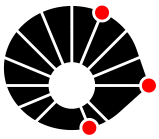
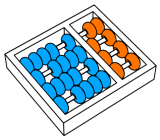




Localização probabilística em robótica

João Guilherme Daros Fidelis *Esther Luna Colombini*

Relatório Técnico - IC-PFG-17-20
Projeto Final de Graduação
2017 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Localização probabilística em robótica

João Guilherme Fidelis* Esther Colombini†

Resumo

O problema de auto localização é fundamental para diversos tipos de robôs autonômos. Os robôs dependem de sua posição local para tomar muitas de suas decisões de quais serão suas próximas ações. Neste trabalho, discutimos o problema de localização voltado para robôs humanoides que participam da tarefa do futebol de robôs da RoboCup. Estudamos o algoritmo probabilístico do filtro de partículas de Monte Carlo e fizemos uma implementação do mesmo que roda num simulador Webots com um modelo virtual do robô Nao da Aldebaran. Chegamos a conclusão que é possível obter boas estimativas de posição, dependendo principalmente do seu modelo de movimentação e modelo de observação. Também mostramos que aumentar o aumento do número partículas do filtro diminui o erro das estimativas.

1 Introdução

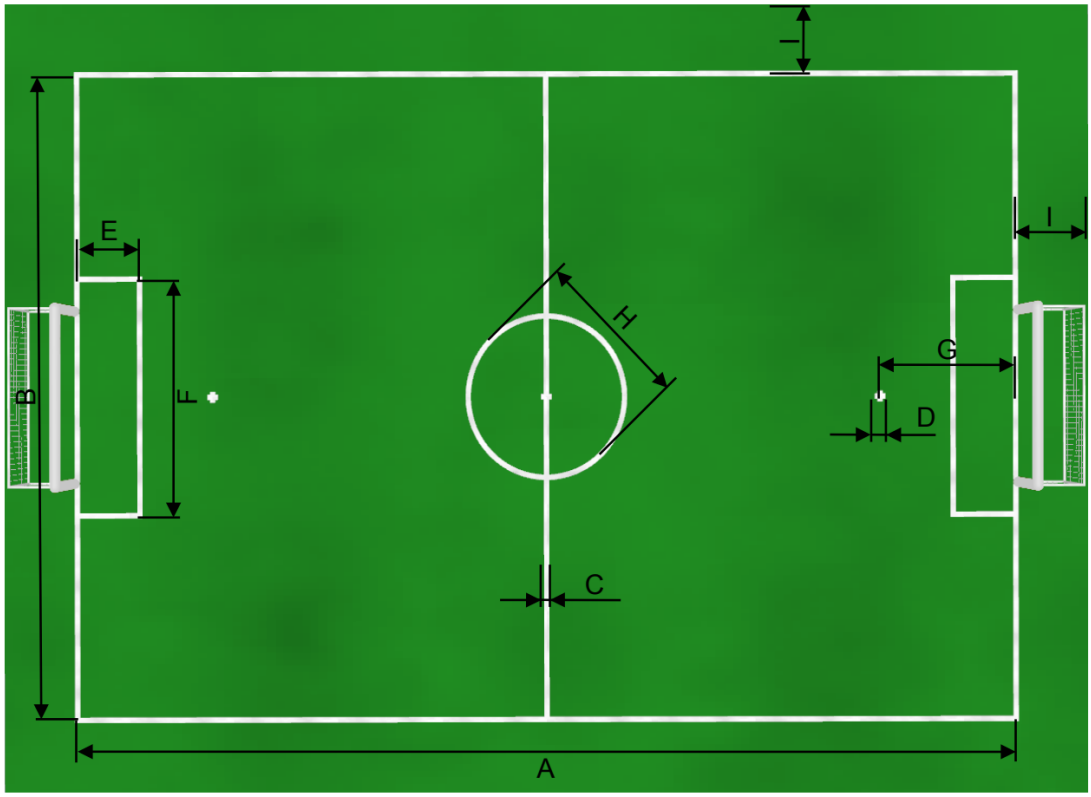
1.1 A RoboCup

A RoboCup[3] é uma competição mundial de robótica que acontece anualmente com o objetivo de promover a robótica e a pesquisa em inteligência artificial ao oferecer um problema desafiador e ao mesmo tempo interessante para o público. O grande objetivo da competição é que até o meio do século XXI, um time de robôs consiga jogar de igual para igual com o time de futebol que for campeão da Copa do Mundo FIFA com as regras do futebol profissional.

Suas competições envolvem não apenas futebol mas também competições que vão desde o regaste de vítimas, tarefas assisitivas até a de danças entre robôs. A liga que envolve futebol é a RoboCup Soccer. Dentro dessa liga há outras sub-ligas, incluindo a a Standard Platform League. Nessa liga, os times utilizam um robô padrão e apenas focam em desenvolver o software utilizado por ele. Desde 2009, apenas robôs Nao, desenvolvidos pela empresa Aldebaran são utilizados. A outra liga, a Humanoid League[4], é dividida em três categorias: *kids* (40-90cm de altura), *teens* (80-140cm de altura) e *adult* (130-180cm de altura). Atualmente, é a liga que está mais perto do grande objetivo de jogar contra jogadores profissionais.

*Inst. de Computação, UNICAMP, 13083-852 Campinas, SP. ra136242@students.ic.unicamp.br
†Inst. de Computação, UNICAMP, 13083-852 Campinas, SP. esther@ic.unicamp.br

Figura 1: Dimensões do campo usado na Standard League da RoboCup



ID	Description	Length (in mm)	ID	Description	Length (in mm)
A	Field length	9000	E	Penalty area length	600
B	Field width	6000	F	Penalty area width	2200
C	Line width	50	G	Penalty cross distance	1300
D	Penalty cross size	100	H	Center circle diameter	1500
			I	Border strip width	700

Os robôs Nao (figura 2) são robôs de formato humanoide que andam sobre duas pernas. A versão mais atual tem 58 centímetros de altura e pesa 4.3 quilos. Ele tem vários motores pelo corpo que funcionam como suas juntas, permitindo que ele realize movimentos complexos, como dar passos, andar, chutar, sentar, levantar, etc.

As regras do futebol de robôs [5] ainda são muito diferentes do futebol profissional de humanos, porém, a cada ano, pequenas mudanças são feitas para deixar ambos mais parecidos. Por exemplo, antigamente, as traves dos gols tinham cores diferentes de cada lado do campo, para facilitar sua diferenciação. Hoje, todas já são brancas. A bola antigamente era laranja, hoje já também é branca (mas ainda é bem menor que uma bola de futebol profissional), essas duas pequenas mudanças já forçam os times a adotarem outras estratégias. O campo tem as dimensões 9 metros de largura e 6 metros de altura. Os gols tem altura

Figura 2: Robô humanóide Nao

de 0.8 metro e 1.6 metro de largura (ver mais detalhes do campo na figura 1).

Alguns dos grandes desafios de robôs humanoides são a movimentação, andar, correr, levantar, chutar, passar a bola, tudo isso mantendo o equilíbrio.

Um dos problemas mais importantes da RoboCup é o de localização, que permite ao robô estimar sua posição no campo, assim como a da bola e outros elementos que podem ser interessantes. Com base na sua posição e orientação, ele pode tomar decisões importantes, por exemplo, um robô pode se comportar de maneira diferente se souber que ele está em seu campo de defesa e que um oponente tem a bola.

A proposta deste trabalho é estudar técnicas e algoritmos de localização probabilísticos comumente utilizadas e conhecidas e aplicar alguns conceitos num experimento prático simulado no simulador Webots [2]. Essas técnicas são muito utilizadas nos robôs autônomos pois dependem apenas de componentes já presentes nele (não precisa de GPS, por exemplo) e tem um bom desempenho computacional.

2 Justificativa

O problema de localização é imprescindível para robôs autônomos, mesmo para os que tem funções completamente diferentes dos robôs da RoboCup. Na RoboCup, esse problema é extremamente importante para que a tomada de decisão seja corretamente realizada. O robô deve saber onde está para saber se deve defender, chutar, se está chutando contra o próprio gol, se está no campo e se foi sequestrado por cometer uma infração, deve poder voltar a compreender o mundo ao seu redor de forma autônoma. Como podemos observar,

é uma funcionalidade importantíssima para a competição e por isso, muitos times gastam muito tempo de desenvolvimento nesse módulo.

3 Objetivos

O objetivo desse trabalho é estudar e entender o problema da localização localização no contexto da RoboCup e implementar o algoritmo mais utilizado, o filtro de partículas de Monte Carlo levando em conta o baixo poder de processamento do robô comparado a outras arquiteturas.

3.1 Definições e Conceitos

3.1.1 O Problema de Localização

Os algoritmos de localização probabilísticos fornecem uma estimativa da posição do robô. Sem a disponibilidade de um GPS, é impossível termos as coordenadas exatas de um sistema no mundo, mas com algoritmos já bastante conhecidos e testados, é possível obter aproximações muito boas.

Para obter bons resultados o robô precisa principalmente de dois módulos funcionando muito bem: movimentação e observação. Ele precisa ter um bom modelo de movimentação e também uma maneira eficaz de extrair características do ambiente ao seu redor e estimar a distância de possíveis pontos de referência (chamados de *landmarks*).

3.1.2 Modelo de Movimentação

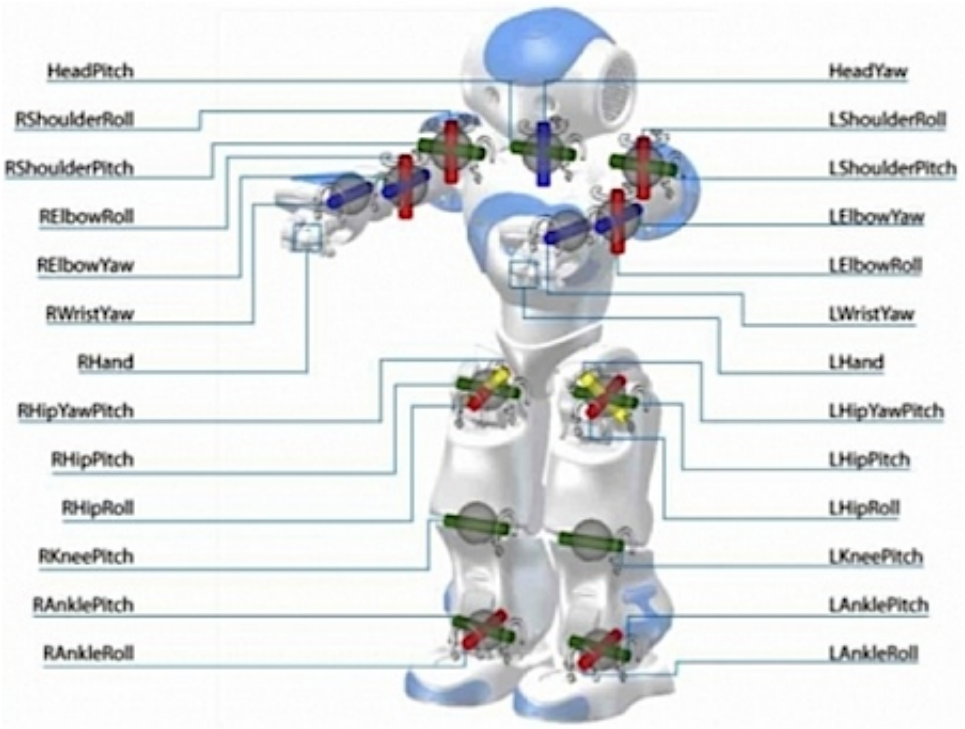
O modelo de movimentação é o componente do robô que dado certo movimento executado pelo robô, estima o seu deslocamento. Para o caso do nosso simulador, ele daria o deslocamento do robô nas coordenadas x, y e θ, ou seja, o deslocamento nos eixos e em rotação.

Para robôs de rodas, esse problema é relativamente simples, porém, para robôs humanoides, ele se torna bem mais complexo. O Nao é um robô humanoide com um conjunto de servomotores que formam e simulam suas juntas (ver figura 3 com os motores disponíveis). Quando o robô quer se locomover, eles aceleram em determinada direção em tempos pré-calculados e fazem o movimento correto para o robô andar. Uma sequência desses movimentos, permite ao robô levantar a perna, dar um passo, chutar, etc.

Devido a essa complexidade, não é simples fazer um modelo de movimentação para um robô humanoide. Há muitas pesquisas exclusivamente voltadas para essa área. Durante nossa pesquisa, tentamos integrar o modelo de movimentação do time B-Human[1], modelo utilizados por eles no RoboCup em seu robô Nao, porém não obtivemos sucesso, devido a complexidade do código e de integrar Python e C++.

Com o intuito de explorar as características do algoritmo de localização, optamos por utilizar um modelo simplificado de movimento. O modelo criado usa o nó supervisor do *Webots* para acessar as coordenadas absolutas do robô Nao simulado no campo para calcular a variação do deslocamento nos eixos x e y. Uma das grandes dificuldades foi mapear o

Figura 3: Motores disponíveis no Nao



ângulo do robô para o ângulo que desejávamos, para obter corretamente a variação no eixo de rotação do robô e para posteriormente verificarmos os resultados obtidos. Isto foi feito obtendo a matriz de rotação do robô e com ela, pudemos descobrir sua rotação em relação ao eixo-z, que é o desejado. Com ela, podemos obter a tangente do ângulo do robô e utilizar a função arcotangente para, após mais algumas transformações, obtermos o ângulo do robô. Definimos que 0 graus seria o robô olhando para o gol direito (ver figura 4) e uma rotação positiva ocorre é no sentido horário. A cada 300 passos de simulação do supervisor, ele envia uma atualização da odometria, que é um vetor de três posições, contendo um Δx, Δy e $\Delta\theta$ desde o último envio.

3.1.3 Modelo de Observação

Outra peça fundamental no trabalho foi o algoritmo que permite identificar e estimar a distância e o ângulo do robô para pontos de referências do mapa através de sua câmera. Utilizamos como base o trabalho prévio conduzido por Magalhães e Colombini [9], que produz ótimos resultados e é capaz de nos informar a distância para o *landmark*, o ângulo do robô para ele e qual o *landmark* (trave esquerda, trave direita ou canto de escanteio). Essa terceira informação é importante para podermos calcular a distância esperada e o ângulo esperado corretamente para *landmark* achado. O único problema do algoritmo é o número de falsos positivos que aparecem e por vezes fazem o erro da estimativa da posição mudar muito, pois achamos haver um *landmark* onde não há. Na figura 5 podemos observar

Figura 4: Robô com rotação de 0 graus de acordo com nosso referencial

um exemplo do momento em que o algoritmo detectou a trave esquerda e direita, já nos fornecendo distância e ângulo para cada. Na figura 6, vemos alguns exemplos de falsos positivos perto do círculo central.

4 Revisão Bibliográfica

O tema de localização de robôs é muito frequente e comum entre robôs autônomos. Como diferentes robôs com outros propósitos tem diferentes sensores disponíveis e variados poderes de processamento, o número de fontes foi bastante restringida. Na RoboCup, times utilizam diversos algoritmos diferentes.

Sobre o tema específico de localização na RoboCup, Nikolaos Kargas [8] fez um trabalho extenso sobre vários algoritmos e sobre o problema de localização em si. Neste trabalho, o autor estuda diferentes tipos de filtros como o de Monte Carlo, o filto de Kalman e o filtro de Kalman Estendido. Após discutir os três métodos, ele discorre sobre o filtro utilizado em seu robô, que acabou sendo um Filtro de Kalman Estendido pois é mais eficiente e permite uma integração dos estados do robô e da bola em um estado compartilhado global de maneira mais fácil. O algoritmo proposto usa Acompanhamento de múltiplas hipóteses onde, quando o robô faz uma previsão de *landmark*, ele pode não ter certeza de qual *landmark* é, criando uma árvore de possibilidades para as possíveis observações do robô. Cada hipótese gera um filtro separado que considera uma pose diferente do robô e uma distribuição Gaussiana diferente. Kargas também estuda as técnicas utilizadas por alguns times e menciona que na época, o B-Human utilizava um filto Monte Carlo com otimizações, como a Particle Swarm

Figura 5: Modelo de Observação detectando as traves esquerda e direita

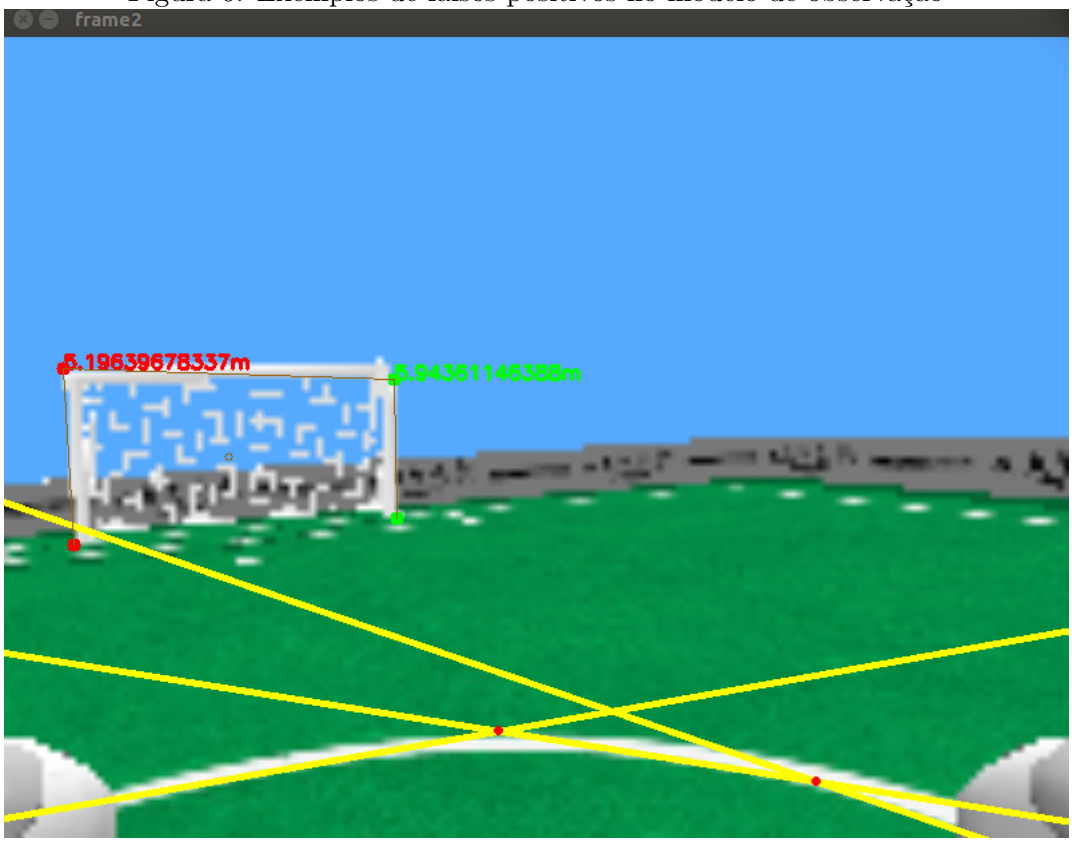
Optimization, utilizando a técnica desenvolvida por Burchardt, Laue e Röfer [7], apesar de hoje em dia utilizar um filtro de Kalman Estendido.

Strasdat, Bennewitz e Behnke [10] estudaram o algoritmo de Monte Carlo, porém com um modelo de observação bem mais robusto. O modelo por eles proposto conseguia extrair as linhas laterais do campo, o círculo central, utilizando a bússola para eliminar ambiguidades. É interessante que eles conseguem dar um valor de confiança para cada observação feita, assim, observações em que o modelo tem mais certeza que estão corretas, tem valores de confiança maiores e contribuem mais para o cálculo dos pesos das partículas. Ele chega em resultados bastante satisfatórios em seus trabalhos.

No trabalho conduzido por Aguias, Máximo e Pinto[6] uma versão padrão do filtro de partículas foi inicialmente utilizada. Entretanto, os autores identificaram que o algoritmo clássico padrão é muito suscetível a falsos positivos, outros erros do modelo de observação e tem problemas para lidar com o "sequestro"do robô. Isso ocorre pois se obtivermos um erro do modelo de observação, partículas ficarão com o peso errado e após o passo de *resample*, as partículas erradas sobrevivem com mais chance e tendem a guiar as estimativas para um valor errado, até o algoritmo se ajustar novamente com o tempo e mais estimativas corretas. Eles fizeram modificações para dar probabilidades às observações vindas do modelo de observação e criaram memórias curtas e longas no algoritmo original, armazenando na memória aquelas observações mais confiáveis, prevenindo que um erro recente do modelo de observação mexa muito nos valores estimados atuais. Porém, o algoritmo não leva em conta landmarks ambíguos, o que é uma limitação e deveria ser adaptado antes de se colocar num robô de verdade.

Os trabalhos [6] e [10] utilizaram funções similares para atualizar o peso de suas partículas, porém com uma pequena diferença. Enquanto os primeiros utilizam o quadrado da diferença entre os valores observados e calculados, os últimos utilizam apenas a diferença absoluta entre eles. A abordagem do grupo do ITA dá mais peso para partículas boas e piora o peso de partículas ruins, o que é algo que pode fazer diferença.

Figura 6: Exemplos de falsos positivos no modelo de observação



5 Desenvolvimento do Trabalho

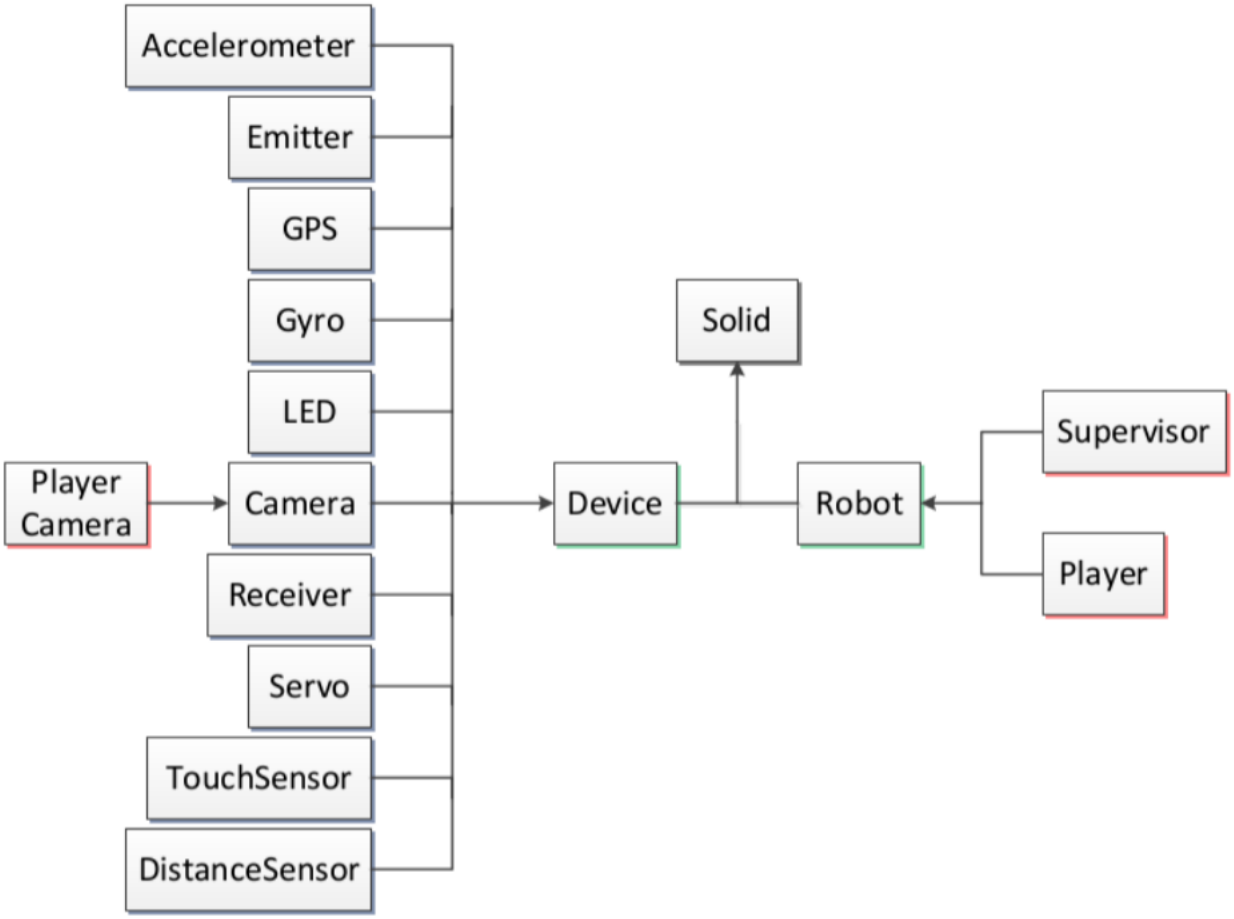
5.1 Configurações do Ambiente

O simulador escolhido para realizar os experimentos foi o *Webots*, da *Cyberbotics*. No ambiente criado, há uma representação do campo de futebol, da bola e do jogador, todos simulados com tamanho real. O *Webots* funciona através de nós, tendo cada nó propriedades próprias que os descrevem e essas propriedades podem ter outros nós. Por exemplo, o nó *Robot*, que é derivado da classe *Solid* (classe que contém propriedades para objetos físicos), tem um nó para sua câmera, para seu GPS, etc. É a partir desses nós que nós temos acesso a dados como a matriz de imagens da câmera. A classe *Solid* tem propriedades que dizem sua posição no mapa e sua rotação. É com um nó supervisor que conseguimos acessos a essas propriedades e podemos calcular a variação em x, y e θ, criando um modelo de movimentação básico.

O campo tem coordenadas que vão de -4.5 a 4.5 no eixo x (refletindo os 9 metros de largura do campo) e no eixo z do *Webots* (que para nós seria chamado de eixo y comumente), vai de -3 a 3.

Podemos observar na figura 7 um exemplo do sistema de nós do Webots e entender

Figura 7: Exemplo de estrutura de nós do Webots



melhor como o robô Nao é composto e simulado no programa.

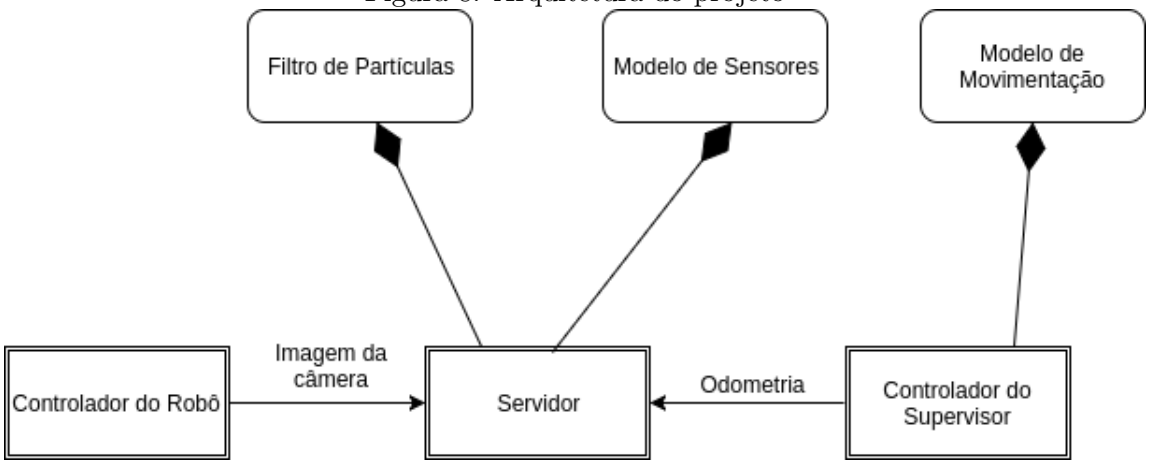
5.2 Arquitetura do Ambiente

Em nosso sistema, temos três arquivos em Python importantes. O primeiro é o controlador do robô, que faz ele andar e tem acesso à câmera. O segundo é o controlador do supervisor que funciona como o modelo de movimentação, enviando ao servidor de imagens a odometria a cada 300 passos de simulação.

O terceiro e mais importante é o servidor de imagens, que contém o modelo de observação, que identifica os pontos de referência e estima o ângulo e a distância para eles, e o filtro de partículas. Ao receber uma odometria do controlador do supervisor, ele aplica as diferenças descritas pela odometria em todas as partículas e continua calculando seus pesos ao ver pontos de referências novos.

O servidor escuta uma porta TCP e espera a conexão dos dois clientes. Ao se conectarem,

Figura 8: Arquitetura do projeto



ele abre uma thread para cada cliente e trata dos pacotes provenientes deles nela. O controlador do supervisor envia de tempos em tempos a odometria e o ângulo atual do robô, enquanto o controlador do robo envia a matriz de imagem que vem de sua câmera e as coordenadas de seu GPS. O servidor recebe essas dados e processa-os.

A figura 8 apresenta os componentes, como se ligam e o que é enviado de um para o outro.

5.3 Filtro de Partículas

O algoritmo de filtro de partículas é, em essência, um filtro Bayesiano onde um número N de partículas, que representam estados aleatórios inicias, é criado. Neste trabalho, os estados de cada partícula são (x, y, θ). Cada patícula também tem um peso, que varia de acordo com a crença de que aquela partícula representa a melhor estimativa de estado ou não, dependendo das observações do modelo de observações. Na geração da partícula, é escolhido um valor aleatório para cada uma dessas variáveis, sendo x no intervalo [-4.5, 4.5], y no intervalo [-3. 3] e θ no intervalo [0, 360). Distribuímos as partículas de forma uniformemente aleatória pelas possíveis coordenadas do campo.

Inicialmente, N partículas foram geradas com o mesmo peso, um. Ao recebermos uma odometria vinda do modelo de movimentação, o deslocamento descrito pela odometria no estado (x, y, θ) é aplicado a todas as partículas.

Neste projeto, consideramos três tipos de pontos de referência (landmarks): trave esquerda, trave direita e canto de escanteio. Quando o modelo de observação detecta algum *landmark*, o mesmo é inserido em uma lista com a distância observada, o ângulo observado e o tipo do *landmark*.

No filtro de partículas, para cada partícula nós calculamos a distância esperada, utilizando uma equação de distância entre pontos simples, e o ângulo esperado. O ângulo esperado é calculado através da equação do ângulo entre dois vetores:

$$x = \frac{\vec{v} \cdot \vec{u}}{\|\vec{v}\| \cdot \|\vec{u}\|} \tag{1}$$

$$\theta = \arccos(x) \tag{2}$$

Para o nosso problema, adotamos um vetor onde a primeira posição é o seno do ângulo que a partícula tem e a segunda posição é o cosseno desse ângulo. O segundo vetor é o vetor da diferença entre a posição do *landmark* para a posição (x, y) da partícula (um vetor que aponta da partícula para o *landmark*). Utilizando as equações (1) e (2), podemos calcular o ângulo esperado entre a partícula e o *landmark*.

Com a distância observada, ângulo observado, distância calculada e ângulo calculado, podemos utilizar a equação de Strasdat, Bennewitz e Behnke[10] para obtermos o peso de uma partícula, dada uma observação e o seu estado.

A equação utilizada é:

$$p_{landmark} = \exp(-\frac{\|d_o - d_e\|}{2}) \cdot \exp(-\frac{\|\theta_o - \theta_e\|}{2}) \tag{3}$$

Sendo d_o a distância observada, d_e a distância esperada, θ_o o ângulo observado e θ_e o ângulo esperado

Esta função é bastante apropriada para o nosso propósito pois quanto menor é a diferença entre os resultados esperados e observados, mais perto de um chega a função. Quanto maior é a diferença, o peso da partícula cai de forma exponencial. Após calcularmos o peso para cada *landmark* observado, combinamos os pesos calculados através de:

$$w_i = \prod p_{te} \cdot p_{td} \cdot p_{esc} \tag{4}$$

Sendo w_i o peso de uma partícula, p_{te} o valor associado daquela partícula à trave esquerda, p_{td} o valor calculado para a trave direita e p_{esc} o valor daquela partícula para o *landmark* de escanteio. Se algum *landmark* não foi visto, ele fica com valor um, não afetando o produtório.

Finalmente, com o peso das partículas definido, fazemos um *resample*, que é um passo de extrema importância no filtro, responsável pela resultado da crença a posteriori. Nele, as partículas são salvas e continuam no filtro de acordo com seu peso, porém de forma aleatória. Aquelas com maior peso, tendem a se manterem vivas. As que não são salvas podem ser geradas aleatoriamente novamente ou, como no nosso algoritmo, são substituídas por cópias de partículas já existentes no filtro, que provavelmente tem um peso maior. Esse passo é muito importante pois ajuda com que partículas mais corretas (de maior peso) sobrevivam em maior número e assim, o resultado da estimativa do estado do robô tende a convergir para um valor mais correto e de forma mais rápida. A estimativa de pose no final é a média ponderada dos estados (x, y, teta) de todas as partículas. O pseudo-código para o algoritmo de localização de Monte Carlo está descrito no Algoritmo 1.

Em nosso pseudo-código (Algoritmo 1), a função $draw\_random\_particle()$ desenha uma partícula com os valores (x, y, θ) distribuídos de forma uniforme pelo campo.

A função $calculate\_weight\_for\_landmark(d_o, \theta_o, tipo)$ utiliza a equação 3 para calcular o peso de tal partícula de acordo com o *landmark* encontrado. A função de *resample* recebe

Algorithm 1 Algoritmo de Monte Carlo

```
1: procedure SIMPLE_MCL
2:     Inicialização
3:     for todas as partículas p em M do
4:         p = draw_random_particle()
5:     Calculando Pesos
6:     for L em landmarks vistos do
7:         d_o = L[0]
8:         θ_o = L[1]
9:         tipo = L[2]
10:        for todas as partículas p em M do
11:            p.peso *= calculate_weight_for_landmark(d_o, θ_o, tipo)
12:    peso_total = 0
13:    for todas as partículas p em M do
14:        peso_total += p.peso
15:    Resample(peso_total, M)
```

a soma total de pesos do filtro e modifica o vetor de partículas M, sendo aquelas com maior peso as com maiores chances de sobreviverem.

5.4 Diferenciação de Landmarks

O algoritmo do modelo de observações nos diz apenas a distância observada, ângulo observado e o tipo do *landmark*, mas não qual é exatamente. A partir do momento que temos duas traves esquerdas idênticas, duas traves direitas idênticas e quatro escanteios iguais, temos muitos problemas de ambiguidade relacionados a simetria do campo.

Duas abordagens foram utilizadas para tratar deste problema. Uma deles foi, antes de calcular os pesos para cada partícula, fazer um sistema de votação onde testávamos a distância calculada de cada partícula para os dois *landmarks* possíveis (por exemplo, trave esquerda do gol "esquerdo"e trave esquerda do gol "direito") e o *landmark* vencedor seria o que mais partículas tivessem uma distância mais próxima da distância observada pelo modelo de observações. Se das 100 partículas 51 votassem que o gol correto que o robô está olhando é o direito, utilizaríamos ele como *landmark* para o cálculo do peso de todas. Porém, isso levou a siatuação onde, na primeira iteração, quando as partículas estavam distribuídas de forma aleatória pelo mapa, se o gol errado vencesse a votação, muitas partículas que eram muito boas e estavam de fato na posição mais correta ficavam com pesos errados e no passo de resample apenas partículas "ruins"sobreviviam e o a estimativa nunca convergia para valores bons.

Na segunda estratégia, a mais simples, os resultados foram melhores. Se sabemos que o *landmark* é um escanteio, por exemplo, testamos a distância da partícula para cada possível posição do escanteio no mapa e guardamos a distâcia calculada mais próxima da observada. Também utilizamos o ângulo calculado para esse *landmark* escolhido por esse critério. Isso pode gerar algumas inconsistências pois o gol encontrado para uma partícula pode não ser

o mesmo que o de outra, para o mesmo *landmark*.

6 Experimentos e Resultados

Os testes foram automatizados para evitar que entradas diferentes produzissem resultados muito diferentes. Mesmo que com as mesmas entradas, devido a rotação do robô de forma involuntária quando andando para frente, isso não pode ser controlado e como é variante e imprevisível, não era possível produzir uma combinação de movimentos idênticos mesmo com as mesmas entradas.

Os experimentos foram feitos com o robô partindo da coordenada (0, 0), ou seja, no círculo central, centro do campo, com diferentes rotações iniciais (no sentido horário, definido previamente): 11 graus, 45 graus e 90 graus. Devido a problemas no OpenCV usado pelo Modelo de Observação, muitas vezes *crashes* prematuros aconteciam, então usamos apenas dados para o qual conseguimos pelo menos 30 atualizações de pesos no vetor de partículas (ou seja, o peso de cada patícula foi atualizado 30 vezes diferentes). Variamos também o número de partículas geradas pelo filtro para vermos se teríamos algum impacto na precisão das estimativas.

Para cada diferente valor de rotação inicial e número de partículas diferentes foram conduzidas 3 simulações com no mínimo 30 atualizações no filtro de partículas. Em cada atualização, o valor da estimativa de posição atual, contendo o (x, y e teta) estimado e os valores atuais do robô (x, y, teta) foi armazenado para futura comparação. Para cada um desses estados, observamos a diferença entre o estimado e o valor real e calculamos uma média das diferenças de todas as medições dos três testes para aquele grau inicial e número de partículas.

Partículas	Erro Médio X	Erro Medio Y	Erro Médio θ	DP Erro X	DP Erro Y	DP Erro θ
100	0.515452	1.941689	63.966194	0.303512	0.344193	41.001778
500	0.318755	0.482954	31.977962	0.150749	0.304204	29.391581
1000	0.108255	0.430408	27.779501	0.109885	0.318679	34.625565

Tabela 1: Dados para testes com $\theta_i = 11$ graus

Na tabela 1 observamos que os erros de X, Y e θ caíram drasticamente conforme aumentamos o número de partículas. O erro médio caiu cerca de cinco vezes em X, quase cinco vezes em Y e um pouco mais de duaz vezes em θ, se compararmos os filtros com 100 e 1000 partículas.

Partículas	Erro Médio X	Erro Médio Y	Erro Médio θ	DP Erro X	DP Erro Y	DP Erro θ
100	0.369013	1.411293	70.197061	0.434384	0.708392	55.308130
500	0.523084	1.934724	49.804047	0.395546	0.941398	50.539249
1000	0.513263	2.236002	38.360232	0.403360	0.888961	28.253801

Tabela 2: Dados para testes com $\theta_i = 45$ graus

Já nas tabelas 2 e 3, para rotação inicial de 45 graus e 90 graus, não notamos uma grande melhora, de certa forma, houve piora em algumas métricas, conforme o número de

Partículas	Erro Médio X	Erro Medio Y	Erro Médio θ	DP Erro X	DP Erro Y	DP Erro θ
100	0.766046	1.243715	75.771592	0.747579	0.563879	35.592377
500	0.261203	1.727628	83.040452	0.277899	1.118833	25.404026
1000	0.385618	1.634717	67.375540	0.381629	1.408556	46.374294

Tabela 3: Dados para testes com $\theta_i = 90$ graus

partículas aumentou. Isso ocorre pois nessas duas configurações, o robô leva muito tempo para observar *landmarks* reais e acontece de registrar alguns *landmarks* falsos positivos que poluem os pesos das partículas e após o resample, partículas que não refletem bem o estado atual do robô sobrevivem por estarem erroneamente com o peso alto.

Observamos que para os testes de 45 graus, o erro médio aumentou em todas as variáveis exceto em θ, conforme o número de partículas aumentou, o que foi surpreendente.

Os altos valores de desvio padrão para quase todos os valores são reflexo do fato de que durante os testes, enquanto o robô olhava e encontrava apenas *landmarks* corretos, o erro diminuía e ficava baixo, porém, ao sair da área e ir para lugares onde encontrava falsos positivos, sua estimativa era contaminada e o erro começava a crescer muito. Por isso, os valores encontrados podem ser muito distantes da média, justificando um alto desvio padrão.

Este comportamento com os falsos-positivos ajuda a explicar o aumento do erro médio no caso de 45 graus e 90 graus. Como temos mais partículas, o filtro resiste mais a mudanças. Quando um falso positivo é observado cedo, o filtro fica com partículas com pesos de acordo com aquele falso positivo. Ao fazer uma nova observação, por ter mais partículas no filtro, essas novas observações tem menos impacto no filtro do que se ele tivesse menos partículas, pois há mais partículas contribuindo para a média ponderada que nos da nossa estimativa final. Com menos partículas, qualquer mudança faz mais efeito na estimativa, fazendo com que caso seja visto um falso positivo e depois um *landmark* correto, demore menos para o filtor se ajustar com essas informações corretas.

Outra hipótese para a causa do desempenho ruim com essas rotações de 45o e 90o, se comparando com os resultados de $\theta_i = 11$ graus, é que a primeira observação do robô nesse caso é a um escanteio e nosso processo de diferenciação de *landmarks* não é muito robusto, fazendo com que partículas erradas estimem fiquem com pesos altos desde a primeira observação. Foi testada essa hipótese fazendo com que o robô informasse para o filtro de partícula, qual *landmark* exato ele estava vendo (incluindo suas coordenadas), para termos certeza que as partículas calculariam distância esperada e ângulo esperado para o *landmark* correto. Porém essa mudança não trouxe resultados bons, o que refutou essa hipótese. Foi feita uma simulação com um filtro de 500 partículas e rotação inicial de 90 graus e o resultado, visto na tabela 4, indica um erro médio muito grande na rotação e em Y.

Partículas	Erro Médio X	Erro Medio Y	Erro Médio θ	DP Erro X	DP Erro Y	DP Erro θ
500	0.373558	3.692411	169.631442	0.485772	0.480491	25.361045

Tabela 4: Dados para testes com $\theta_i = 90$ graus e robô passando para o filtro qual o *landmark* observado

Para vermos se o desempenho do filtro de 100 partículas é realmente melhor do que o de 1000 partículas quando a rotação inicial é 90 graus, executamos o teste mais vezes e analisamos cada rodada separadamente.

Teste	Erro Médio X	Erro Medio Y	Erro Médio θ
1	1.332009	0.879561	103.594207
2	1.329695	1.304658	54.855036
3	0.062738	1.391539	76.173352
4	0.441256	1.559879	60.188987
5	0.367362	0.139170	15.117687
6	0.234999	2.682297	60.189420
7	0.436979	1.119430	56.190269
8	0.889214	0.761387	99.074575
9	0.567084	1.948849	80.909181
10	0.981894	2.655215	66.939608

Tabela 5: Dados para testes com $\theta_i = 90$ graus e 100 partículas

Teste	Erro Médio X	Erro Medio Y	Erro Médio θ
1	0.199659	1.255442	64.590630
2	0.769619	2.618150	84.789096
3	0.181164	0.911235	46.462462
4	0.307883	0.641667	72.434127
5	0.120552	0.676549	24.847971
6	0.293788	0.996593	47.468835
7	0.237513	0.655006	68.971646
8	0.318901	0.616482	41.827624
9	0.499596	0.725518	34.436462
10	0.344210	0.476304	35.910712

Tabela 6: Dados para testes com $\theta_i = 90$ graus e 1000 partículas

Comparando as tabelas 5 e 6, verificamos que o algoritmo com 1000 partículas funciona melhor, na média. A partir da tabela 7, que contém as médias do valores da tabela 5 e 6, confirmamos que na média, o filtro com 1000 partículas teve resultados melhores.

7 Trabalhos Futuros

Considerando os experimentos realizados verificou-se que o filtro de partículas é viável para aplicações online de localização de robôs autônomos, mas que, seria interessante estudar como técnicas de diferenciação de *landmarks* podem influenciar os resultados obtidos. Também seria útil estudar como o algoritmo consegue se comportar em caso de *kidnapping*,

Partículas	Média do Erro Médio X	Média do Erro Medio Y	Média do Erro Médio θ
100	0.664323	1.444198	67.323232
1000	0.327289	0.957295	52.173957

Tabela 7: Média dos dados das tabelas 5 e 6

o que não tivemos tempo de fazer nesse trabalho. Seria de profundo interesse estudar melhor como esse algoritmo se comportaria com um modelo de movimentação real, com um modelo de observação mais robusto e no robô de verdade.

8 Conclusão

Através deste trabalho, podemos concluir com o algoritmo de Monte Carlo é excelente para conseguir estimativas de posição para o robô Nao na tarefa do futebol da RoboCup. Vimos nos nossos experimentos alguns fatos importantes: quanto mais partículas no filtro melhor são suas estimativas, na média, e que o filtro é bastante sensível a falsos positivos. Também foi possível observar que a qualidade deste algoritmo depende da qualidade e número de landmarks extraídos da cena.

Para obtermos resultados melhores, seria necessário um modelo de obervação mais robusto, que fosse capaz de identificar e estimar distâncias para outros *landmarks*, como as faixas do campo. Isso solucionaria alguns dos problemas de nosso algoritmo que tende a não ter resultados bons enquanto o robô olha para as laterais do campo, longe dos escanteios e gols. Ali também, tendem a aparacer muitos falsos positivos, o que poderia ser melhorado no modelo de observação também.

Antes de usar o algoritmo em um robô, seria bom achar um valor ótimo para o número de partículas em seu filtro. O algoritmo tem desempenho que tem relação linear com o tamanho do vetor, logo, é necessário achar um valor que garanta um tempo de execução bom e um erro aceitável.

O código desenvolvido neste trabalho está disponível em: https://github.com/jgfidelis/Self-Localization-Nao-Robot.

Referências

[1] B-Human Robotics Team Website. `https://www.b-human.de/index.html`. Acessado em: 11/2017.

[2] Cyberbotic's Webots Simulator. `https://www.cyberbotics.com/overview`. Acessado em: 11/2017.

[3] RoboCup Federation. `http://www.robocup.org/`. Acessado em: 11/2017.

[4] RoboCup Humanoid League Website. `https://www.robocuphumanoid.org/`. Acessado em: 11/2017.

[5] RoboCup Standard Platform League 2017 Rules. `http://spl.robocup.org/wp-content/uploads/downloads/Rules2017.pdf`. Acessado em: 11/2017.

[6] Luis Aguiar, Marcos Máximo, and Samuel Pinto. Monte carlo localization for robocup 3d soccer simulation league.

[7] Armin Burchardt, Tim Laue, and Thomas Röfer. Optimizing particle filter parameters for self-localization. In *Robot Soccer World Cup*, pages 145–156. Springer, 2010.

[8] Nikolaos Kargas. *Robust Localization for the RoboCup Standard Platform League*. PhD thesis, Technical University of Crete, Greece, 2013.

[9] Gabriel Magalhães and Esther Colombini. Detecção de objetos no futebol de robôs. Technical Report IC-PFG-17-08, Institute of Computing, University of Campinas, 2017.

[10] Hauke Strasdat, Maren Bennewitz, and Sven Behnke. Multi-cue localization for soccer playing humanoid robots. In *Robot Soccer World Cup*, pages 245–257. Springer, 2006.