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

Trabalho Prático 2 - Mapeamento

Prazo: 10/10/2020

Considerações gerais

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

A avaliação do trabalho será feita através da análise do funcionamento das implementações no simulador MobileSim.

Entregue via Moodle um arquivo compactado com os códigos desenvolvidos, e posteriormente o trabalho deverá ser apresentado ao professor em horário a combinar.

Instruções sobre implementação de occupancy grids no framework

O framework disponibilizado neste trabalho contem uma classe chamada `Grid` descrita no arquivo `Grid.cpp`, que dentre outras coisas define funções para desenhar o mapa na tela.
A versão atual do framework contém 4 modos de visualização do mapa, que podem ser alternados pressionando a tecla 'v' ou 'b'.

0. Mapa com LOG-ODDS usando LASER *(default)*
1. Mapa usando SONAR
2. Mapa com HIMM usando LASER
3. Mapa com classificação das células (posteriormente usaremos para planejamento de movimentos)

OBS: por enquanto todos os mapas estão vazios, pois falta implementar as funções de mapeamento.

O grid é composto por células (classe `Cell`) contendo atributos de ocupação. A escala do grid (que indica quantas células correspondem a um metro) pode ser obtida através da função:

```
int scale = grid->getMapScale();
```

Por padrão, a escala é 10, logo se, por exemplo, o robô fizer uma leitura de 5 metros com o laser, o método de mapeamento deverá atualizar uma distância de 50 células.

Para atualizar o mapa (independente do método implementado), é preciso acessar as células dentro do campo-de-visão do sensor que está sendo utilizado, cujo alcance máximo é dado por `base.getMaxLaserRange()` ou `base.getMaxSonarRange()`. Tal valor é dado em metros, logo lembre-se de multiplicar por `scale` para obter o valor correspondente em número de células. Não é preciso checar células mais distantes do robô do que o alcance máximo do sensor, conforme mostra a Figura 1.

Uma posição (x,y) em metros é mapeada para uma célula do grid (i,j), multiplicando-se (x,y) pela escala do mapa. Por exemplo, considerando `scale=10`:

- posição (0.0, 0.0) corresponde à célula (0,0)
- posição (1.0, -2.0) corresponde à célula (10,-20)
- posições (0.30, 0.50) e (0.33, 0.57) correspondem à célula (3,5)

O acesso à célula situada na posição (i, j) é feito atraves da função:

```
Cell* c = grid->getCell(i,j);
```

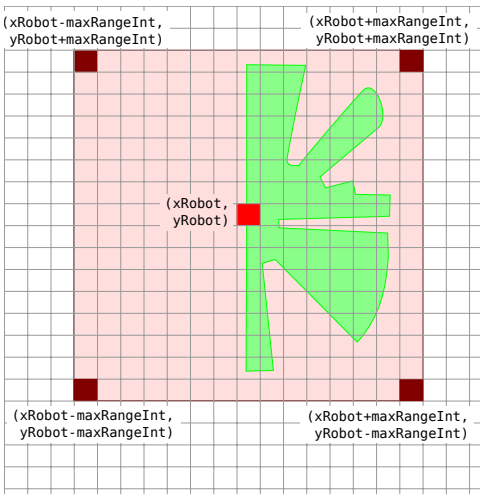


Figura 1: Células a serem atualizadas durante o mapeamento.

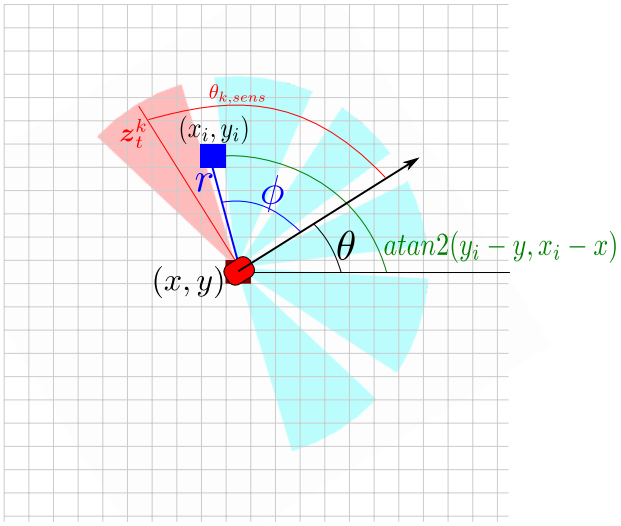


Figura 2: Atualização de uma célula

A atualização da ocupação das células deve ser feita seguindo o modelo inverso de sensor visto na "*Aula 07 - Grades de Ocupação*", exemplificado na Figura 2. Para uma dada célula na posição (x_i, y_i) é preciso:

1. Computar a distância r até a célula onde está o robô. Como essa distância é dada em número de células, lembre-se de dividí-la por `scale` para convertê-la para metros e poder compará-la com as medidas dos sensores.

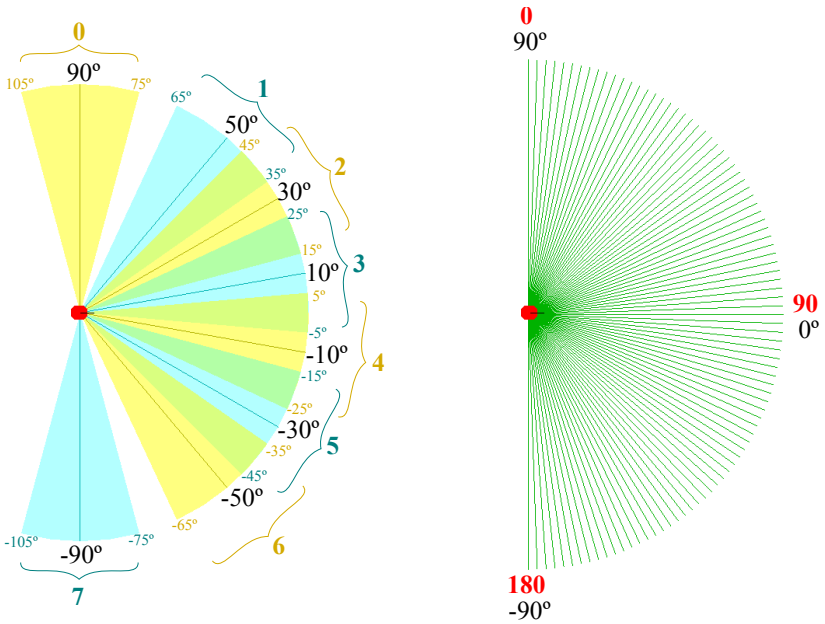
2. Computar a orientação ϕ da célula em relação ao robô em coordenadas locais (ou seja, já descontando a orientação θ do robô).
   OBS: Todos os ângulos computados devem estar devidamente normalizados (entre $-180°$ e $180°$). Para auxiliar use a função: `phi = normalizeAngleDEG(phi);` descrita em `Utils.cpp`

3. Encontrar a medida do sensor k mais próxima da orientação da célula em relação ao robô. Isso depende se está sendo utilizado laser ou sonar. A Figura 3 mostra as configurações do sonar e do laser. Para facilitar o trabalho, são fornecidas as funções `base.getNearestSonarBeam(phi)` ou `base.getNearestLaserBeam(phi)`, que retornam o índice da medida mais próxima do ângulo `phi`. Se a orientação da célula for menor do que $-90°$ o sensor mais próximo será o último à direita. Se a orientação da célula for maior que $90°$ o sensor mais próximo será o último à esquerda.

4. Atualizar a ocupação da célula como ocupada ou livre dependendo da região do sensor em que se enquadrar. Para isso deve-se testar se a célula está dentro da abertura do campo-de-visão do sensor através da diferença entre a orientação ϕ da célula e a orientação da medida k, dada pelas funções `base.getAngleOfSonarBeam(k)` ou `base.getAngleOfLaserBeam(k)`. E também verificar se a distância r é próxima ou menor da medida pelo k-ésimo sensor, dada pelas funções `base.getKthSonarReading(k)` ou `base.getKthLaserReading(k)`.

1. Implementação do método de mapeamento com LOG-ODDS usando LASER

Complete a função `mappingWithLogOddsUsingLaser` da classe `Robot` no arquivo `Robot.cpp`. Essa função deverá fazer a atualização dos valores de ocupação das células no campo de visão do laser do robô.



(a) Sonar (b) Laser

Figura 3: Configurações dos sensores sonar e laser.

Utilize $\lambda_r = 0.1$ (isto é, $0.1m$) como a largura dos obstáculos e $\lambda_\phi = 1.0$ (isto é, 1 grau) como abertura do sensor.

Defina valores fixos de ocupação para áreas ocupadas (p_{occ}) e áreas livres (p_{free})

$$0.0 < p_{free} < 0.5 < p_{occ} < 1.0$$

Os valores escolhidos impactarão na velocidade de atualização da ocupação das células. Na prática, serão utilizados os valores em *log-odds*, isto é $l = \log \frac{p}{1-p}$. Há uma função pronta para fazer essa conversão:
`lo = getLogOddsFromOccupancy(p);`

Dada uma célula qualquer do grid (ex. `Cell* c = grid->getCell(i,j);`), podemos acessar os valores de *log-odds* e ocupação dela em `c->logodds` e `c->occupancy` respectivamente. Embora o método de atualização seja aplicado sobre `c->logodds`, como desejamos visualizar o passo-a-passo do mapeamento também precisamos atualizar constantentemente `c->occupancy`, pois este valor (normalizado entre 0 e 1) que é ilustrado no mapa em tons de cinza, conforme implementado em `drawCell` em `Grid.cpp`. A conversão de *log-odds* para probabilidades pode ser feita através de:
`c->occupancy = getOccupancyFromLogOdds(c->logodds);`

2. Implementação do método de mapeamento HIMM usando LASER

De forma análoga à questão 1, complete a função `mappingWithHIMMUsingLaser` da classe `Robot` no arquivo `Robot.cpp`. Essa função deverá fazer a atualização dos valores de ocupação das células no campo de visão do laser do robô.

Utilize novamente $\lambda_\phi = 1.0$ (isto é, 1 grau) como abertura do sensor e $\lambda_r = 0.2$ (isto é, $0.2m$) como a largura dos obstáculos. Isso fará as paredes do mapa construído com HIMM ficarem mais grossas.

A ocupação de cada célula é dada por um contador (valor inteiro) e acessada em `c->himm`. Sempre que a célula cair em uma região de obstáculo do sensor, incremente o contador em 3 unidades. Sempre que a célula cair em uma região livre decremente o contador em 1 unidade. Limite o valor máximo de ocupação em 15 e mínimo em 0.

Apertando a tecla 'g', o valor `c->himm` de cada célula é mostrado no mapa. Isso está definido em `drawText` em `Grid.cpp`.

3. Implementação do método de mapeamento usando SONAR

De forma análoga às questões anteriores, complete a função `mappingUsingSonar` da classe `Robot` no arquivo `Robot.cpp`. Essa função deverá fazer a atualização dos valores de ocupação das células no campo de visão do sonar do robô.

Agora utilize $\lambda_r = 0.5$ (isto é, $0.5m$) como a largura dos obstáculos e $\lambda_p hi = 30.0$ (isto é, 30 graus) como abertura do sensor. É interessante que a largura dos obstáculos seja bem maior, pois o sonar é um sensor menos preciso e que captura menos informação a cada instante.

Implemente o modelo de sensor invertido proposto por Murphy, que dá maior peso as células mais próximas do robô (distância definida por r) e mais próximas do eixo acústico do sonar (ângulo definido por α). Considere R como sendo o alcance máximo do sensor e β como sendo metade da abertura do sensor ($\lambda_\phi/2$).

$$OccUpdate = 0.5\times \left(\frac{\frac{R-r}{R} + \frac{\beta-\alpha}{\beta}}{2} \right) + 0.5 \qquad \text{(Região 1) valores de 0.5 a 1}$$

$$OccUpdate = 0.5\times \left(1.0 - \frac{\frac{R-r}{R} + \frac{\beta-\alpha}{\beta}}{2} \right) \qquad \text{(Região 2) valores de 0 a 0.5}$$

ATENÇÃO: neste método a atualização da ocupação de cada célula é feita através de um produto de probabilidades.

$$Occ = \frac{OccUpdate \times Occ}{(OccUpdate \times Occ) + ((1.0 - OccUpdate) \times (1.0 - Occ))}$$

É importante evitar que o valor de Occ chegue em 0 ou 1, pois nesse caso ele nunca mais pode ser alterado. Por exemplo, após a atualização da ocupação limite o valor para ficar entre 0.01 e 0.99.

Para não sobrescrever os valores de ocupação atualizados pelo método da questão 1, acesse os valores de ocupação de cada célula em `c->occupancySonar`.

4. Implementação de método de limiarização da ocupação das células para classificá-las

O framework contém uma classe chamada `Planning` descrita no arquivo `Planning.cpp`, que nos trabalhos futuros será responsável por realizar a atualização do planejamento de caminhos do robô. A função `run` dessa classe `Planning` roda continuamente em uma *thread* diferente da *thread* principal (que roda a função `run` da classe `Robot` - responsável pelo controle do robô e mapeamento do ambiente). Embora o planejamento de caminhos só venha a ser desenvolvido futuramente, ele precisará de um mapa com as células já devidamente classificadas. Portanto, complete a função `updateCellsTypes` da classe `Planning`.

Primeiro, as células devem ser classificadas quanto ao tipo de ocupação (`occType`), que pode ser:

- **não-explorada**: `c->occType = UNEXPLORED` (mostrada em cinza)
- **livre**: `c->occType = FREE` (mostrada em amarelo bem claro)
- **obstáculo**: `c->occType = OCCUPIED` (mostrada em marrom escuro)

Você deve varrer todas as células ao redor do robô dentro do alcance máximo dos sensores (definido na classe `Planning` pela variável `maxUpdateRange`). Para cada célula varrida é preciso analisar o valor de ocupação obtido por um método de mapeamento (pode ser tanto o HIMM quanto o log-odds usando laser - evite sonar). Todas células começam como UNEXPLORED, e no momento em que deixam de ser, só devem alternar entre FREE ou OCCUPIED. Defina limiares adequados para modificar a classificação das células.

Dica: Use limiares diferentes para alternar de FREE para OCCUPIED (e vice-versa) para evitar que pequenas variações na ocupação em valores próximos do limiar causem oscilações na classificação.

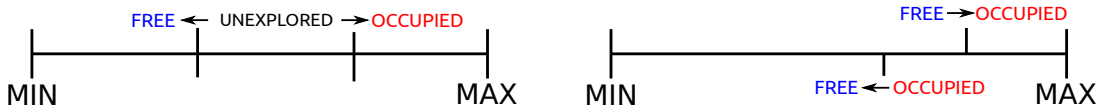


Figura 2: Escolha de limiares para alternar a classificação das células do grid.

Em um segundo momento, classifique as células quanto ao tipo de função no planejamento (`planType`), que por enquanto pode ser:

- **fronteira**: `c->planType = FRONTIER` (mostrada em verde)
- **perigo**: `c->planType = DANGER` (mostrada em vermelho)
- **regular**: `c->planType = REGULAR` (que não se enquadram nos tipos anteriores)

Considere como célula de fronteira, toda célula não-explorada que é adjacente a uma célula livre. Células de fronteira indicam regiões que o robô deve ir para completar o mapeamento do ambiente (um mapa está completo quando não há mais células de fronteira atingíveis pelo robô).

Por fim, considere como célula de perigo toda célula livre que é muito próxima a uma célula de obstáculo. Estas células são navegáveis caso seja necessário, mas devem ser evitadas pois há um risco grande de colisões. Utilize uma distância de 3 células para definí-las, ou seja, marque todas células livres que estão a 1, 2, ou 3 células de distância de obstáculos.