The Constant Gardener

GUILHERME CARREIRO GOMES (180074)*, LUÍSA MADEIRA CARDOSO*, MATEUS CORADINI SANTOS *

*Aluno especial - Mestrado

E-mails: karreiro@gmail.com, lu.madeira2@gmail.com, mateuscoradini@gmail.com

***Resumo* – O objetivo deste trabalho é implementar um sistema de localização e planejamento de rotas para um robô móvel. Para localização, utilizou-se o filtro Kalman extendido, combinando a estimativa provida pela odometria com a observação da localização de uma base. Para o planejamento e execução de rotas, utilizou-se o algoritmo A* combinado com o comportamento "Go to Goal". Também discutem-se possíveis melhorias para a implementação desenvolvida, utilizando técnicas de aprendizado por reforço. O filtro de Kalman provou-se uma técnica eficiente para computar a localização do robô, especialmente quando mais de uma base é utilizada fusão sensorial. As técnicas utilizadas para implementação do módulo de planejamento de caminhos também atingiram êxito.**

***Palavras-chave* – V-REP Pioneer Localização KF EKF A* A-Star GoToGoal Aprendizado por reforço**

I. INTRODUÇÃO

O desenvolvimento de um robô jardineiro envolve uma grande quantidade de desafios, entre eles: a movimentação do robô com base em seu modelo cinemático, a percepção de sua localização, a identificação da saúde das plantas através de uma câmera, a identificação do tipo de planta tratada, o planejamento do percurso do robô (para visitar as plantas periodicamente), a atuação sobre as plantas (para a adição de adubo ou água, por exemplo) e a interface com o usuário, permitindo a flexibilização de diversos recursos.

Neste projeto, isolamos duas questões essenciais para um robô desse tipo. O planejamento da rota, necessário para atingir todas as plantas, e o desenvolvimento da implementação de um modelo de localização altamente preciso. Para possibilitar a implementação dessas duas provas de conceito, certas premissas foram assumidas, desacoplando-as de um cenário maior e mais complexo. Tais premissas serão discutidas e apresentadas ao longo do artigo.

Todas as simulações expostas foram realizadas no simulador *V-REP* utilizando o *Pioneer P3-DX*. As implementações foram feitas utilizando Java 8 e Python 3.5. O módulo de localização encontra-se em https://github.com/luwood/MO810-vrep-python, já o módulo relacionado ao planejamento de caminhos pode ser acessado em https://github.com/karreiro/pathfinding-lab, por fim, o projeto final integrando os dois módulos com o *V-REP* encontra-se em https://github.com/mateuscoradini/final-project.

Nas seções subsequentes, abordaremos respectivamente: II) Localização, onde Luísa discute as técnicas e os resultados obtidos utilizando odometria, uma base para fusão sensorial e os conceitos que envolvem a implementação do filtro de Kalman; III) Planejamento e execução de rotas, onde Guilherme expõe os desafios envolvendo a implementação do algoritmo A* em um cenário envolvendo um robô móvel; em IV) Q-Learning e Aprendizado, Mateus apresenta técnicas para evoluirmos a solução apresentada utilizando aprendizado para coordenação dos comportamentos; e finalmente em V) todos integrantes concluem interpolando todos os tópicos abordados.

II. LOCALIZAÇÃO

A. Odometria

O cálculo da odometria é realizado com base na estimativa de velocidade das rodas. Cada roda possui um *encoder* que provê sua posição angular. Através da coleta temporal desta informação é possível determinar sua velocidade utilizando a seguinte fórmula:

$$V = \frac{\Delta\theta}{\Delta time} R$$

Em que $\Delta\theta$ representa a diferença angular entre posições do *encoder* durante um intervalo de tempo $\Delta time$ e R é o raio da roda. É importante destacar que o cálculo da diferença angular deve levar em conta a orientação do giro e o universo em que os ângulos estão.

Dada a velocidade de cada roda, pode-se calcular a velocidade linear e angular do robô através da fórmula:

$$V = \frac{V_r + V_l}{2}$$

$$\omega = \frac{V_r - V_l}{D}$$

Em que V_r é a velocidade da roda direita, V_l é a roda esquerda, D é a distância entre as rodas, V é a velocidade linear e ω é a velocidade angular.

A pose do robô no momento t depende da pose anterior, em $t-1$, e pode ser calculada através das equações:

$$x_t = x_{t-1} + (\Delta s * cos(\theta_{t-1} + \frac{\Delta\theta}{2}))$$

$$y_t = y_{t-1} + (\Delta s * sin(\theta_{t-1} + \frac{\Delta\theta}{2}))$$

$$\theta = \theta_{t-1} + \Delta\theta$$

A implementação do cálculo da odomoetria é feita pela classe *OdometryPoseUpdater*. Para fins práticos a pose inicial do robô é obtida com a leitura do *Ground Truth*. O cálculo da velocidade da roda encontra-se em uma classes distinta chamada *Wheel*. A orientação do giro é obtida utilizando a hipótese que a diferença angular deve ser sempre menor do que π.

B. Localizando a Base

A ideia inicial deste projeto era permitir que a localização do robô fosse obtida a partir da comunicação com uma base. O princípio seria semelhante a tecnologia utilizada no StarGazer (HagiSonic): o robô envia um sinal e a base o reflete. Deste modo é possível determinar a distância entre os dois objetos.

1) Teoria: Através da distância de um único ponto, é impossível determinar sua localização precisa. Considerando o sistema local de coordenadas do robô, pode-se ver na figura 1 que a base poderia estar em qualquer ponto do círculo determinado pela distância calculada entre o sensor e a base. Portanto, são necessários mais sensores no robô para determinar a localização da base.

Com três sensores é possível determinar a posição da base através da resolução de um sistema linear de equações. Na figura 2 podemos ver que o círculo que parte de cada sensor, se intersecta em um único ponto.

A equação de cada uma das circunferências pode ser descrita da seguinte forma:

$$r_1^2 = (x - x_1)^2 + (y - y_1)^2 \tag{1}$$

$$r_2^2 = (x - x_2)^2 + (y - y_2)^2 \tag{2}$$

$$r_3^2 = (x - x_3)^2 + (y - y_3)^2 \tag{3}$$

Para encontrar o ponto de intersecção entre as três circunferências, é necessário encontrar os valores de x e y combinando as três equações quadráticas em um sistema de duas equações lineares. Subtraindo (2) de (1) e (3) de (1):

$$2x(x_2-x_1)^2+2y(y_2-y_1)^2+(x_1^2-x_2^2)+(y_1^2-y_2^2)-(r_1^2-r_2^2) = 0$$

$$2x(x_3-x_1)^2+2y(y_3-y_1)^2+(x_1^2-x_3^2)+(y_1^2-y_3^2)-(r_1^2-r_3^2) = 0$$

A solução do sistema é a localização da base considerando o sistema de coordenadas local do robô.

2) Transformação de coordenadas: Para implementação do filtro de Kalman utilizado neste trabalho é necessário identificar a posição da base no sistema de coordenadas globais. A transformação do sistema de coordenadas locais do robô para o sistema global é dado pela rotação (4) seguida da translação(5) do ponto(x, y), no qual dx, dy e α são dados pela pose do robô.

$$\begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5}$$

3) Implementação: O robô possui três transceptores em seu topo que estão dispostos como mostrado na figura 2. A base se encontra nas coordenadas $(0, 0)$ do sistema global de referência. A distância entre os tranceptores e a base é calculada utilizando o módulo de cálculo de distâncias provido pelo simulador V-REP. É importante ressaltar que este cálculo de distâncias em uma simulação mais verossímil precisaria ser implementado.

A implementação do cálculo de intersecção das três circunferências é o método *calculatePoint* no módulo *AngleUniverse*. Ele é utilizado pela classe *BaseDetector* para calcular a posição da base dadas as distâncias obtidas dos transceptores. A base é representada pela classe *DetectedBase* que possui o método *getAbsolutePosition* que realiza a transformação das coordenadas locais para as coordenadas globais dada uma determinada pose.

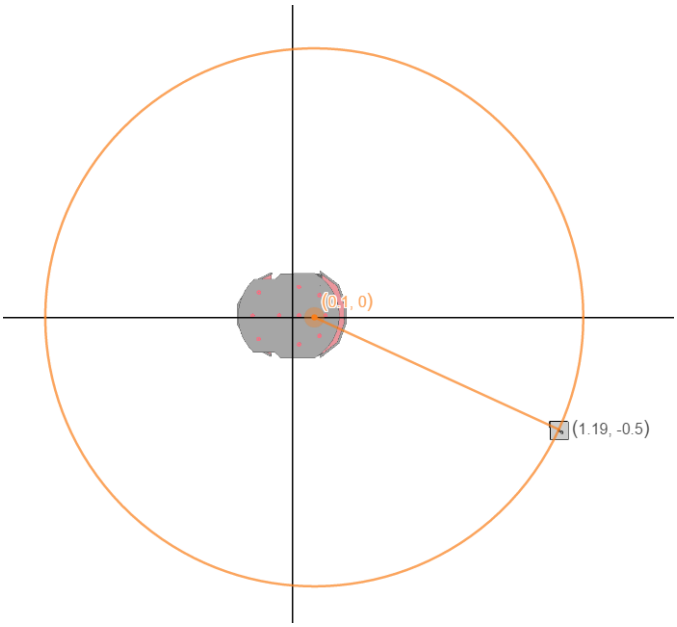

Figura 1. Robô com apenas um sensor de distância da base

C. Filtro de Kalman

O filtro de Kalman (KF) é uma implementação de filtros *Bayesianos* que realiza remoção de ruídos e predição de valores num sistema de estados contínuos. O interessante desta técnica é sua capacidade de combinar diferentes estimativas e suas respectivas covariâncias, computando uma distribuição Gaussiana baseada apenas em estados anteriores.

Por definição, o KF trabalha com probabilidades lineares. Sua versão extendida (EFK) trabalha com a hipótese de as funções que modelam as probabilidades não são lineares. Como um robô tipicamente pode realizar uma trajetória circular, o modelo mais indicado é o EKF.

A lógica do EKF está descrita pelo algoritmo 1, onde $\bar{\mu}_t$ pode ser entendido como a estimativa do estado no tempo t,

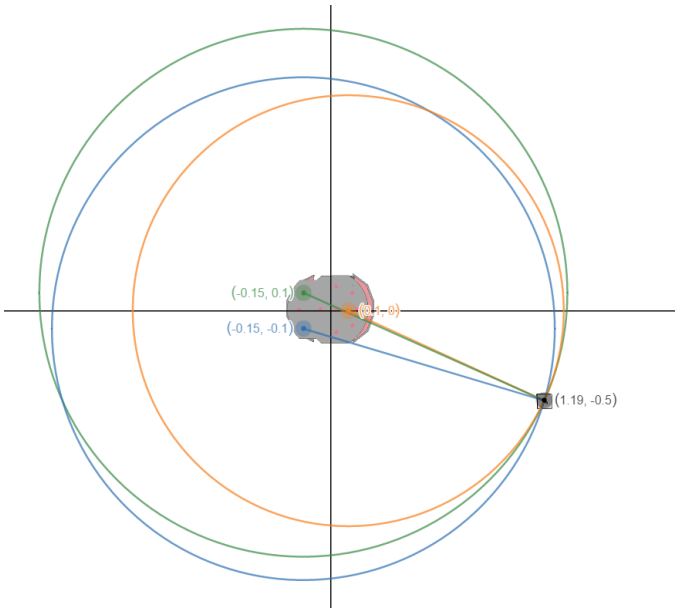

Figura 2. Robô três sensores de distância da base

Algorithm 1 Extended Kalman filter $\bar{\mu}_t, \mu_{t-1}, \Sigma_{t-1}, \Sigma_{\Delta t}, z_t$

$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^-1$
$\mu_t = \bar{\mu}_t + K_t(\bar{z}_t - z_t)$
$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$
return μ_t, Σ_t

μ_{t-1} o cálculo do estado no tempo $t-1$, Σ_{t-1} a covariância calculada em $t-1$ e z_t são as observações no tempo t.

1) Localização com EKF: Esta sessão é dedicada a explicar como o filtro de Kalman Extendido pode ser utilizado para realizar a localização do robô com o auxílio da detecção de *landmarks*. As fórmulas utilizadas foram primariamente retiradas da tabela 7.2 do livro *Probabilistic Robotics*[**?**]. Como suporte também foi utilizada uma apresentação realizada em 2014[**?**].

O objetivo da utilização do filtro é a melhoria na cálculo da pose do robô. Portanto, na primeira linha do algoritmo 1, μ_t representa a pose do robô no instante de tempo t . A estimativa de μ_t, $\bar{\mu}_t$, é dada pelo computação da odometria.

A primeira linha do algoritmo pode ser entendida como o modelo de erro da odometria. Portanto, foi acrescentado mais um fator em sua composição. Sua forma final é dada pela equação:

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + V_t \Sigma_{\Delta t} V_t^T + R_t$$

Os valores de G_t, $Sigma_{\Delta t}$, V_t e R_t estão descritos abaixo:

$$\beta_t = \theta_{t-1} + \frac{\Delta\theta_t}{2}$$

$$G_t = \begin{bmatrix} 1 & 0 & -\Delta s * sin(\beta_t) \\ 0 & 1 & \Delta s * cos(\beta_t) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\Sigma_{\Delta t} = \begin{bmatrix} K_s|\Delta s_t| & 0 \\ 0 & K_t|\Delta\theta_t| \end{bmatrix}$$

$$D = wheelsDistance$$

$$V_t = \begin{bmatrix} \frac{1}{2}cos(\beta_t) - \frac{\Delta s}{2D}sin(\beta_t) & \frac{1}{2}cos(\beta_t) + \frac{\Delta s}{2D}sin(\beta_t) \\ \frac{1}{2}sin(\beta_t) + \frac{\Delta s}{2D}cos(\beta_t) & \frac{1}{2}sin(\beta_t) - \frac{\Delta s}{2D}cos(\beta_t) \\ \frac{1}{D} & \frac{1}{D} \end{bmatrix}$$

$$R_t = \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\theta^2 \end{bmatrix}$$

A segunda linha do algoritmo calcula a matriz K_t, conhecida como o ganho de Kalman. Essa matriz pode ser entendida como o mecanismo que indica se deve-se confiar no valor estimado (odometria) ou na observação dos sensores (z). No cenário proposto, os sensores são capazes de estimar a posição de *landmarks* cuja posição real é conhecida anteriormente. A matriz Q representa a distribuição dos erros das observações. A matriz H_t é a matriz Jacobiana do cálculo de z. Portanto, para cada *landmark* que encontra-se em (L_x, L_y), as matrizes H e Q recebem duas linhas:

$$x_t = \bar{\mu}_t[x], y_t = \bar{\mu}_t[y]$$

$$q = (L_x - x_t)^2 + (L_y - y_t)^2$$

$$H_t = \begin{bmatrix} -\frac{L_x - x_t}{\sqrt{q}} & -\frac{L_y - y_t}{\sqrt{q}} & 0 \\ \frac{L_y - y_t}{q} & -\frac{L_x - x_t}{q} & -1 \end{bmatrix}$$

$$Q_t = \begin{bmatrix} \sigma_{Id}^2 & 0 \\ 0 & \sigma_{I\theta}^2 \end{bmatrix}$$

A terceira linha do algoritmo realiza o cálculo de μ_t. A equação $\bar{z}_t - z_t$ calcula um valor chamado de *inovação* ou resíduo. Em termos práticos ele apenas estima a diferença entre onde o *landmark* deveria estar e onde ele está. Para cada *landmark* utilizado duas linhas são acrescentadas as matrizes:

$$z = \begin{bmatrix} L_{range} \\ L_{bearing} \end{bmatrix}$$

$$\bar{z} = \begin{bmatrix} l_{range} \\ l_{bearing} \end{bmatrix}$$

Onde L representa a posição real do *landmark* e l a posição calculada através dos sensores. As funções de distância e inclinação podem ser calculadas através das equações:

$$p_{range} = \sqrt{(p_x - x_t)^2 + (p_y - y_t)^2}$$

$$p_{bearing} = arctan2((p_y - y_t), (p_x - x_t)) - \theta_t$$

2) Implementação: A implementação do filtro extendido de Kalman encontra-se na classe *KalmanFilterPoseUpdater*. É importante notar que este componente utiliza o cálculo da odometria e sempre realiza a atualização do último valor calculado pela mesma.

Os valores das constantes utilizadas estão descritos na tabela I

Tabela I
CONSTANTES UTILIZADAS

Variável	Valor
K_s	0.1
K_t	0.1
σ_x	1
σ_y	1
σ_θ	1
σ_{Id}	0.5
$\sigma_{I\theta}$	0.1

D. Resultados

Os experimentos realizados com as técnicas de estimativa de pose do robô utilizam o algoritmo de Braitenberg para movimentação do mesmo.

1) Odometria: O gráfico comparando a posição real do robô e a posição calculada através da odometria pose ser visto na figura 3. É possível observar que o trajeto em linha reta obtido pela odometria é preciso. Porém assim que a primeira curva é realizada a diferença entre as posições começa a divergir. A diferença torna-se maior a cada iteração devido aos erros acumulados. A tabela II mostra a evolução do erro no cálculo da orientação durante um determinado período de teste.

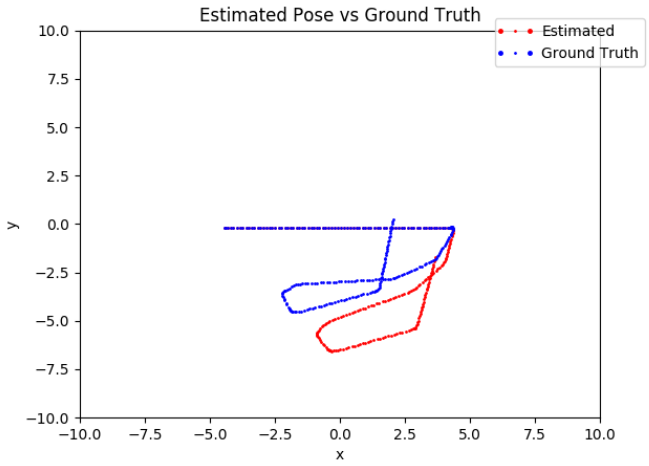

Figura 3. Odometria

Pode-se concluir que a odometria é um método de estimativa extremamente suscetível a erros acumulados. Para tornar este método viável seria necessário necessário realizar correções no cálculo da orientação. A utilização de uma bússola, por exemplo, poderia auxiliar nesta computação.

2) Filtro de Kalman Extendido: A figura 4 mostra a comparação entre a posição estimada pela odometria e o filtro de Kalman e a posição real do robô quando apenas uma base é utilizada. Pode-se observar a evidente melhoria no trajeto calculado em comparação com a figura 3 (cenário que utiliza apenas a odometria). Porém também fica evidente que em alguns pontos o erro na predição da rota é maior do que em outros.

Tabela II
DIFERENÇA θ EM GRAUS - A CADA 200MS

Real	Odometria	Erro
-0.424	-0.406	0.018
-0.515	-0.488	0.027
-0.783	-0.718	0.065
-1.182	-1.065	0.117
-2.033	-1.818	0.215
-3.048	-2.724	0.324
-4.355	-3.861	0.494
-6.122	-5.416	0.706
-7.843	-6.953	0.89
-9.851	-8.747	1.104
-12.613	-11.227	1.386
-14.864	-13.288	1.576
-18.003	-16.129	1.874
-22.479	-20.084	2.395

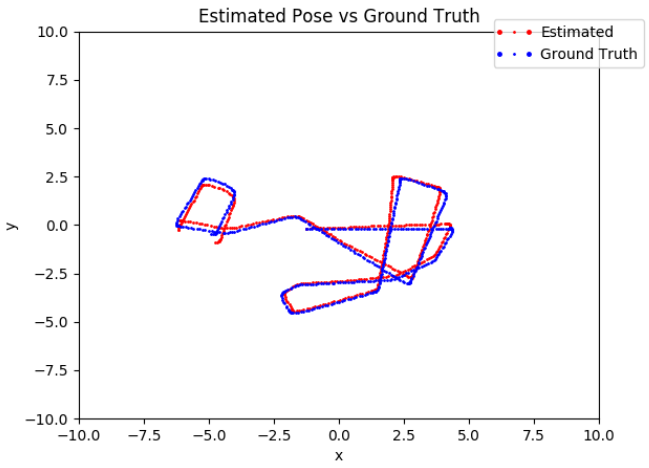

Figura 4. EKF: Utilizando 1 base

Na tentativa de melhorar o cálculo da trajetória, foi acrescentada mais uma base, na posição $(3, -3)$. O resultado desta iteração pode ser visto na figura 9. A rota torna-se mais precisa do que quando utilza-se apenas uma base.

Acrescentando mais uma base no ambiente, na coordenada $(4, 6)$, o cálculo da trajetória fica muito próximo da trajetória real, como pode ser notado na figura 6.

III. PLENEJAMENTO E EXECUÇÃO DE ROTAS

O conceito de planejamento de caminhos é bastante amplo em robótica, englobando desde tópicos relacionados a inteligência artificial até teoria de controle. Entretanto, em suma, tal conceito refere-se técnicas para mover um robô de determinando ponto a outro [**?**].

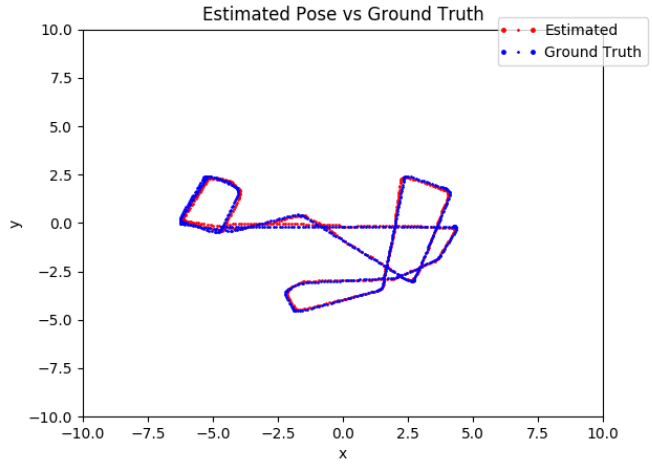

Figura 5. EKF: Utilizando 2 bases

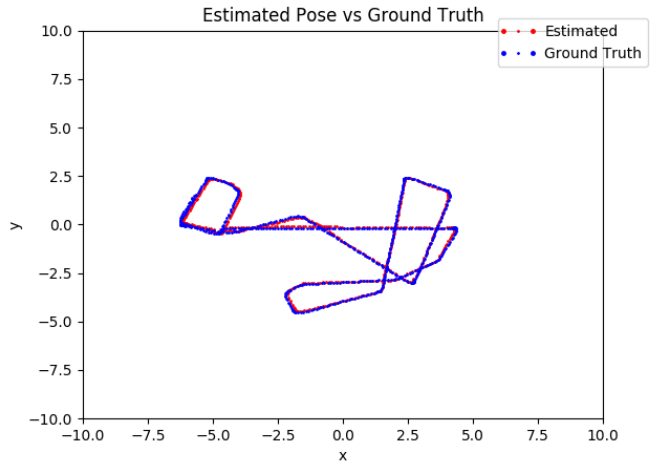

Figura 6. EKF: Utilizando 3 bases

O módulo de planejamento de rota funciona considerando a premissa da existência de um grid com informações do ambiente. Tal discretização deve conter coordenadas de paredes, obstáculos, plantas e qualquer outro entrave que influencie na rota do robô. Dessa maneira, o robô pode ser capaz de visitar todas as plantas, desviando de eventuais obstáculos.

Com esta premissa bem definida, note que este módulo possui dois grandes desafios, analisados nas próximas subseções: planejar rotas e seguir determinada trajetória, até o objetivo definido.

A. *Planejamento de rota*

Para realizar o planejamento dos caminhos, chegou-se a conclusão que o algoritmo A* (ou A-Star, ou ainda, A-Estrela) seria o mais apropriado para a realização de tal tarefa, pois é uma das melhores estratégias para planejamento de rotas que pode ser aplicado em um grid métrico. Combinando conceitos do algoritmo do melhor caminho com heurística, o A-Star possui ótimos resultados [**?**], num tempo bastante razoável, considerando o escopo do problema abordado.

A base do A* é sua função de avaliação de nó, que calcula a relevância de cada nó no grid, considerando sua origem e o destino na trajetória. Tal função pode ser representada da seguinte maneira:

$$f(n) = g(n) + h(n)$$

O elemento ”g(n)”representa o custo da locomoção do robô entre nó avaliado e o nó de origem. Enquanto o ”h(n)”representa o custo de locomoção do nó avaliado até o nó de destino, repare que apesar de tal valor ser alcançável, uma heurística como Manhattan, distância Euclideana, ou ainda Chebyshev, apresenta-se muito mais eficiente e barata [**?**].

De modo genérico, podemos dizer que o comportamento do A* funciona basicamente da seguinte maneira:

- Primeiramente elenca o primeiro nó, e gradativamente se expande para os nós vizinhos;
- Cada vizinho sub-sequente é avaliado, caso não seja um obstáculo, pela função do A*, organizando os vizinhos de acordo com o valor obtido;
- Esse processo continua até que o nó destino seja atingido;
- Para que o caminho até o nó destino seja armazenado, é necessário um processo a parte [**?**].

Dessa maneira, é possível obter um caminho válido a partir de um grid com obstáculos, apenas possuindo um ponto de origem e um ponto de destino.

B. *Execução de rota*

Para permitir que o robô siga a rota, implementou-se um algoritmo ”Go to Goal”, que resumidamente, recebe uma coordenada qualquer, e auxilia o robô a chegar o mais perto possível de tal ponto.

Considerando o modelo cinemático do Pioneer, para que ele siga determinada coordenada, é necessário que seu ângulo se alinhe ao ângulo da coordenada desejada. Desta maneira, conforme o robô se desloca, alinha-se ao objetivo definido, conseguindo assim se aproximar cada vez mais da coordenada desejada.

Para realizarmos o alinhamento do ângulo do Pioneer com seu objetivo, utilizamos um controlador PID, acrônimo para ”Proportional, Integral and Derivative”, tais elementos descrevem os três elementos básicos de um controlador PID, exercendo funções específicas no sistema [**?**].

O componente ”Proportional”auxilia o sistema evitando que reaja exageradamente a pequenos estímulos; já o componente ”Integral”auxilia adicionando uma precisão a longo prazo com os erros acumulados ao longo do tempo; por fim, o componente ”Derivative”auxilia em respostas rápidas a erros que aumentam repentinamente, entretanto este componente pode ser zero se o erro for relativamente constante [**?**]. Desta maneira, com o sinal de controle gerado pelo controlador PID, é possível ajustar a intensidade dos motores das rodas, alinhando o robô com o objetivo desejado.

C. Implementação

Com o algoritmo A* selecionado como melhor solução, ele foi primeiramente implementado em um módulo externo como prova de conceito. Em tal projeto, definiu-se além da implementação do algoritmo, a estrutura de persistência do grid e de outros detalhes relacionados ao planejamento de trajetórias.

O módulo planejador demonstrou possuir performance aderente ao contexto abordado, necessitado de apenas 170ms em média para planejar uma rota em um grid de 50 colunas, 50 linhas e diversos obstáculos. Sua complexidade é O(E) onde ”E”refere-se ao número de células no grid. Considerando que fizemos pouquíssimas otimizações e que nosso robô não exigia um tempo menor que esse, concluímos a implementação da PoC seria boa o suficiente para ser transferida ao projeto principal (veja a figura 7). Não utilizou-se diagonais no planejamento das rotas, a fim de facilitar comportamentos relacionados a eventuais colisões.

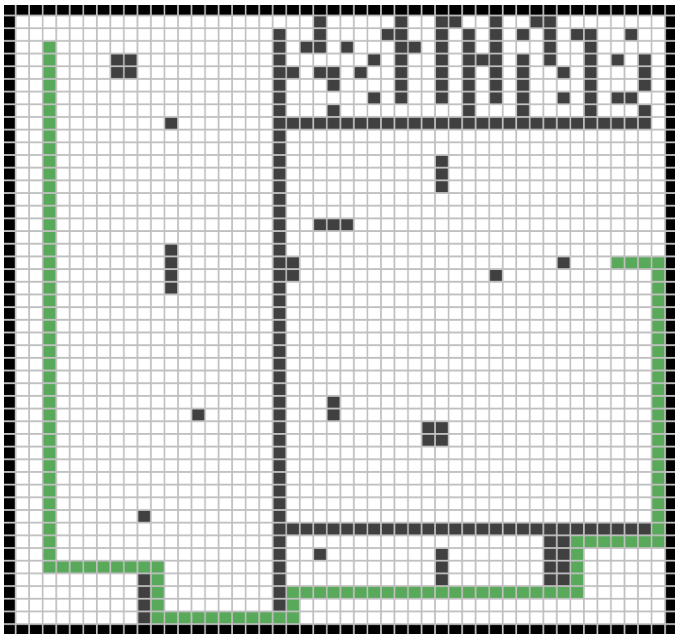

Figura 7. Prova de conceito do algoritmo A*

O próximo passo para integração foi traduzir o grid na cena do V-REP. Discretizou-se o cenário (conforme podemos ver na Figura 8) e criou-se um módulo que traduz as coordenadas das células para coordenadas no V-REP.

Para seguir a trajetória definida, o robô basicamente recebe uma lista de coordenadas, representado o caminho a ser percorrido, e as desempilha nó a nó, até que o objetivo seja atingido. Dessa maneira, conseguimos permitir que o robô visitasse todas as plantas do cenário controlado proposto.

A única especificidade adotada para atingir tal objetivo, refere-se a utilização de certas células alcunhadas de ?Padding?. Tais células são interpretadas pelo A-Star como obstáculos, porém demarcam regiões que não merecem ser

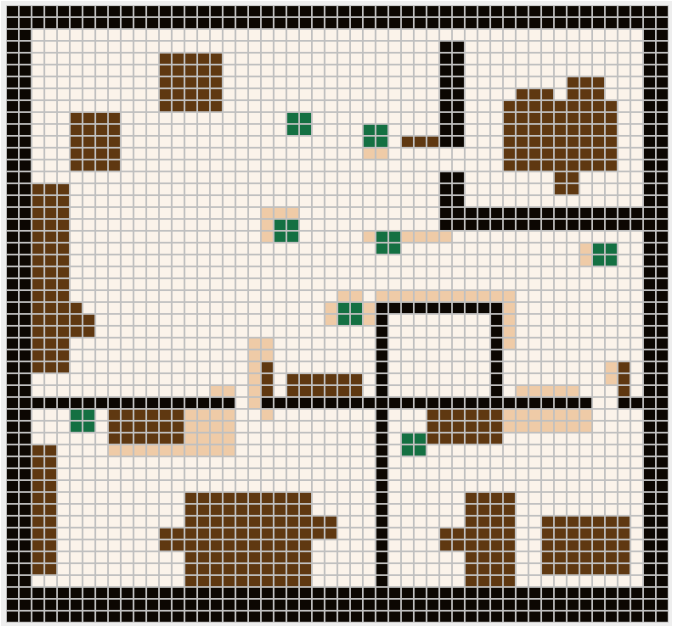

Figura 8. Cena discretizada em grid

visitadas, pois submeteriam o robô a uma condição de risco, como por exemplo, passar perto demais de determinada aqui. Com a lapidação do Go to Goal e de um grid mais preciso, este recurso certamente seria dispensável.

IV. Q-Learning e Aprendizado

Ao iniciar o modelo deliberativo de comportamento, traçamos como meta inicial abordar o aprendizado para chaveamento entre os comportamentos ”Wall Follower”e ”Avoid Obstacle”, representados no código pelas classes SimpleWallFollower e SimpleAvoidObstacle.

Dessa maneira, iniciou-se o o processo de estudo que coordenaria os comportamentos de acordo com o algoritmo de aprendizado por reforço Q-Learning desenvolvido por Watkins [**?**], que segundo Watkins [**?**], capitulo 3, ”Um processo de decisão Markov, ou cadeia controlada de Markov, cosiste em quatro partes: um estado-espaço S, uma função A que possibilita ações para cada estado, uma função de transição T e uma função de recompensa R.”, assim, com base nesta técnica, utilizada para achar a ótima seleção de ações baseadas neste processo. Considerando a equação segundo Geramifard, Walsh, Tellex e Chowdhary [**?**]:

$$Q(st, at) < -Q(st, at)+\alpha t[rt+1+\lambda maxQ(St+1, a)-Q(st, at)]$$

Onde: Q(st, at) é o valor anterior, e αt é a taxa de aprendizado e como valor aprendido na expressão, $rt + 1$ é a recompensa observada após executar a ação at, λ é o fator de desconto e $maxQ(St + 1, a)$ é a estimativa de ação futura, descontando-se o valor anterior.

Assim abordamos o modelo de aprendizado, mapeando os possíveis estados, como definido na figura 9. Onde 1

representara o estado inicial parado, 2 o comportamento de wall-follower, 3 o avoid-obstacle e 4 uma situação de risco.

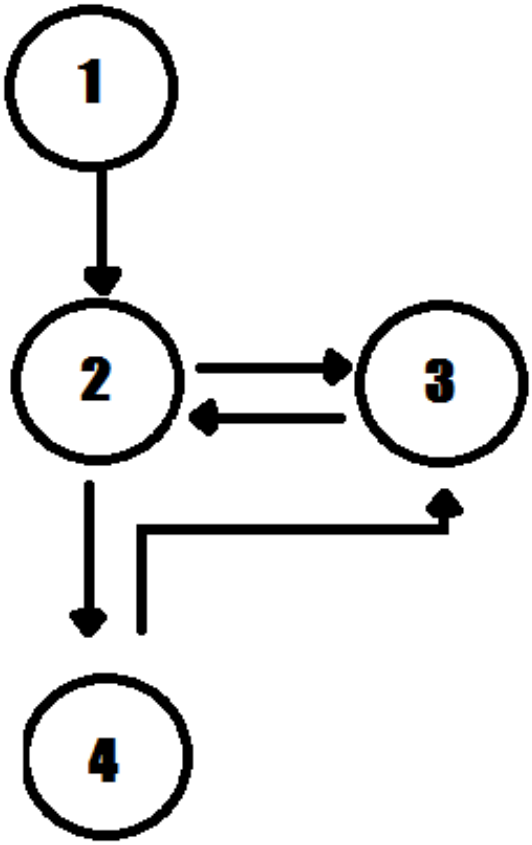

Figura 9.

O modelo não se mostrou efetivo para o comportamento esperado de exploração, era necessário mais que um estado ação de simples coordenação de comportamento, entretanto uma implementação para entender os passos do algoritmo encontra-se no projeto. Segundo o algoritmo de O algoritmo traduzido para linguagem estruturada, seria segundo [**?**]:

- Para cada ciclo do estado de treino
- Selecione um estado randomico inicial.
- Enquanto (o estado final não chegar), faça:
  - Selecione uma possivel ação, considerando o proximo estado.
  - Recupere o valor maximo de Q(recompensa) para o valor considerado proximo estado.
  - Compute valor da recompensa: Q(estado, ação) = Q(estado, ação) + alpha * (R(estado ,ação) + gamma * Max(proximo estado, todas ações) ? Q(estado, ação)).
  - Coloque o valor de estado no estado corrente.

A matriz de recompensa a ações assim se fazia ineficiente para o resultado não mapeado que apresentamos como premissa pois, pois não recebia recompensas por exploração e sim por situações de risco e então alteraria para avoid-obstacle para ganhar recompensa exploratória e sempre continuava no mesmo estado, estando assim em um loop de estado ações pouco explorado, pois ao perceber uma situação de risco a única alternativa de recompensa seria alternar seu comportamento para evitar os obstáculos. Testes com 500 iterações, num mapa de 3 colunas por 3 linhas, para exemplificar o goal, podem ser vistas na classe QLearningTheory:

- Print result
- out from Stopped: 0 0 0 0
- out from WallFollowing: 0 0 -1 0
- out from Danger: 0 0 0 0
- out from AvoidObstacle: 0 1 0 0,9
- showPolicy
- from Stopped goto Stopped
- from WallFollowing goto WallFollowing
- from Danger goto Danger
- from AvoidObstacle goto WallFollowing
- Time: 0.029 sec.

As tentativas de implementação do algoritmo citado não convergiram, considerando o modelo de recompensa de situação de risco e então começamos a pensar na integração das partes exploratórias de outros algoritmos e de um modelo futuro para exploração baseada no modelo de aprendizado e no algoritmo de Q-Learning. Este modelo deve criar um conjunto de ações com melhor recompensa definida por exploração de localização, como por exemplo, usar o mapeamento do ambiente e odometria para distinguir os objetos localizados e assim denominar o labirinto e escolhas a fazer. Iniciei a integração dessa classe com o objetivo de modelar o ambiente do grid como conhecemos, usando a classe PathPlanner, podíamos gerar um grid genérico, para então efetuar os cálculos para descobrir uma maneira de coordenar os comportamentos. Então iniciamos a modificação dos comportamentos para andar no grid de maneira condenada, acima, abaixo, lateral esquerda, lateral direita, e verificar essas condições para validar se não havia obstáculos, que podem ser vista na classe QLearningBeahiavor, mas não foi finalizado a implementação e ficou como melhoria futura.

V. Conclusão

O filtro extendido de Kalman mostrou-se uma excelente maneira de realizar correções no calculo da odometria. A hipótese de estimar a pose do robô utilizando apenas uma base mostrou-se válida, porém exibe erros consideráveis que podem prejudicar os comportamentos do robô que dependem desta estimativa. Os experimentos deixaram claro que quanto mais *landmarks* são acrescentados, mais precisa fica a estimativa da pose.

O módulo de planejamento se mostrou bastante eficiente nesse cenário controlado. Entretanto, uma implementação com um grid com células ainda menores, com maior refinamento no momento de calcular o caminho, certamente garantiria que

o robô pudesse acessar pontos específicos do cenário com maior precisão. O módulo de planejamentos poderia contar com outros comportamentos, ativados de acordo com faixas de coordenadas para garantir uma navegação mais dinâmica. Além disso, o Go to Goal também pode ser ainda mais refinado.

A integração do processo de localização com o módulo de planejamento se deu através de serviços Python consumidos por um cliente Java. Dessa maneira conseguimos garantir bastante desacoplamento entre as implementações, utilizando o melhor de cada tecnologia durante o desenvolvimento desta prova de conceito..