsistemas. Para alimentar o sistema o corpo do robô suporta até 3 baterias, dando uma autonomia de 8 a 10 horas.

A especificação completa deste modelo pode ser consultada no site da fabricante em MobileRobotics (2014)

Figura 4.1: Pioneer 3-DX

4.1.2 Laser Sick LMS-200

Sensores lasers são utilizados para detecção em alta resolução de superfícies. Eles utilizam feixes de laser para calcular a distância a partir do tempo de reflexão destes feixes. Este modelo em particular (Figura 4.2), fabricado pela empresa *Sick Sensor Inteligence*, é para uso interno apenas e possui 180^o de visão sendo 1^o por feixe, possuindo uma precisão de 10 milímetros. Seu alcance é, de no máximo, 80 metros com 600% de refletividade do objeto. Ele pode escanear o ambiente com uma frequência máxima de 75 Hz

Na robótica ele é utilizado para geração de mapas, detecção de obstáculos e navegação em geral. Ele possui um suporte para ser acoplado no P3-DX, como visto na Figura 4.3. A especificação completa deste sensor pode ser consultada em (SICK, 2014)

Figura 4.2: Sensor laser Sick LMS-200

4.2 SOFTWARE UTILIZADO

Nesta seção serão descritas as ferramentas de softwares e bibliotecas utilizadas neste trabalho. Para controle e acesso aos dados dos sensores foram utilizadas as bibliotecas *ARIA* e *ARNL* juntamente com um manipulador de mapas, *Mapper3*, um simulador, *MobileSim*, e um software de controle remoto, MobileEyes. Para manter o histórico de tarefas do previsor foi utilizado o Software de Gerência de Banco de Dados, ou SGBD , SQlite3.

Figura 4.3: Sensor laser Sick LMS-200 instalado no robô

4.2.1 ARIA

ARIA, do inglês *Advanced Robotics Interface for Applications*, é uma biblioteca de código aberto e gratuita desenvolvida pela fabricante do robô *P3-DX* para acesso de alto nível do mesmo. Desenvolvida em C++, e com interfaces para Java e Python, ela contém muitas funções para acesso aos componentes e sensores do robô e suas configurações. Sua API é vastamente documentada com descrições de suas classes, funções e variáveis e exemplos de uso das mesmas, sendo uma fonte indispensável de consulta.

A biblioteca também fornece funções para enviar e receber informações pela rede. Aplicações clientes, como por exemplo o *MobileEyes*, podem enviar comandos para o robô e receber várias informações. Esta biblioteca é multi plataforma, podendo ser utilizada em sistemas Linux e Windows podendo o mesmo programa ser compilado para ambos sem alteração no código.

4.2.2 ARNL

ARNL, do inglês *Advanced Robotics Interface for Navigation and Localization* é a biblioteca em C++ também desenvolvida pela fabricante do robô para fornecer funções avançadas de localização e navegação. Com ela é possível executar várias tarefas relacionadas a esse assunto, principalmente a utilização de mapas e planejamento de rotas. Juntamente com um sensor laser é possível com esta biblioteca gerar leituras de ambientes para posteriormente poderem ser transformadas em um mapa.

Sobre estes mapas é possível então planejar rotas ente pontos, teleguiar seguramente o robô, ou simplesmente deixá-lo 'passeando' pelo mapa. Informações sobre o planejamento de rota podem ser passadas para aplicações clientes pela rede do mesmo modo que pelo *ARIA*. Esta biblioteca não é de código aberto e só pode ser adquirida juntamente com a aquisição de equipamentos que contenham em sua licença de uso a permissão de sua aquisição.

4.2.3 Mapper3

Este software é de propriedade da fabricante do robô e é utilizado para criação e edição de mapas, como visto na Figura 4.4, estes podendo serem criados pelo próprio usuário ou gerados de leituras de sensores lasers para criação mapas de alta fidelidade. No último caso é preciso navegar pelo robô de modo que a trajetória e os dados dos sensores laser sejam salvos. Com estas informações o software então consegue gerar automaticamente o mapa. Com estes mapas em mãos é possível adicionar várias informações como áreas proibidas, pontos de interesses (objetivos) e obstáculos. Os mapas então podem ser carregados no robô e enviados para aplicações clientes.

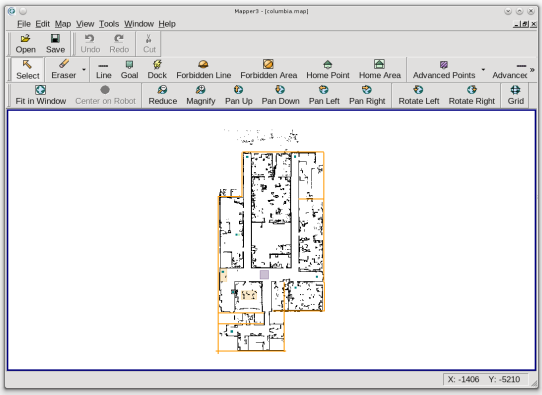


Figura 4.4: Interface do aplicativo Mapper3

4.2.4 MobileSim

O simulador MobileSim é uma plataforma para simulação de robôs, voltados para os modelos da mesma fabricante do P3-DX. É possível simular robôs em seus ambientes utilizando mapas criados pelo Mapper3, como visto na Figura 4.5. Foi desenvolvido baseado no simulador Stage, do projeto Player.

O simulador não pode ser executado em conjunto com um robô real, uma vez que a biblioteca ARIA primeiramente tenta se conectar com o robô real e, se não encontrado, ao simulador. Isso é possível pois ambos utilizam uma porta TCP para comunicação. O simulador já vem com alguns robôs modelados, incluindo o P3-DX, juntamente com seus sensores, o que viabiliza uma simulação muito fiel ao mundo real.

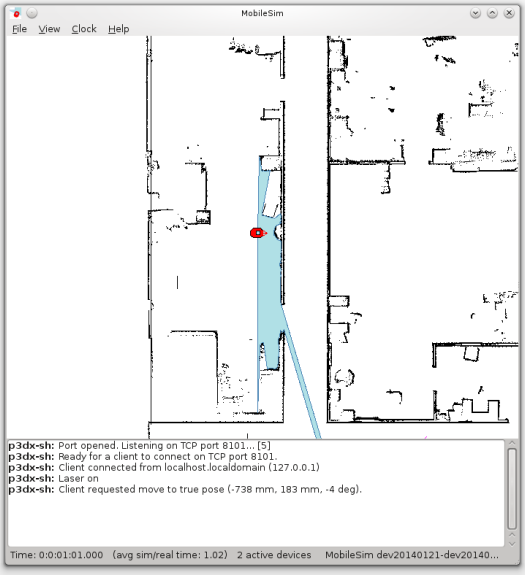


Figura 4.5: Interface do simulador MobileSim

4.2.5 MobileEyes

MobileEyes é um aplicativo cliente para se conectar aos robôs que disponibilizarem um servidor de informações pela rede utilizando uma conexão TCP/IP. Com este aplicativo é possível controlar o robô remotamente e visualizar as informações que o robô fornece, incluindo o mapa sendo utilizado. Através deste aplicativo é possível, como visto na Figura 4.6, enviar o robô à algum objetivo, visualizar o planejamento de rotas, informações de posições, dentre outros.

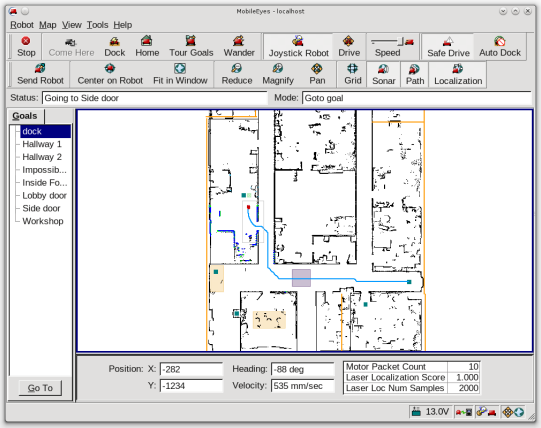


Figura 4.6: Aplicativo MobileEyes exibindo o planejamento de rota

4.2.6 SQlite3

O SGBD SQlite3 foi escolhido pela sua simplicidade e principalmente pela sua portabilidade. Os banco de dados (BD) do SQlite3 são arquivos de texto, criptografados ou não, que podem ser movimentados de um computador para outro sem nenhum problema. Esse SGBD, apesar de não possuir muitas funcionalidades quando comparado com outros SGBD's, cumpre todos os requisitos requeridos pelo previsor. Existe também uma biblioteca para acessar estes BD de dentro de aplicações em C/C++, sendo um grande atrativo para o desenvolvimento deste trabalho.

4.2.7 Kompex

A biblioteca *Kompex* é um *wrapper* orientado a objetos em C++ para facilitar a utilização da biblioteca de acesso ao SQlite3 do C++, fornecendo funções de mais alto nível ao programador. A biblioteca é de código aberto e gratuita com uma documentação vasta e com exemplos de uso. O maior atrativo desta biblioteca é que ela facilita a escrita do código e o mantém mais limpo, ajudando na leitura do mesmo.

4.3 DEFINIÇÃO DO BANCO DE DADOS

Como já mencionado anteriormente o banco de dados contém tanto informações essenciais para o funcionamento dos mecanismos tanto quanto informações auxiliares para métrica de comparação. A estrutura da tabela do mecanismo de base e do proposto são bem similares e diferem apenas em três colunas. O mecanismo de base possui as seguintes informações:

id: O id serial na tabela (Inteiro)

taskId: O id da tarefa a qual os dados pertencem (Inteiro)

startTime: A data de início da tarefa (string ISO 8601)

deadline: A data limite para término da tarefa (string ISO 8601)

distanceToGoal A distância estimada ao início da tarefa para o objetivo (double)

endTime: A data de término da tarefa (string ISO 8601)

distQ1: A distância total percorrida no marco q_1 (double)

predictorAnswer: A resposta do previsor (booleano)

success: Indica se a tarefa foi completada dentro do *deadline* (booleano)

Similarmente nós temos para a tabela do mecanismo proposto as seguintes colunas:

id: O id serial na tabela (Inteiro)

taskId: O id da tarefa a qual os dados pertencem (Inteiro)

startTime: A data de início da tareda (string ISO 8601)

deadline: A data limite para término da tarefa (string ISO 8601)

distQ1: A distância total percorrida no marco q_1 (double)

distQ2: A distância total percorrida no marco q_2 (double)

distQ3: A distância total percorrida no marco q_3 (double)

distQ4: A distância total percorrida no marco q_4 (double)

endTime: A data de término da tarefa (string ISO 8601)

predictorAnswer: A resposta do previsor (booleano)

success: Indica se a tarefa foi completada dentro do *deadline* (booleano)

Essas são as únicas tabelas que existem no BD e são suficientes para fazer uma análise estatística comparando as duas abordagens, como será visto no capítulo 5. Dessas informações apenas as colunas *predictorAnswer* e *success* não são necessárias para o funcionamento do previsor e são utilizadas apenas para verificação das taxas de acerto e erro da previsão.

4.4 IMPLEMENTAÇÃO DO BD

Para implementar essas informações no previsor foram abstraídas as informações do banco e as operações sobre essas informações e para isso foram criados definições juntamente com as operações básicas *insert*, *update*, *select* e *delete* para as tabelas *baseline* e *tasks* (B.13 e B.14 respectivamente) e modelos para manipulação de dados dessas mesmas tabelas (B.4 e B.12 respectivamente) juntamente com uma estrutura para representar esses dados.

Também para auxiliar na manipulação do banco de dados, uma vez que a biblioteca Kompex ainda foi considerada de baixo nível foram implementadas classes dentro do *namespace DbUtils* (B.6) para tornar o acesso ao BD mais orientado a objetos. Essa abstração é dependente da biblioteca kompex, mas as definições de tabelas e modelos não e pode-se usar com outros BD's.

Como descrito no Capítulo 3 o mecanismo termina sua execução ao prever que uma perda do *deadline* irá ocorrer, porém, apesar de manter essa ideia, a implementação do mecanismo não segue diretamente essa abordagem. Por isso nos modelos as consultas sempre filtram linhas onde não houve

uma perda efetiva do *deadline*, independente da resposta do previsor, sendo que assim buscamos no BD apenas informações que resultaram em um ambiente desejado de execução.

Vale ressaltar que essa abordagem foi adotada para avaliar a taxa de acertos e erros do mecanismo. Quando o mesmo for utilizado em produção, se for escolha de quem implantou o mecanismo, o robô irá tomar uma ação de correção e não irá mais terminar a tarefa em questão e essa entrada não seria salva no banco de dados. Então na próxima execução da mesma tarefa só irá constar dados de tarefas completadas com sucesso.

4.5 INTERFACES

Para não fixar nenhuma tecnologia na implementação a não ser a linguagem de programação toda a implementação é baseada em interfaces de comunicação, onde são definidas as funções que uma suposta especialização deve implementar. Em outras palavras o código não se importa como as funções são executadas, contanto que elas estejam implementadas.

Um exemplo é o arquivo RobotInterface.h (B.11) que define as funções que um robô deve fornecer ao mecanismo e uma especialização, AriaRobotInterface.cpp (B.17), para o robô Pioneer utilizado no trabalho. Outro arquivo definindo uma interface para gerar mensagens dentro do programa foi o LoggerInterface.h (B.7) com uma especialização no arquivo AriaLogger.cpp (B.16).

Com essa abordagem fica simples de se utilizar o mecanismo com outras bibliotecas ou até mesmo plataformas robóticas, bastando essas conseguirem fornecer as informações necessárias para o funcionamento do mecanismo.

4.6 ALGORITMOS

Antes de entrar em detalhes da implementação é preciso introduzir o tema da execução paralela dos mecanismos. Para se poder comparar os dois previsores eles precisam executar a mesma tarefa para conterem os mesmos dados durante a execução. As implementações dos mecanismos são independentes uma da outra mas as execuções são feitas ao mesmo tempo, no caso desse trabalho utilizando uma *thread* diferente para cada mecanismo.

Como os mecanismos tem que esperar até um marco para serem ativados as *threads* são colocadas para dormir até esse marco chegar. A utilização de *threads* é válida pelo fato de que com essa abordagem é possível que as

execuções dos mecanismos, se ocorrerem ao mesmo tempo, possam ser feitas em núcleos diferentes do processador.

Para ser possível configurar o previsor sem ter que alterar o código fonte ou criar um interface de configuração foi adotado um sistema de arquivo de configuração, como mostarado na listagem 4.1.

Listagem 4.1: Exemplo de arquivo de configuração

```
filename = "Configuration File Sample";

// Parameters
params =
{
    deadline_seconds = 90; //Deadline em segundos
    database = "./db/predictor_limit_25.db"; //Caminho onde será criado/lido o
        ↪ banco de dados
    limitDb = 25; //Limita as queries em 25 resultados
};
```

Para interfacear os mecanismos com o robô Pioneer foi criada a classe *ArPredictorTask* (B.3), que estende a classe *ArASyncTask* pertencente à biblioteca ARIA, que tem como função executar o seu conteúdo de modo assíncrono com as outras tarefas do robô. Se observado nessa classe verifica-se que existe um atributo chamado *training* que tem como objetivo alternar entre o modo de treinamento e execução do mecanismo.

Para executar os mecanismos no robô foi utilizado um programa de exemplo que vem junto com a biblioteca ARNL, onde o mesmo foi alterado para incluir o código necessário para se criar a instância da classe acima descrita. Para instanciar essa classe é preciso passar para o construtor 3 parâmetros sendo eles uma referência para uma instância do robô, uma referência para o planejador de rotas e uma *string* representando o diretório, com caminho completo ou relativo, onde o arquivo de configuração a utilizado se encontra e opcionalmente um ponteiro para uma função a ser executada quando o previsor indicar que irá ocorrer uma perda do deadline.

Nesse construtor são criadas instâncias do mecanismo de base e o previsor proposto, como serão descritos posteriormente. Uma vez que essa classe é colocada em execução ela que cuida dos tempos para invocar os previsores no momento certo, verificando se o modo de execução do programa é de treinamento ou não.

O modo de treinamento é utilizado para popular o banco de dados sem a preocupação da resposta dos mecanismos e portanto não são executados os cálculos de previsão. Nesse modo os dados são lidos do robô e atualizados no banco de dados nas colunas correspondentes quando os marcos são atingidos. Por exemplo, ao chegar no marco q_1 é lida a distância percorrida e esse dado é salvo na coluna “distQ1” na linha da tarefa em questão.

Em modo normal de execução os mecanismos seguem suas definições

matemáticas, lembrando que foi delegado ao modelo retornar os dados válidos para o previsor. Quando iniciada a execução da classe *ArPredictorTask* nesse modo ela dispara duas *threads*.

A primeira executa uma função que prepara o previsor proposto para uma nova tarefa e controla o tempo de ativação dos marcos e, quando os marcos forem alcançados, ela invoca o previsor para efetuar os cálculos. A segunda *thread* executa uma função que, similarmente executa o mecanismo de base. A implementação dessa classe pode ser vista em B.19

Os cálculos, como dito anteriormente, seguem fielmente a descrição matemática e podem ser verificados em B.24 para o mecanismo proposto e em B.21 para o mecanismo de base.

5 SIMULAÇÃO

Nesse capítulo serão descritos os passos efetuados para executar as simulações dos mecanismos apresentados, sendo eles a preparação do ambiente de execução, coleta de dados e execução.

5.1 PREPARAÇÃO DO AMBIENTE DE EXECUÇÃO

Como descrito anteriormente, o ambiente escolhido para testes foi o terceiro andar do INE, onde era possível fazer testes com o robô real de forma controlada. Uma vez escaneado e processado um mapa, foram estabelecidos dois objetivos para o robô alcançar: a porta de entrada do LAPESD e o Elevador, como já mostrados na Figura 3.1, lembrando que as tarefas para esse trabalho são o deslocamento de um ponto A a um ponto B e do ponto B de volta ao ponto A, sendo que os pontos A e B são respectivamente a porta do LAPESD e o Elevador

Para ser possível controlar o robô remotamente foi necessário configurar o mesmo para utilizar a infraestrutura de rede sem fio fornecida pela universidade, visto que utilizar uma conexão cabeada entre um computador e o robô tornaria inviável a execução.

Configurada a rede foi necessário também instalar todas a bibliotecas necessárias no robô para se poder executar o programa principal. Com a preparação pronta foi então executado o programa principal para iniciar a próxima etapa, a de coleta de dados. Porém o processo se demonstrou demasiadamente lento por causa da velocidade de movimento do robô. O que foi decidido então foi fazer algumas coletas com o robô real e depois comparar com os dados coletado pelo simulador.

Após 25 execuções de cada tarefas pelo robô real e 25 execuções pelo simulador, superando algumas expectativas, ficou demonstrado que não era possível definir o método de coletas dos dados coletados olhando apenas para os valores lidos pois os mesmos são extremamente semelhantes. A partir desse resultado todo os passos seguintes foram executados no simulador *MobileSim*.

Sendo assim todos os processos de preparo do robô tiveram que ser feitos no equipamento onde serão executadas as simulações, no caso um notebook HP Pavillion dv-5 2040-br com 4Gb de RAM, processador Intel Core i3 M 350 de 2,27GHz de *clock*, HD SAMSUNG HM500JJ de 500Gb. Após esse preparo foi possível então começar a coleta de dados.

5.2 COLETA DE DADOS

O processo de coleta de dados foi bem simples uma vez que que ficou definido a quantidade de entradas inicial no banco de dados, 400 no total sendo 200 para cada tarefa. Esse valor foi estipulado pelo fato de que o limite máximo de tamanho do histórico ficou estipulado em 200 e os dados mais antigos serão substituídos pelos mais novo, logo com uma população inicial maior que 400 os dados mais antigos nunca seriam utilizados. Esses resultados obtidos então serão utilizados como dados iniciais para todos os testes com diferentes tamanhos de histórico.

Com a coleta feita e a observação dos seguintes valores obtidos ficou estipulado que o *deadline* das tarefas seria de 90 segundos, um valor próximo ao da tarefa 1 (de A para B) porém com uma certa folga na tarefa 2 (de B para A). A pequena diferença de distância percorrida nas duas tarefas, aproximadamente 200 mm, foi devido à configuração do mapa onde o planejador de rotas fazia um pequeno desvio quando indo de A para B.

- Tempo médio de A até B = 88,0972 segundos
- Distância média de A até B = 33410,6395 milímetros
- Tempo médio de B até A = 80,4687 segundos
- Distância média de B até A = 33198,0971 milímetros

Após definido o valor do deadline foram criados 4 arquivos de de configuração, um para cada valor de limite do banco de dados, e copiados esses dados iniciais para 4 arquivos diferentes de BD.

5.3 EXECUÇÃO

Uma vez definidos os parâmetros de execução para iniciar a simulação foram necessários 3 outros programas além do que executaria o mecanismo. Primeiramente foi necessário executar o simulador MobileSim para receber dois robôs no ambiente. Um desses robôs seria um outro programa de exemplo, que veio junto com a biblioteca Aria, que simplesmente executa um robô que fica vagando pelo ambiente sem um objetivo a alcançar, cujo objetivo é simular uma anomalia no ambiente.

Sendo assim, foi executado o programa que contém os mecanismos de previsão, que no simulador ficou esperando os comandos para iniciar suas tarefas, esses que foram provenientes do programa MobileEyes. Nesse programa existe um comando para fazer com que o robô fique se movendo entre

os objetivos definidos no mapa. Como só há dois objetivos definidos no mapa utilizado ele ficou se deslocando entre o LAPESD e o Elevador.

Como o fator aleatório do ambiente já estava definido não era preciso ficar observando as simulações e era possível deixar rodando durante longos períodos de tempo. Contanto foi nessa etapa que as equações de previsão foram sendo aprimoradas até chegarem no seu formato final, como apresentado no capítulo 3 e foram necessárias mais de 120 horas de experimentos até se obter resultados satisfatórios.

Após amadurecidas essas equações foram realizadas mais de 400 execuções, além dos dados iniciais, para cada limite de banco de dados, ou seja, mais de 1600 deslocamentos, sendo então gastos mais de 40 horas de simulação.

Como era praticamente impossível supervisionar todas as execuções dos trajetos ocasionalmente apareciam situações em que o previsor de rota decidia que não existiam alternativas quando o robô executando o algoritmo proposto encontrava o robô errante. Nesses casos eram retiradas essas entradas do banco de dados uma vez que eles não iriam altear a execução do mecanismo.

6 RESULTADOS

Neste capítulo estão apresentados os dados coletados durante as simulações juntamente com suas análises. Foram feitas análises de desempenho das tarefas de modo individual e também uma taxa global de desempenho que contempla as duas tarefas, sempre tendo como referência o algoritmo de base.

Na Figura 6.1a pode-se verificar que o resultado global para um histórico de 25 entradas é aproximadamente 10% melhor que a taxa do algoritmo de base. No entanto a taxa de falsos positivos também aumentou, o que pode ser indesejado em alguns casos onde o resultado de uma perda de *deadline* pode ter um impacto negativo muito grande, porém quando analisamos individualmente as tarefas percebemos que há uma grande diferença de resultados entre elas.

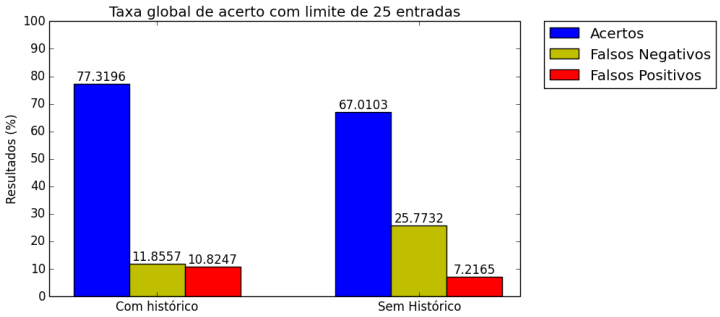
Começando pela tarefa 0 pode-se verificar na Figura 6.1b que o previsor teve uma taxa de acerto muito próxima ao algoritmo de base. No entanto as taxas de erro sofreram uma grande alteração, onde a taxa de falsos negativos aumentou em 4 vezes e a de falsos positivos diminuiu aproximadamente pela metade.

Já na tarefa 1, como visto na Figura 6.1c, é muito mais perceptível a diferença de performance dos mecanismos. Enquanto o mecanismo de base teve uma taxa de acerto de apenas 48,9583% a taxa de acertos do mecanismo proposto foi de 71,8750%, um ganho de aproximadamente 23%. Essa variação também é bem perceptível nas taxas de erro, onde os falsos positivos tiveram uma queda de mais de 4 vezes e os falsos positivos tiveram um grande aumento de 8 vezes.

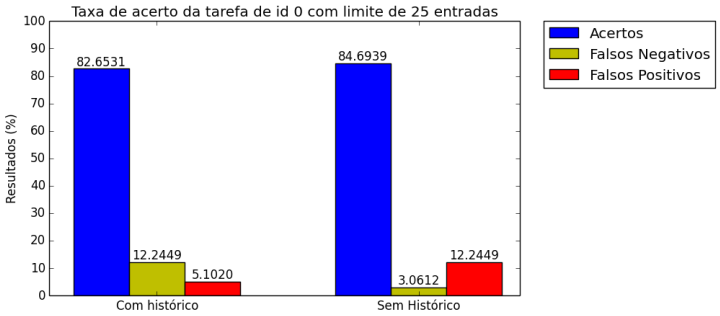
Se formos fazer a mesma análise sobre os dados para as tarefas executadas com um histórico de 50 entradas, como mostram os gráficos na Figura 6.2 vamos notar que o mesmo padrão se segue. Nas taxas globais o mecanismo proposto teve uma performance 10% melhor em relação a taxa de acerto e, apesar da taxa de falsos positivos terem diminuído em quase 15% em relação ao algoritmo de base houve um aumento de aproximadamente 5,5% nos falsos positivos.

Partindo para a análise individual das tarefas notamos que na tarefa 0 a taxa de acerto do mecanismo proposto, apesar de um pouco abaixo, é muito próxima à taxa do mecanismo de base. Houve um pequeno aumento na taxa de falsos positivos e uma pequena queda na taxa de falsos positivos.

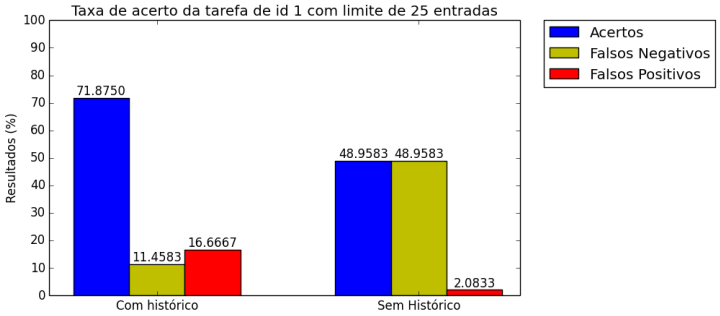
A tarefa 1 também segue o mesmo padrão da sua correspondente de 25 entradas no histórico, tendo um aumento de 20% na taxa de acerto, uma queda muito significativa de 34% nos falsos negativos mas tendo um aumento



(a) Taxas globais de acerto e erro



(b) Taxas de acerto e erro da tarefa 0



(c) Taxas de acerto e erro tarefa 1

Figura 6.1: Taxas de acerto e erro com histórico de 25 entradas

de 14% nos falsos positivos, os quais são inexistentes no algoritmo de base.

E fazendo as mesmas análises para as tarefas executadas com 100 e 200 entradas no histórico vamos perceber o mesmo padrão. Na análise global o mecanismo proposto tem uma taxa de acerto e de falsos positivos superior ao algoritmo de base enquanto a taxa de falsos negativos é inferior.

Na tarefa 0 para essas mesmas quantidades de dados nota-se, do mesmo modo que nas análises anteriores, que as taxas de acerto entre os mecanismos permanecem muito próximas uma da outra enquanto há uma elevação de falsos negativos e uma diminuição de falsos positivos por parte do mecanismo proposto. E novamente há uma elevação considerável das taxas de acerto e de falsos positivos, apesar da diminuição também considerável dos falsos negativos na tarefa 1.

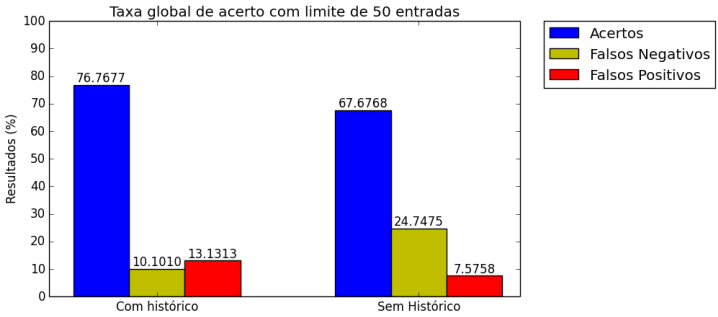
Sendo assim fica evidente que as taxas de acerto para a tarefa com um *deadline* mais folgado para ambos mecanismos são muito parecidas e mais altas que as taxas da tarefa com *deadline* mais apertado. Contudo o mecanismo proposto se mostrou mais eficiente nesse último caso, contanto que se considere apenas as taxas de acerto. Isso porque sempre houve um aumento considerável na taxa de falsos positivos que, como dito anteriormente, pode ser indesejável.

E é justamente nessa parte que o tamanho do histórico mostrou seu impacto interno ao mecanismo. A Tabela 6.1 mostra a relação entre a quantidade de entradas no histórico e as taxas de acerto e falsos positivos, sendo possível chegar a conclusão de que com o aumento do histórico houve uma pequena queda da taxa de acerto mas a relação entre essa taxa e a taxa de falsos positivos aumenta, ou seja, para cada falso positivo se tem uma quantidade maior de acertos.

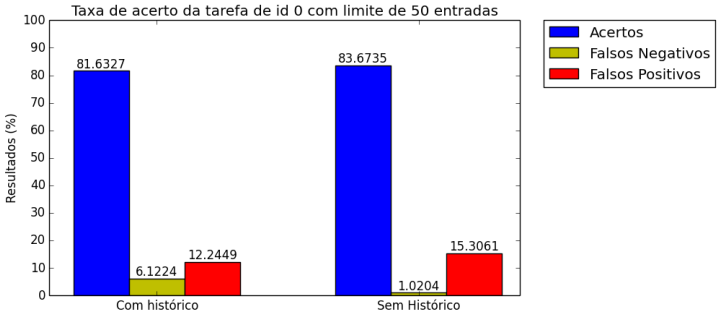
Tamanho	Tarefa	Taxa de acerto (TA)	Falsos Positivos (FP)	TA / FP
25	0	82,6531	5,1020	16,2000
	1	71,8750	16,6667	4,3125
50	0	81,6327	12,2449	6,6666
	1	72,0000	14,0000	5.1428
100	0	78,0220	16,4835	4,7333
	1	70,6522	26,0870	2.7084
200	0	86,9347	8,0402	10,8125
	1	70,6468	9,4527	7.4737

Tabela 6.1: Relação do tamanho do histórico com as taxas de acerto e falsos positivos

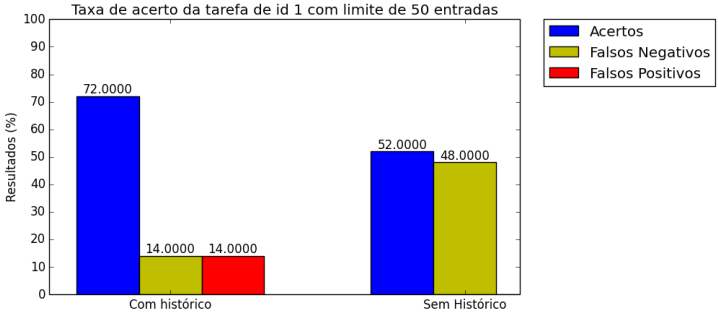
Nos gráficos 6.5a e 6.5b fica mais fácil ver que variando o tamanho do



(a) Taxas globais de acerto e erro



(b) Taxas de acerto e erro da tarefa 0

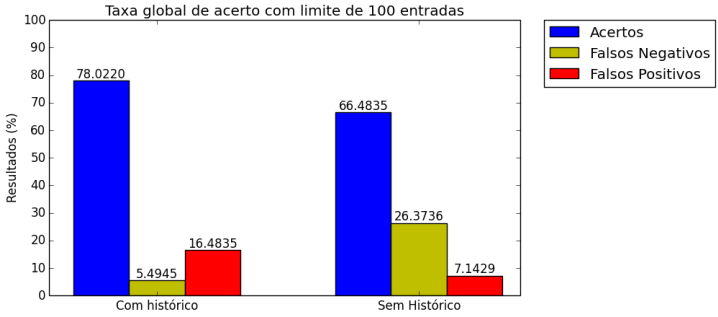


(c) Taxas de acerto e erro tarefa 1

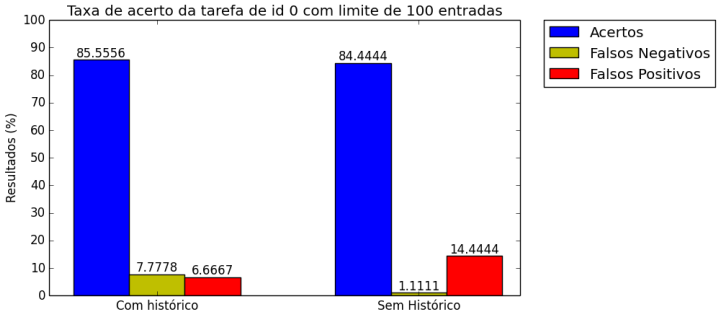
Figura 6.2: Taxas de acerto e erro com histórico de 50 entradas

histórico não houve variação considerável na taxa de acerto e que na tarefa 0, onde o *deadline* é mais folgado a performance dos dois mecanismos foi bem constante com valores entre 81% e 87%, tendo uma diferença máxima entre valores de 5,3020% para o mecanismo proposto e 1,0204% para o mecanismo de base.

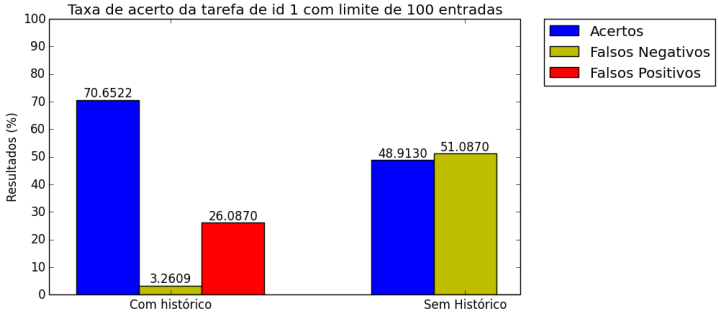
Já na tarefa 1, onde o *deadline* é mais apertado o mecanismo proposto manteve valores mais uniformes, entre 70,6468% e 72%, mantendo uma diferença máxima de 1,3532% enquanto o algoritmo de base se manteve bem abaixo desses valores com um mímino de 48,9130% e um máximo de 57,2139 tendo uma variação de 8,3009%.



(a) Taxas globais de acerto e erro

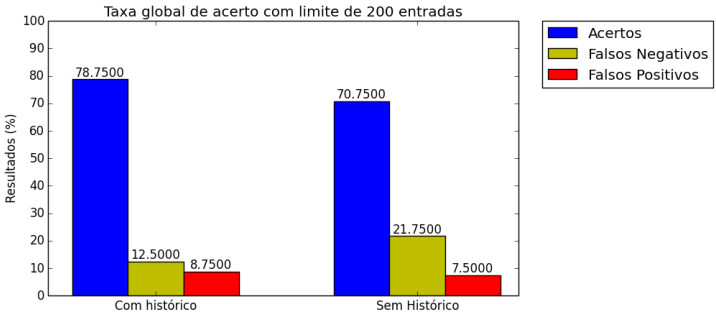


(b) Taxas de acerto e erro da tarefa 0

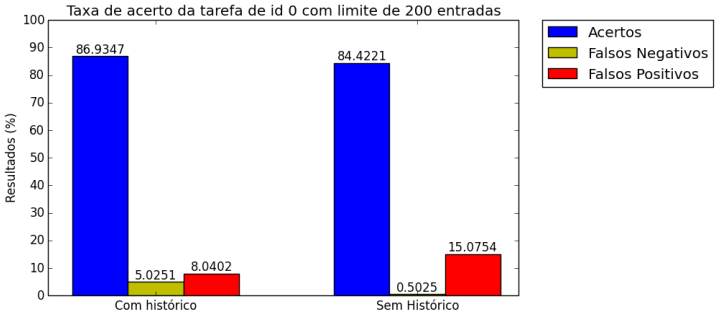


(c) Taxas de acerto e erro tarefa 1

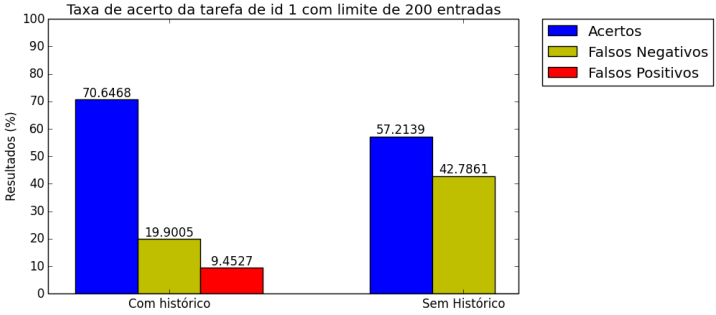
Figura 6.3: Taxas de acerto e erro com histórico de 100 entradas



(a) Taxas globais de acerto e erro

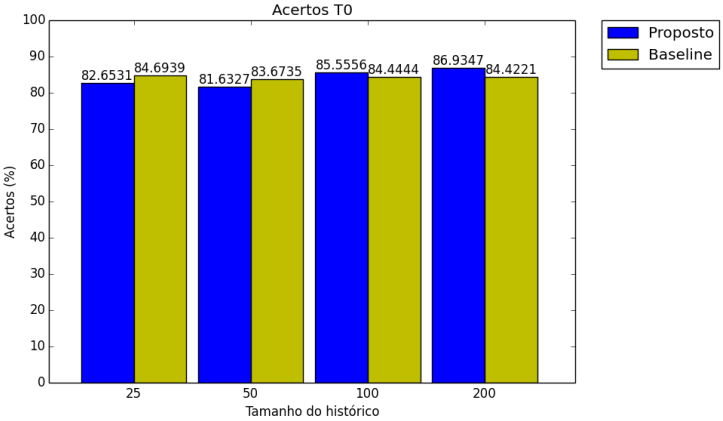


(b) Taxas de acerto e erro da tarefa 0

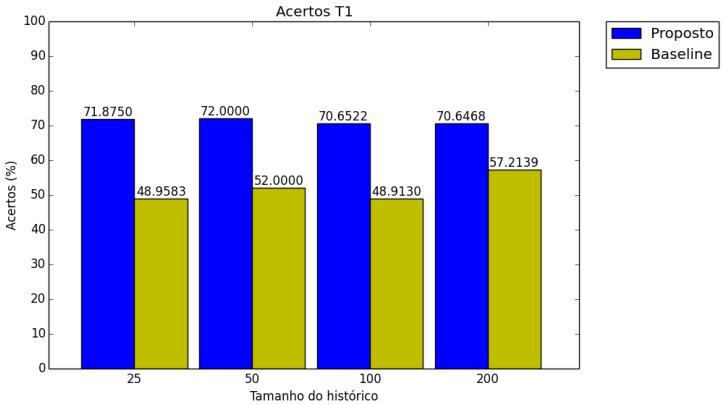


(c) Taxas de acerto e erro tarefa 1

Figura 6.4: Taxas de acerto e erro com histórico de 200 entradas



(a) Taxas de acerto tarefa 0



(b) Taxas de acerto tarefa 1

Figura 6.5: Taxas de acertos das tarefas

7 CONCLUSÃO

Este trabalho descreveu, implementou e analisou um mecanismo de previsão de cumprimento de tarefas de alto nível em robótica móvel. A proposta foi implementada sobre a plataforma robótica Pioneer P3-DX em C++ como uma tarefa para ser executada de modo assíncrono em relação às outras tarefas do robô.

O objetivo de descrever e implementar o mecanismo foram cumpridos nos capítulos 3 e 4, respectivamente. O objetivo de simulação foi cumprido no capítulo 5 e a análise dos resultados obtidos foi cumprida no capítulo 6.

Com esses objetivos cumpridos ficou claro que para uma tarefa com um *deadline* folgado o previsor tem um desempenho pior do que o algoritmo de base e adiciona características indesejadas. Porém com um deadline mais justo o trabalho proposto apresenta uma performance superior com o efeitos colateral de maior quantidade de erros indesejados.

7.1 TRABALHOS FUTUROS

Nesta seção serão apresentadas algumas propostas de trabalhos futuros que foram identificados na fase final do projeto onde não cabiam as alterações ou que não foram abordados.

7.1.1 Melhorias no banco de dados

Ao final do projeto foram identificadas algumas deficiências na estrutura do banco de dados, sendo elas:

- A coluna “taskId” não possui valor semântico
- Os nomes das colunas não indicam o tipo ou unidade dos dados

Talvez fosse interessante alterar a estrutura do banco para sanar essas demandas, como talvez criar uma tabela auxiliar para conter informações das tarefas e relacionar com a coluna acima mencionada.

7.1.2 Comparação com técnicas de Inteligência Artificial

Como descrito no início do trabalho, o mecanismo proposto visa fornecer uma alternativa de baixo custo computacional à técnicas de inteligência

artificial. Como essa área abrange muitas outras acabou não pertencendo ao escopo deste trabalho, portanto seria de interesse comparar o mecanismo proposto com alguma técnica de IA que se aplique.

REFERÊNCIAS BIBLIOGRÁFICAS

NG, J.; BRäUNL, T. Performance comparison of bug navigation algorithms. *Journal of Intelligent and Robotic Systems*, Springer Netherlands, v. 50, n. 1, p. 73–84, 2007. ISSN 0921-0296. Disponível em: <http://dx.doi.org/10.1007/s10846-007-9157-6>.

ASIMOV, I. *I, Robot*. Bantam Books, 1950. (Robot series). Disponível em: <http://books.google.com.br/books?id=MD0GAQAAIAAJ>.

SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. Scituate, MA, USA: Bradford Company, 2011. ISBN 978-0-262-01535-6.

NASSIRAEI, A. A. F. et al. Concept and design of a fully autonomous sewer pipe inspection mobile robot "kantaro". In: *Robotics and Automation, 2007 IEEE International Conference on*. [S.l.: s.n.], 2007. p. 136–143. ISSN 1050-4729.

FARINES, J.-M.; FRAGA, J. da S.; OLIVEIRA, R. da Silva de. Sistemas de tempo real. *Escola de Computação, IME-USP, São Paulo-SP*, 2000.

KOPETZ, H. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. [S.l.]: Springer US, 2011. (Real-Time Systems Series). ISSN 1867-321X. ISBN 978-1-4419-8236-0.

FAPESP. 11 2007. Disponível em: <http://revistapesquisa.fapesp.br/2007/11/01/robos-aquaticos/>.

DUDEK, G.; JENKIN, M. *Computational Principles of Mobile Robotics*. Segunda edição. Cambridge University Press, 2010. ISBN 978-0-521-69212-0. Disponível em: <http://www.cambridge.org/9780521692120>.

BETKE, M.; GURVITS, L. Mobile robot localization using landmarks. *Robotics and Automation, IEEE Transactions on*, v. 13, n. 2, p. 251–263, Apr 1997. ISSN 1042-296X.

DISSANAYAKE, M. et al. A solution to the simultaneous localization and map building (slam) problem. *Robotics and Automation, IEEE Transactions on*, v. 17, n. 3, p. 229–241, Jun 2001. ISSN 1042-296X.

MOBILEROBOTICS. *Seekur Jr Outdoor Manipulator*. 2014. Disponível em: <http://www.mobilerobots.com/ResearchRobots/SeekurJr_with_Outdoor_Manipulator.aspx>.

FERREIRA, A. et al. An approach to avoid obstacles in mobile robot navigation: the tangential escape. *Sba: Controle & Automação Sociedade Brasileira de Automatica*, scielo, v. 19, p. 395 – 405, 12 2008. ISSN 0103-1759. Disponível em: <http://www.scielo.br/scielo.php?pid=S0103-17592008000400003&script=sci_arttext>.

KAMON, I.; RIVLIN, E.; RIMON, E. A new range-sensor based globally convergent navigation algorithm for mobile robots. In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*. [S.l.: s.n.], 1996. v. 1, p. 429–435 vol.1. ISSN 1050-4729.

MEI, Y. et al. A case study of mobile robot's energy consumption and conservation techniques. In: *Advanced Robotics, 2005. ICAR '05. Proceedings., 12th International Conference on*. [S.l.: s.n.], 2005. p. 492–497.

PEDRAM, M.; WU, Q. Battery-powered digital cmos design. In: *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*. [S.l.: s.n.], 1999. p. 72–76.

WANG, B. et al. Power-aware real-time task scheduling with feedback control for mobile robots. In: *Industrial Electronics and Applications, 2008. ICIEA 2008. 3rd IEEE Conference on*. [S.l.: s.n.], 2008. p. 1240 –1245. ISBN 978-1-4244-1717-9.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. S. de. Deadline missing prediction in systems based on distributed threads. *JRI 2008 - EnRI-LARS2008 - 5th Latin American Robotics Symposium*, 2008. Disponível em: <http://www.das.ufsc.br/~romulo/artigos/Patricia-Montez-Romulo-Lars2008.pdf>.

TSUBOUCHI, T.; ARIMOTO, S. Behavior of a mobile robot navigated by an ldquo;iterated forecast and planning rdquo; scheme in the presence of multiple moving obstacles. In: *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. [S.l.: s.n.], 1994. p. 2470–2475 vol.3.

MIURA, J.; UOZUMI, H.; SHIRAI, Y. Mobile robot motion planning considering the motion uncertainty of moving obstacles. In: *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*. [S.l.: s.n.], 1999. v. 4, p. 692–697. ISSN 1062-922X.

MICHAUD, F.; MATARIć, M. Learning from history for behavior-based mobile robots in non-stationary conditions. *Machine Learning*, Kluwer Academic Publishers, v. 31, n. 1-3, p. 141–167, 1998. ISSN 0885-6125. Disponível em: <http://dx.doi.org/10.1023/A

MOBILEROBOTICS. *Especificação Pioneer P3-DX*. 2014. Disponível em: <`http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx`>.

SICK. *Laser scanners LMS2xx / LMS200 / Indoor / Short Range*. 2014. Disponível em: <`https://mysick.com/PDF/Create.aspx?ProductID=33755&Culture=en-US`>.

ANEXO A – Descrição do Banco de Dados

A.1 SCRIPTS

```
CREATE TABLE baseline(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    taskId INTEGER NOT NULL,
    startTime TEXT NOT NULL,
    deadline TEXT NOT NULL,
    distanceToGoal DOUBLE NOT NULL,
    endTime TEXT DEFAULT NULL,
    distanceWalkedQ1 DOUBLE DEFAULT NULL,
    predictorAnswer INT DEFAULT NULL,
    success INT DEFAULT NULL
);

CREATE TABLE tasks(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    taskId INTEGER NOT NULL,
    startTime TEXT NOT NULL,
    deadline TEXT NOT NULL,
    distQ1 DOUBLE DEFAULT NULL,
    distQ2 DOUBLE DEFAULT NULL,
    distQ3 DOUBLE DEFAULT NULL,
    distQ4 DOUBLE DEFAULT NULL,
    endTime TEXT DEFAUL NULL,
    predictorAnswer INT DEFAULT NULL,
    success INT DEFAULT NULL
);
```

ANEXO B – Código Fonte

B.1 HEADERS

Listagem B.1: AriaLogger.h

```
#ifndef ARIALOGGER_H_
#define ARIALOGGER_H_

#include "interfaces/LoggerInterface.h"
#include <string>

using namespace std;

/**
 * ARIA implementatios of the LoggerInterface.
 */
class AriaLogger: public LoggerInterface
{
private:
    void log(string, string); /**< Display the text to be logged*/

public:
    AriaLogger(LogLevel level=ALL);

    /**
     * @see info
     */
    void info(const char* message);

    /**
     * Format the message with the '<INFO>' prefix
     *
     * @param message The message to be formatted
     */
    void info(string message);

    /**
     * @see warning
     */
    void warning(const char* message);

    /**
     * Format the message with the '<WARN>' prefix
     *
     * @param message The message to be formatted
     */
    void warning(string message);

    /**
     * @see error
     */
    void error(const char* message);

    /**
     * Format the message with the '<ERROR>' prefix
     *
     * @param message The message to be formatted
     */
    void error(string message);
```

```
    /**
     * @see trace
     */
    void trace(const char* message);

    /**
     * Format the message with the '<TRACE>' prefix
     *
     * @param message The message to be formatted
     */
    void trace(string message);

};

#endif /* ARIALOGGER_H_ */
```

Listagem B.2: AriaRobot.h

```
#ifndef ARIA_ROBOT_INTERFACE_H_
#define ARIA_ROBOT_INTERFACE_H_

#include "Aria.h"
#include "Arnl.h"
#include "interfaces/RobotInterface.h"

/**
 * ARIA implementatio of the RobotInterface
 */
class AriaRobot: public RobotInterface
{
protected:
    ArRobot * robot_; /**< Pointer to the ArRobot object */
    ArPathPlanningTask * planning_; /**< Ponter to the ArPathPlanningTask object */
    double lastDistanceRead_; /**< Last reading of distance to goal */
    double lastWalkedDistRead_; /**< Last total distance walked read */

public:
    /**
     * Constructor.
     * Initialize the robot_ and planning_ pointers
     *
     * @param robot The ArRobot pointer
     * @param pathPlanning The pointer to the ArPathPlanningTask
     */
    AriaRobot(ArRobot * robot, ArPathPlanningTask * pathPlanning);

    /**
     * TODO: Document
     */
    void init();

    /**
     * Returns the estimated distance to goal with a clear path
     *
     * @return The distance in meters
     */
    double distanceToGoal();

    /**
     * Returns the estimated time of arrival to goal with a clear path
     *
```

```
     * @return The time in seconds
     */
    double timeToGoal();

    /**
     * Returns the total distance walked based on the robot odometer
     *
     * @return The distance, in meters, the robot walked in the current task
     */
    double distanceWalked();

    /**
     * @returns If the goal has been achived
     */
    bool goalAchived();

    /**
     * Return the id of the current tasks
     *
     * @return The id of the current task
     */
    int currentTaskId();
};

#endif /* ARIA_ROBOT_INTERFACE_H_ */
```

Listagem B.3: ArPredictorTask.h

```
#ifndef ARIAPREDICTORTASK_H_
#define ARIAPREDICTORTASK_H_

#include <baselinePredictor.h>
#include "ArASyncTask.h"
#include "AriaRobot.h"
#include "Predictor.h"
#include "AriaLogger.h"
#include <thread>

/**
 * Implentation of a asynchronous task to call the predictor
 */
class ArPredictorTask: public ArASyncTask
{
protected:
    Predictor* predictor_; /**< Pointer to the predictor algorithm */
    BaselinePredictor *baseline_; /**< Pointer for the baseline predicto */
    AriaLogger* logger_; /**< TODO: Document */
    AriaRobot robot_; /**< Implentation of the RobotInteface */
    string configFilePath_; /**< Path to the config file */
    PredictorConfig *config_; /**< Pointer to a config object */
    ArFunctor* failAction_; /**< Function to be executed on action failed*/
    bool training_; /**< Define if the task will run in training mode*/

public:
    /**
     * Class constructor
     *
     * @param robot Pointer to the robot instance
     * @param planning Pointer to the path planning task
     */
    ArPredictorTask(ArRobot* robot, ArPathPlanningTask* planning, string cfgFile="");
```

```
    /**
     * Class constructor
     *
     * @param robot Pointer to the robot instance
     * @param planning Pointer to the path planning task
     * @param failFunction Function to be called uppon deadline miss prediction
     */
    ArPredictorTask(ArRobot* robot, ArPathPlanningTask* planning, string cfgFile,
          ↪ ArFunctor* failFunction);

    /**
     * Deletes the Predictor instance
     */
    ~ArPredictorTask();

    /**
     * Loop to run the predictor algorithm
     * @param param Any param used at the algorithm implentation
     * @return Any value return by the algorithm implentation
     */
    void* runThread(void* param);

    /**
     * Sets the function to be run on negative prediction
     *
     * @param failFunction Function to be run at negative prediction
     */
    void failAction(ArFunctor* failFunction);

    /**
     * Defines if the predictor will run in trainig mode
     *
     * @param value If the predictor should run in training mode or not
     */
    void setTrainig(bool value);

    /**
     * TODO: Document
     */
    void updateOnGoalCompleted();

    /**
     * TODO: Document
     */
    void runPredictor();

    /**
     * TODO: Document
     */
    void runBaseline();
};

#endif /* ARIAPREDICTORTASK_H_ */
```

Listagem B.4: baseline.h

```
/*
 * baseline.h
 *
 * Created on: 05/04/2014
```

```
 * Author: rafael
 */

/**
 * TODO: Document the whole file
 */
#ifndef BASELINE_H_
#define BASELINE_H_

#include <models/baseline.h>
#include <models/tables/tbBaseline.h>
#include "utils/dbHelpers.h"

typedef TbBaseline::BaselineData BaselineData;
typedef DbHelpers::Column Column;
typedef DbHelpers::Condition Condition;
typedef DbHelpers::ResultSet ResultSet;

class BaselineModel {
public:
    BaselineModel(LoggerInterface *logger);
    int insert(BaselineData data);
    int update(BaselineData data, list<Condition> conditions);
    int update(BaselineData data, Condition condition);
    int currentId();
    void updateCurrentId_();
    int updateCurrentTaskId();

protected:
    std::string tbDefinition_; /**< 'tasks' table definition */
    int currentId_;
    int currentTaskId_;
    LoggerInterface *logger_;

protected:
    void createTable();
};

#endif /* BASELINE_H_ */
```

Listagem B.5: baselinePredictor.h

```
/*
 * baselinePredictor.h
 *
 * Created on: 06/04/2014
 * Author: rafael
 */
//TODO: Document whole file
#ifndef BASELINEPREDICTOR_H_
#define BASELINEPREDICTOR_H_
#include <models/baseline.h>
#include "PredictorConfig.h"
#include "interfaces/RobotInterface.h"
#include "interfaces/LoggerInterface.h"

class BaselinePredictor {
protected:
    PredictorConfig* config_;
    RobotInterface* robot_;
    LoggerInterface* logger_;
```

```
    BaselineModel* baselineModel_;
    double distanceToGoal_;
    double distanceWalkedQ1_;
    int currentTaskId_;
    int result_;

public:
    BaselinePredictor(RobotInterface* robot, LoggerInterface* logger, PredictorConfig *
        ↪ config);
    ~BaselinePredictor();
    int prepareNewTask(PredictorConfig *config);
    void registerEndTask();
    double timeToDeadline();
    void runPrediction();
};

#endif /* BASELINEPREDICTOR_H_ */
```

Listagem B.6: dbHelpers.h

```
/*
 * dbHelpers.h
 *
 * Created on: 01/07/2013
 * Author: rafael
 */


/*
 * TODO: Document whole file
 */

#ifndef DBHELPERS_H_
#define DBHELPERS_H_

#include "kompex/KompexSQLiteStatement.h"
#include "kompex/KompexSQLiteDatabase.h"
#include "kompex/KompexSQLiteException.h"
#include "interfaces/LoggerInterface.h"
#include <string>
#include <map>
#include <list>
#include <boost/any.hpp>
#include <iostream>

using namespace std;

typedef Kompex::SQLiteStatement SQLiteStatement;
typedef Kompex::SQLiteDatabase SQLiteDatabase;
typedef Kompex::SQLiteException SQLiteException;

class DbHelpers
{
protected:
    static SQLiteDatabase* db_; /**< Pointer to the Kompex::SQLiteDatabase database connection */

public:
    static bool openDb(string dbName, LoggerInterface* logger);
    class ResultSet
    {
    public:
```

```
    typedef class Data_
    {
    public:
        map<string, boost::any> fields;
    } Data;
public:
    ResultSet()
    {
        this->init();
    }
    bool empty();
    int length();
    void setValue(SQLiteStatement& stmt, Data* data, string column);
    void addRow(Data);
    int getInt(string column);
    double getDouble(string column);
    string getText(string column);
    void* getBlob(string column);
    bool end();

    ResultSet& operator++() // ++var
    {
        it_++;
        return *this;
    }
    ResultSet& operator--()
    {
        it_--;
        return *this;
    }
    ResultSet operator++(int) // var++
    {
        ResultSet s(*this);
        ++(*this);
        return s;
    }
    ResultSet operator--(int)
    {
        ResultSet s(*this);
        --(*this);
        return s;
    }
    void init();
protected:
    list<Data> rows_;
    list<Data>::iterator it_;
};

class Table
{
public:
    string name;
    string alias;
    Table(string name_, string alias_ = "") :
                    name(name_), alias(alias_)
    {
    }
    ;
    friend bool operator<(const Table &t1, const Table &t2)
    {
        return false;
```

```
    }
};

class Column
{
public:
    string name;
    string alias;
    Column(string name_, string alias_ = "") :
                    name(name_), alias(alias_)
    {
    }
    ;
    friend bool operator<(const Column &t1, const Column &t2)
    {
        return false;
    }
};

class Condition
{
public:
    string field;
    string operation;
    string value;
    Condition(string field_, string operation_, string value_) :
                    field(field_), operation(operation_), value(value_) {}
    ;
    friend bool operator<(const Condition &t1, const Condition &t2)
    {
        return false;
    }
};

class Join
{
public:
    string type;
    string table;
    list<Condition> on;
    string alias;
    Join(string table_, string type_ = "", list<Condition> on_ = list<Condition>(),
                    string alias_ = "") :
                    type(type_), table(table_), on(on_), alias(alias_)
    {
    }
    ;
    friend bool operator<(const Join &t1, const Join &t2)
    {
        return false;
    }
};

class Statement
{
protected:
    SQLiteStatement stmt_; /**< Object to run queries and statements */
public:
    Statement();
    Statement(string sql);
    void insert(string table, map<string, string> fields);
```

```
        void update(string table, map<string, string> fields, list<Condition> conditions);
        void update(string table, map<string, string> fields, Condition condition);
        void remove(string table, list<Condition> conditions);
        void remove(string table, Condition condition);
        void rawSQL(string sql);
        bool run();
    };

    class Query: public Statement
    {
    protected:
        friend ostream& operator<<(ostream& out, const Query& q);
        ResultSet* result_;
        string query_;
        list<Column> columnList_;
        list<Table> tableList_;
        list<Join> joinList_;
        list<Condition> conditionList_;
        string limit_;
        list<string> orderBy_;
        bool asc_;

    protected:
        string buildQuery_() const;

    public:
        Query();
        ~Query();
        Query* select(list<Column> columns);
        Query* select(Column column);
        Query* select(string column = "*", string alias = "");
        Query* from(list<Table> tables);
        Query* from(Table table);
        Query* from(string table, string alias = "");
        Query* fromSubquery(Query query);
        Query* join(list<Join> query_joins);
        Query* join(Join join);
        Query* join(string table, string type = "", string alias = "", list<Condition>
            ↪ condition =
                    list<Condition>());
        Query* where(list<Condition> conditions);
        Query* where(Condition condition);
        Query* where(string field, string operation, string value);
        Query* limit(int limit, int offset=0);
        Query* orderBy(string column, bool asc=true);
        Query* orderBy(list<string> columns, bool asc=true);
        ResultSet fetch();
        string str();
        void clear();
    };

protected:
    static string expandColumns_(list<Column> columns);
    static string expandTables_(list<Table> tables);
    static string expandConditions_(list<Condition> conditions);
    static string expandJoin_(list<Join> join);
    static string expandOrderBy_(list<string> columns, bool asc);

};

#endif /* DBHELPERS_H_ */
```

Listagem B.7: LoggerInterface.h

```
#ifndef LOGGERINTERFACE_H_
#define LOGGERINTERFACE_H_

#include <string>

using namespace std;

typedef enum {
    INFO = 1,
    TRACE = 2,
    WARNING = 4,
    ERROR = 8,
    ALL = 15

} LogLevel;

/** Logger interface definition. Interface that all logger mechanisms must
 * implement to be used at the predictor
 */
class LoggerInterface
{
protected:
    int logLevel_ = ALL; /**< Log Level */

public:
    /**
     * Destructor
     */
    virtual ~LoggerInterface() {};

    /**
     * Logs a info message.
     * @param text the message to be logged
     */
    virtual void info(const char* text) = 0;

    /**
     * Logs a info message.
     * @param text the message to be logged
     */
    virtual void info(string text) = 0;

    /**
     * Logs a warning message.
     * @param text the message to be logged
     */
    virtual void warning(const char* text) = 0;

    /**
     * Logs a warning message.
     * @param text the message to be logged
     */
    virtual void warning(string text) = 0;

    /**
     * Logs an error message.
     * @param text the message to be logged
     */
    virtual void error(const char* text) = 0;
```

```
    /**
     * Logs an error message.
     * @param text the message to be logged
     */
    virtual void error(string text) = 0;

    /**
     * Logs a trace message.
     * @param text the message to be logged
     */
    virtual void trace(const char* text) = 0;

    /**
     * Logs a trace message.
     * @param text the message to be logged
     */
    virtual void trace(string text) = 0;

    /**
     * Set the level of loggin
     * @param level The level of loggin
     */
    void logLEvel(int level) {
        logLevel_ = level;
    }

    /**
     * Return the current log level
     */
    int logLevel() const {
        return logLevel_;
    }
};

#endif /* LOGGERINTERFACE_H_ */
```

Listagem B.8: path.h

```
#ifndef PATH_H_
#define PATH_H_

#include <string>

using namespace std;

/**
 * Collection of utility functions
 */
namespace Utils
{
/**
 * Creath a directory with all parents, if needed
 *
 * @param path The absolute or relative path to be created
 */
void createPath(string path);

/**
 * Creath a directory with all parents, if needed
 *
 * @param path The absolute or relative path to be created
```

```
 */
void createPath(const char* path);
}
#endif /* PATH_H_ */
```

Listagem B.9: Predictor.h

```
#ifndef ARIA_PREDICTOR_H_
#define ARIA_PREDICTOR_H_

#include <vector>
#include <functional>
#include <exception>

#include "utils/time.h"
#include "Aria.h"
#include "interfaces/LoggerInterface.h"
#include "interfaces/RobotInterface.h"
#include "models/tasks.h"
#include "PredictorConfig.h"

using namespace Kompex;

class no_such_prediction_func : public exception {
protected:
    int pos_;

public:
    no_such_prediction_func(int pos): pos_(pos) {}

    virtual const char* what() const throw() {
        string response = "No function to call at index " + to_string(pos_);
        return response.c_str();
    }
};

/**
 * Implementation of a simple predictor example
 */
class Predictor {
protected:
    // runtime data
    double distanceToGoal_; /**< Distance to goal provided by the planned path */
    double initialDistanceToGoal_; /**< Initial distance to goal provided by the planned path */
    int timeToGoal_; /**< Time to goal provided by the planned path */
    int currentId; /**< The current task id at the database */
    int result_; /**< Representation of the result of the prediction; -1 = not set; 0 = fail; 1 = success */
    int limitDb_; /**< Limits the DB queries results */

protected:
    typedef std::function<bool (Predictor*, double)> predictionFunc;
    std::vector<predictionFunc> predictionFunctions_;
    PredictorConfig* config_; /**< Pointer to the PredictorConfig object */
    RobotInterface* robot_; /**< Pointer to the RobotInterface object */
    LoggerInterface* logger_; /**< Pointer to the object that implements the LoggerInterface */
    TaskModel* tasksModel_; /**< Pointer to the TasksModel object */
    string cfgFile_; /**< Path to the file containig the parameters for the execution */

public:
    /**
     * Creates a database connection, the models and initializes the attributes
```

```
 *
 * @param robot Pointer to the robot instance
 * @param logger Pointer to the logger instance
 * @param config Pointr to a config object
 */
Predictor(RobotInterface* robot, LoggerInterface* logger, PredictorConfig *config);

/**
 * Deletes all the objects instantiated by the class.
 */
~Predictor();

/**
 * Prepare the predictor to run a new task
 *
 * @param config Pointr to a config object
 *
 * @return The current task Id
 */
int prepareNewTask(PredictorConfig *config);

/**
 * TODO: Document
 */
bool invalidPredictionFunction(double pos) {
    throw new no_such_prediction_func(pos);
}

/**
 * TODO: Document
 */
bool runPredictionQ(int n, double fracDeadline);

/**
 * Runs the prediction algorithm for 1/5 of the deadline
 *
 * @return The prediction result
 */
bool runPredictionQ1(double fracDeadline);

/**
 * Runs the prediction algorithm for 2/5 of the deadline
 *
 * @return The prediction result
 */
bool runPredictionQ2(double fracDeadline);

/**
 * Runs the prediction algorithm for 3/5 of the deadline
 *
 * @return The prediction result
 */
bool runPredictionQ3(double fracDeadline);

/**
 * Updates the distance at the n/5 part of the deadline
 *
 * @param n the nth part of the deadline to be updated
 */
void updateQ(int n);
```

```
    /**
     * Returns the time in seconds to the deadline
     *
     * @return TODO: Document
     */
    double timeToDeadline();

    /**
     * TODO: Document
     */
    void registerEndTask(bool training);
};

#endif /* ARIA_PREDICTOR_H_ */
```

Listagem B.10: PredictorConfig.h

```
#ifndef PREDICTOR_CONFIG_H_
#define PREDICTOR_CONFIG_H_

#include <string>

using namespace std;

/**
 * Class to parse a configuration file and store its values
 */
class PredictorConfig
{
protected:
    string deadline_; /**< ISO timestamp representing the Deadline */
    int limitDb_; /**< limits the number of results at DB query */
    int maxVel_; /**< Maximum robot velocity in meters/second */
    string databaseName_; /**< Database to connect to */

public:
    /**
     * Reads the configuration file and store its values
     *
     * @param filename Name of the file to be read
     */
    PredictorConfig(const string& fileName);

    /**
     * Return the task deadline
     *
     * @return The deadline in a formated string
     */
    string deadline() const;

    /**
     * Returns the maximum velocity, in cm/s the robot can reach
     *
     * @return The maxium velocity in cm/s
     */
    int maxVel() const;

    /**
     * Returns the database name the predictor should connect to
     *
     * @returns The name of the database (with relative of absolute path)
```

```
     */
    string databaseName() const;

    /**
     * @returns The value to limit de DB query
     */
    int limitDb() const;

protected:

    /**
     * Parse the config file and sets the class variables
     *
     * @param fileName The path of the file to be used
     */
    void parseConfigFile(const string& fileName);
};

#endif /* PREDICTOR_CONFIG_H_ */
```

Listagem B.11: RobotInterface.h

```
#ifndef ROBOT_INTERFACE_H_
#define ROBOT_INTERFACE_H_

/** Robot interface definition. Robot interface that all robot proxies must
 * implement to be used at the predictor.
 */
class RobotInterface
{
public:
    /**
     * Destructor
     */
    virtual ~RobotInterface(){};

    /**
     * Executes a routine to prepare the robot for a new task
     */
    virtual void init() = 0;

    /**
     * Returns the current estimated distance to goal
     *
     * @return The estimated distance to goal in Meters with one decimal point
     */
    virtual double distanceToGoal() = 0;

    /**
     * Returns the current estimated time to reach the goal
     *
     * @return The ETA to goal in seconds
     */
    virtual double timeToGoal() = 0;

    /**
     * Returns the distance walked in the current task
     *
     * @return The total distace walked in Milimeters with one decimal point
     */
    virtual double distanceWalked() = 0;
```

```
    /**
     * Return the id of the current tasks
     *
     * @return The id of the current task
     */
    virtual int currentTaskId() = 0;
};

#endif /* ROBOT_INTERFACE_H_ */
```

Listagem B.12: tasks.h

```
#ifndef TASKS_H_
#define TASKS_H_

#include <models/tables/tbTasks.h>
#include <utils/dbHelpers.h>
#include <list>
#include <string>

using namespace std;

typedef TbTasks::TaskData TaskData;
typedef DbHelpers::Column Column;
typedef DbHelpers::Condition Condition;
typedef DbHelpers::ResultSet ResultSet;

/**
 * Class responsible to deal with the database requests regarding the tasks data.
 */
class TaskModel {
public:
    /**
     * Initializes the [statement](@ref stmt_) object and creates the table 'tasks'
     */
    TaskModel(LoggerInterface *logger);

    /**
     * Insert an entry at the 'tasks' table.
     *
     * @param data The information that will be inserted in th database
     * @return The status of the transaction
     */
    int insert(TaskData data);

    /**
     * Update an entry at the 'tasks' table.
     *
     * @param data The information that will be updated in th database
     * @param conditions The condition(s) to filter the affected rows
     * @return The status of the transaction
     */
    int update(TaskData data, list<Condition> conditions);
    int update(TaskData data, Condition condition);


    /**
     * Update an entry at the 'tasks' table.
     *
     * @param n the n'nth art of the deadline
```

```
     * @param distance The distance walked at n/5 of the deadline
     * @return The status of the transaction
     */
    int updateCurrentTaskQ(int n, double distance);

    /**
     * TODO: Document
     */
    ResultSet select(list<Column> fields, list<Condition> conditions = list<Condition
        ↪ >());
    ResultSet select(list<Column> fields, Condition condition);

    /**
     * TODO: Document
     */
    int currentId();

    /**
     * TODO: Document
     */
    void updateCurrentId_();

    /**
     * TODO: Document
     */
    int updateCurrentTaskId();

    /**
     * TODO: Document
     */
    double avgDistQ(int n, int limit=0);

    /**
     * TODO: Document
     */
    double avgSpeedQ(int n, double time, int limit=0);

    /**
     * TODO: Document
     */
    double avgSpeedQ4(double spentTime, int limit=0);

    /**
     * TODO: Document
     */
    double minDistQ(int n, int limit=0);

protected:
    string tbDefinition_; /**< 'tasks' table definition */
    int currentId_; /**< TODO: Document */
    int currentTaskId_; /**< TODO: Document */
    LoggerInterface *logger_;

protected:
    /**
     * Creates the 'tasks' table
     *
     * Creates the table according to tbDefinition_
     */
    void createTable();
};
```

```
#endif /* TASK_H_ */
```

Listagem B.13: tbBaseline.h

```
/*
 * tbBaseline.h
 *
 * Created on: 05/04/2014
 * Author: rafael
 */

/**
 * TODO: Document whole file
 */

#ifndef TBBASELINE_H_
#define TBBASELINE_H_

#include <map>
#include <string>

class TbBaseline {
public:
    /**
     * Struct that represents a row in the 'baseline' table
     */
    class BaselineData
    {
    public:
        BaselineData(): id(-1), taskId(-1), startTime(""), deadline(""),
            endTime(""), distanceToGoal(-1), distanceWalkedQ1(-1),
            predictorAnswer(-1), success(-1) {}
        int id;
        int taskId;
        std::string startTime;
        std::string deadline;
        std::string endTime;
        double distanceToGoal;
        double distanceWalkedQ1;
        int predictorAnswer;
        int success;
    public:
        std::map<std::string, std::string> toStringMap();
    };
    static std::string definitionTbBaseline;

};

#endif /* TBBASELINE_H_ */
```

Listagem B.14: tbTasks.h

```
#ifndef TBTASKs_H_
#define TBTASKs_H_

#include <map>
#include <string>

using namespace std;
```

```
class TbTasks
{
public:
    /**
     * Struct that represents a row in the 'tasks' table
     */
    class TaskData
    {
    public:
        typedef map<string, string> dataMap;
        int id; /**< Unique indentification in the table (PK) */
        int taskId; /**< Id of the task */
        string startTime; /**< Start time of the task */
        string deadline; /**< The task deadline, in seconds */
        double distQ1; /**< The distance walked at 1/5 of the deadline */
        double distQ2; /**< The distance walked at 2/5 of the deadline */
        double distQ3; /**< The distance walked at 3/5 of the deadline */
        double distQ4; /**< The distance walked at task completion */
        string endTime; /**< The time the task ended */
        int predictorAnswer; /**< The resposse of the predictor*/
        int success; /**< Indicate the success or failure of the task */
    public:

        TaskData(): id(-1), taskId (-1), startTime(""), deadline(""),
        distQ1(-1), distQ2(-1), distQ3(-1), distQ4(-1), endTime(""),
        predictorAnswer(-1), success(-1) {}
        /**
         * TODO: Document
         */
        dataMap toStringMap();
    };
    static string definitionTbTasks; /**< Declaration of the 'tasks' table structure */
};

#endif /* TBTASKs_H_ */
```

Listagem B.15: time.h

```
/*
 * time.h
 *
 * Created on: 24/06/2013
 * Author: rafael
 */

#ifndef TIME_H_
#define TIME_H_

#include <string>
#include <ctime>

using namespace std;

namespace Utils
{
const time_t when = time(NULL);
const struct tm utc = *gmtime(&when);
const struct tm lcl = *localtime(&when);

/**
```

```
 * TODO: Document
 */
const int gmt_hour_offset = lcl.tm_hour - utc.tm_hour;

/**
 * TODO: Document
 */
const int gmt_min_offset = lcl.tm_min - utc.tm_min;

/**
 * Returns a timestamp in the YYYY-MM-DDTHH:MM:SSTZ format (ISO 8601)
 *
 * @param gmt Tell if the generated timestamp is in GMT
 * @param date The to be converted to a timestamp
 *
 * @return The formated string
 */
string isoTimestamp(bool gmt=false, time_t date=time(NULL));

/**
 * Returns de difference, in seconds, from two timestamps
 *
 * @return The difference between two timestamps, in seconds
 */
double timeDiff(string beggining, string end);

/**
 * Converts a iso timestamp to a time_t format
 *
 * @param datetime The string to be converted
 *
 * @return A time_t structure representing the string
 */
time_t stringToDateTime(string datetime);

/**
 * TODO: Document
 */
void sleep_for(int usec);
}
#endif /* TIME_H_ */
```

B.2 IMPLEMENTAÇÕES

Listagem B.16: AriaLogger.cpp

```
#include "AriaLogger.h"
#include "Aria.h"
#include <string>
#include <iostream>

using namespace std;

AriaLogger::AriaLogger(LogLevel level) {
    this->logLevel_ = level;
}

void AriaLogger::log(string type, string text) {
    type.append(" ");
```

```
    type.append(text);
    ArLog::logPlain(ArLog::Terse, type.c_str());
}

void AriaLogger::info(const char* text) {
    string text_(text);
    this->info(text_);
}

void AriaLogger::info(string text) {
    if (this->logLevel_ & INFO) {
        string type = "<INFO>";
        this->log(type, text);
    }
}

void AriaLogger::error(const char* text) {
    string text_(text);
    this->error(text_);
}

void AriaLogger::error(string text) {
    if (this->logLevel_ & ERROR) {
        const char * type = "<ERROR>";
        this->log(type, text);
    }
}

void AriaLogger::trace(const char* text) {
    string text_(text);
    this->trace(text_);
}

void AriaLogger::trace(string text) {
    if (this->logLevel_ & TRACE) {
        string type = "<TRACE>";
        this->log(type, text);
    }
}

void AriaLogger::warning(const char* text) {
    string text_(text);
    this->warning(text_);
}

void AriaLogger::warning(string text) {
    if (this->logLevel_ & WARNING) {
        string type = "<WARN>";
        this->log(type, text);
    }
}
```

Listagem B.17: AriaRobot.cpp

```
#include "Aria.h"
#include "ArPredictorTask.h"

AriaRobot::AriaRobot(
        ArRobot * robot, ArPathPlanningTask * planning) :
        robot_(robot), planning_(planning), lastDistanceRead_(0),
        lastWalkedDistRead_(0) {}
```

```
void AriaRobot::init() {
    lastDistanceRead_ = distanceToGoal();
    lastWalkedDistRead_ = robot_->getOdometerDistance();
}

double AriaRobot::distanceToGoal() {
    planning_->lock();
    double dist = planning_->estimateDistanceToGoal();
    planning_->unlock();
    return dist;
}

double AriaRobot::timeToGoal() {
    planning_->lock();
    double time = planning_->estimateTimeToGoal();
    planning_->unlock();
    return time;
}

double AriaRobot::distanceWalked() {
    robot_->lock();
    double dist = robot_->getOdometerDistance() - lastWalkedDistRead_;
    robot_->unlock();
    return dist;
}

bool AriaRobot::goalAchived() {
    return planning_->computeProgress() > 0.999;
}

int AriaRobot::currentTaskId() {
    int id = -1;
    ArPose pos = planning_->getCurrentGoal();
    ArPose elevador(6674, 20817, 91.726953);
    ArPose lapesd(-6781, -87, -177.638241);
    if (pos == elevador) {
        id = 1;
    } else if (pos == lapesd) {
        id = 0;
    }
    return id;
}
```

Listagem B.18: arnlServer.cpp

```
/*
 Adept MobileRobots Advanced Robotics Navigation and Localization (ARNL)
 Version 1.7.5.1

 Copyright (C) 2004, 2005 ActivMedia Robotics LLC
 Copyright (C) 2006, 2007, 2008, 2009 MobileRobots Inc.
 Copyright (C) 2010, 2011, 2012 Adept Technology, Inc.

 All Rights Reserved.

 Adept MobileRobots does not make any representations about the
 suitability of this software for any purpose. It is provided "as is"
 without express or implied warranty.

 The license for this software is distributed as LICENSE.txt in the top
```

```
 level directory.

 robots@mobilerobots.com
 Adept MobileRobots
 10 Columbia Drive
 Amherst, NH 03031
 800−639−9481

 */

/**
 This is an example of using ARNL and ArNetworking to provide a
 server program that remote clients, such as MobileEyes, can connect
 to, and which includes most of the features available in ARNL.

 You can make a copy of this program's source file in the 'examples' directory
 and use it as the basis for your own program.

 MobileEyes sends requests to this server program to go to goals,
 get the current map, set configuration parameters, manually relocalize,
 request the current state of the robot and sensors, etc. This example server
 includes almost all features provided by ARNL (or SONARNL or MOGS), including
 localization, path planning and navigation to a map goal or abritrary point,
 sharing information with other servers peer−to−peer or through
 a "central server", using IR and bumper sensors, stopping the robot
 if localization fails, global replanning if a path is not followable,
 use of special SICK laser reflector beacons, as well as various networking
 services for MobileEyes and other clients, such as diagnostic visualizations,
 access to configuration parameters, access to camera control and
 video images (if SAVServer is running), special debugging commands
 ("custom commands"), file uploading/downloading and gathering raw laser
 scans using Mapper3, and safe and unsafe teleoperation.
 */

#include "Aria.h"
#include "ArNetworking.h"
#include "Arnl.h"

#include "ArLocalizationTask.h"
#include "ArDocking.h"

//Predictor includes
#include "ArPredictorTask.h"
#include "Predictor.h"
#include "AriaRobot.h"
#include "AriaLogger.h"

#include "boost/filesystem/path.hpp"
#include "boost/filesystem/operations.hpp"

#include "utils/dbHelpers.h"

namespace fs = boost::filesystem;

void logOptions(const char *progname)
{
    ArLog::log(ArLog::Normal, "Usage: %s [options]\n", progname);
    ArLog::log(ArLog::Normal,
                "[options] are any program options listed below, or any ARNL
                    ↪ configuration");
```

```
    ArLog::log(ArLog::Normal, "parameters as -name <value>, see params/arnl.p for list
        ↪ .");
    ArLog::log(ArLog::Normal, "For example, -map <map file>.");
    Aria::logOptions();
}

class PredictorCallbacks
{
protected:
    ArPathPlanningTask* pathPlanning_;
    ArPredictorTask* predictor_;
    ArPose lastGoal_;
public:
    PredictorCallbacks(ArPathPlanningTask* pathPlanning, ArPredictorTask* predictor):
        pathPlanning_(pathPlanning), predictor_(predictor){};

    void negativePrediction() {
        ArLog::log(ArLog::Terse, "negative prediction, going home");
        string str = (pathPlanning_->pathPlanToGoal("dock")) ? "goal exist" : "goal doesn
            ↪ 't exist";
        ArLog::log(ArLog::Terse, str.c_str());
    }

    void taskCompleted(ArPose pose) {
        ArLog::log(ArLog::Terse, "<INFO> Updating end task\n");
        predictor_->updateOnGoalCompleted();
    }

    void taskFailed(ArPose pose) {
        ArLog::log(ArLog::Terse, "<INFO> Task failed, stopping predictor\n");
        predictor_->stopRunning();
    }

    void stateChanged() {
        ArPathPlanningTask::PathPlanningState state = pathPlanning_->getState();
//  char* txt;
//  pathPlanning_->getStatusString(txt, 300);
//  ArLog::log(ArLog::Terse, txt);
        if (state == ArPathPlanningTask::MOVING_TO_GOAL) {
            if (pathPlanning_->getCurrentGoal() != lastGoal_) {
                if (predictor_->getRunningWithLock()) {
                    predictor_->stopRunning();
                }
                predictor_->runAsync();
            }
            lastGoal_ = pathPlanning_->getCurrentGoal();
        }
    }

    void taskCanceled(ArPose pose) {
        double x = pose.getX();
        double y = pose.getY();
        stringstream ss;
        ss << "<INFO> Canceling task to pose (" << x << ", " << y << ")\n";
        ArLog::log(ArLog::Terse, ss.str().c_str());
        if (predictor_->getRunningWithLock()) {
            predictor_->stopRunning();
        }
    }

};
```

```
int main(int argc, char **argv)
{
    // Initialize Aria and Arnl global information
    Aria::init();
    Arnl::init();

    // The robot object
    ArRobot robot;

    // Parse the command line arguments.
    ArArgumentParser parser(&argc, argv);

    std::string configFile("./config/default.cfg");
    const char *cfgFile = NULL;
    if (parser.checkParameterArgumentString("c", &cfgFile) && cfgFile != NULL) {
        configFile = string(cfgFile);
    }

    fs::path full_path_cfg = fs::system_complete(fs::path(configFile.c_str()));

    if (!fs::exists(full_path_cfg)) {
        std::string log = "Config file " + full_path_cfg.native() +
                "" not found! Exiting...";
        ArLog::log(ArLog::Normal, log.c_str());
        return 0;
    } else {
        std::string log = "Config file " + full_path_cfg.native() + " found!";
        ArLog::log(ArLog::Normal, log.c_str());
    }

    bool training = parser.checkArgument("training");

    ArLog::log(ArLog::Terse, (training) ? "Executing in training mode" :
            "Executing in normal mode");

    // Set up our simpleConnector, to connect to the robot and laser
    //ArSimpleConnector simpleConnector(&parser);
    ArRobotConnector robotConnector(&parser, &robot);

    // Connect to the robot
    if (!robotConnector.connectRobot()) {
        ArLog::log(ArLog::Normal, "Error: Could not connect to robot... exiting");
        Aria::exit(3);
    }

    // Set up where we'll look for files. Arnl::init() set Aria's default
    // directory to Arnl's default directory; addDirectories() appends this
    // "examples" directory.
    char fileDir[1024];
    ArUtil::addDirectories(fileDir, sizeof(fileDir), Aria::getDirectory(), "examples");

    // To direct log messages to a file, or to change the log level, use these calls:
    //ArLog::init(ArLog::File, ArLog::Normal, "log.txt", true, true);
    //ArLog::init(ArLog::File, ArLog::Verbose);

    // Add a section to the configuration to change ArLog parameters
// ArLog::addToConfig(Aria::getConfig());
    struct tm t;
    ArUtil::localtime(&t);
    stringstream ss;
```

```
ss << "log/predictor_" << t.tm_mday << "." << t.tm_mon+1 << "." << t.tm_year + 1900
        << "-" << t.tm_hour << "-" << t.tm_min << "-" << t.tm_sec << ".out";
ArLog::init(ArLog::File, ArLog::Terse, ss.str().c_str(), true, true, true);

// Our networking server
ArServerBase server;

// Set up our simpleOpener, used to set up the networking server
ArServerSimpleOpener simpleOpener(&parser);

// the laser connector
ArLaserConnector laserConnector(&parser, &robot, &robotConnector);

// Tell the laser connector to always connect the first laser since
// this program always requires a laser.
parser.addDefaultArgument("-connectLaser");

// Load default arguments for this computer (from /etc/Aria.args, environment
// variables, and other places)
parser.loadDefaultArguments();

// Parse arguments
if (!Aria::parseArgs() || !parser.checkHelpAndWarnUnparsed()) {
    logOptions(argv[0]);
    Aria::exit(1);
}

// This causes Aria::exit(9) to be called if the robot unexpectedly
// disconnects
ArGlobalFunctor1<int> shutdownFunctor(&Aria::exit, 9);
robot.addDisconnectOnErrorCB(&shutdownFunctor);

// Create an ArSonarDevice object (ArRangeDevice subclass) and
// connect it to the robot.
ArSonarDevice sonarDev;
robot.addRangeDevice(&sonarDev);

// This object will allow robot's movement parameters to be changed through
// a Robot Configuration section in the ArConfig global configuration facility.
ArRobotConfig robotConfig(&robot);

// Start the robot thread.
robot.runAsync(true);

// connect the laser(s) if it was requested, this adds them to the
// robot too, and starts them running in their own threads
if (!laserConnector.connectLasers()) {
    ArLog::log(ArLog::Normal, "Could not connect to all lasers... exiting\n");
    Aria::exit(2);
}

// find the laser we should use for localization and/or mapping,
// which will be the first laser
robot.lock();
ArLaser *firstLaser = robot.findLaser(1);
if (firstLaser == NULL || !firstLaser->isConnected()) {
    ArLog::log(ArLog::Normal,
                "Did not have laser 1 or it is not connected, cannot start
                    ↪ localization and/or mapping... exiting");
    Aria::exit(2);
}
```

```
robot.unlock();

/* Create and set up map object */

// Set up the map object, this will look for files in the examples
// directory (unless the file name starts with a /, \, or .
// You can take out the 'fileDir' argument to look in the program's current directory
// instead.
// When a configuration file is loaded into ArConfig later, if it specifies a
// map file, then that file will be loaded as the map.
ArMap map(fileDir);
// set it up to ignore empty file names (otherwise if a configuration omits
// the map file, the whole configuration change will fail)
map.setIgnoreEmptyFileName(true);
// ignore the case, so that if someone is using MobileEyes or
// MobilePlanner from Windows and changes the case on a map name,
// it will still work.
map.setIgnoreCase(true);

/* Create localization and path planning threads */

ArPathPlanningTask pathTask(&robot, &sonarDev, &map);

ArLog::log(ArLog::Normal, "Creating laser localization task");
// Laser Monte-Carlo Localization
ArLocalizationTask locTask(&robot, firstLaser, &map);

// Set some options on each laser that the laser connector
// connected to.
std::map<int, ArLaser *>::iterator laserIt;
for (laserIt = robot.getLaserMap()->begin(); laserIt != robot.getLaserMap()->end();
      ↪ laserIt++) {
    int laserNum = (*laserIt).first;
    ArLaser *laser = (*laserIt).second;

    // Skip lasers that aren't connected
    if (!laser->isConnected())
        continue;

    // add the disconnectOnError CB to shut things down if the laser
    // connection is lost
    laser->addDisconnectOnErrorCB(&shutdownFunctor);
    // set the number of cumulative readings the laser will take
    laser->setCumulativeBufferSize(200);
    // add the lasers to the path planning task
    pathTask.addRangeDevice(laser, ArPathPlanningTask::BOTH);
    // set the cumulative clean offset (so that they don't all fire at once)
    laser->setCumulativeCleanOffset(laserNum * 100);
    // reset the cumulative clean time (to make the new offset take effect)
    laser->resetLastCumulativeCleanTime();

    // Add the packet count to the Aria info strings (It will be included in
    // MobileEyes custom details so you can monitor whether the laser data is
    // being received correctly)
    std::string laserPacketCountName;
    laserPacketCountName = laser->getName();
    laserPacketCountName += " Packet Count";
    Aria::getInfoGroup()->addStringInt(laserPacketCountName.c_str(), 10,
                    new ArRetFunctorC<int, ArLaser>(laser, &ArLaser::getReadingCount))
                        ↪ ;
}
```

```
/* Start the server */

// Open the networking server
if (!simpleOpener.open(&server, fileDir, 240)) {
    ArLog::log(ArLog::Normal, "Error: Could not open server.");
    exit(2);
}

// Add range device which uses forbidden regions given in the map to give virtual
// range device readings to ARNL. (so it avoids obstacles
// detected by this device)
ArForbiddenRangeDevice forbidden(&map);
robot.addRangeDevice(&forbidden);
pathTask.addRangeDevice(&forbidden, ArPathPlanningTask::CURRENT);

robot.unlock();

// Action to slow down robot when localization score drops but not lost.
ArActionSlowDownWhenNotCertain actionSlowDown(&locTask);
pathTask.getPathPlanActionGroup()->addAction(&actionSlowDown, 140);

// Action to stop the robot when localization is "lost" (score too low)
ArActionLost actionLostPath(&locTask, &pathTask);
pathTask.getPathPlanActionGroup()->addAction(&actionLostPath, 150);

// Arnl uses this object when it must replan its path because its
// path is completely blocked. It will use an older history of sensor
// readings to replan this new path. This should not be used with SONARNL
// since sonar readings are not accurate enough and may prevent the robot
// from planning through space that is actually clear.
ArGlobalReplanningRangeDevice replanDev(&pathTask);

// Service to provide drawings of data in the map display :
ArServerInfoDrawings drawings(&server);
drawings.addRobotsRangeDevices(&robot);
drawings.addRangeDevice(&replanDev);

/* Draw a box around the local path planning area use this
 (You can enable this particular drawing from custom commands
 which is set up down below in ArServerInfoPath) */
ArDrawingData drawingDataP("polyLine", ArColor(200, 200, 200), 1, 75);
ArFunctor2C<ArPathPlanningTask, ArServerClient *, ArNetPacket *> drawingFunctorP(&
    ↪ pathTask,
            &ArPathPlanningTask::drawSearchRectangle);
drawings.addDrawing(&drawingDataP, "Local Plan Area", &drawingFunctorP);

/* Show the sample points used by MCL */
ArDrawingData drawingDataL("polyDots", ArColor(0, 255, 0), 100, 75);
ArFunctor2C<ArLocalizationTask, ArServerClient *, ArNetPacket *> drawingFunctorL(&
    ↪ locTask,
            &ArLocalizationTask::drawRangePoints);
drawings.addDrawing(&drawingDataL, "Localization Points", &drawingFunctorL);

// "Custom" commands. You can add your own custom commands here, they will
// be available in MobileEyes' custom commands (enable in the toolbar or
// access through Robot Tools)
ArServerHandlerCommands commands(&server);

// These provide various kinds of information to the client:
ArServerInfoRobot serverInfoRobot(&server, &robot);
```

```
  ArServerInfoSensor serverInfoSensor(&server, &robot);
  ArServerInfoPath serverInfoPath(&server, &robot, &pathTask);
  serverInfoPath.addSearchRectangleDrawing(&drawings);
  serverInfoPath.addControlCommands(&commands);

  // Provides localization info and allows the client (MobileEyes) to relocalize at a given
  // pose:
  ArServerInfoLocalization serverInfoLocalization(&server, &robot, &locTask);
  ArServerHandlerLocalization serverLocHandler(&server, &robot, &locTask);

  // If you're using MobileSim, ArServerHandlerLocalization sends it a command
  // to move the robot's true pose if you manually do a localization through
  // MobileEyes. To disable that behavior, use this constructor call instead:
  // ArServerHandlerLocalization serverLocHandler(&server, &robot, true, false);
  // The fifth argument determines whether to send the command to MobileSim.

  // Provide the map to the client (and related controls):
  ArServerHandlerMap serverMap(&server, &map);

  // These objects add some simple (custom) commands to 'commands' for testing and debugging:
  ArServerSimpleComUC uCCommands(&commands, &robot); // Send any command to the
      ↪ microcontroller
  ArServerSimpleComMovementLogging loggingCommands(&commands, &robot); // configure
      ↪ logging
  ArServerSimpleComLogRobotConfig configCommands(&commands, &robot); // trigger logging
      ↪ of the robot config parameters
// ArServerSimpleServerCommands serverCommands(&commands, &server); // monitor networking
    ↪ behavior (track packets sent etc.)

  // service that allows the client to monitor the communication link status
  // between the robot and the client.
  //
  ArServerHandlerCommMonitor handlerCommMonitor(&server);

  // service that allows client to change configuration parameters in ArConfig
  ArServerHandlerConfig handlerConfig(&server, Aria::getConfig(),
                  Arnl::getTypicalDefaultParamFileName(), Aria::getDirectory());

  /* Set up the possible modes for remote control from a client such as
   * MobileEyes:
   */

  // Mode To go to a goal or other specific point:
  ArServerModeGoto modeGoto(&server, &robot, &pathTask, &map, locTask.getRobotHome(),
                  locTask.getRobotHomeCallback());

  // Mode To stop and remain stopped:
  ArServerModeStop modeStop(&server, &robot);

  // Cause the sonar to turn off automatically
  // when the robot is stopped, and turn it back on when commands to move
  // are sent. (Note, if using SONARNL to localize, then don't do this
  // since localization may get lost)
  ArSonarAutoDisabler sonarAutoDisabler(&robot);

  // Teleoperation modes To drive by keyboard, joystick, etc:
  ArServerModeRatioDrive modeRatioDrive(&server, &robot);
// ArServerModeDrive modeDrive(&server, &robot); // Older mode for compatability

  // Prevent normal teleoperation driving if localization is lost using
  // a high-priority action, which enables itself when the particular mode is
```

```
// active.
// (You have to enter unsafe drive mode to drive when lost.)
ArActionLost actionLostRatioDrive(&locTask, &pathTask, &modeRatioDrive);
modeRatioDrive.getActionGroup()->addAction(&actionLostRatioDrive, 110);

// Add drive mode section to the configuration, and also some custom (simple) commands:
modeRatioDrive.addToConfig(Aria::getConfig(), "Teleop settings");
modeRatioDrive.addControlCommands(&commands);

// Wander mode (also prevent wandering if lost):
ArServerModeWander modeWander(&server, &robot);
ArActionLost actionLostWander(&locTask, &pathTask, &modeWander);
modeWander.getActionGroup()->addAction(&actionLostWander, 110);

// This provides a small table of interesting information for the client
// to display to the operator. You can add your own callbacks to show any
// data you want.
ArServerInfoStrings stringInfo(&server);
Aria::getInfoGroup()->addAddStringCallback(stringInfo.getAddStringFunctor());

// Provide a set of informational data (turn on in MobileEyes with
// View->Custom Details)

Aria::getInfoGroup()->addStringInt("Motor Packet Count", 10,
                new ArConstRetFunctorC<int, ArRobot>(&robot, &ArRobot::
                    ↪ getMotorPacCount));

Aria::getInfoGroup()->addStringDouble("Laser Localization Score", 8,
                new ArRetFunctorC<double, ArLocalizationTask>(&locTask,
                                &ArLocalizationTask::getLocalizationScore), "%.03f");
Aria::getInfoGroup()->addStringInt("Laser Loc Num Samples", 8,
                new ArRetFunctorC<int, ArLocalizationTask>(&locTask,
                                &ArLocalizationTask::getCurrentNumSamples), "%4d");

// Setup the dock if there is a docking system on board.
ArServerModeDock *modeDock = NULL;
modeDock = ArServerModeDock::createDock(&server, &robot, &locTask, &pathTask);
if (modeDock != NULL) {
    modeDock->checkDock();
    modeDock->addAsDefaultMode();
    modeDock->addToConfig(Aria::getConfig());
    modeDock->addControlCommands(&commands);
}

// Make Stop mode the default (If current mode deactivates without entering
// a new mode, then Stop Mode will be selected)
modeStop.addAsDefaultMode();

/* Services that allow the client to initiate scanning with the laser to
 create maps in Mapper3 (So not possible with SONARNL): */

ArServerHandlerMapping handlerMapping(&server, &robot, firstLaser, fileDir, "", true
    ↪ );

// make laser localization stop while mapping
handlerMapping.addMappingStartCallback(
                new ArFunctor1C<ArLocalizationTask, bool>(&locTask,
                                &ArLocalizationTask::setIdleFlag, true));

// and then make it start again when we're doine
handlerMapping.addMappingEndCallback(
```

```
                    new ArFunctor1C<ArLocalizationTask, bool>(&locTask,
                                &ArLocalizationTask::setIdleFlag, false));

    // Make it so our "lost" actions don't stop us while mapping
    handlerMapping.addMappingStartCallback(actionLostPath.getDisableCB());
    handlerMapping.addMappingStartCallback(actionLostRatioDrive.getDisableCB());
    handlerMapping.addMappingStartCallback(actionLostWander.getDisableCB());

    // And then let them make us stop as usual when done mapping
    handlerMapping.addMappingEndCallback(actionLostPath.getEnableCB());
    handlerMapping.addMappingEndCallback(actionLostRatioDrive.getEnableCB());
    handlerMapping.addMappingEndCallback(actionLostWander.getEnableCB());

    // don't let forbidden lines show up as obstacles while mapping
    // (they'll just interfere with driving while mapping, and localization is off anyway)
    handlerMapping.addMappingStartCallback(forbidden.getDisableCB());

    // let forbidden lines show up as obstacles again as usual after mapping
    handlerMapping.addMappingEndCallback(forbidden.getEnableCB());

    /*
     // If we are on a simulator, move the robot back to its starting position,
     // and reset its odometry.
     // This will allow localizeRobotAtHomeBlocking() below will (probably) work (it
     // tries current odometry (which will be 0,0,0) and all the map
     // home points.
     // (Ignored by a real robot)
     //robot.com(ArCommands::SIM_RESET);
     */

    // create a pose storage class, this will let the program keep track
    // of where the robot is between runs... after we try and restore
    // from this file it will start saving the robot's pose into the
    // file
    ArPoseStorage poseStorage(&robot);
    /// if we could restore the pose from then set the sim there (this
    /// won't do anything to the real robot)... if we couldn't restore
    /// the pose then just reset the position of the robot (which again
    /// won't do anything to the real robot)
    if (poseStorage.restorePose("robotPose"))
        serverLocHandler.setSimPose(robot.getPose());
    else
        robot.com(ArCommands::SIM_RESET);

    /* File transfer services: */

#ifdef WIN32
    // Not implemented for Windows yet.
    ArLog::log(ArLog::Normal, "Note, file upload/download services are not implemented
        ↪  for Windows; not enabling them.");
#else
    // This block will allow you to set up where you get and put files
    // to/from, just comment them out if you don't want this to happen
    // /*
    ArServerFileLister fileLister(&server, fileDir);
    ArServerFileToClient fileToClient(&server, fileDir);
    ArServerFileFromClient fileFromClient(&server, fileDir, "/tmp");
    ArServerDeleteFileOnServer deleteFileOnServer(&server, fileDir);
    // */
#endif
```

```
/* Video image streaming, and camera controls (Requires SAVserver or ACTS) */

// Forward any video if either ACTS or SAV server are running.
// You can find out more about SAV and ACTS on our website
// http://robots.activmedia.com. ACTS is for color tracking and is
// a seperate product. SAV just does software A/V transmitting and is
// free to all our customers. Just run ACTS or SAV server before you
// start this program and this class here will forward video from the
// server to the client.
ArHybridForwarderVideo videoForwarder(&server, "localhost", 7070);

// make a camera to use in case we have video. the camera collection collects
// multiple ptz cameras
ArPTZ *camera = NULL;
ArServerHandlerCamera *handlerCamera = NULL;
ArCameraCollection *cameraCollection = NULL;

// if we have video then set up a camera
if (videoForwarder.isForwardingVideo()) {

    cameraCollection = new ArCameraCollection();
    cameraCollection->addCamera("Cam1", "PTZ", "Camera", "PTZ");

    videoForwarder.setCameraName("Cam1");
    videoForwarder.addToCameraCollection(*cameraCollection);

    camera = new ArVCC4(&robot); //, invertedCamera, ArVCC4::COMM_UNKNOWN, true, true);
    // To use an RVision SEE camera instead:
    // camera = new ArRVisionPTZ(&robot);
    camera->init();

    handlerCamera = new ArServerHandlerCamera("Cam1", &server, &robot, camera,
                    cameraCollection);

    pathTask.addGoalFinishedCB(
                    new ArFunctorC<ArServerHandlerCamera>(handlerCamera,
                                    &ArServerHandlerCamera::
                                        ↪ cameraModeLookAtGoalClearGoal));
}

// After all of the cameras / videos have been created and added to the collection,
// then start the collection server.
//
if (cameraCollection != NULL) {
    new ArServerHandlerCameraCollection(&server, cameraCollection);
}

/* Load configuration values, map, and begin! */

// When parsing the configuration file, also look at the program's command line options
// from the command-line argument parser as well as the configuration file.
// (So you can use any argument on the command line, namely -map.)
Aria::getConfig()->useArgumentParser(&parser);

// Read in parameter files.
ArLog::log(ArLog::Normal, "Loading config file %s into ArConfig...",
                Arnl::getTypicalParamFileName());
if (!Aria::getConfig()->parseFile(Arnl::getTypicalParamFileName())) {
    ArLog::log(ArLog::Normal, "Trouble loading configuration file, exiting");
    Aria::exit(5);
}
```

```
// Warn about unknown params.
if (!simpleOpener.checkAndLog() || !parser.checkHelpAndWarnUnparsed()) {
    logOptions(argv[0]);
    Aria::exit(6);
}

// Warn if there is no map
if (map.getFileName() == NULL || strlen(map.getFileName()) <= 0) {
    ArLog::log(ArLog::Normal, "");
    ArLog::log(ArLog::Normal,
               "### No map file is set up, you can make a map with the
                    ↪ following procedure");
    ArLog::log(ArLog::Normal,
               " 0) You can find this information in README.txt or docs/
                    ↪ Mapping.txt");
    ArLog::log(ArLog::Normal, " 1) Connect to this server with MobileEyes");
    ArLog::log(ArLog::Normal, " 2) Go to Tools->Map Creation->Start Scan");
    ArLog::log(ArLog::Normal, " 3) Give the map a name and hit okay");
    ArLog::log(ArLog::Normal, " 4) Drive the robot around your space (see docs/
         ↪ Mapping.txt");
    ArLog::log(ArLog::Normal, " 5) Go to Tools->Map Creation->Stop Scan");
    ArLog::log(ArLog::Normal, " 6) Start up Mapper3");
    ArLog::log(ArLog::Normal, " 7) Go to File->Open on Robot");
    ArLog::log(ArLog::Normal, " 8) Select the .2d you created");
    ArLog::log(ArLog::Normal, " 9) Create a .map");
    ArLog::log(ArLog::Normal, " 10) Go to File->Save on Robot");
    ArLog::log(ArLog::Normal, " 11) In MobileEyes, go to Tools->Robot Config");
    ArLog::log(ArLog::Normal, " 12) Choose the Files section");
    ArLog::log(ArLog::Normal,
               " 13) Enter the path and name of your new .map file for the
                    ↪ value of the Map parameter.");
    ArLog::log(ArLog::Normal, " 14) Press OK and your new map should become the map
         ↪  used");
    ArLog::log(ArLog::Normal, "");
}

// Print a log message notifying user of the directory for map files
ArLog::log(ArLog::Normal, "");
ArLog::log(ArLog::Normal, "Directory for maps and file serving: %s", fileDir);

ArLog::log(ArLog::Normal, "See the ARNL README.txt for more information");
ArLog::log(ArLog::Normal, "");

// Do an initial localization of the robot. It tries all the home points
// in the map, as well as the robot's current odometric position, as possible
// places the robot is likely to be at startup. If successful, it will
// also save the position it found to be the best localized position as the
// "Home" position, which can be obtained from the localization task (and is
// used by the "Go to home" network request).
locTask.localizeRobotAtHomeBlocking();

// Start the networking server's thread
server.runAsync();

// PREDICTOR BLOCK

ArPredictorTask predictorTask(&robot, &pathTask, configFile);

PredictorCallbacks callbacks(&pathTask, &predictorTask);
```

```
    ArFunctor1C<PredictorCallbacks, ArPose> goalDone(callbacks, &PredictorCallbacks::
        ↪ taskCompleted);
    ArFunctor1C<PredictorCallbacks, ArPose> goalFailed(callbacks, &PredictorCallbacks
        ↪ ::taskFailed);
    ArFunctor1C<PredictorCallbacks, ArPose> goalInterrupted(callbacks, &
        ↪ PredictorCallbacks::taskCanceled);
    ArFunctorC<PredictorCallbacks> statusChanged(callbacks, &PredictorCallbacks::
        ↪ stateChanged);

    ArFunctorC<PredictorCallbacks> predictionFailed(callbacks, &PredictorCallbacks::
        ↪ negativePrediction);

    pathTask.addGoalDoneCB(&goalDone);
    pathTask.addGoalFailedCB(&goalFailed);
    pathTask.addStateChangeCB(&statusChanged);
    pathTask.addGoalInterruptedCB(&goalInterrupted);

    predictorTask.failAction(&predictionFailed);

    predictorTask.setTrainig(training);

    // END PREDICTOR BLOCK

    // Add a key handler so that you can exit by pressing
    // escape. Note that this key handler, however, prevents this program from
    // running in the background (e.g. as a system daemon or run from
    // the shell with "&") -- it will lock up trying to read the keys;
    // remove this if you wish to be able to run this program in the background.
    ArKeyHandler *keyHandler;
    if ((keyHandler = Aria::getKeyHandler()) == NULL) {
        keyHandler = new ArKeyHandler;
        Aria::setKeyHandler(keyHandler);
        robot.lock();
        robot.attachKeyHandler(keyHandler);
        robot.unlock();
        puts("Server running. To exit, press escape.");
    }

    // Enable the motors and wait until the robot exits (disconnection, etc.) or this program is
    // canceled.
    robot.enableMotors();
    robot.waitForRunExit();
    ArLog::log(ArLog::Terse, "robo saiu");
    Aria::exit(0);
}
```

Listagem B.19: ArPredictorTask.cpp

```
#include "Aria.h"
#include "Predictor.h"
#include "ArPredictorTask.h"
#include "AriaLogger.h"
#include "utils/time.h"
#include <thread>

ArPredictorTask::ArPredictorTask(ArRobot *robot, ArPathPlanningTask *planning,
        string cfgFile) : robot_(robot, planning), training_(false) {
    failAction_ = NULL;
    configFilePath_ = cfgFile;
    config_ = new PredictorConfig(cfgFile);
    logger_ = new AriaLogger();
```

```
    DbHelpers::openDb(config_->databaseName(), logger_);
    baseline_ = new BaselinePredictor(&robot_, logger_, config_);
    predictor_ = new Predictor(&robot_, logger_, config_);
}

ArPredictorTask::ArPredictorTask(ArRobot *robot, ArPathPlanningTask *planning,
        string cfgFile, ArFunctor* fail) :
        robot_(robot, planning), training_(false) {
    failAction_ = fail;
    config_ = new PredictorConfig(cfgFile);
    DbHelpers::openDb(config_->databaseName(), logger_);
    logger_ = new AriaLogger();
    baseline_ = new BaselinePredictor(&robot_, logger_, config_);
    predictor_ = new Predictor(&robot_, logger_, config_);
}

ArPredictorTask::~ArPredictorTask() {
    delete logger_;
    delete predictor_;
}

void* ArPredictorTask::runThread(void*) {
    config_ = new(config_) PredictorConfig(configFilePath_);

    std::thread predictorT(&ArPredictorTask::runPredictor, this);

    if (!training_) {
        std::thread baselineT(&ArPredictorTask::runBaseline, this);
        if(baselineT.joinable()) {
            baselineT.join();
        }
    }
    if(predictorT.joinable()) {
        predictorT.join();
    }

    return 0;
}

void ArPredictorTask::failAction(ArFunctor* failAction) {
    failAction_ = failAction;
}

void ArPredictorTask::setTrainig(bool value) {
    training_ = value;
}

void ArPredictorTask::updateOnGoalCompleted() {
    predictor_->registerEndTask(training_);
    if (!training_) {
        baseline_->registerEndTask();
    }
}

void ArPredictorTask::runPredictor() {
    int id = predictor_->prepareNewTask(config_);
    double timeToDeadline = predictor_->timeToDeadline();
    double timeBetweenCalls = timeToDeadline / 5.;
    double timeBetweenCallsUsec = timeBetweenCalls * 1000;
    int qn = 1;
    string log = "Starting new task with Id " + to_string(id);
```

```
    logger_->info(log);

    log = "time to deadline: " + to_string(timeToDeadline);
    logger_->info(log);

    log = "sleeping for: " + to_string(timeBetweenCallsUsec) + " usec";
    logger_->info(log);
    if (training_) {
        while (qn < 4 && this->getRunningWithLock()) {
            Utils::sleep_for(timeBetweenCallsUsec);
            if (this->getRunningWithLock()) {
                predictor_->updateQ(qn);
                qn++;
            }
        }
    } else {
        bool prediction = true;
        while (qn < 4 && prediction && this->getRunningWithLock()) {
            Utils::sleep_for(timeBetweenCallsUsec);
            if (this->getRunningWithLock() && prediction) {
                prediction = predictor_->runPredictionQ(qn, timeBetweenCalls);
                qn++;
            }
        }
        log = "exiting predictor of task with Id " + to_string(id);
        logger_->info(log);
    }
}

void ArPredictorTask::runBaseline() {
    baseline_->prepareNewTask(config_);
    logger_->trace("Runnning baseline");
    double timeToDeadline = baseline_->timeToDeadline();
    double timeBetweenCalls = timeToDeadline / 2;
    timeBetweenCalls *= 1000;
    Utils::sleep_for(timeBetweenCalls);
    if (this->getRunningWithLock()) {
        baseline_->runPrediction();
    }
}
```

Listagem B.20: baseline.cpp

```
/*
 * baseline.cpp
 *
 * Created on: 05/04/2014
 * Author: rafael
 */

#include <models/baseline.h>

BaselineModel::BaselineModel(LoggerInterface *logger) :
        tbDefinition_(TbBaseline::definitionTbBaseline), logger_(logger) {
    createTable();
    updateCurrentId_();
}

int BaselineModel::insert(BaselineData data) {
    try {
        DbHelpers::Statement stmt;
```

```
        map<string, string> fields(data.toStringMap());
        stmt.insert("baseline", fields);
        updateCurrentId_();
        return currentId_;
    } catch (SQLiteException &e) {
        e.Show();
        return -1;
    }
}

void BaselineModel::updateCurrentId_() {
    try {
        DbHelpers::Query query;
        query.select("seq")->from("sqlite_sequence")->where("name", "=",
                "'baseline'");
        DbHelpers::ResultSet res = query.fetch();
        if (res.empty()) {
            currentId_ = 0;
        } else {
            currentId_ = res.getInt("seq");
        }
    } catch (SQLiteException &e) {
        e.Show();
    }
}

int BaselineModel::currentId() {
    return currentId_;
}

int BaselineModel::updateCurrentTaskId() {
    updateCurrentId_();
    DbHelpers::Query query;
    string id = to_string(currentId_);
    query.select("taskId")->from("tasks")->where("id", "=", id);
    ResultSet res = query.fetch();
    currentTaskId_ = -1;
    try {
        currentTaskId_ = res.getInt("taskId");
    } catch (SQLiteException &e) {
        e.Show();
    } catch (boost::bad_any_cast &bec) {
        cout << "cast error on currentTaskId\n";
    }
    return currentTaskId_;
}

int BaselineModel::update(BaselineData data, list<Condition> conditions) {
    map<string, string> fields(data.toStringMap());
    DbHelpers::Statement stmt;
    stmt.update("baseline", fields, conditions);
    return 1;
}

int BaselineModel::update(BaselineData data, Condition condition) {
    list<Condition> l = { condition };
    return update(data, l);
}

void BaselineModel::createTable() {
    try {
```

```
        DbHelpers::Statement stmt;
        stmt.rawSQL(tbDefinition_);
        stmt.run();
    } catch (SQLiteException &e) {
        e.Show();
    }
}
```

Listagem B.21: baselinePredictor.cpp

```
/*
 * baselinePredictor.cpp
 *
 * Created on: 06/04/2014
 * Author: rafael
 */

#include <baselinePredictor.h>
#include "utils/time.h"

BaselinePredictor::BaselinePredictor(RobotInterface* robot, LoggerInterface* logger,
      ↪ PredictorConfig *config) :
        config_ (config), robot_(robot), logger_(logger),
        distanceToGoal_(-1), distanceWalkedQ1_(-1), currentTaskId_(-1), result_(-1) {
    baselineModel_ = new BaselineModel(logger);
}

BaselinePredictor::~BaselinePredictor() {
    delete config_;
    delete baselineModel_;
}

double BaselinePredictor::timeToDeadline() {
    double diff = Utils::timeDiff(Utils::isoTimestamp(false), config_->deadline());
    return diff;
}

void BaselinePredictor::registerEndTask() {
    BaselineData data;
    string id = to_string(currentTaskId_);
    data.endTime = Utils::isoTimestamp();
    data.success = timeToDeadline() >= 0;
    data.predictorAnswer = this->result_;
    data.distanceWalkedQ1 = this->distanceWalkedQ1_;
    Condition cond("id", "=", id);
    baselineModel_->update(data, cond);
}

int BaselinePredictor::prepareNewTask(PredictorConfig *config) {
    config_ = config;
    BaselineData data;
    data.taskId = robot_->currentTaskId();
    data.startTime = Utils::isoTimestamp();
    data.deadline = config_->deadline();
    this->distanceToGoal_ = robot_->distanceToGoal();
    data.distanceToGoal = this->distanceToGoal_;

    try {
        currentTaskId_ = baselineModel_->insert(data);
        return currentTaskId_;
    } catch (SQLiteException &e) {
```

```
        e.Show();
        return -1;
    }
}

void BaselinePredictor::runPrediction() {
    this->distanceWalkedQ1_ = robot_->distanceWalked();
    this->result_ = this->distanceWalkedQ1_ >= this->distanceToGoal_ / 2;
}
```

Listagem B.22: dbHelpers.cpp

```
/*
 * DbHelpers.cpp
 *
 * Created on: 01/07/2013
 * Author: rafael
 */

#include "utils/dbHelpers.h"
#include "utils/path.h"

SQLiteDatabase* DbHelpers::db_ = NULL;

bool DbHelpers::openDb(string dbName, LoggerInterface* logger)
{
    bool newDb = false;
    try {
        DbHelpers::db_ = new SQLiteDatabase(dbName, SQLITE_OPEN_READWRITE, 0);
        string info = "File " + dbName + " found!";
        logger->info(info);
    } catch (SQLiteException &e) {
        Utils::createPath(dbName);
        string wrn = "File " + dbName + " not found, creating...";
        logger->warning(wrn);
        DbHelpers::db_ = new SQLiteDatabase(
                dbName,
                SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, 0
            );
        newDb = true;
    }

    return newDb;
}

string DbHelpers::expandColumns_(list<Column> columns)
{
    ostringstream result;
    result << "SELECT ";
    if (columns.empty()) {
        result << "* ";
    } else {
        for (Column &c : columns) {
            if (&c != &columns.front()) {
                result << ",";
            }
            result << c.name;
            if (c.alias != "") {
                result << " as " << c.alias;
            }
        }
```

```
        result << " ";
    }
    return result.str();
}

string DbHelpers::expandTables_(list<Table> tables)
{
    ostringstream result;
    result.str("");
    result << "FROM ";
    if (tables.empty()) {
// throw Kompex::SQLiteException; TODO: Será?
    } else {
        for (Table &t : tables) {
            if (&t != &tables.front()) {
                result << ",";
            }
            result << t.name;
            if (t.alias != "") {
                result << " as " << t.alias;
            }
        }
        result << " ";
    }
    return result.str();
}

string DbHelpers::expandJoin_(list<Join> join)
{
    ostringstream result;
    result.str("");

    for (Join j : join) {
        result << j.type << " JOIN " << j.table << " ";
        if (!j.on.empty()) {
            result << "on " << DbHelpers::expandConditions_(j.on) << " ";
        }
        if (j.alias != "") {
            result << "as " << j.alias << " ";
        }
    }
    return result.str();
}

string DbHelpers::expandConditions_(list<Condition> conditions)
{
    string resultStr;
    ostringstream result;
    result.str("");
    if (conditions.empty()) {
// throw Kompex::SQLiteException; TODO: Será?
    } else {
        result << "WHERE ";
        for (Condition c : conditions) {
            result << c.field << " " << c.operation << " "
                   << c.value << " AND ";
        }

        resultStr = result.str();
        resultStr.erase(resultStr.end() - 5, resultStr.end());
    }
```

```
    return resultStr;
}

string DbHelpers::expandOrderBy_(list<string> columns, bool asc)
{
    stringstream ss;
    string order;
    for (string &column : columns) {
        if (&column != &columns.front()) {
            ss << ",";
        }
        ss << column;
    }
    if (!columns.empty()) {
        order = " ORDER BY " + ss.str() + ((asc) ? " ASC " : " DESC ");
    }
    return order;
}
```

Listagem B.23: path.cpp

```
#include "utils/path.h"
#include <string>
#include <stdlib.h>

using namespace std;

void Utils::createPath(string path)
{
    unsigned found = path.find_last_of("/\\");
    string dir = path.substr(0, found);
    string cmd("mkdir --parent ");
    cmd.append(dir);
    system(cmd.c_str());
}

void Utils::createPath(const char* path)
{
    string str(path);
    createPath(str);
}
```

Listagem B.24: Predictor.cpp

```
/*
 * aria_predictor.cpp
 *
 * Created on: 25/05/2013
 * Author: rafael
 */

#include <kompex/KompexSQLiteException.h>
#include <Predictor.h>
#include <utils/dbHelpers.h>
#include <utils/time.h>
#include <string>

using namespace std;
using namespace Kompex;
```

```
Predictor::Predictor(RobotInterface * robot, LoggerInterface * logger,
        PredictorConfig *config) : config_(config), robot_(robot), logger_(logger){
    result_ = -1;
    distanceToGoal_ = robot_->distanceToGoal();
    initialDistanceToGoal_ = distanceToGoal_;
    timeToGoal_ = robot_->timeToGoal();
    tasksModel_ = new TaskModel(logger);
    currentId = -1;
    limitDb_ = -1;
    predictionFunctions_.push_back(&Predictor::invalidPredictionFunction);
    predictionFunctions_.push_back(&Predictor::runPredictionQ1);
    predictionFunctions_.push_back(&Predictor::runPredictionQ2);
    predictionFunctions_.push_back(&Predictor::runPredictionQ3);
}

Predictor::~Predictor() {
    delete tasksModel_;
}

int Predictor::prepareNewTask(PredictorConfig *config) {
    robot_->init();
    config_ = config;
    string deadline = "Deadline: " + config_->deadline();
    logger_->info(deadline);
    distanceToGoal_ = robot_->distanceToGoal();
    timeToGoal_ = robot_->timeToGoal();

    limitDb_ = config_->limitDb();

    TaskData data;
    data.startTime = Utils::isoTimestamp();
    data.deadline = config_->deadline();
    data.taskId = robot_->currentTaskId();

    try {
        currentId = tasksModel_->insert(data);
        tasksModel_->updateCurrentTaskId();
        return currentId;
    } catch (SQLiteException &e) {
        e.Show();
        return -1;
    }
}

bool Predictor::runPredictionQ(int n, double fracDeadline) {
    bool will_complete;
    try {
        predictionFunc function = predictionFunctions_.at(n);
        will_complete = function(this, fracDeadline);
        string log = "q" + to_string(n)
                + ": will_complete: " + to_string(will_complete);
        logger_->trace(log);
        this->result_ = will_complete;
    } catch (std::out_of_range &e) {
        invalidPredictionFunction(n);
    }
    return will_complete;
}

bool Predictor::runPredictionQ1(double fracDeadline) {
    double remainingDist = robot_->distanceToGoal();
```

```
    double distWalked = robot_->distanceWalked();
    double avgDistQ1 = tasksModel_->avgDistQ(1, limitDb_);
    double deltaDist = distWalked - avgDistQ1;

    bool will_complete = (deltaDist >= 0);
    if (!will_complete) {
        double avgSpeedQ1 = tasksModel_->avgSpeedQ(1, fracDeadline, limitDb_);
        double avgSpeedQ2 = tasksModel_->avgSpeedQ(2, fracDeadline, limitDb_);
        double avgSpeedQ3 = tasksModel_->avgSpeedQ(3, fracDeadline, limitDb_);
        double avgSpeedQ4 = tasksModel_->avgSpeedQ4(fracDeadline * 3, limitDb_);

        double tq1 = -deltaDist / avgSpeedQ1;

        double sumQ = (avgSpeedQ2 + avgSpeedQ3 + avgSpeedQ4) / 3;
        double estimated = ceil(tq1 + (remainingDist + deltaDist) / sumQ);
        double deadline = timeToDeadline();

        time_t estimated_t = time(NULL) + estimated;

        will_complete = estimated <= deadline;

        // ##### LOGGING #####

        stringstream ss;
        ss << "Q1 data:" << endl
            << "\tremainingDist: " << remainingDist << endl
            << "\tdistWalked: " << distWalked << endl
            << "\tdeltaDist: " << deltaDist << endl
            << "\tavgSpeedQ1: " << avgSpeedQ1 << endl
            << "\tavgSpeedQ2: " << avgSpeedQ2 << endl
            << "\tavgSpeedQ3: " << avgSpeedQ3 << endl
            << "\tavgSpeedQ4: " << avgSpeedQ4 << endl
            << "\testimated: " << estimated << endl
            << "\testimated_t: " << Utils::isoTimestamp(false, estimated_t) << endl
            << "\ttime to deadline: " << deadline << endl
            << "\twill: " << will_complete;
        logger_->trace(ss.str());

        // ##### END LOGGING #####

    }
    updateQ(1);
    return will_complete;
}

bool Predictor::runPredictionQ2(double fracDeadline) {
    double remainingDist = robot_->distanceToGoal();
    double distWalked = robot_->distanceWalked();
    double avgDistQ2 = tasksModel_->avgDistQ(2, limitDb_);
    double deltaDist = distWalked - avgDistQ2;
    bool will_complete = (deltaDist >= 0);
    if (!will_complete) {
        double avgSpeedQ2 = tasksModel_->avgSpeedQ(2, fracDeadline, limitDb_);
        double avgSpeedQ3 = tasksModel_->avgSpeedQ(3, fracDeadline, limitDb_);
        double avgSpeedQ4 = tasksModel_->avgSpeedQ4(fracDeadline * 3, limitDb_);
        double sumQ = (avgSpeedQ3 + avgSpeedQ4) / 2;

        double tq2 = -deltaDist / avgSpeedQ2;

        double estimated = ceil(tq2 + (remainingDist + deltaDist) / sumQ);
        double deadline = timeToDeadline();
```

```
        time_t estimated_t = time(NULL) + estimated;

        will_complete = estimated <= deadline;
        stringstream ss;

        // ##### LOGGING #####

        ss << "Q2 data: "<< endl
            << "\tremainingDist: " << remainingDist << endl
            << "\tdistWalked: " << distWalked << endl
            << "\tavgDistQ2: " << avgDistQ2 << endl
            << "\tdeltaDist: " << deltaDist << endl
            << "\tavgSpeedQ2: " << avgSpeedQ2 << endl
            << "\tavgSpeedQ3: " << avgSpeedQ3 << endl
            << "\tavgSpeedQ4: " << avgSpeedQ4 << endl
            << "\testimated: " << estimated << endl
            << "\testimated_t: " << Utils::isoTimestamp(false, estimated_t) << endl
            << "\ttime to deadline: " << deadline << endl
            << "\twill: " << will_complete;
        logger_->trace(ss.str());

        // ##### END LOGGING #####

    }
    updateQ(2);
    return will_complete;
}

bool Predictor::runPredictionQ3(double fracDeadline) {
    double remainingDist = robot_->distanceToGoal();
    double distWalked = robot_->distanceWalked();
    double avgDistQ3 = tasksModel_->avgDistQ(3, limitDb_);
    double deltaDist = distWalked - avgDistQ3;
    bool will_complete = (deltaDist >= 0);
    if (!will_complete) {
        double avgSpeedQ3 = tasksModel_->avgSpeedQ(3, fracDeadline, limitDb_);
        double avgSpeedQ4 = tasksModel_->avgSpeedQ4(fracDeadline * 3, limitDb_);

        double tq3 = -deltaDist / avgSpeedQ3;

        double estimated = ceil(tq3 + (remainingDist + deltaDist) / avgSpeedQ4);
        double deadline = timeToDeadline();

        time_t estimated_t = time(NULL) + estimated;

        will_complete = estimated <= deadline;

        // ##### LOGGING #####

        stringstream ss;
        ss << "Q3 data: "<< endl
            << "\tremainingDist: " << remainingDist << endl
            << "\tdistWalked: " << distWalked << endl
            << "\tavgDistQ3: " << avgDistQ3 << endl
            << "\tdeltaDist: " << deltaDist << endl
            << "\tavgSpeedQ4: " << avgSpeedQ4 << endl
            << "\testimated: " << estimated << endl
            << "\testimated_t: " << Utils::isoTimestamp(false, estimated_t) << endl
            << "\ttime to deadline: " << deadline << endl
            << "\twill: " << will_complete;
```

```
        logger_->trace(ss.str());

        // ##### END LOGGING #####
    }
    updateQ(3);
    return will_complete;
}

double Predictor::timeToDeadline() {
    time_t now = time(NULL);
    double diff = Utils::timeDiff(Utils::isoTimestamp(false, now),
            config_->deadline());
    return diff;
}

void Predictor::updateQ(int n) {
    tasksModel_->updateCurrentTaskQ(n, robot_->distanceWalked());
}

void Predictor::registerEndTask(bool training) {
    TaskData data;
    stringstream id;
    id << currentId;
    data.distQ4 = robot_->distanceWalked();
    data.endTime = Utils::isoTimestamp();
    data.success = timeToDeadline() >= 0;
    data.predictorAnswer = this->result_;
    Condition cond("id", "=", id.str());
    tasksModel_->update(data, cond);
// if (!training) {
//  tasksModel_->removeFirstTask();
// }
}
```

Listagem B.25: PredictorConfig.cpp

```
#include "PredictorConfig.h"
#if defined(_WIN32) || defined(__WIN32__) || defined(WIN32)
#include "config/libconfig.h"
#else
#include <libconfig.h>
#endif

#include <iostream>
#include <string>

#include "utils/time.h"

using namespace std;

PredictorConfig::PredictorConfig(const string& fileName)
{
    parseConfigFile(fileName);
}

void PredictorConfig::parseConfigFile(const string& fileName) {
    config_t cfg;
    config_setting_t* setting;
    const char* file_name = fileName.c_str();
    config_init(&cfg);
    if (config_read_file(&cfg, file_name)) {
```

```
        setting = config_lookup(&cfg, "params");
        if (setting != NULL) {
            const char* tmp;
            long long int tmp_deadline;
            if (config_setting_lookup_int64(setting, "deadline_seconds",
                    &tmp_deadline)) {
                time_t now = time(NULL);
                now += tmp_deadline;
                deadline_.assign(Utils::isoTimestamp(false, now));
            } else if (config_setting_lookup_string(setting, "deadline_iso",
                    &tmp)) {
                deadline_.assign(tmp);
            }
            if (config_setting_lookup_int(setting, "max_vel", &maxVel_)) {
            }
            if (config_setting_lookup_int(setting, "limitDb", &limitDb_)) {
            }
            if (config_setting_lookup_string(setting, "database", &tmp)) {
                databaseName_.assign(tmp);
            }
        }

        config_destroy(&cfg);
    } else {
        cout << "\n\nErro ao verificar configurações\n\n";
    }
}

string PredictorConfig::deadline() const
{
    return deadline_;
}

int PredictorConfig::maxVel() const
{
    return maxVel_;
}

string PredictorConfig::databaseName() const
{
    return databaseName_;
}

int PredictorConfig::limitDb() const
{
    return limitDb_;
}
```

Listagem B.26: query.cpp

```
/*
 * query.cpp
 *
 * Created on: 06/07/2013
 * Author: rafael
 */

#include "utils/dbHelpers.h"
#include <typeinfo>

DbHelpers::ResultSet DbHelpers::Query::fetch() {
```

```
    try {
        query_ = buildQuery_();
        result_ = new ResultSet();
        stmt_.Sql(query_);
        int total_results = stmt_.GetNumberOfRows();
        for (int i = 0; i < total_results; i++) {
            stmt_.FetchRow();
            ResultSet::Data data;
            for (Column c : columnList_) {
                string name = (c.alias != "") ? c.alias : c.name;
                result_->setValue(stmt_, &data, name);
            }
            result_->addRow(data);
        }
        result_->init();
        stmt_.FreeQuery();
    } catch (SQLiteException &e) {
        e.Show();
        result_ = new ResultSet();
    }
    return *result_;
}

string DbHelpers::Query::str()
{
    return buildQuery_();
}

DbHelpers::Query::Query()
{
    result_ = 0;
    asc_ = true;
}

DbHelpers::Query::~Query() {
    delete result_;
}

DbHelpers::Query* DbHelpers::Query::select(list<Column> columns)
{
    list<Column> l(columns);
    columnList_.merge(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::select(Column column)
{
    list<Column> l;
    l.push_front(column);
    this->select(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::select(string column, string alias)
{
    list<Column> l;
    Column c(column, alias);
    l.push_front(c);
    this->select(l);
    return this;
}
```

```
DbHelpers::Query* DbHelpers::Query::from(list<Table> tables)
{
    list<Table> l(tables);
    tableList_.merge(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::from(Table table)
{
    list<Table> l;
    l.push_front(table);
    this->from(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::from(string table, string alias)
{
    list<Table> l;
    Table t(table, alias);
    l.push_front(t);
    this->from(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::fromSubquery(Query query)
{
    string q = query.str();
    q.erase(q.end() - 1, q.end());
    return from("("+ q + ")");
}
DbHelpers::Query* DbHelpers::Query::join(list<Join> joins)
{
    list<Join> l(joins);
    joinList_.merge(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::join(Join join)
{
    list<Join> l;
    l.push_front(join);
    this->join(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::join(string table, string type, string alias, list<
    ↪ Condition> conditions)
{
    list<Join> l;
    Join j(table, type, conditions, alias);
    l.push_front(j);
    this->join(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::where(list<Condition> conditions)
{
    list<Condition> l(conditions);
    conditionList_.merge(l);
    return this;
```

```
}

DbHelpers::Query* DbHelpers::Query::where(Condition condition)
{
    list<Condition> l;
    l.push_front(condition);
    this->where(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::where(string field, string operation, string value)
{
    list<Condition> l;
    Condition c(field, operation, value);
    l.push_front(c);
    this->where(l);
    return this;
}

DbHelpers::Query* DbHelpers::Query::limit(int limit, int offset) {
    string s("limit ");
    s += to_string(limit);
    if (offset > 0) {
        s + "," + to_string(offset);
    }
    limit_ = s;
    return this;
}

DbHelpers::Query* DbHelpers::Query::orderBy(string column, bool asc) {
    list<string> l(1,column);
    return orderBy(l, asc);
}

DbHelpers::Query* DbHelpers::Query::orderBy(list<string> columns, bool asc) {
    asc_ = asc;
    orderBy_.insert(orderBy_.end(), columns.begin(), columns.end());
    return this;
}

string DbHelpers::Query::buildQuery_() const {
    string res = DbHelpers::expandColumns_(columnList_);
    res += DbHelpers::expandTables_(tableList_);
    res += DbHelpers::expandJoin_(joinList_);
    res += DbHelpers::expandConditions_(conditionList_);
    res += DbHelpers::expandOrderBy_(orderBy_, asc_);
    res += limit_;
    res += ";";
    return res;
}

ostream& operator<<(ostream& out, const DbHelpers::Query& q)
{
    string res(q.buildQuery_());

    return out << res;
}

void DbHelpers::Query::clear() {
    this->columnList_.clear();
    this->conditionList_.clear();
```

```
    this->joinList_.clear();
    this->tableList_.clear();
    delete this->result_;
}
```

Listagem B.27: resultSet.cpp

```
/*
 * resultSet.cpp
 *
 * Created on: 06/07/2013
 * Author: rafael
 */

#include "utils/dbHelpers.h"

void DbHelpers::ResultSet::init()
{
    it_ = rows_.begin();
}

enum ValueTypes
{
    INT = 1,
    DOUBLE = 2,
    STRING = 3,
    BLOB = 4,
    NULL_ = 5
};

void DbHelpers::ResultSet::setValue(SQLiteStatement& stmt, Data* data, string column)
{
    switch (stmt.GetColumnType(column)) {
    case INT:
        data->fields[column] = stmt.GetColumnInt(column);
        break;
    case DOUBLE:
        data->fields[column] = stmt.GetColumnDouble(column);
        break;
    case STRING:
        data->fields[column] = stmt.GetColumnString(column);
        break;
    case BLOB:
        data->fields[column] = stmt.GetColumnBlob(column);
        break;
    case NULL_:
        data->fields[column] = NULL;
        break;
    }
}

bool DbHelpers::ResultSet::empty()
{
    return rows_.empty();
}

int DbHelpers::ResultSet::length()
{
    return rows_.size();
}
```

```
void DbHelpers::ResultSet::addRow(Data data)
{
    rows_.push_back(data);
}

int DbHelpers::ResultSet::getInt(string column)
{
    return boost::any_cast<int>(it_->fields[column]);
}

double DbHelpers::ResultSet::getDouble(string column)
{
    return boost::any_cast<double>(it_->fields[column]);

}

string DbHelpers::ResultSet::getText(string column)
{
    return boost::any_cast<string>(it_->fields[column]);
}

void* DbHelpers::ResultSet::getBlob(string column)
{
    return boost::any_cast<void *>(it_->fields[column]);
}

bool DbHelpers::ResultSet::end()
{
    return it_ == rows_.end();
}
```

Listagem B.28: Statement.cpp

```
/*
 * Statement.cpp
 *
 * Created on: 03/08/2013
 * Author: rafael
 */

#include "utils/dbHelpers.h"

DbHelpers::Statement::Statement(): stmt_(DbHelpers::db_) {}

DbHelpers::Statement::Statement(string sql): stmt_(DbHelpers::db_) {
    stmt_.SqlStatement(sql);
}

void DbHelpers::Statement::rawSQL(string sql)
{
    stmt_.SqlStatement(sql);
}

void DbHelpers::Statement::insert(string table, map<string, string> fields)
{
    ostringstream sql, field, value;
    sql << "INSERT INTO " << table;
    field << "(";
    value << "(";
    map<string, string>::iterator it;
    for (it = fields.begin(); it != fields.end(); it++){
```

```
        if (it != fields.begin()) {
            field << ", ";
            value << ", ";
        }
        field << '"' << it->first << '"';
        value << '"' << it->second << '"';
    }
    field << ")";
    value << ")";
    sql << field.str() << " VALUES " << value.str() << ";";
    stmt_.Sql(sql.str());
    stmt_.ExecuteAndFree();
}

void DbHelpers::Statement::update(string table, map<string, string> fields, Condition
    ↪ condition) {
    list<Condition> l = {condition};
    update(table, fields, l);
}

void DbHelpers::Statement::update(string table, map<string, string> fields, list<
    ↪ Condition> conditions)
{
    stringstream sql;
    sql << "UPDATE " << table << " SET ";
    for (auto &field : fields) {
        sql << field.first << " = '" << field.second << "', ";
    }
    string partial = sql.str();
    partial.erase(partial.end()-2, partial.end());
    sql.clear();
    sql.str("");
    sql << partial;
    if (!conditions.empty()) {
        sql << " WHERE ";
    }
    for (Condition c : conditions) {
        sql << c.field << c.operation << c.value << " AND ";
    }
    partial = sql.str();
    partial.erase(partial.end()-4, partial.end());
    stmt_.Sql(partial);
    stmt_.ExecuteAndFree();
}

void DbHelpers::Statement::remove(string table, list<Condition> conditions)
{
    ostringstream partialSql, field, value;
    partialSql << "DELETE FROM " << table;
    if (!conditions.empty()) {
        partialSql << " WHERE ";
        for (Condition cond : conditions){
            partialSql << cond.field << cond.operation << cond.value << " AND ";
        }
    }
    string sql = partialSql.str();
    sql.erase(sql.end()-5, sql.end());
    stmt_.Sql(sql);
    stmt_.ExecuteAndFree();
}
```

```
void DbHelpers::Statement::remove(string table, Condition condition)
{
    list<Condition> l = { condition };
    remove(table, l);
}

bool DbHelpers::Statement::run()
{
    stmt_.ExecuteAndFree();
    return true;
}
```

Listagem B.29: tasks.cpp

```
#include <kompex/KompexSQLiteException.h>
#include <models/tasks.h>
#include <iostream>
#include <map>

TaskModel::TaskModel(LoggerInterface *logger) :
        tbDefinition_(TbTasks::definitionTbTasks),
        logger_(logger) {
    createTable();
    updateCurrentId_();
}

int TaskModel::insert(TaskData data) {
    try {
        DbHelpers::Statement stmt;
        map<string, string> fields(data.toStringMap());
        stmt.insert("tasks", fields);
        updateCurrentId_();
        return currentId_;
    } catch (SQLiteException &e) {
        e.Show();
        return -1;
    }
}

int TaskModel::update(TaskData data, Condition condition) {
    list<Condition> l = {condition};
    return update(data, l);
}

int TaskModel::update(TaskData data, list<Condition> conditions) {
    map<string, string> fields(data.toStringMap());
    DbHelpers::Statement stmt;
    stmt.update("tasks", fields, conditions);
    return 1;
}

int TaskModel::updateCurrentTaskQ(int n, double distance) {
    string key = "distQ" + to_string(n);

    map<string, string> fields;
    fields[key] = to_string(distance);

    DbHelpers::Statement stmt;
    stmt.update("tasks", fields, Condition("id", "=", to_string(currentId_)));
    return 1;
}
```

```
double TaskModel::minDistQ(int n, int limit) {
    double minDist = -1;
    try {
        string id = to_string(currentId_);
        string taskId = to_string(currentTaskId_);

        DbHelpers::Query query, subquery;

        string columnSubquery = "distQ" + to_string(n);
        subquery.select(columnSubquery)
                ->from("tasks")
                ->where("id", "<", id)
                ->where("taskId" , "=", taskId)
                ->where("success", "=", "1")
                ->orderBy("id", false);
        if (limit > 0) {
            subquery.limit(limit);
        }

        string colMin = "min(" + columnSubquery + ")";
        query.select(colMin, "minDist")->fromSubquery(subquery);

        ResultSet res = query.fetch();
        minDist = res.getDouble("minDist");
    } catch (SQLiteException &e) {
        e.Show();
    } catch (boost::bad_any_cast &bec) {
        cout << "cast error on minDistQ(n)\n";
    }
    return minDist;
}

double TaskModel::avgDistQ(int n, int limit) {
    double avg = -1;
    try {
        string id = to_string(currentId_);
        string taskId = to_string(currentTaskId_);
        string columnSubquery = "distQ" + to_string(n);
        string column = "avg(" + columnSubquery + ")";

        DbHelpers::Query query, subquery;

        subquery.select(columnSubquery)
                ->from("tasks")
                ->where("id", "<", id)
                ->where("taskId" , "=", taskId)
                ->where("success", "=", "1")
                ->orderBy("id", false);

        if(limit > 0) {
            subquery.limit(limit);
        }

        query.select(column, "avg")->fromSubquery(subquery);

        ResultSet res = query.fetch();
        avg = res.getDouble("avg");
    } catch (SQLiteException &e) {
        e.Show();
    } catch (boost::bad_any_cast &bec) {
```

```
        cout << "cast error on avgDistQ(n)\n";
    }
    return avg;
}

double TaskModel::avgSpeedQ(int n, double time, int limit) {
    double avg = -1;
    DbHelpers::Query query;
    DbHelpers::Query subquery;

    string taskId = to_string(currentTaskId_);
    string id = to_string(currentId_);

    string columnSubquery = "distQ" + to_string(n);
    string colAvg = "avg((" + columnSubquery;

    if (n > 1) {
        string subtraction = "distQ" + to_string(n - 1);
        colAvg += " - " + subtraction;
        columnSubquery += "," + subtraction;
    }

    colAvg += ")) / " + to_string(time);

    try {
        subquery.select(columnSubquery)
                ->from("tasks")
                ->where("taskId" , "=", taskId)
                ->where("id", "<", id)
                ->where("success", "=", "1")
                ->orderBy("id", false);

        if(limit > 0) {
            subquery.limit(limit);
        }

        query.select(colAvg, "avg")->fromSubquery(subquery);
        ResultSet res = query.fetch();
        avg = res.getDouble("avg");
    } catch (SQLiteException &e) {
        e.Show();
    } catch (boost::bad_any_cast &bec) {
        cout << "cast error on avgSpeedQ(n)\n";
    }

    return avg;
}

double TaskModel::avgSpeedQ4(double spentTime, int limit) {
    double avg = -1;
    DbHelpers::Query query, subquery;

    string taskId = to_string(currentTaskId_);
    string id = to_string(currentId_);

    Column distQ4("distQ4");
    Column distQ3("distQ3");
    Column endTIme("endTIme");
    Column startTime("startTime");

    list<Column> columns;
```

```
    columns.push_back(distQ4);
    columns.push_back(distQ3);
    columns.push_back(endTIme);
    columns.push_back(startTime);

    subquery.select(columns)
            ->from("tasks")
            ->where("taskId" , "=", taskId)
            ->where("id", "<", id)
            ->where("success", "=", "1")
            ->orderBy("id", false);

    if(limit > 0) {
        subquery.limit(limit);
    }

    string interval = "(julianday(endTime) - julianday(startTime)) "
                      "* 24 * 60 * 60 - " + to_string(spentTime);
    string colAvg = "avg((distQ4 - distQ3) / (" + interval + "))";
    try {
        query.select(colAvg, "avg")->fromSubquery(subquery);

        ResultSet res = query.fetch();
        avg = res.getDouble("avg");
    } catch (SQLiteException &e) {
        e.Show();
    }
    return avg;
}

void TaskModel::createTable() {
    try {
        DbHelpers::Statement stmt;
        stmt.rawSQL(tbDefinition_);
        stmt.run();
    } catch (SQLiteException &e) {
        e.Show();
    }
}

void TaskModel::updateCurrentId_() {
    try {
        DbHelpers::Query query;
        query.select("seq")->from("sqlite_sequence")->where("name", "=",
                "'tasks'");
        DbHelpers::ResultSet res = query.fetch();
        if (res.empty()) {
            currentId_ = 0;
        } else {
            currentId_ = res.getInt("seq");
        }
    } catch (SQLiteException &e) {
        e.Show();
    }
}

int TaskModel::currentId() {
    return currentId_;
}

int TaskModel::updateCurrentTaskId() {
```

```
    updateCurrentId_();
    DbHelpers::Query query;
    string id = to_string(currentId_);
    query.select("taskId")
            ->from("tasks")
            ->where("id", "=", id);
    ResultSet res = query.fetch();
    currentTaskId_ = -1;
    try{
        currentTaskId_ = res.getInt("taskId");
    } catch (SQLiteException &e) {
        e.Show();
    } catch (boost::bad_any_cast &bec) {
        cout << "cast error on currentTaskId\n";
    }
    return currentTaskId_;
}

ResultSet TaskModel::select(list<Column> fields, Condition condition) {
    list<Condition> l = {condition};
    return select(fields, l);
}

ResultSet TaskModel::select(list<Column> fields, list<Condition> conditions) {
    DbHelpers::Query query;
    query.select(fields)->from("tasks");
    if (!conditions.empty()) {
        query.where(conditions);
    }
    ResultSet res = query.fetch();
    return res;
}
```

Listagem B.30: tbBaseline.cpp

```
/*
 * tbBaseline.cpp
 *
 * Created on: 05/04/2014
 * Author: rafael
 */

#include <models/tables/tbBaseline.h>
#include <sstream>
#include "utils/time.h"

#include <iostream>

std::string TbBaseline::definitionTbBaseline =
        "CREATE TABLE IF NOT EXISTS baseline("
        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
        "taskId INTEGER NOT NULL,"
        "startTime TEXT NOT NULL,"
        "deadline TEXT NOT NULL,"
        "distanceToGoal DOUBLE NOT NULL,"
        "endTime TEXT DEFAULT NULL,"
        "distanceWalkedQ1 DOUBLE DEFAULT NULL,"
        "predictorAnswer INT DEFAULT NULL,"
        "success INT DEFAULT NULL"
        ");";
```

```
std::map<std::string, std::string> TbBaseline::BaselineData::toStringMap()
{
    std::map<std::string, std::string> data;
    if (this->id != -1) {
        data["id"] = to_string(this->id);
    }
    if (this->taskId != -1) {
        data["taskId"] = to_string(this->taskId);
    }
    if (this->startTime != "") {
        data["startTime"] = this->startTime;
    }
    if (this->endTime != "") {
        data["endTime"] = this->endTime;
    }
    if (this->distanceToGoal != -1) {
        data["distanceToGoal"] = to_string(this->distanceToGoal);
    }
    if (this->distanceWalkedQ1 != -1) {
        data["distanceWalkedQ1"] = to_string(this->distanceWalkedQ1);
    }
    if (this->deadline != "") {
        data["deadline"] = this->deadline;
    }
    if (this->success != -1) {
        data["success"] = to_string(this->success);
    }
    if (this->predictorAnswer != -1) {
        data["predictorAnswer"] = to_string(this->predictorAnswer);
    }
    return data;
}
```

Listagem B.31: tbTasks.cpp

```
#include <models/tables/tbTasks.h>
#include <utils/time.h>
#include <sstream>

string TbTasks::definitionTbTasks = "CREATE TABLE IF NOT EXISTS tasks("
        "id INTEGER PRIMARY KEY AUTOINCREMENT,"
        "taskId INTEGER NOT NULL,"
        "startTime TEXT NOT NULL,"
        "deadline TEXT NOT NULL,"
        "distQ1 DOUBLE DEFAULT NULL,"
        "distQ2 DOUBLE DEFAULT NULL,"
        "distQ3 DOUBLE DEFAULT NULL,"
        "distQ4 DOUBLE DEFAULT NULL,"
        "endTime TEXT DEFAUL NULL,"
        "predictorAnswer INT DEFAULT NULL,"
        "success INT DEFAULT NULL"
        ");"; /**< Implementation of the 'tasks' table structure */

map<string, string> TbTasks::TaskData::toStringMap()
{
    map<string, string> data;
    if (this->id != -1) {
        data["id"] = to_string(this->id);
    }
    if (this->taskId != -1) {
        data["taskId"] = to_string(this->taskId);
```

```
    }
    if (this->startTime != "") {
        data["startTime"] = this->startTime;
    }
    if (this->deadline != "") {
        data["deadline"] = this->deadline;
    }
    if (this->distQ1 != -1) {
        data["distQ1"] = to_string(this->distQ1);
    }
    if (this->distQ2 != -1) {
        data["distQ2"] = to_string(this->distQ2);
    }
    if (this->distQ3 != -1) {
        data["distQ3"] = to_string(this->distQ3);
    }
    if (this->distQ4 != -1) {
        data["distQ4"] = to_string(this->distQ4);
    }
    if (this->endTime != "") {
        data["endTime"] = this->endTime;
    }
    if (this->success != -1) {
        data["success"] = to_string(this->success);
    }
    data["predictorAnswer"] = to_string(this->predictorAnswer);
    return data;
}
```

Listagem B.32: time.cpp

```
/*
 * time.cc
 *
 * Created on: 24/06/2013
 * Author: rafael
 */

#include <time.h>
#include "utils/time.h"
#include <sstream>
#include <string>
#include <vector>

#include <chrono>
#include <thread>

#include <iostream>

using namespace std;
using chrono::system_clock;

string Utils::isoTimestamp(bool gmt, time_t date) {
    time_t t = date;
    tm* ptm;
    string tz;
    if (gmt) {
        ptm = gmtime(&t);
        tz = "Z";
    } else {
        ptm = localtime(&t);
```

```
        int gmtoff = ptm->tm_gmtoff / 3600.0;
        tz = to_string(gmtoff);
        if (gmtoff > -10 && gmtoff < 10) {
            tz.insert(tz.size() - 1, "0");
        }
        string minuts((ptm->tm_gmtoff % 3600 != 0) ? ":30" : ":00");
        tz.append(minuts);
    }

    int year = ptm->tm_year + 1900;

    string mon((ptm->tm_mon < 9) ? "-0" : "-");
    string day((ptm->tm_mday < 10) ? "-0" : "-");
    string hour((ptm->tm_hour < 10) ? "0" : "");
    string min((ptm->tm_min < 10) ? "0" : "");
    string sec((ptm->tm_sec < 10) ? "0" : "");

    ostringstream stream;
    stream << year << mon << (ptm->tm_mon + 1) << day << ptm->tm_mday << "T"
           << hour << ptm->tm_hour << ":" << min << ptm->tm_min << ":" << sec
           << ptm->tm_sec << tz;
    return stream.str();
}

time_t Utils::stringToDateTime(string datetime) {
    struct tm dt;
    string tz = datetime.substr(19, 6);
    long offset;
    if (tz == "Z" || tz == "") {
        offset = 0;
    } else {
        int intHour;
        int intMinutes;
        bool negative = tz.substr(0, 1) == "-";
        stringstream hour(tz.substr(1, 2));
        stringstream minutes(tz.substr(3, 2));
        hour >> intHour;
        minutes >> intMinutes;
        offset = intHour * 3600 + intMinutes * 60;
        if (negative) {
            offset *= -1;
        }
    }

    string format = "%FT%T%z";
    strptime(datetime.c_str(), format.c_str(), &dt);

    dt.tm_isdst = 0;

    dt.tm_hour += gmt_hour_offset;
    dt.tm_min += gmt_min_offset;

    time_t t = mktime(&dt);
    t -= offset;
    return t;

}

double Utils::timeDiff(string start, string end) {
    time_t startTime = Utils::stringToDateTime(start);
    time_t endTime = Utils::stringToDateTime(end);
```

```
    return difftime(endTime, startTime);
}

void Utils::sleep_for(int millisec) {
    chrono::milliseconds duration(millisec);
    this_thread::sleep_for(duration);
}
```

ANEXO C – Documentação

A documentação do Previsor pode ser encontrada em `http://inf.ufsc.br/~rafael.pedretti/predictor_doc` para consulta online ou em `http://inf.ufsc.br/~rafael.pedretti/predictor_doc.pdf`

ANEXO D – Artigo

Previsão Temporal de Navegação Para Robôs Móveis Autônomos

Rafael Elias Pedretti[1]

[1]Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Campus Universitário – Florianópolis – SC – Brazil

Abstract. *The mobile robotic is a very broad area which is present in all types of environments. There are robots for patrolling, reconnaissance, entertainment and many other functions, all of which can be terrestrial, aquatic, aerial and each with its specific function, with its type of navigation, communication and interaction with the outside world. Precisely because there are so many different techniques and technologies it's difficult to create prediction mechanisms for these tasks when there is time and/or autonomy constraints. The purpose of this document, through qualitative analysis of the factors already mentioned, is to expose the conceptual and technological factors that are involved in design decisions for the creation of such mechanisms.*

Resumo. *A robótica móvel é uma área muito abrangente que est presente em todos os tipos de ambientes. Existem robôs de patrulhamento, reconhecimento, entretenimento e muitas outras funções, todos podendo ser de ambientes terrestres, subaquáticos, aéreos e cada um com sua função específica, com seu tipo de navegação, comunicação e interação com o mundo externo. Justamente por haver tantas técnicas e tecnologias diferentes é difícil criar mecanismos de previsão de cumprimento dessas tarefas quando há restrições de tempo e/ou autonomia. O objetivo deste documento, através de análises qualitativas dos fatores já mencionados, é expor os fatores conceituais e tecnológicos que estão envolvidos nas decisões de projeto para a criação de tais mecanismos.*

1. Introdução

Não é de hoje que a humanidade sonha com robôs para os auxiliar com tarefas diversas. A palavra *robótica* foi muito popularizada na década de 50 pelo autor de obras de ficção científica Isaac Asimov com seu conto Eu, Robô, do inglês *I, Robot* [Asimov 1950] e tantas outras obras. Desde então a televisão e o cinema também fizeram uso de robôs como a aclamada franquia *Guerra nas Estrelas* com o robô diplomata C-3PO e a série *Battlestar Galactica* com sua raça de robôs inteligentes, os *Cylons*.

Apesar de serem obras de ficção o conceito ficou muito presente na indústria com seus braços robóticos que executam tarefas mais complexas com mais precisão que um ser humano e a partir desse ponto houve uma demanda muito grande desse tipo de equipamento. É possível observar que a robótica está inserida em quase todos os meios: na medicina auxiliando cirurgias, exploração de ambientes inóspitos, na guerra com os VANT's (Veículos Aéreos Não Tripulados), entre outros.

Este trabalho é voltado para a robótica móvel, onde os robôs não são fixados em local, como os robôs de uma montadora de carros, por exemplo. Atualmente existem

vários tipos de robôs móveis [Siegwart and Nourbakhsh 2011], sendo os mais conhecidos os com pernas, rodas e aéreos, cada um com um objetivo e modos de operação.

Os robôs móveis podem ser segmentados em duas grandes classes: os tele-operados e os autônomos. Os tele-operados são robôs que precisam de um operador humano para controlá-lo remotamente, enquanto os autônomos são robôs que conseguem se locomover e executar suas funções sem intervenções humanas.

Para um robô autônomo ter uma utilidade, ele deve ser capaz de executar as tarefas que lhe foram passadas de forma eficiente e eficaz. As atividades que um robô deve executar variam de simplesmente "passear" por um mapa sem esbarrar em nada até inspecionar tubulações [Nassiraei et al. 2007].

A proposta desse trabalho é fornecer à esses robôs móveis um mecanismo que os possibilitem de dizer se irão conseguir cumprir suas tarefas dentro do prazo estipulado quando houver uma restrição de tempo de execução (um *deadline*).

2. Robótica

A robótica é uma área jovem, com raízes nas engenharias, computação ciências cognitivas e sociais dentre outras. A robótica alcançou grande prestígio na área de fabricação industrial com os braços robóticos, ou manipuladores, capazes de rapidamente executar tarefas repetitivas como solda ponto e pintura [Siegwart and Nourbakhsh 2011].

Apesar da sua grande utilidade os manipuladores ainda tem uma desvantagem: a falta de mobilidade. Com essa demanda foram criados os robôs móveis, com diferentes capacidades de locomoção para atingir o maior número de áreas possíveis, criando novas possibilidades como por exemplo a inspeção interna de tubulações, onde para um ser humano seria muito perigoso ou de difícil acesso. Sendo assim, a robótica móvel melhorou as condições de trabalho e segurança dos envolvidos nas tarefas.

Estes robôs podem ser classificados em 3 grupos, de acordo com sua locomoção e 2 grupos de acordo com o modo de operação. Na locomoção o robô pode ser terrestre, aéreo ou aquático. no modo de operação ele pode ser teleoperado ou autônomo, que é o foco deste trabalho. Para realizar suas tarefas essas ferramentas dispoem de uma gama de sensores e atuadores para poderem interagir com o ambiente em que se encontram.

Com esses sensores e atuadores disponíveis os robôs podem navegar e se localizar em ambientes, conhecidos ou não, de modo que evitem colisões e consigam cumprir suas tarefas. Porém apenas completrar a missão que lhe foi passada as vezes não é suficiente, é preciso que ela seja terminada dentro de um limite de tempo, *deadline*.

2.1. Trabalhos Correlatos

São muitos os estudos na área de previsão de perda de *deadline*, desde tarefas de baixo nível até as de mais alto nível. Em [Plentz et al. 2008] é proposto um mecanismo de previsão para tarefas com *deadlines soft* e *firm* em uma aplicação com um sistema de *threads* distribuídas.

[Tsubouchi and Arimoto 1994] e [Miura et al. 1999] descrevem técnicas para melhorar o planejamento de rotas durante a execução da tarefa em ambientes dinâmicos. Essas abordagens utilizam modelos matemáticos e estatísticos para analisar o ambiente e

tomar as decisçoes necessárias. Na mesma linha de pesquisa [Miura et al. 1999] utiliza um método de busca em uma árvore de decisões com pesos nas arestas para calcular a rota até o destino final com o provável menor tempo de execução.

Com técnicas um pouco mais avançadas é possível que o robô aprenda a escolher melhores caminhos do que o atual baseado na relação entre tarefas executadas e ações corretivas tomadas e o tempo gasto em execuções anteriores para avaliar a performance da execução atual [Michaud and Matari 1998]. Mantendo um histórico de ações executadas e utilizando uma métrica de comparação é possível que o robô decida quando ele deve tomar um caminho diferente de ações para completar suas tarefas de um modo mais eficiente que o atual.

3. Proposta

Os mecanismos apresentados neste trabalho foram desenvolvidos com o seguinte cenário como ambiente de execução: A tarefa que o robo irá executar é se locomover de um ponto A a um ponto B em um ambiente conhecido com obstáculos dinâmicos. Foram então desenvolvidos dois mecanismos de previsão, o proposto e um mais simples, cujo propósito é servir de base de comparação. A abordagem escolhida para os previsores foi a temporal, que consistem em certos marcos do *deadline* executar a previsão.

3.1. Algoritmo de Base

Para manter essa abordagem simples foi definido que esse marco seria sempre na metade do *deadline* e quando esse tempo fosse alcançado seria então executado o cálculo da previsão. Esse marco foi escolhido para que quando uma resposta for dada, o robô ou o operador humano tenham tempo de tomar as ações necessárias. Decompondo a linha de tempo de uma tarefa podemos então definir três marcos, sendo eles o início, 50% do *deadline* e o fim da tarefa. Para futuras referências chamemos esses marcos de q_0, q_1 *e* q_2, respectivamente.

Em q_0 os dados do robô que serão utilizados, sendo eles a distância estimada d_e até o objetivo, a distância percorrida d na tarefa e o tempo decorrido t desde o inicio da tarefa, são reinicializados para começar a nova execução. Nesse ponto então temos que o robô percorreu uma distância estimada d_e qualquer, uma distância percorrida d de 0 mm e um tempo decorrido t de 0 segundos e então o previsor "dorme" até q_1 ser alcançado.

Quando esse marco for atingindo é executada a previsão da seguinte maneira: Se em q_1 a distância percorrida d for maior ou igual à distância estimada d_e assume-se que o robô irá chegar ao seu destino dentro do *deadline*, caso contrário a previsão é de não concluir a tempo a tarefa. Definindo esses termos matematicamente chegamos na função 1.

$$\text{Vai concluir} = \begin{cases} \text{Sim} & d \geq \frac{de}{2} \\ \text{Não} & d < \frac{de}{2} \end{cases} \tag{1}$$

Diferente dos marcos anteriores q_2 não tem relação direta com o *deadline* uma vez que ele é um marco de evento de término da tarefa, ou seja, para cada execução da tarefa q_0 será sempre em 0% do *deadline*, q_1 será sempre em 50% do *deadline* mas q_2

pode assumir qualquer valor de tempo pois ele indica quando a tarefa terminou. E é nesse marco que é verificado se a resposta do previsor foi condizente com a realidade.

3.2. Algoritmo Proposto

O algoritmo de base é bem simples e justamente por isso surgiram algumas questões sobre ele. A primeira questão foi a existência de um único ponto de execução do cálculo do previsor. Por que não executar esse cálculo em mais pontos para poder detectar mais cedo uma possível perda do *deadline* e consequentemente tomar as ações necessárias para diminuir o impacto dessa perda?

Essa é a primeira alteração incorporada no mecanismo proposto. Ao invés de executar a previsão aos 50% do *deadline* essa verificação é executada 3 vezes ao longo do percurso e do mesmo modo que no algoritmo de base vamos dar um nome para cada marco temporal, de q_0 a q_4, onde q_0 é o início da tarefa, q_1, q_2, e q_3 são os marcos onde serão executados o cálculo de previsão, sendo eles respectivamente em $\frac{1}{5}$, $\frac{2}{5}$ e $\frac{3}{5}$ do *deadline*, e q_4 o fim da tarefa. Chamemos essa fração de $\frac{1}{5}$ do *deadline* de f_d para futuras referências

Pensando dessa forma foi adotado um sistema de histórico de tarefas para armazenar informações relevantes ao trajeto. Essas informações então serão consultadas e compiladas para fornecer as informações necessárias para o previsor dar uma resposta condizente com aquela tarefa no estado em que o ambiente se encontra. Para esse trabalho o histórico vai possuir informações essenciais para o funcionamento do previsor e algumas informações que serão utilizadas apenas para métricas de comparação entre o mecanismo de base e o proposto. As informações essenciais são as seguintes:

- Um identificador da tarefa;
- A data de início e fim da tarefa;
- A distância percorrida acumulada em cada marco temporal (q_1, q_2, q_3 e q_4);

Agora que foram identificados os pontos chaves do mecanismo podemos definir formalmente o seu funcionamento, ou seja, como ele irá se comportar matematicamente, definindo as fórmulas e equações utilizadas nos cálculos que serão executados. Primeiramente vamos começar definindo alguns símbolos.

Comecemos definindo d_{q_i} como a distância percorrida até o marco q_i, com i variando de 1 a 3. Assim podemos também definir $\overline{d_{q_i}}$ como a média dos d_{q_i}'s. Similarmente ao mecanismo de base então podemos verificar se no marco em questão a distância percorrida pela tarefa é maior ou igual à média daquele marco, ou seja, a diferença Δd_{q_i} é maior ou igual a 0 de modo que:

$$\text{Previsão em } q_i = \begin{cases} \text{Vai Concluir} & \text{se } \Delta d_{q_i} \geq 0 \\ \text{Não vai concluir} & \text{se } \Delta d_{q_i} < 0 \end{cases} \tag{2}$$

Mas somente essa verificação precisa ser aprimorada para ficar mais robusta e por isso iremos introduzir uma segunda parte na equação 2. Essa alteração contempla a média das velocidades $\overline{v_{q_j}}$ em cada marco subsequente ao seu. Podemos escrever essa média das velocidades médias v_{m_i}:

$$v_{m_i} = \left(\sum_{j=i}^{4} \frac{\overline{d}_{q_{j+1}} - \overline{d}_{q_j}}{\Delta t} \right) / (4 - i) \tag{3}$$

onde Δt representa o tempo gasto desde a última verificação e tem como valor a seguinte função:

$$\Delta t = \begin{cases} f_d & \text{se } i \in [1, 2] \\ \\ (\overline{fim} - \overline{inicio}) - 3 * f_d & \text{se } i = 3 \end{cases} \tag{4}$$

Como o mecanismo se baseia muito em médias a quantidade de referências a serem consultadas influencia no seu funcionamento. Se tivermos poucas amostras no histórico uma pequena variação nos dados gera uma diferença muito grande nos valores de médias e se tivermos uma quantidade muito grande de amostras uma variação nos dados terá pouco impacto. Como decidir então qual o tamanho ideal do tamanho desse histórico? Para responder essa pergunta foi pensando em comparar a execução do mecanismo com diferentes tamanhos de histórico.

Para testas os resultados com diferentes tamahos de histórico foram executadas 4 simulações com diferentes tamanhos de histórico, sendo eles 25, 50, 100 e 200 entradas, mas para fins de estatística o banco de dados irá conter mais entradas do que essas quantidades.

Sendo assim, um novo registro sempre irá sobrescrever o registro mais antigo fazendo com que os dados representem sempre o estado mais recente do ambiente. Isso faz com que o mecanismo tenha um certo nível de adaptabilidade ao ambiente. Por fim ficou estabelecido que as características principais do mecanismo são:

- A utilização de vários pontos de verificação
- A utilização de um histórico para utilização nos cálculos
- Um tamanho fixo de entradas no histórico

4. Simulação e Resultado

Primeiramente foi preparado o ambiente de testes mapeando o terceiro andar do Departamento de Informática e Estatistica da Universidade Federal de Santa Catarina, utilizando o robô Pioneer P3-DX[MobileRobotics 2014] em conjunto com o laser SICK[SICK 2014] e o software Mapper3.

Uma vez tendo o mapa processado foram definidos objetivos no mesmo cujo o robô teria que alcançar, como mostra a Figura 1. Pronto o ambiente foram feitas as coletas de dados e dimensionamento dos *deadlines*das tarefassendo elas o deslocamento do ponto LAPESD(A) até ELEVADOR(B) e o trajeto contrário, com as seguintes distâncias e tempos obtidos.

- Tempo médio de A até B = 88,0972 segundos;
- Distância média de A até B = 33410,6395 milímetros;
- Tempo médio de B até A = 80,4687 segundos;

Figure 1. Mapa do terceiro andar do INE

- Distância média de B até A = 33198,0971 milímetros;

As 4 simulações foram executadas utilizando o simulador MobileSim do seguinte modo: um robô executava as tarefas descritas com os dois mecanismos apresentados enquanto outro vagava aleatoriamente pelo mapa gerando alterações na rota prevista pelo primeiro robô.

Os resultados obtidos indicam que o previsor proposto tem um desempenho superior ou igual ao mecanismo de base, dependendo da folga do *deadline*, como mostram a Figura 2, Figura 3 e Figura 4. Esse padrão se repetiu para todos os tamanhos de histórico.

A variação do tamanho histórico se mostrou de baixa relevância, com omostra a Tabela 1.

5. Considerações Finais

Neste trabalho foi apresentado um mecanismo de previsão temporarl de navegação e sua implementação. Seu objetivo é indicar se um robô irá conseguir completar sua tarefa dentro do prazo estupulado. Com esses objetivos cumpridos ficou claro que para uma tarefa com um *deadline*folgado o previsor tem um desempenho pior do que o algoritmo de base e adiciona características indesejadas. Porém com um deadline mais justo o trabalho proposto apresenta uma performance superior com o efeitos colateral de maior quantidade de erros indesejados.

Como melhoria deste trabalho sugere-se uma reformulação dos cálculos para uti-

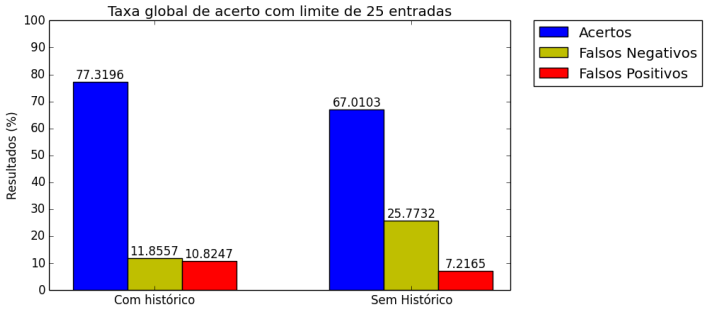


Figure 2. Taxas de acerto e erro com histórico de 25 entradas

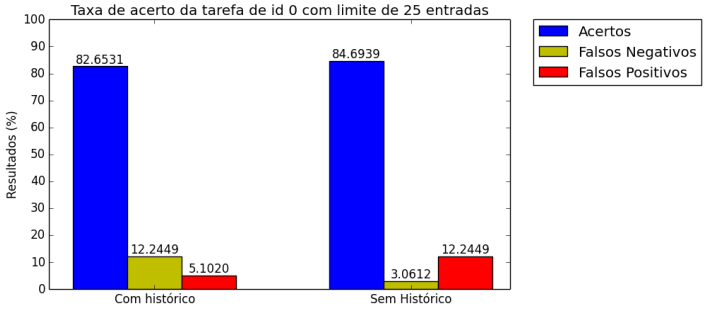


Figure 3. Taxas de acerto e erro da tarefa 0

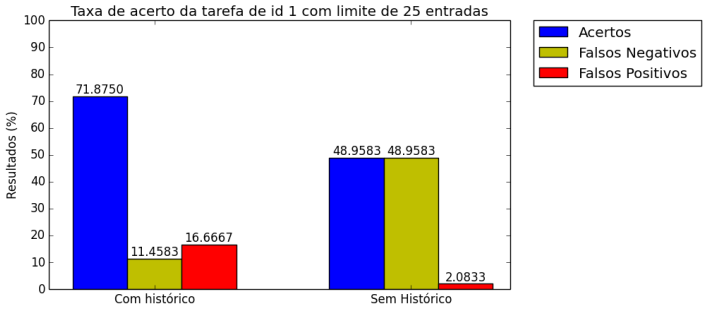


Figure 4. Taxas de acerto e erro tarefa 1

Tamanho	Tarefa	Taxa de acerto (TA)	Falsos Positivos (FP)	TA / FP
25	0	82,6531	5,1020	16,2000
	1	71,8750	16,6667	4,3125
50	0	81,6327	12,2449	6,6666
	1	72,0000	14,0000	5.1428
100	0	78,0220	16,4835	4,7333
	1	70,6522	26,0870	2.7084
200	0	86,9347	8,0402	10,8125
	1	70,6468	9,4527	7.4737

Table 1. Relação do tamanho do histórico com as taxas de acerto e falsos positivos

lizar o conjunto de dados proposto. Também seria de grande valia comparar esse mecanismo com alguma técnica de inteligência artificial que se aplique nas mesmas condições.

References

Asimov, I. (1950). *I, Robot*. Robot series. Bantam Books.

Michaud, F. and Matari, M. (1998). Learning from history for behavior-based mobile robots in non-stationary conditions. *Machine Learning*, 31(1-3):141–167.

Miura, J., Uozumi, H., and Shirai, Y. (1999). Mobile robot motion planning considering the motion uncertainty of moving obstacles. In *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on*, volume 4, pages 692–697.

MobileRobotics (2014). Especificação pioneer p3-dx.

Nassiraei, A. A. F., Kawamura, Y., Ahrary, A., Mikuriya, Y., and Ishii, K. (2007). Concept and design of a fully autonomous sewer pipe inspection mobile robot "kantaro". In *Robotics and Automation, 2007 IEEE International Conference on*, pages 136–143.

Plentz, P. D. M., Montez, C., and de Oliveira, R. S. (2008). Deadline missing prediction in systems based on distributed threads. *JRI 2008 - EnRI-LARS2008 - 5th Latin American Robotics Symposium*.

SICK (2014). Laser scanners lms2xx / lms200 / indoor / short range.

Siegwart, R. and Nourbakhsh, I. R. (2011). *Introduction to Autonomous Mobile Robots*. Bradford Company, Scituate, MA, USA.

Tsubouchi, T. and Arimoto, S. (1994). Behavior of a mobile robot navigated by an ldquo;iterated forecast and planning rdquo; scheme in the presence of multiple moving obstacles. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 2470–2475 vol.3.