UNIVERSIDADE D
COIMBRA

MESTRADO EM ENGENHARIA E DE COMPUTADORES

Relatório de Sistema Robóticos Autónomos

Utilização de Campos de Forças Virtuais (VFF) e Histograma de Forças Virtuais (VFH) para a Navegação de Robôs Móveis

Duarte de Sousa Cruz, 2017264057
João Pedro Chaves Castilho, 2017263424

Universidade de Coimbra
03 de Abril de 2022

1 Introduction

Neste trabalho foi-nos pedido para implementar dois algoritmos de navegação de plataformas móveis e construção de mapas: Campos de Forças Virtuais(*VFF*) e Histograma de Forças Virtuais(*VFH*). Estes algoritmos têm como objectivo fazer com que a plataforma *TurtleBot*, conhecendo a sua posição atual, consiga navegar num ambiente com obstáculos, conseguindo desviar-se dos mesmos.

O seu princípio de funcionamento é simples e consiste em associar a cada célula de uma grelha de ocupação um valor, que representa a confiança de ocupação dessa célula tendo por base as leituras efetuadas pelos sensores que equipam o robô.

2 Estratégia de Movimento - Seguimento de um percurso

A estratégia de movimento implementada pretende dotar a capacidade da plataforma de seguir um percurso definido mediante o conhecimento prévio do mapa. Este algoritmo consiste na perseguição do robô de um ponto objetivo variável, que se vai deslocar ao longo percurso com velocidade constante. A plataforma vai se adaptando à orientação desse ponto objetivo.

A distância em relação ao ponto objetivo (x^*, y^*) vai ser definida pelo erro:

$$e(k) = \sqrt{(x(k)^* - x(k))^2 + (y(k)^* - y(k))^2} - d(k)^* \tag{1}$$

Para o controlo da velocidade linear usamos um controlador proporcional + integral:

$$v(k) = k_v * e(k) + k_i * \int et(k)dt \tag{2}$$

O segundo controlador orienta a plataforma de forma a estar orientada para o ponto objectivo (x^*, y^*):

$$\phi(k) = \tan^{-1} \frac{y^* - y(k)}{x^* - x(k)} \tag{3}$$

A velocidade angular fica então sob acção de um controlador proporcional:

$$w(k) = k_s\Big(\phi(k) \perp \theta(k)\Big), \quad k_s > 0 \tag{4}$$

3 Campos de Forças Virtuais(*VFF*)

O planeamento de trajetória baseado em Campos de Forças Virtuais consiste num princípio simples e poderoso. A plataforma é considerada uma partícula que é inserida num campo de potenciais gerados pelo ponto objetivo e pelos obstáculos no mapa. O ponto objetivo vai gerar uma força atrativa e os obstáculos vão gerar forças repulsivas, fazendo com que esta seja atraída para o ponto e repelida pelos obstáculos do mapa.

A combinação da força atrativa com a soma de todas as forças repulsivas vai gerar uma força resultante que navega o robô para o ponto objetivo. Esta força vai ser dada então por:

$$F_R = F_{att} + \sum F_{rep} \tag{5}$$

A força atrativa vai ser:

$$F_{att} = F_{ca}\Big(\frac{x_t - x_o}{d_t}, \frac{y_t - y_o}{d_t}\Big) \tag{6}$$

sendo F_{ca} uma força constante. O valor para esta força constante é arbitrário, mas tem de ser escolhido criteriosamente. O valor que encontramos ser ótimo para esta constante foi $F_{ca} = 20$. Ficamos então com a componente em x e y da força de atração.

A força repulsiva que uma certa célula de ocupação exerce no robô é dada por:

$$F_{rep} = F_{cr}\frac{C_{i,j}}{d^2(i,j)}\left(\frac{x_j - x_o}{d(i,j)}, \frac{y_j - y_o}{d(i,j)}\right) \tag{7}$$

O valor que encontramos serem estáveis para os ganhos k_v, k_i e k_s no *VFF*, foram respectivamente 0.5, 0.3 e 0.7.

3.1 VFF.m

Esta função que implementamos calcula a força resultante que dita o movimento do robô. Tem como parâmetros de entrada o robô, as células com obstáculo da área ativa e a posição objetivo final. Retorna as duas componentes em x e y da força resultante. Por sua vez, esta função chama duas outras funções criadas, *getAtract.m* e *getRepulsive.m*. Estas duas funções retornam respetivamente a força atrativa em x e y, e a força repulsiva resultante em x e y. Após termos estas forças, a força resultante vai ser igual à soma da força atrativa com a força repulsiva.

```
% Gets the atractive and the repulsive forces
[fax,fay] = getAtract(tbot,goalPose);
[fox,foy] = getRepulsive(tbot,x_active, y_active);

% Sums the forces
frx=fax+fox;
fry=fay+foy;
```

3.2 getAtract.m

Esta função consiste no cálculo da força atrativa da forma descrita na eq.6. Esta função tem como parâmetros de entrada o objeto robô e as coordenadas da posição de objetivo final. Vai retornar a força atrativa em x e em y.

3.3 getRepulsive.m

Esta função vai calcular a força repulsiva da forma descrita na eq.7. Esta função tem como parâmetro de entrada o objeto robô e as coordenadas dos pontos com obstáculo que se encontram dentro da janela ativa. O ciclo *for* vai calcular o versor e a as duas componentes x e em y da força para todos os pontos que entram na função. No fim a força repulsiva vai ser a soma de todas as forças e vai retornar a mesma.

```
%GETREPULSIVE gets the repulsive force for VFF
Fcr=-0.5;
[x1, y1, ~] = tbot.readPose();
N=15;

fx=zeros(1,size(x_active,1));
fy=zeros(1,size(y_active,1));

for i=1:size(x_active,1)
    verx=(x_active(i)-x1)/(sqrt((x_active(i)-x1)^2+(y_active(i)-y1)^2));
    very=(y_active(i)-y1)/(sqrt((x_active(i)-x1)^2+(y_active(i)-y1)^2));

    fx(i)=(Fcr*1*verx)/(sqrt((x_active(i)-x1)^2+(y_active(i)-y1)^2));
    fy(i)=(Fcr*1*very)/(sqrt((x_active(i)-x1)^2+(y_active(i)-y1)^2));
end

fox=sum(fx);
foy=sum(fy);
```

3.4 getActiveArea.m

Esta é uma função auxiliar criada por nós para obter as células ativas, numa dada área ativa em volta do robô. Apenas são retornadas as coordenadas das células ativas com obstáculos.

```
function [active_cells] = getActiveArea(robotPose,map,N)

[robot_x_cell, robot_y_cell] = world2grid(robotPose(1), robotPose(2),size(map,1));

low_limit_x = max(1,floor(robot_x_cell-N/2));
high_limit_x = min(size(map,1),floor(robot_x_cell+N/2));
low_limit_y = max(1,floor(robot_y_cell-N/2));
high_limit_y = min(size(map,1),floor(robot_y_cell+N/2));

[xc, yc] = find(map);
cond_x = (xc>low_limit_x & xc<high_limit_x);
cond_y = (yc>low_limit_y & yc<high_limit_y);
cell_x = xc(cond_x & cond_y);
cell_y = yc(cond_x & cond_y);
active_cells = [cell_x, cell_y];

window_cm = N/2*5;
x1=robotPose(1)-window_cm/100;
y1=robotPose(2)-window_cm/100;
x2=robotPose(1)+window_cm/100;
y2=robotPose(2)+window_cm/100;

sx = [x1, x2, x2, x1, x1];
sy = [y1, y1, y2, y2, y1];

figure(1)
hold on
plot(sx, sy,'b--',"LineWidth",2)

end
```

3.5 grid2world.m

Converte coordenadas do mapa discretizado para as coordenadas do mundo.

```
x = cell_x.*4./map_size;
y = cell_y.*4./map_size;
```

3.6 moveVFF.m

Esta é a função principal para todo o movimento do robô. É aqui que implementámos toda a estratégia de controlo descrita na primeira secção.

Começamos com a colocação dos pontos objetivos no mapa e o desenho dos obstáculos. Se colocarmos mais que 1 ponto objetivo, todos os pontos selecionados antes do último serão *via points*. Para os *via points* não exigimos ao robô que anule o erro. No entanto, para o ponto objetivo final o erro tem de ser menos que 10 cm para contar como sucesso.

```
n_points = input('How many points?');
figure(1);
[map_y,map_x] = find(map);
scatter(map_y./20,map_x./20,120,"black","filled")
axis([0, 4, 0, 4]) % the limits for the current axes [xmin xmax ymin ymax]
grid on;
[xi,yi]=ginput(n_points);
```

```
for j=1:n_points
    goalPose(j,1) = xi(j);
    goalPose(j,2) = yi(j);
    hold on
    if (j == n_points)
        plot(goalPose(j,1),goalPose(j,2),'gx', 'MarkerSize', 5); %display locations of points
    else
        plot(goalPose(j,1),goalPose(j,2),'bx', 'MarkerSize', 5); %display locations of points
    end
end
```

Depois vamos criar o ponto objetivo variável que vai depender da força e da posição atual do robô. Para o ponto objetivo final o erro tem de ser menos que 10 cm para contar como sucesso.

```
nPose(1) = x + frx;
nPose(2) = y + fry;
```

Continuamos com a obtenção da próxima posição usando a estratégia de movimento descrita anteriormente para calcularmos a velocidade linear e angular da plataforma.

3.7 Resultados e Observações

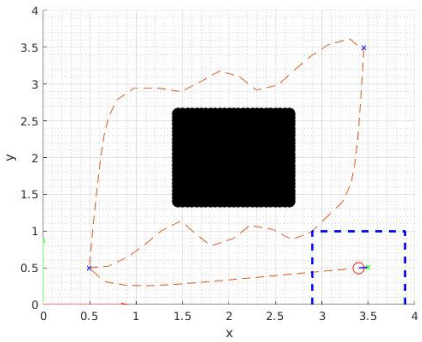

Figure 1: Mapa do quadrado

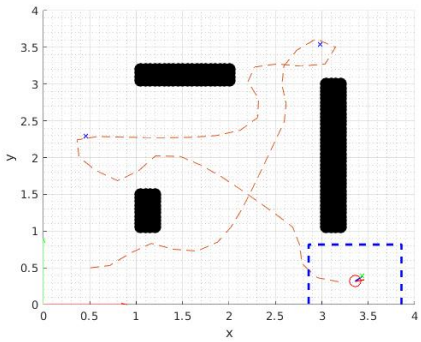

Figure 2: Mapa de obstáculos

Na figura.1 e figura.2 podemos verificar que o robô consegue ter o comportamento desejado e conseguir passar por todos os pontos intermédios (marcados com uma cruz azul) sem ficar preso num mínimo local e chegar ao ponto objetivo final(marcado com uma cruz verde). Verificamos assim que as equações de movimento estão bem implementadas e as constantes otimizadas.

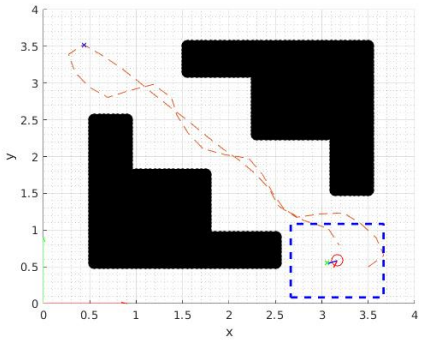

Figure 3: Mapa de passagem estreita

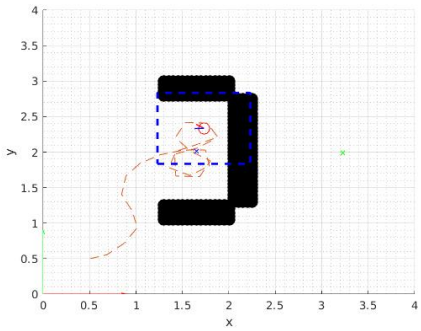

Figure 4: Mapa em forma de u mal sucedido

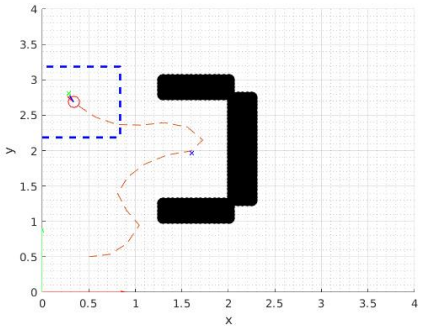

Figure 5: Mapa em forma de u bem-sucedido

Na figura.3 e figura.4 colocamos o robô em dois cenários complicados para o algoritmo dos *VFF*. Apesar de a própria natureza do algoritmo dificultar o andamento em passagens estreitas, conseguimos otimizar as nossas constantes para conseguir passar sem problemas. Por último, no mapa em formato de u conseguimos ver bem um problema deste algoritmo (mínimo local). O mínimo local é quando o robô fica preso numa trajetória circular e não consegue sair devido a estar sofrer forças de todos os obstáculos à sua volta e não consegue definir um percurso para contornar os mesmos. Se mudarmos o ponto objetivo final para as costas do robô, como mostra a figura.5, este consegue sair do u com bastante facilidade.

4 Histograma de Forças Virtuais (VFH)

O algoritmo de Histograma de Forças Virtuais (VFH), baseado no VFF, calcula direções de movimento livres de obstáculos para que a nossa plataforma móvel consiga chegar ao ponto desejado. É considerado um planeador de trajetórias local, pois não tem em consideração o cálculo de um caminho ótimo.

A implementação do Algoritmo VFH, consiste em 3 partes:

1. Construção de um mapa cartesiano discreto que represente as células com obstáculos e células livres.
2. Construção de um histograma polar de 1 dimensão, através do mapa discreto obtido na etapa anterior. Este histograma polar vai representar as densidades dos obstáculos em cada sector angular, em redor do robô.
3. Obtenção dos vales candidatos e da direção do movimento.

Destas etapas, a etapa 1 não é abordada na nossa implementação, pois já nos é fornecido o mapa discretizado do ambiente.

4.1 getPolarHistogram.m

Nesta função, construimos o histograma polar. Primeiro, calculámos o ângulo das células com obstáculo relativamente ao robô:

$$\tan^{-1}(\frac{y_i - y_o}{x_i - x_o}) \tag{8}$$

em que y_i e x_i representam as coordenadas cartesianas das células com obstáculos, e y_o e x_o representam as coordenadas atuais do robô.

```
beta_cells = atan2((world_y-y),(world_x-x));
beta_cells = beta_cells+2*pi*(beta_cells<0);
```

Após termos os ângulos das células, vamos calcular a magnitude de cada célula ativa, através da fórmula:

$$m(i,j) = c_{i,j}^2(a - b * d_{i,j}) \tag{9}$$

em que $c_{i,j}$ é o valor de certeza da célula ativa, no nosso caso será sempre igual a 1, pois necessitamos da primeira etapa do algoritmo. As constantes a e b são escolhidas para que $a - b * d_{max} = 0$, onde $d_{max} = \sqrt{2}(w_s - 1)/2$ é a distância entre a célula ativa mais afastada e o centro do robô.

```
dist_cells = sqrt((world_x-x).^2+(world_y-y).^2);
dmax = sqrt(2)*(window_size-1)/2;
a = dmax;
b = 1;
m = 1.*(a-b.*dist_cells);
```

Após termos o ângulo e a magnitude de cada célula ativa, vamos associar cada célula ativa a um sector do histograma e calcular a densidade de obstáculos em cada sector. O histograma polar vai ter uma resolução angular α, tal que $n = 360/\alpha$ representa o número de *bins* do histograma polar. Para calcularmos o sector correspondente usámos:

$$k = INT(\frac{\beta_{i,j}}{\alpha}), \quad k \in [1, n] \tag{10}$$

Para cada sector k a densidade de obstáculos é calculada através:

$$h_k = \sum_{i,j} m_{i,j} \tag{11}$$

```
k = ceil(beta_cells/alpha);
h = zeros(1,2*pi/alpha);
for i=1:size(m,1)
    h(k(i)) = h(k(i)) + m(i);
end
```

Após termos o histograma construido é nos sugeridos pelo *'paper'* do algoritmo que este histograma seja suavizado, seguindo a seguinte fórmula:

$$h'_k = \frac{h_{k-l} + 2h_{k-l+1} + ... + lh_k + ... + 2h_{k+l+1} + k_{k+l}}{2l+1} \tag{12}$$

em que l é o coeficiente de suavização do histograma e quanto maior o valor mais setores serão afetados por esta suavização. A nossa implementação deste filtro pode ser vista abaixo:

```
L = 2;
h_length = size(h,2);
h_padded = [zeros(1,L),h(1,:),zeros(1,L)];
hp_sum = zeros(1,h_length);
div = ones(1,h_length);
weightArray = [1:L,L:-1:1];
div = div*(2*L+1);
div_mod = [L:-1:1, zeros(1,h_length-(2*L)), 1:L];
div = div-div_mod;

for i=1+L:h_length+L
    hp_sum(i-L) = sum(h_padded((i-L):(i+L-1)).*weightArray);
end
h_smooth = hp_sum./div;
```

Um exemplo do histograma pode ser visualizado na imagem 7. Neste caso usamos $\alpha = 10$ e para a suavização do histograma $l = 2$.

4.2 getSteeringDirection.m

Após termos o histograma suavizado construido, resta nos obter a direção ótima para o movimento. Começamos por calcular qual o sector objetivo, ou seja, em qual sector se encaixa a direção correspondente ao ponto objetivo.

$$\theta_{target} = \tan^{-1} \frac{y_{goal} - y_i}{x_{goal} - x_i} \tag{13}$$

$$k_{target} = INT(\frac{\theta_{target}}{\alpha}) \tag{14}$$

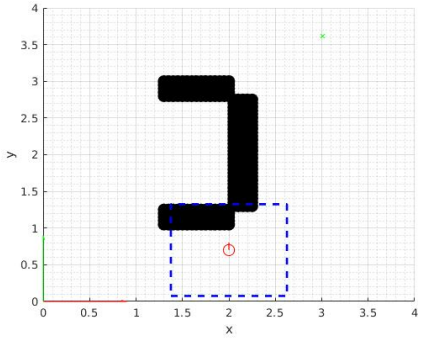

Figure 6: Robô num mapa com obstáculos

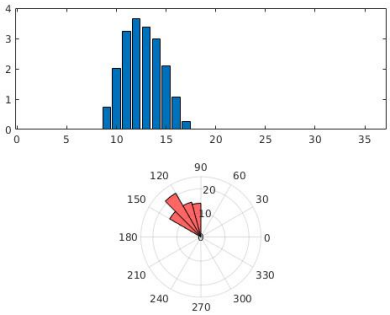

Figure 7: Histograma correspondente

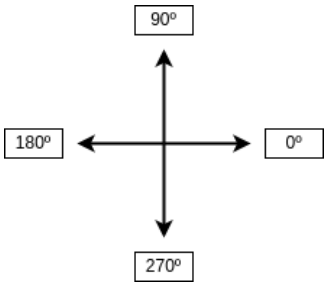

Figure 8: Direções

```
target_t = atan2(goal_pose(2)-y,goal_pose(1)-x);
target_t = target_t + 2*pi*(target_t<0);
k_target = ceil(target_t/alpha);
if k_target == 0
    k_target = 1;
end
```

Depois de termos a direção objetivo, vamos encontrar os vales do histograma. Definimos como vales, secções consecutivas do histograma que não possuem quaisquer obstáculos. Para encontrarmos estes vales, primeiro criamos um histograma auxiliar de 1s e 0s, e de seguida encontramos os sectores em que acontecem passagens de 0 para 1 e de 1 para 0.

```
Hp1 = zeros(1,size(h_smooth,2));
for j = 1:size(h_smooth,2)
    if h_smooth(j) > 0
        Hp1(j) = 1;
    end
end

b1=(find(diff(Hp1)==-1)); %Find beginning of consecutive clear sectors
b1 = b1 + 1;
b2=(find(diff(Hp1)==1)); %Find end of consecutive clear sector

passages = [b1' b2'];
for i = 1:size(passages,1)
    if (abs(passages(i,1)-passages(i,2)) > 3)
        real_passages(i,:) = passages(i,:);
    end
end
```

Depois de descobrirmos os vales temos de encontrar qual o vale ótimo a escolher, para isso temos de calcular a distancia dos vales para o sector objectivo:

```
dist=min(mod((real_passages - k_target),36), mod((k_target - real_passages),36));
[r,c] = find(dist==min(dist(:)),1);
chosenValley = real_passages(r,:)
k_chossen = chosenValley(1,c);
```

De seguida, caracterizamos o vale em vale estreito ou vale largo. Para isso definimos um parâmetro s_{max}, no nosso caso, para $\alpha = 10$ um $s_{max} = 9$. Se a largura do vale for maior que s_{max} então o vale é considerado vale largo, se a largura for menor que s_{max} é considerado um vale estreito.

Se o vale selecionado for largo, então a direção vai ser calculada através da seguinte fórmula:

$$\theta_{go} = k_n \pm 0.5 * s_{max} * \alpha \tag{15}$$

onde k_n é o sector mais à borda pertencente ao vale. Pode ser observado pela fórmula que quanto maior for s_{max} mais o robô se vai afastar dos obstáculos.

Se o vale for considerado estreito, a seguinte fórmula é aplicada para calcular a direção a seguir:

$$\theta_{go} = 0.5 * (k_n + k_f) \tag{16}$$

onde k_n é o sector mais à borda pertencente ao vale, e k_f o sector correspondente à outra "ponta" do vale.

```
smax = 9;
if (abs(chosenValley(1) - chosenValley(end)) > smax)
    fprintf("WIDE VALLEY\n");
    if (k_chossen >chosenValley(1))
        o_theta = k_chossen*rad2deg(alpha)-0.5*smax*rad2deg(alpha);
    else
        go_theta = k_chossen*rad2deg(alpha)+0.5*smax*rad2deg(alpha);
    end
else
    fprintf("NARROW VALLEY\n");
    go_theta = 0.5*(chosenValley(1)*rad2deg(alpha)+chosenValley(end)*rad2deg(alpha));
end
```

5 Resultados e Observações

Realizamos alguns testes para testar a robustez da nossa implementação. Usamos os mesmos mapas que nos testes do VFF.

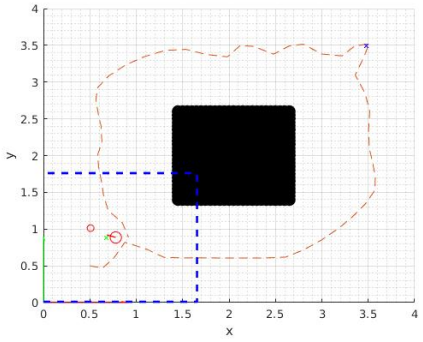

Figure 9: Mapa do Quadrado

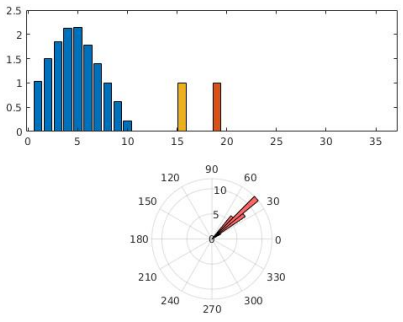

Figure 10: Histograma correspondente

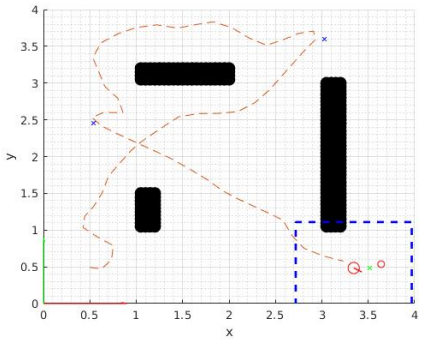

Figure 11: Mapa de obstáculos

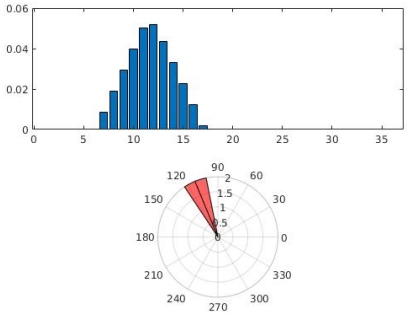

Figure 12: Histograma correspondente

Podemos ver nas Figuras 9 e 11 que o algoritmo implementado consegue concretizar o movimento, enquanto se desvia também dos obstáculos.

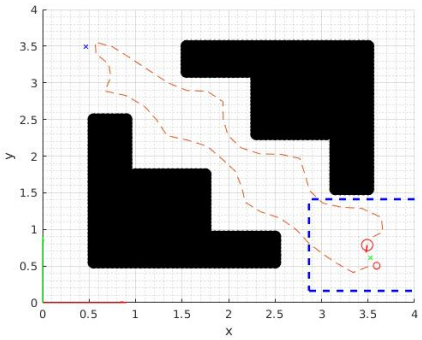

Figure 13: Mapa de passagem estreita

No mapa do corredor, vemos que este teve um comportamento diferente do caso dos VFF. Vemos que o robô contornou mais juntos dos obstáculos.

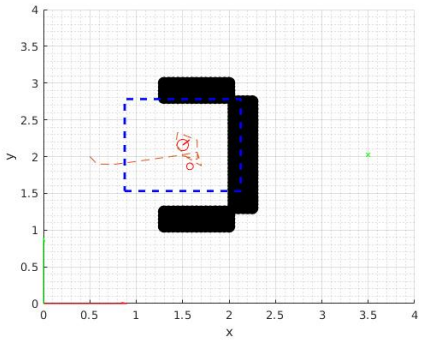

Figure 14: Mapa em forma de u mal sucedido

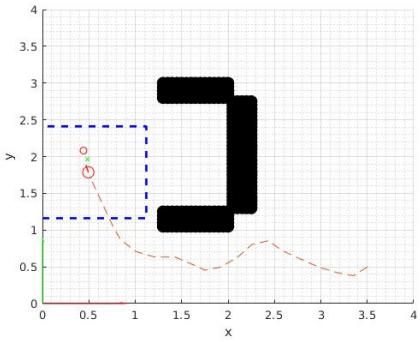

Figure 15: Mapa em forma de u bem-sucedido

Podemos também ver, que tal como no caso dos VFF, o algoritmo encontra um mínimo local no mapa da Figura 14. Isto deve-se ao facto de serem algoritmos que apenas atuam ao nível local e não global.