Trabalho Prático 2 - Planejamento e navegação

Alunos: Davi Fraga Marques Neves
Alexis Duarte Guimarães Mariz

Universidade Federal de Minas Gerais - UFMG

2025

Introdução

O objetivo deste trabalho é implementar diferentes algoritmos de planejamento de caminhos, utilizando técnicas de navegação e controle dos robôs no simulador CoppeliaSim. É preciso planejar o caminho de dois robôs, um holonômico(Robotino) e o outro diferencial(Pioneer P3DX). Os algoritmos são testados em diferentes mapas para verificar o desempenho de cada algoritmo, mostrando as suas vantagens e desvantagens.

Os algoritmos implementados foram Roadmap, usando A* para busca do melhor caminho e Campos Potenciais. Ambos os algoritmos partem de uma heurística determinística: não importa quantas vezes forem executados, os robôs farão sempre o mesmo caminho em um mesmo mapa.

O primeiro algoritmo é o algoritmo de campos potenciais. A intuição desse algoritmo é transformar cada ponto do mapa em um vetor. Essa ideia vem dos campos magnéticos da física onde um imã possui maior ou menor atração dependendo de sua posição relativa. Nessa abordagem, o mapa vai sendo lido conforme o robô anda através de sensores. Ou seja, o mapa só é percebido quando a simulação começa e é mais próximo de uma situação real, onde não conhecemos todo o ambiente e os obstáculos.

O segundo algoritmo, Roadmaps, transforma cada pixel da imagem em um nó de um grafo. Os pixels correspondentes aos objetos são inválidos, ou seja, não fazem parte do grafo. Com isso, todos os nós do grafo são posições livres de obstáculos. A partir desse grafo, utiliza-se um algoritmo de busca, nesse caso, A*. Com essa abordagem, o robô precisa conhecer todas as posições do mundo, inclusive a posição exata dos obstáculos. Nessa abordagem, o caminho que o robô irá percorrer é calculado antes de começar a simulação.

Implementação - campos potenciais

Objetivo e parada

Para implementar esse algoritmo, primeiro precisamos coletar a distância do robô ao goal. O loop principal que mantém a simulação rodando é uma condição que verifica se a distância do robô ao goal é uma distância mínima aceitável. Assim, o robô só para quando a distância é suficientemente pequena. Caso contrário ele fica andando procurando rotas.

```
      while np.linalg.norm(x_goal - q[:2]) > 0.5: # enquanto não estiver a
50 cm do goal
```

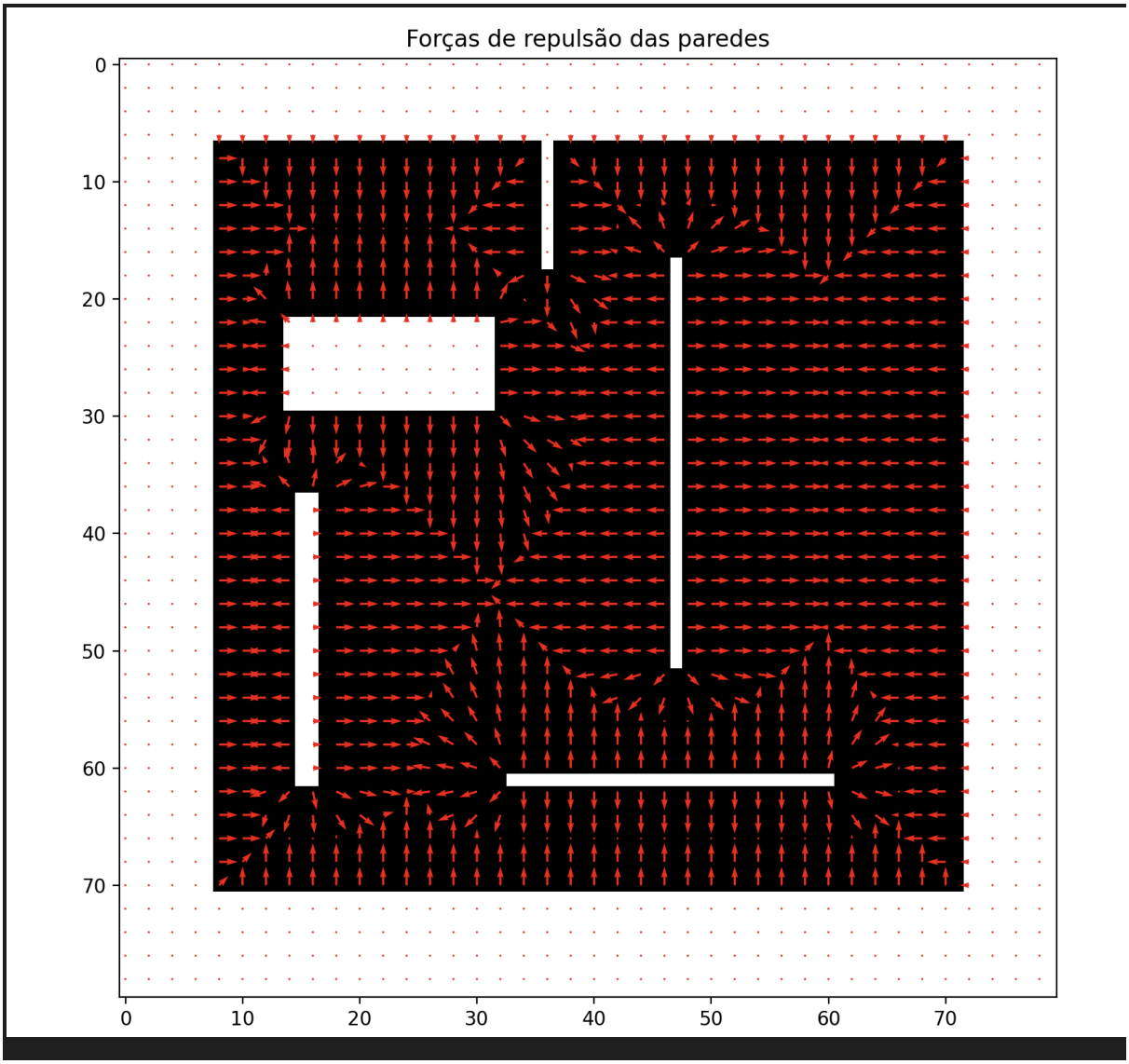
Tomada de decisão

Para que o pioneer decida para onde ele vai depende do cálculo do vetor na sua posição atual seguindo a fórmula: A “força resultante” é a força de atração ao goal (maior quanto mais perto) subtraída da força de repulsão, decorrente da leitura dos sensores. Quanto mais perto de um obstáculo, mais é “empurrado” na direção oposta).
Esse algoritmo pode travar o robô caso o mapa o deixe em uma região onde o campo não “aponta para fora” em momento algum também conhecido como mínimo global. Logo, não é um algoritmo completo. Entretanto, esse algoritmo pode ser bem eficiente na maioria dos casos onde não conhecemos o ambiente completamente.

```
      ### 1. FORÇA DE REPULSÃO ###
            F_rep = np.zeros(2)
            for sensor in sensorHandles:
                state, detectedPoint = sim.readProximitySensor(sensor)
                if state:  # obstáculo detectado
                    detectedPoint = np.array(detectedPoint[:3])
                    dist = np.linalg.norm(detectedPoint)
                    if dist < 0.5:  # considerar apenas obstáculos próximos
                        direction = -detectedPoint[:2] / dist  # vetor para
longe do obstáculo (2D)
                        F_rep += K_rep * direction * (1.0 / dist**2)  #
força repulsiva proporcional

### 2. FORÇA DE ATRAÇÃO ###
      F_att = K_att * (x_goal - q[:2])

### 3. FORÇA RESULTANTE ###
F = F_att + F_rep  # atração + repulsão
```



A imagem acima ilustra a direção das forças de repulsão dos obstáculos.

Calibragem:

A calibragem foi feita manualmente através de experimentação. É um desafio equilibrar as forças de atração, repulsão e o tamanho do raio de repulsão.

```
# parametros para calibrar
K_att = 1.0
K_rep = 0.5
kv = 1.5
kw = 3.5

# limites e parâmetros do comportamento
```

```
    d0 = 0.2 # raio da repulsão
    v_max = 0.6
    w_max = 1.5
```

Implementação - roadmap

Objetivo e parada

Da mesma maneira, o objetivo do algoritmo é alcançar o goal. No entanto, como estamos mapeando uma imagem para um grafo, a condição de parada é chegar no nó mais próximo do goal.

Tomada de decisão

A imagem é pré-processada e transformada em grafo antes mesmo de começar a simulação. Com isso, escolhemos o algoritmo A* de caminhamento em grafos para busca do melhor caminho. Esse é um algoritmo completo e ótimo.
O código a seguir implementa o algoritmo A*.

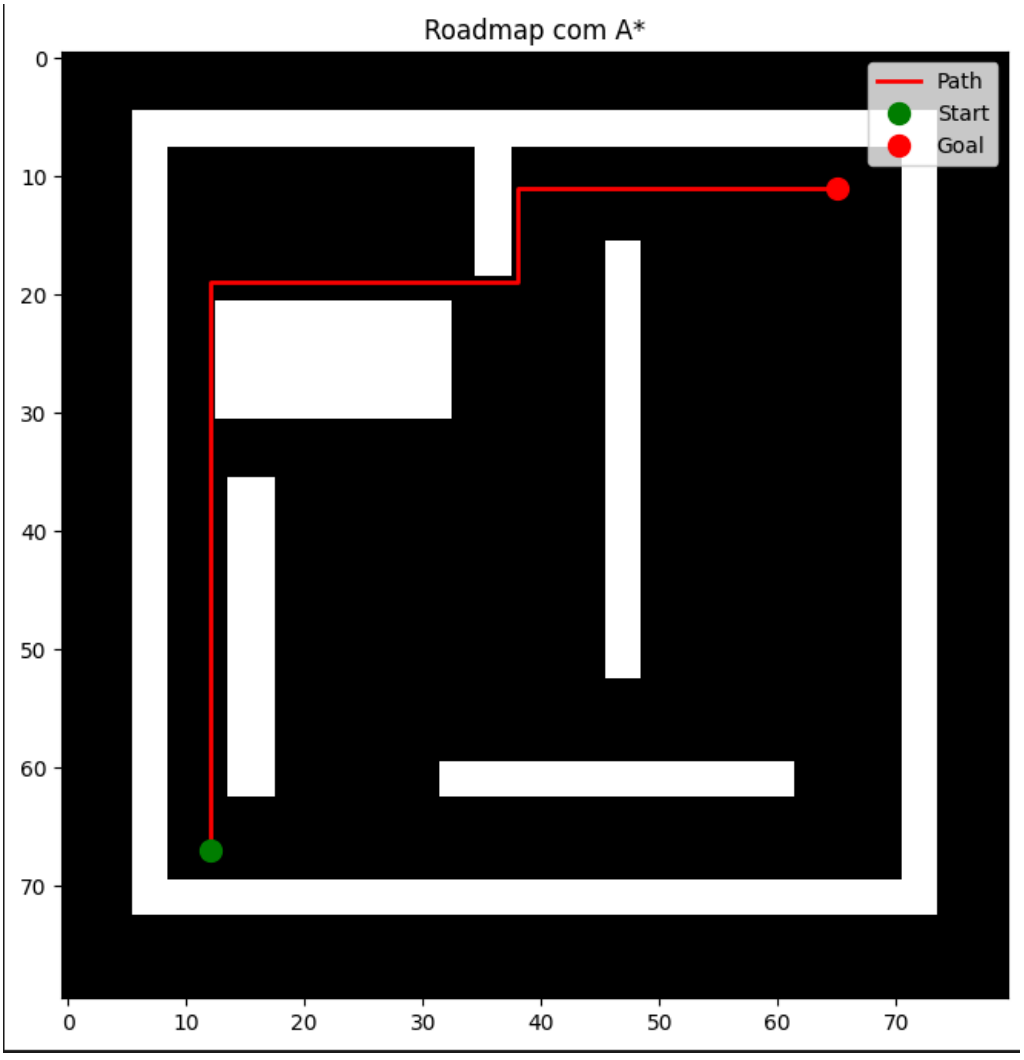
```
import heapq

def a_star(mask, start, goal):
   (R,C) = mask.shape
   pq = [] #(f,g,node,parent)
   heapq.heappush(pq, (0, 0, start, None))
   came_from = {}
   gbest = {start: 0}

   while pq:
       f, g, node, parent = heapq.heappop(pq)
       if node in came_from:  # já visitado com custo melhor
           continue
       came_from[node] = parent
       if node == goal:
           break

       for nb in neighbors_4(mask, node):
           g2 = g + 1 # custo por passo
           if nb not in gbest or g2 < gbest[nb]:
               gbest[nb] = g2
               f2 = g2 + heuristic(nb, goal)
               heapq.heappush(pq, (f2, g2, nb, node))
```

Esse algoritmo é uma **busca guiada —** chamada de busca best-first, porque expande primeiro o nó com menor custo total estimado**.** Esse algoritmo tem crescimento exponencial no pior caso e pode ficar lento se tiver que explorar muitos caminhos diferentes, mas garantidamente sempre retorna a melhor solução - menor custo/distância - caso exista uma.



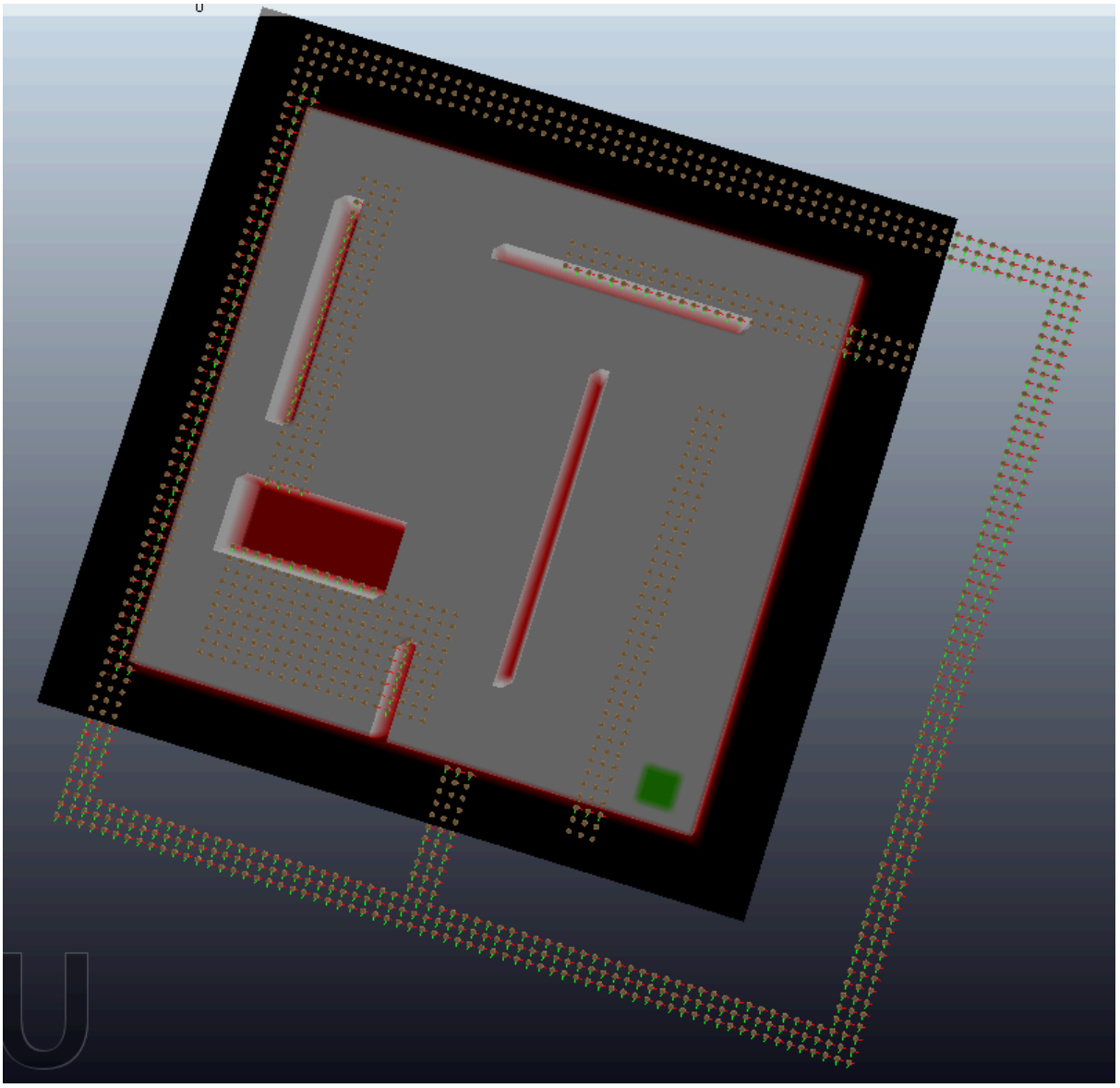
A imagem acima ilustra o caminho retornado pelo A* para a imagem geradora da cena utilizada no trabalho.

Dificuldades

A primeira dificuldade do trabalho foi a criação do mapa. Mesmo com diversos tutoriais, é difícil mapear os pixels da imagem para as coordenadas do simulador. Ao gerar um mapa no CoppeliaSim utilizando um Heighfield, o mapa e os obstáculos são considerados um objeto só, impossibilitando usar a API para obter as coordenadas. Para contornar esse problema, posicionamos sistemas de coordenadas(ReferenceFrame) nas bordas da parede para obter uma estimativa da posição de cada obstáculo. Como os obstáculos são gerados a partir de uma imagem, é possível calcular uma aproximação de cada coordenada. Utilizamos uma imagem 80x80 pixels e a escala 16x16m no simulador para que cada pixel correspondesse a uma célula quadrada de 20cm. Sabendo que o robotino tem largura de aproximadamente 13,5cm, garantimos que ele cabe em uma célula - nó do grafo do Roadmap. Utilizamos ReferenceFrame em outros momentos para ajustar a posição e verificar se o mapeamento estava correto.

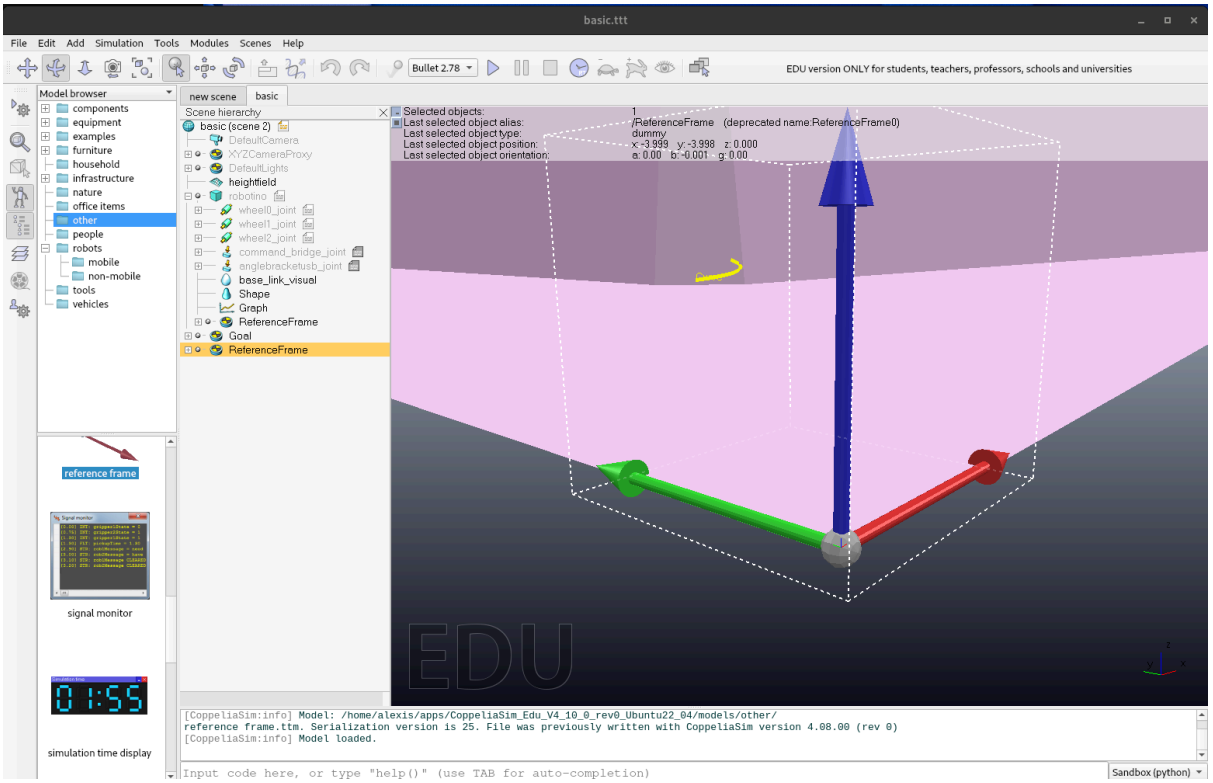
```
    LARGURA_IMAGEM = 80
    WORLD_ORIGIN = (-8.0, -8.0)
    LARGURA_PIXEL_MUNDO = 0.2

    def px_to_world(col_px, row_px):
        row_from_bottom = (LARGURA_IMAGEM-1) - row_px
        x = WORLD_ORIGIN[0] + ((col_px) * (LARGURA_PIXEL_MUNDO))
        y = WORLD_ORIGIN[1] + ((row_from_bottom) *
(LARGURA_PIXEL_MUNDO))
        return float(x), float(y)
```



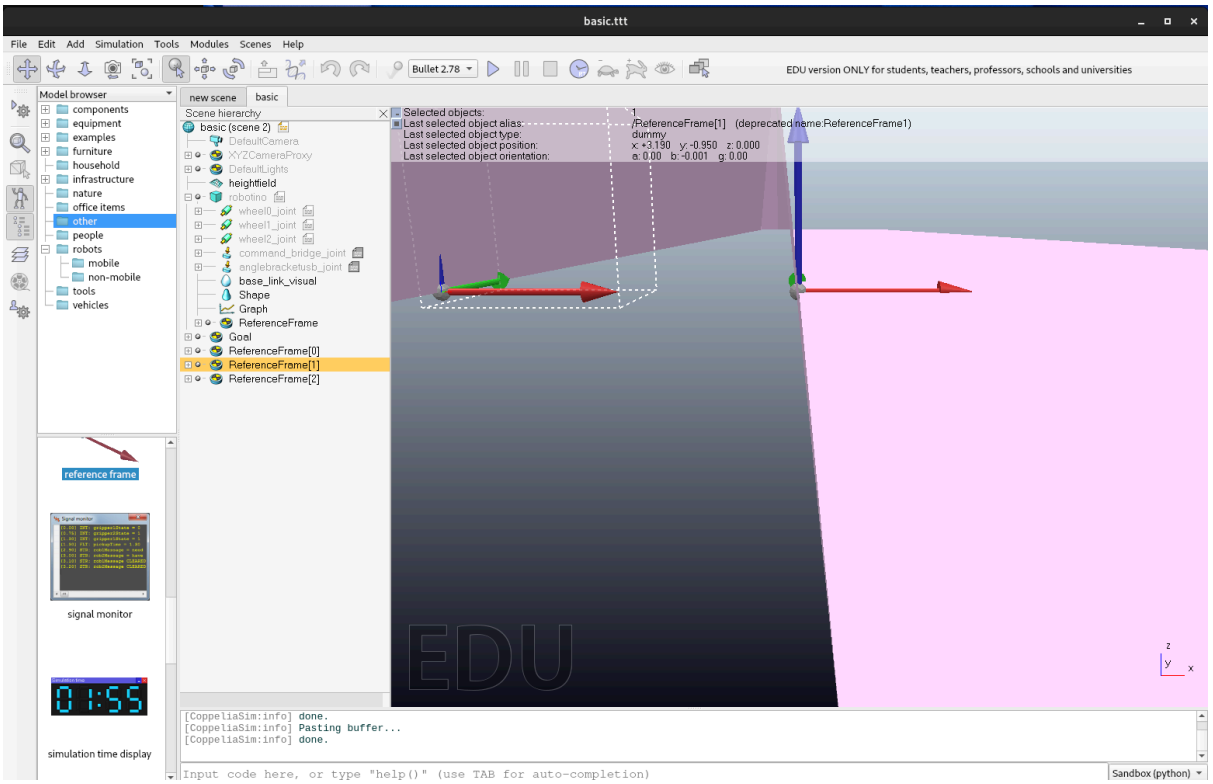
A imagem acima mostra um exemplo de erro de calibragem do mapa visto de baixo. Nesse ângulo de visualização, os obstáculos são os buracos em vermelho. Cada ponto amarelo é um ReferenceFrame representando um pixel de obstáculo, que deveriam estar nos obstáculos em vermelho. Esse erro aconteceu por alguns fatores, como problemas de escala e deslocamento do Heightmap que representa o mundo.

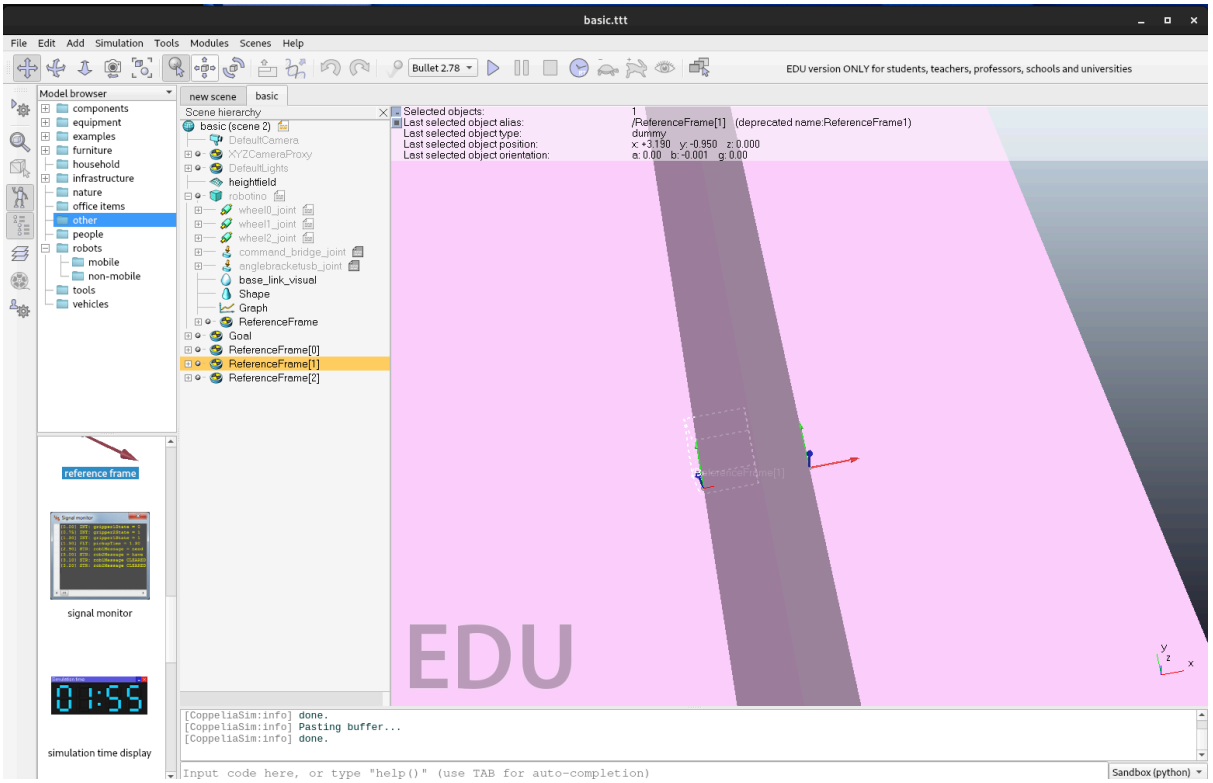
Outro problema, foi a largura das paredes. Primeiro, tentamos medir manualmente(novamente usando ReferenceFrame para obter estimativas) a largura de uma parede de 1 pixel. No entanto, a largura da parede de 2 pixels não correspondia ao dobro da largura da parede de 1 pixel. Isso se tornou um problema grave e foi resolvido tornando os pixels em volta dos obstáculos inválidos.



A imagem acima mostra como obtemos a coordenada correspondente ao pixel (0,0) da imagem. Nesse caso, x=-4 e y=-4.

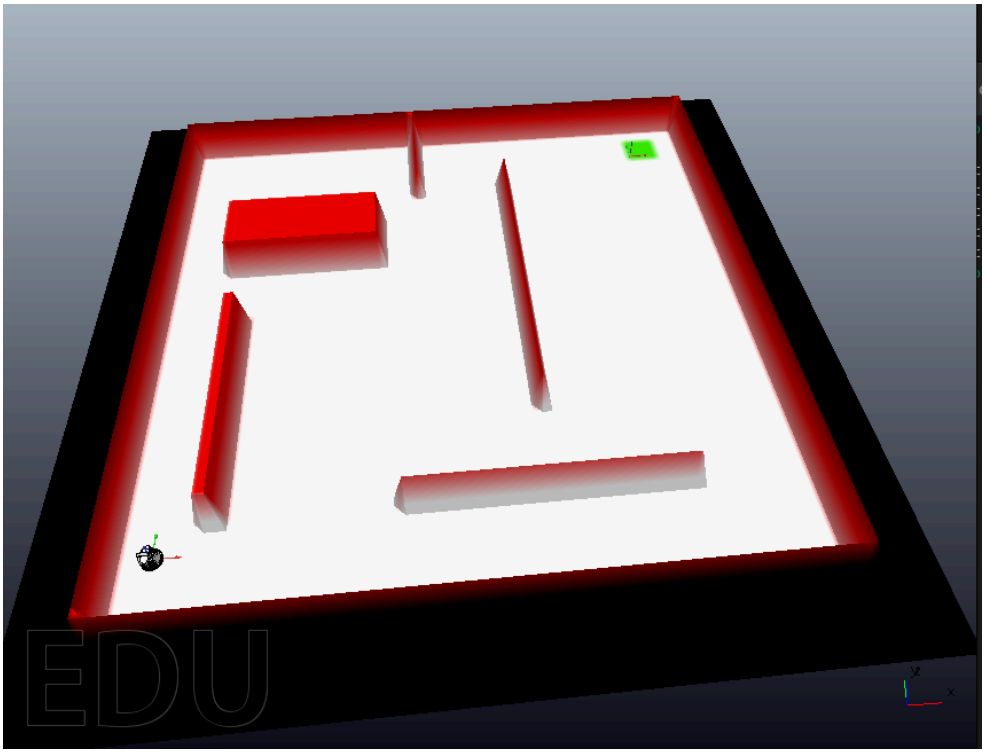
As imagens abaixo ilustram a forma como obtemos a largura de cada parede, calculando a diferença de posição entre os ReferenceFrame posicionados nas bordas da parede.

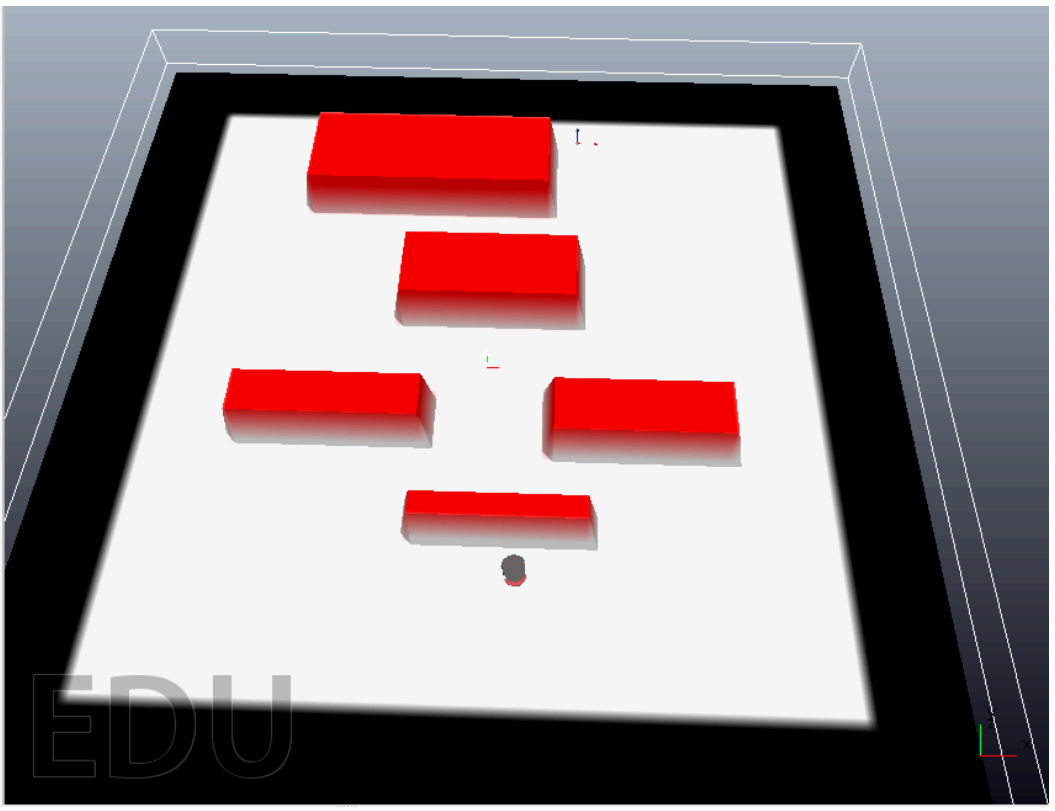




Testes

Ambos os algoritmos foram testados nas cenas a seguir:





O algoritmo Roadmaps obteve desempenho melhor, andando pelo menor caminho. Já o algoritmo de Campos Potenciais encontrou dificuldades e ficou quase batendo na parede(sem encostar) até encontrar o Goal. O que cada um levou para chegar ao Goal não pode ser comparável, pois os robôs andaram em velocidades diferentes. No entanto, o caminho do Roadmaps foi superior ao Campos Potenciais.

Conclusão

O algoritmo dos campos potenciais é muito bom quando não há conhecimento prévio da cena. O desempenho do algoritmo varia de acordo com o mapa e a calibragem das forças: para cada mapa, uma calibragem diferente pode ser melhor. Para mapas que se parecem mais com um circuito sinuoso podem operar incrivelmente bem. Porém, mapas que tem características mais próximas de um labirinto (maior complexidade), pode ser que o robô agarre e não encontre a saída. Já a abordagem do roadmap com o algoritmo A* é completa e ótima - sempre encontra o melhor caminho. Entretanto, essa abordagem depende de ter acesso ao mapa completo previamente.