UNIVERSIDADE D
COIMBRA

MESTRADO EM ENGENHARIA E DE COMPUTADORES

Relatório de Sistema Robóticos Autónomos

Localização baseada em grelhas (Grid-map-based) usando RP-LIDAR + EKF

Duarte de Sousa Cruz, 2017264087
João Pedro Chaves Castilho, 2017263424

Universidade de Coimbra
5 de Junho de 2022

1 Introdução

Neste trabalho era-nos pedido para implementarmos o algoritmo de localização *Extended Kalman Filter* (EKF). Com este algoritmo, sabendo o mapa do ambiente e um conjunto de medidas sensoriais, conseguimos, fazer uma estimação para a posição da nossa plataforma móvel. Este problema é dos mais críticos no estudo da robótica móvel, pois sem ele não é possível termos plataformas moveis completamente autónomas.

Para testarmos a nossa implementação do EKF vamos realizar 3 testes diferentes:

- Localização com odometria corrompida (alterando a baseline);
- Localização com ruído gaussiano na leitura nos *encoders* e alteração da baseline;
- Localização com odometria corrompida e estimativa inicial errada;

Para que o algoritmo seja validado tem que ser visível que as observações sensoriais influenciam a estimação da posição de forma positiva.

2 Implementação do *Extended Kalman Filter*

A implementação do algoritmo foi dividida em duas fases: **previsão** e **atualização**.

2.1 Previsão

Nesta fase o objetivo é, tendo o valor das leituras dos encoders, fazer uma previsão inicial da posição do robô, usando o modelo do sistema. Este modelo do sistema é definido por:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_k = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{k-1} + \begin{bmatrix} D_k * \cos(\theta_{k-1} + \frac{D_t}{2}) \\ D_k * \sin(\theta_{k-1} + \frac{D_t}{2}) \\ D_t \end{bmatrix} \tag{1}$$

em que:

$$D_k = \frac{r_t + l_t}{2} \tag{2}$$

$$D_t = \frac{r_t - l_t}{baseline} \tag{3}$$

onde r_t e l_t são respetivamente a contagem dos encoders da roda direita e da esquerda, respetivamente, e *baseline* corresponde à distância entre as rodas. É neste passo, onde corrompemos a leitura dos enconders alterando o valor da *baseline*. Para os nossos testes em vez de usarmos o valor correto de 16 centímetros usamos 16, 3 centímetros.

No *Matlab* estas equações estão implementadas na função `getDeadReckoning` apresentada no seguinte bloco de código.

```
function [dead_reckoning] = getDeadReckoning(state_vector, right_ticks, left_ticks, baseline)
    Dk = (right_ticks + left_ticks) / 2;
    Dt = (right_ticks - left_ticks) / baseline;

    dead_reckoning = state_vector + [Dk * cos(state_vector(3) + (Dt / 2));
                                     Dk * sin(state_vector(3) + (Dt / 2));
                                     Dt];
end
```

O passo seguinte é calcular a matriz de covariância associada à previsão. Para isso usamos a seguinte equação:

$$C_p(k+1|k) = J_p \cdot C_p(k|k) \cdot J_p^T + J_u * C_u(k) * J_u^T \tag{4}$$

onde J_p e J_u correspondem aos jacobianos associados à 1 com respeito a $p = [x, y, \theta]$ e $u = [D_k, D_t]$ e é igual a:

$$J_p = \begin{bmatrix} 1 & 0 & -D_k \cdot \sin(\theta + \frac{D_t}{2}) \\ 0 & 1 & D_k \cdot \cos(\theta + \frac{D_t}{2}) \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$J_u = \begin{bmatrix} \cos(\frac{D_t}{2} + \theta) & -(D_k * \sin(\frac{D_t}{2} + \theta))/2 \\ \sin(\frac{D_t}{2} + \theta) & (D_k * \cos(\frac{D_t}{2} + \theta))/2 \\ 0 & 1 \end{bmatrix} \tag{6}$$

As matrizes C_p e C_u são as matrizes cujos valores diagonais dizem respeito às incertezas das respectivas variáveis. Para a matriz C_p é necessário definir um valor inicial. No caso da matriz C_u o valor é diferente em cada ciclo, que lemos os enconders:

$$C_u(k) = \begin{bmatrix} k_r|r_t| & 0 \\ 0 & k_l|l_t| \end{bmatrix} \tag{7}$$

em que k_r e k_l são constantes que definimos e que o seu valor define o quão confiamos nas medidas dos encoders.

A função no *Matlab* que criamos para calcular a covariância pode ser vista abaixo:

```
function motionError = getMotionError(C_p,C_u,state_vector, sr, sl, b)
Dk = (sr + sl) / 2
Dt = (sr - sl) / b

J_p = [1 0 -Dk * sin(state_vector(3) + Dt / 2)
       0 1 Dk * cos(state_vector(3) + Dt / 2)
       0 0 1]

J_u = [cos(Dt/2 + state_vector(3)), -(Dk*sin(Dt/2 + state_vector(3)))/2
       sin(Dt/2 + state_vector(3)),  (Dk*cos(Dt/2 + state_vector(3)))/2
       0,1]

motionError = J_p * C_p * J_p' + J_u * C_u * J_u';

end
```

2.2 Observações e Update

Nesta fase, usamos as observações feitas pelo robô para atualizar a nossa previsão feita na etapa da previsão.

O primeiro passo nesta fase é conseguir calcular as observações que seriam obtidas quando o robô está na posição prevista. Para isso usamos o modelo do sensor Lidar, que pode ser visto como se fossem muitos sensores sonar, neste caso, como o nosso Lidar tem uma resolução de 1° serão 360 sonares. O modelo do sensor é o seguinte:

$$g_i(k) = min\sqrt{(x_s(k) + x_{mj}(k))^2 + ((y_s(k) + y_{mj}(k))^2}, \quad 1 \leq j \leq r \tag{8}$$

em que (x_s, y_s) representam as coordenadas do sensor no referencial do mundo, e (x_{mj}, y_{mj}) são as coordenadas das células do mapa.

Este modelo pode ser implementado de várias maneiras. A maneira que encontramos mais simples de implementar vai ser explicada a seguir.

O problema que temos é encontrar as células ocupadas onde o Lidar supostamente iria bater se o robô estivesse na posição prevista na fase da previsão. Para isso começamos por encontrar as células que correspondem à distância máxima das leituras do Lidar, no nosso caso, 2 metros.

Após termos as células que correspondem à distância máxima para cada ângulo da resolução do Lidar, usamos o algoritmo de *bresenham* [1] para percorrermos todas as células dessa linha, até encontrarmos uma com um valor de ocupação igual a 1. Caso não haja nenhuma célula com valor de ocupação igual a 1, será atribuído o valor de `NaN`. Depois basta calcular o valor de $g_i(k)$ para a célula encontrada, usando a equação 8.

Para além de calculado o valor de $g_i(k)$, temos também de calcular o jacobiano associado a esta medida. O jacobiano é dado por:

$$J_g = \begin{bmatrix} \frac{x_s - x_{mj}}{\sqrt{((x_s - x_{mj})^2 + (y_s - y_{mj})^2}} \\ \frac{y_s - y_{mj}}{\sqrt{((x_s - x_{mj})^2 + (y_s - y_{mj})^2}} \\ 0 \end{bmatrix} \tag{9}$$

Depois disto, calculamos a inovação v, que é a diferença entre as medidas do Lidar e todos os valores de $g_i(k)$ calculados, e a respetiva covariância da inovação $s_i(k+1)$, que é igual a:

$$s_i(k+1) = J_g C_p(k+1|k) J_g^T + R_i(k+1) \tag{10}$$

onde $R_i(k+1)$ é a incerteza da medida do Lidar que no nosso caso corresponde a 3.5% da leitura do Lidar.

De modo a estabelecer correspondência entre as medidas do sensor e as medidas preditas, utiliza-se o critério de validação proposto por Leonard&Durrant-White e apresentado nas aulas,

$$v_i(k+1) \cdot s_i^{-1}(k+1) v_i^T(k+1) \leq e^2 \tag{11}$$

se a condição for satisfeita então a correspondência diz-se valida sendo usada para o estágio da atualização.

```
v = g_true - g_est;
[row,~] = find(~(isnan(v)));

e = 1;
for i = 1:size(row,1)
    idx = row(i);
    R_aux = 0.035*g_true(idx);
    s = J_g(:,:,idx)*C_p*J_g(:,:,idx)'+R_aux;
    e_gait = v(idx)*(1/s)*v(idx)';
    if (e_gait < e^2)
        v_valid(i)=v(idx);
        valid_J_g(i,:) = J_g(:,:,idx);
        R(i) = R_aux;
    end
end
```

Na fase final são usadas as correspondências que passaram as validações da eq 11 para obtermos a matriz de todas as inovações e a matriz de todos os jacobianos:

$$v(k+1) = \begin{bmatrix} o_1(k+1) - \hat{o_1}(k+1) \\ \vdots \\ o_j(k+1) - \hat{o_j}(k+1) \end{bmatrix} \tag{12}$$

$$\nabla g = \begin{bmatrix} \nabla g_1 \\ \vdots \\ \nabla g_j \end{bmatrix} \tag{13}$$

em que j é o número de correspondências validas.

Depois calculamos a matriz concatenada da inovação:

$$S(k+1) = \nabla g \cdot C_p(k+1|k) \cdot \nabla g^T + R(k+1) \tag{14}$$

A nova posição é calculada com a seguinte equação:

$$\hat{X}(k+1|k+1) = \hat{X}(k+1|k) + G(k+1)v(k+1) \tag{15}$$

e a respetiva matriz de covariância final:

$$C(k+1|k+1) = C(k+1|k) - G(k+1)S(k+1)G^T(k+1) \tag{16}$$

onde $G(k+1)$ é chamado o ganho de Kalman, obtido através de:

$$G(k+1) = C(k+1)\nabla g^T S^{-1}(k+1) \tag{17}$$

```
inov_cov = valid_J_g*C_p*valid_J_g'+diag(R);
Kalman_gain = C_p*valid_J_g'*pinv(inov_cov);

dead_reckoning = dead_reckoning + Kalman_gain*v_valid';
C_p = C_p-Kalman_gain*inov_cov*Kalman_gain';
```

3 Resultados

Em todos os casos de testes é dado ao robô os comandos de velocidade para que este efetue uma trajetória circular com raio de 1 metro.

3.1 Odometria corrompida

Neste teste, alteramos o valor da baseline do robô para $16,3$ centímetros em vez dos corretos 16 centímetros.

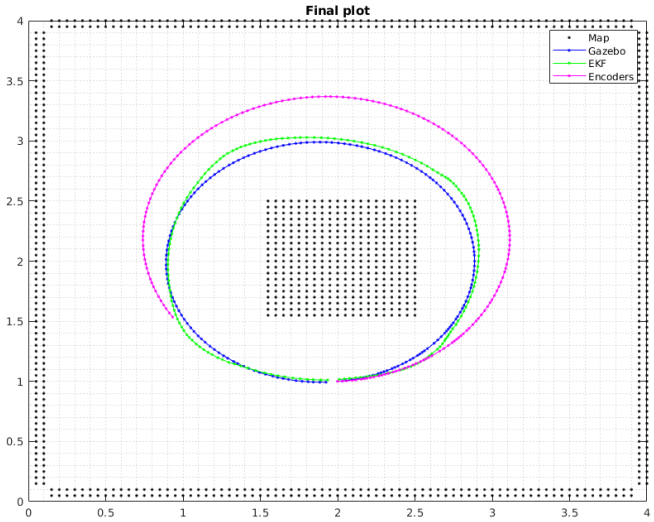

Figure 1: Teste do EFK com odometria corrompida

Podemos observar pela Fig. 1 que a nossa implementação do EKF tem o comportamento desejado.

A cor de rosa conseguirmos ver a trajetória obtida apenas pelo modelo dinâmico do sistema, ou seja, apenas com a fase da predição e não com a fase da observação e atualização. Vemos que devido ao valor incorreto da baseline, esta trajetória não corresponde à desejada com os comandos enviados.

A azul conseguimos observar o movimento real que o robô efetua no simulador Gazebo. Portanto, o objetivo seria que a trajetória estimada pelo EKF fosse igual a esta. Tal pode ser constatado a verde no gráfico.

Neste teste foram usadas as seguintes matrizes inicias para C_p e C_u:

$$C_p = \begin{bmatrix} 0.025 & 0 & 0 \\ 0 & 0.025 & 0 \\ 0 & 0 & 1^\circ \end{bmatrix}$$

$$C_u = \begin{bmatrix} 0.001|r_t| & 0 \\ 0 & 0.001|l_t| \end{bmatrix}$$

3.2 Ruído gaussiano na medida dos encoders

Neste teste inserimos ruído gaussiano na medida dos encoders usando a função Matlab:

```
[dsr, dsl, pose2D, timestamp] = tbot.readEncodersDataWithNoise([0.005, 0.005])
```

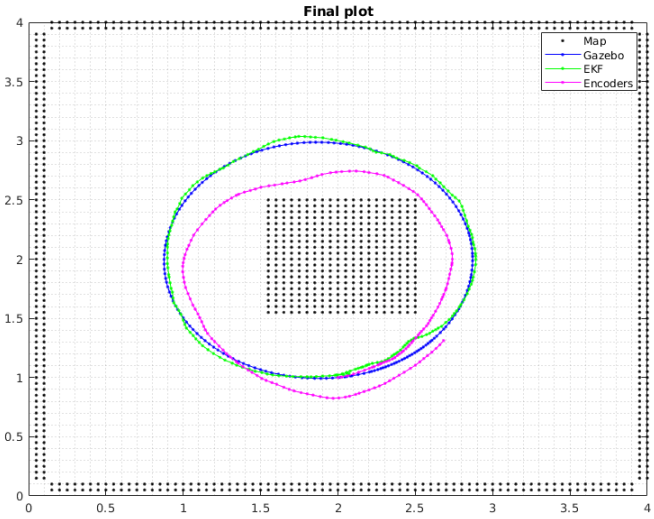

Figure 2: Teste do EFK com ruído gaussiano

Podemos ver que a trajetória a cor-de-rosa é bastante diferente da anterior, com alguma componente aleatória, como era esperado. Vemos que o EKF também teve um bom resultado, pois a trajetória a verde pouco se afasta da trajetória azul.

Para esta simulação foram usadas as seguintes matrizes:

$$C_p = \begin{bmatrix} 0.08 & 0 & 0 \\ 0 & 0.08 & 0 \\ 0 & 0 & 1^\circ \end{bmatrix}$$

$$C_u = \begin{bmatrix} 0.0005|r_t| & 0 \\ 0 & 0.0005|l_t| \end{bmatrix}$$

Podemos ver que para esta simulação usamos um valor para C_p superior para a incerteza de x e y.

3.3 Odometria corrompida e estimativa inicial incorreta

Para que o EKF funcione como esperado, é necessário definirmos uma estimativa inicial para a posição do robô. Até agora esta estimativa tem sido 100% correta. Vamos agora testar com uma estimativa errada. Vamos colocar a estimativa inicial igual a $(1.75, 0.5)$.

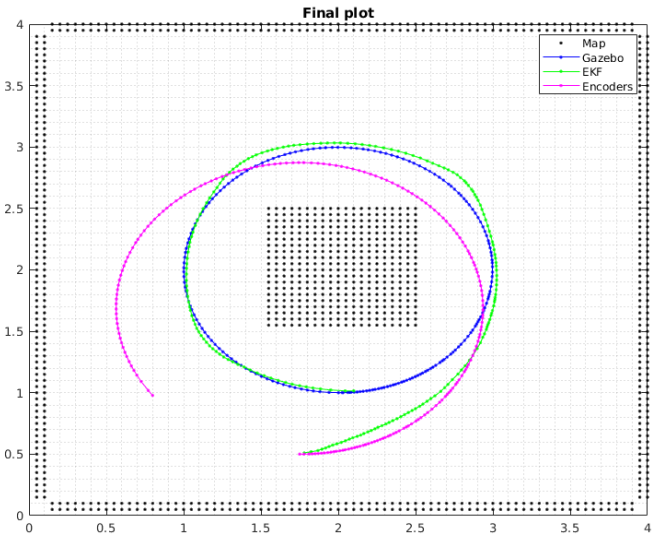

Figure 3: Teste do EFK com odometria corrompida e estimativa inicial incorreta

Como pode ser visto pela Fig. 3 o EKF consegue rapidamente encontrar a posição real do robô, mesmo estando a estimativa inicial incorreta.

4 Conclusão

Face aos resultados obtidos e posterior discussão podemos concluir que a nossa implementação do *Extended Kalman Filter* foi realizada com sucesso, obtendo bons resultados com diferentes ruídos.

Para testarmos ainda mais a nossa implementação, poderíamos realizar também testes com uma trajetória retangular em vez de circular, noutro mapa.

References

[1] J. E. Bresenham. "Algorithm for computer control of a digital plotter". In: *IBM Systems Journal* 4.1 (1965), pp. 25–30. DOI: 10.1147/sj.41.0025.