Universidade Federal do Rio Grande do Sul
Instituto de Informática
INF01034 - Robótica II
Prof. Renan Maffei

Trabalho Prático 4 - Localização + Planejamento

Prazo: 24/06/2019

Considerações gerais

O trabalho consiste em guiar o robô até um objetivo pré-definido de um ambiente conhecido, no entanto, sem conhecer a pose inicial do robô. Será preciso localizar o robô no ambiente para computar um caminho que leve o robô até o objetivo da forma mais eficiente possível. A localização deverá ser feita usando um filtro de partículas, conforme mostra a figura abaixo.

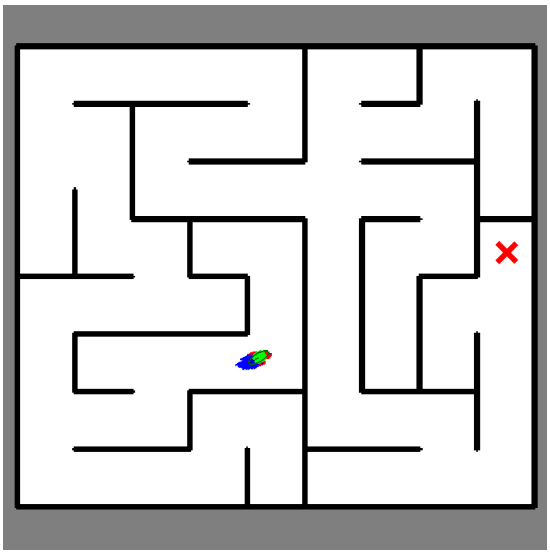

Figura 1: O robô, cuja pose deve ser estimada pela média das partículas, deverá chegar ao objetivo marcado com um X.

O trabalho deverá ser feito preferencialmente em DUPLA ou TRIO (senão INDIVIDUALMENTE).

A avaliação do trabalho será feita em aula através da análise do funcionamento das implementações.

Além de apresentar o funcionamento da implementação daquilo que for pedido, entregue via Moodle um arquivo compactado com os códigos desenvolvidos.

1. Instruções sobre implementação da localização de Monte Carlo no framework

Foi adicionada ao framework utilizado nos trabalhos da disciplina uma classe chamada `MCL` descrita no arquivo `MCL.cpp`. Esta classe é responsável por carregar um mapa do ambiente onde o robô se encontra e executar um filtro de partículas sobre este mapa para resolver o problema de localização global.

Para rodar o programa é preciso passar como parâmetro o nome do mapa (que deverá estar dentro da pasta DiscreteMaps).
Ex: `-m labirinto1.txt`

Exemplo de funcionamento via **terminal**:

```
../build-make/program sim -m labirinto3.txt
```

Exemplo de funcionamento via **QtCreator**:

- vá em **Projects->Build & Run->Run**
- cole em '**Command line arguments**':
  `sim -m labirinto3.txt`

Note que o nome do mapa do arquivo .txt (da pasta DiscreteMaps) passado como parâmetro para o programa deve ser coerente com o arquivo .map (da pasta Maps) aberto no simulador MobileSim.

A versão atual do framework contém 5 modos de visualização do mapa, que podem ser alternados pressionando a tecla 'v'.

0. Mapa do MCL mostrando as partículas e a posição objetivo *(default)*
1. Mapa mostrando a classificação do ambiente, as fronteiras e o menor caminho escolhido
2. Mapa mostrando o campo potencial local
3. Mapa com LOG-ODDS usando LASER
4. Mapa com HIMM usando LASER
5. Mapa usando SONAR

OBS: por enquanto todos os mapas estão em branco (com exceção do mapa do MCL que é conhecido), pois vocês precisam colocar a função de mapeamento que desejam usar.

Na classe MCL, já estão implementadas a inicialização do mapa (método `readMap`), inicialização das partículas (método `initParticles`), desenho do mapa com as partículas (método `draw`) e cálculo da média e covariância das partículas (método `updateMeanAndCovariance`).

- Por padrão o filtro possui 10000 partículas. É possível alterar este número no construtor.
- A visualização das partículas pode (ou não) considerar transparência, o que auxilia na percepção quando muitas partículas ocupam o mesmo local. Para ligar/desligar a transparência, pressione `t`.
- O programa mostra em verde o caminho gerado pela odometria a partir da posição inicial correta. Note que no ambiente simulado, a trajetória está correta, enquanto que no ambiente real, a trajetória vai entortando com o tempo.

Também está implementado o método `run`, chamado a cada instante t pela classe `Robot`, que é composto pelas etapas de **sampling**, **weighting** e **resampling**. Por enquanto, essas três funções estão vazias, logo o objetivo deste trabalho é implementá-las corretamente.

1.1. Implementação da etapa de SAMPLING

Complete a função `sampling` da classe `MCL`.

O objetivo é propagar cada uma das partículas de acordo com o modelo de movimento baseado em odometria visto em aula (Algoritmo 4 da Aula 19). Para isso deve-se usar a odometria informada pelo robô (descrita pela `Action u`) composta por três componentes: `rot1, trans, rot2`.
OBS: a geração dessas três componentes (linhas 1-3 do algoritmo) já é feita pela classe Robot.

Cada partícula deve gerar uma movimentação diferente (aleatória), mas próxima aos valores definidos por `u`. Para isso, primeiro é preciso gerar 3 distribuições normais de média zero e variâncias dadas em função de `rot1, trans, rot2` conforme descritas no algoritmo. Então para cada partícula, deve-se gerar uma amostra de cada uma das 3 distribuições. Por fim, encontra-se a nova posição da partícula usando os valores amostrados de `rot1, trans, rot2`.

DICA: como no início o programa começa com milhares de partículas em todas as posições e direções do mapa, fica difícil de depurar se algo está errado. Portanto, para testar se essa etapa do algoritmo está OK, diminua a quantidade de partículas no construtor (por exemplo, para 10 partículas). Altere a inicialização das partículas em `initParticles`, fixando em ZERO o ângulo de cada partícula. Aí tente analisar se o movimento delas faz sentido em comparação com a trajetória em verde. Se estiver OK, o *sampling* deve estar correto.

1.2. Implementação da etapa de IMPORTANCE WEIGHTING

Complete a função `weighting` da classe `MCL`.

Nesta etapa, o objetivo é avaliar cada uma das partículas, ou seja, determinar a probabilidade de adequação de cada partícula à distribuição esperada.

Parte 1:

A parte mais fácil da pesagem é colocar peso zero para todas as partículas que após a movimentação ficarem sobre alguma célula não-livre. Observe como a checagem é feita na inicialização das partículas (método `initParticles`). Todas as partículas que ficarem com peso zero serão eliminadas na etapa de *resampling*.

Parte 2:

A segunda parte, bem mais complicada, é avaliar quão boa é uma partícula considerando as observações feitas pelo robô, ou seja, computar probabilidades de acordo com o modelo de observação $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t^{[p]}, \boldsymbol{m})$. Na prática, devemos comparar as observações do robô (dadas pelas leituras do laser `z`) e as observações esperadas da partícula na posição $\boldsymbol{x}_t^{[p]}$.

Sabemos que a cada instante o robô faz múltiplas leituras com o laser:

$$\boldsymbol{z}_t = (\boldsymbol{z}_t^1, \boldsymbol{z}_t^2 \cdots, \boldsymbol{z}_t^K)^T$$

Como as leituras são independentes, a probabilidade final associada à particula p pode ser aproximada pelo produto das probabilidades individuais:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t^{[p]}, \boldsymbol{m}) = \prod_{k=1}^{K} p(\boldsymbol{z}_t^k \mid \boldsymbol{x}_t^{[p]}, \boldsymbol{m})$$

E a probabilidade de uma medição individual pode ser definida de acordo com o modelo visto em aula (Basic Beam Model). Aqui vamos simplificar o modelo e dizer que para uma dada observação medida pelo robô $\boldsymbol{z}_t^k$ e a observação esperada $\boldsymbol{z}_t^{k*}$ pela partícula, a probabilidade de semelhança entre as observações é computada por:

$$\mathcal{N}(\boldsymbol{z}_t^k, \boldsymbol{z}_t^{k*}, var) = \frac{1}{\sqrt{2\pi var}} e^{-\frac{1}{2}\frac{(\boldsymbol{z}_t^k - \boldsymbol{z}_t^{k*})^2}{var}}$$

onde var é a variância (dada em m^2) da distribuição associada às medições.

OBS: Você deve definir essa variância. Valores muito altos podem fazer com que o filtro demore a convergir, pois medições diferentes (boas ou ruins) terão pesos parecidos. Também podem fazer com que mesmo após convergência as partículas fiquem muito espalhadas. No entanto, valores muito pequenos farão com que quase todas as medições tenham peso zero, e eliminarão muitas hipóteses boas no início do processo. Esse caso é muito pior, pois o filtro não tem como se recuperar.

$\boldsymbol{z}_t^k$ é o k-ésimo valor do vetor de observações `z`, informado como entrada para a função. Já $\boldsymbol{z}_t^{k*}$, o valor esperado da k-ésima medição feita na posição da partícula p deve ser computado usando ray-casting sobre o mapa. Para poupar trabalho, o framework fornece a função `computeExpectedMeasurement(int index, Pose p)`, então basta informar o indíce da medição que se deseja estimar e a posição da partícula, que a função retorna a distância (em m) que se espera medir no mapa.

DICA: Essa etapa é a mais demorada do filtro. Para acelerar você não precisa computar a probabilidade das 181 medições do laser. Uma dica é pular de 10 em 10, ou de 20 em 20. Ou seja, a probabilidade final pode ser aproximada pela $prob(z_0) \cdot prob(z_{10}) \cdot prob(z_{20}) \cdots prob(z_{180})$

Parte 3:

Por fim é preciso normalizar os pesos das partículas, ou seja, calcular a soma de todos os pesos e dividí-los por essa soma. Note que se a soma der ZERO, a divisão não pode ser feita, nesse caso deixe todas as partículas com o mesmo peso $(1/N)$. Porém isso é um sinal de que a variância no cálculo do peso está muito pequena.

1.3. Implementação da etapa de RESAMPLING

Complete a função `resampling` da classe `MCL`.

Nesta etapa você deve reamostrar o conjunto de partículas, dando maior prioridade para partículas de alto peso. Então gere um novo vetor de partículas selecionando aleatóriamente (com reposição) as melhores partículas do conjunto atual. No final substitua o conjunto atual pelo conjunto reamostrado de partículas.

Implemente o amostrador de baixa variância (algoritmo 4 da Aula 21) pois este é o mais eficiente dentre os métodos vistos em aula.

2. Estratégia de planejamento

É preciso desenvolver uma estratégia de planejamento de movimento para guiar o robô até o objetivo começando de uma posição desconhecida qualquer do mapa. Note que a odometria (da classe `Robot`) que vinhamos usando até então começa sempre da pose (0,0,0), **e ela obviamente não corresponde à pose correta do robô no mapa!** A posição do objetivo dentro do mapa, dada em metros, é conhecida e descrita tanto na variável `goal` da classe `MCL` quanto em `goalPose` da classe `Planning`.

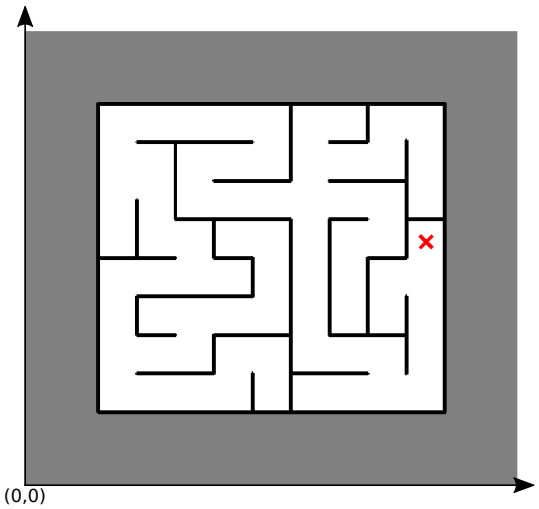

Figura 2: Origem do sistema de coordenadas do mapa de MCL.

O mapa do ambiente usado na localização é descrito em `CellOccType** mapCells` da classe `MCL` e pode, por exemplo, ser acessado na classe `Robot` usando `mcl->mapCells`.

Por fim, a posição média das partículas (também dada em metros, seguindo o mesmo sistema de coordenadas do mapa mostrado na Figura 2) é dada em `Pose meanParticlePose` da classe `MCL`, computada no método `updateMeanAndCovariance`. Esta posição é visualizada em verde claro no mapa do MCL. Quando o filtro de partículas converge para a localização certa, a média das partículas é uma ótima estimativa da posição do robô no mapa.

Além da média, também é computada a matriz de covariância das partículas descrevendo a incerteza da distribuição. A incerteza é visualizada através de uma elipse verde-escuro, cuja orientação é dada por

`covAngle` (obtido a partir dos autovetores da matriz de covariância). Os comprimentos dos eixos maior e menor da elipse são dados, respectivamente, por `covMajorAxis` e `covMinorAxis` (obtidos a partir dos autovalores da matriz de covariância).

Tente desenvolver a estratégia mais eficiente possível, ficando a vontade para explorar todas as informações disponíveis e modificar aquilo que for conveniente no código do framework. A dica inicial é usar como base a exploração implementada no trabalho prático III, mas note que uma exploração simples guiada a fronteiras se mostrará ineficiente.