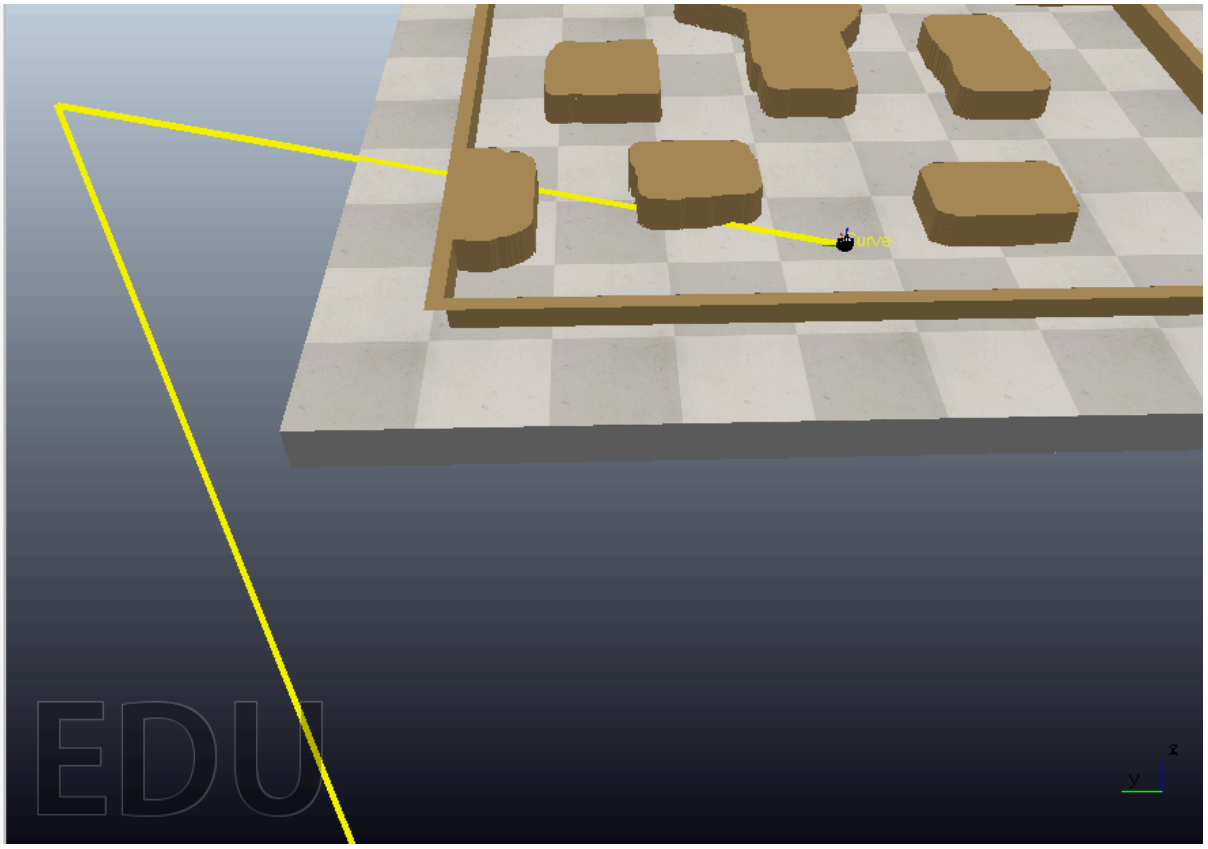
UNIVERSIDADE FEDERAL
DE MINAS GERAIS



TRABALHO PRÁTICO 2
PLANEJAMENTO E NAVEGAÇÃO

ALUNOS:
FELIPPE VELOSO MARINHO
MATRÍCULA: 2021072260

JOÃO VITOR MATEUS SILVA
MATRÍCULA: 2020425801

DISCIPLINA: ROBÓTICA MÓVEL

1. Introdução: detalhamento do problema e visão geral sobre o funcionamento do programa.

2. Implementação: descrição detalhada sobre a implementação. Deve ser discutido as estruturas de dados e algoritmos utilizados (de preferência com diagramas ilustrativos), bem como decisões tomadas relativas aos casos e detalhes que porventura estejam omissos no enunciado.

4. Testes: descrição dos testes realizados e ilustração dos resultados obtidos (não edite os resultados). Você deve propor experimentos considerando diferentes cenários.

5. Conclusão: comentários gerais sobre o trabalho e as principais dificuldades encontradas.

6. Bibliografia: bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites, etc.

1 - Introdução

O presente trabalho tem como objetivo documentar a implementação dos métodos e códigos, visando a familiarização com os conceitos básicos de descrição espacial, transformações e a utilização do software CoppeliaSim para simulação. Para facilitar a integração e o controle das simulações, assim como nas aulas, foi empregada uma API oferecida pelo próprio software. Neste caso, utilizou-se a zmqRemoteApi, uma versão mais atualizada e robusta do que aquela empregada em aulas anteriores. Optou-se pela linguagem Python devido à maior familiaridade e à disponibilidade de materiais de referência.

O principal propósito deste trabalho é colocar em prática os conhecimentos adquiridos em planejamento e navegação ao longo da disciplina. Para isso, foram explorados dois casos de teste distintos.

Ambos os algoritmos auxiliam um robô a sair de um ponto inicial até uma posição final (goal). Para isso, cada uma das estratégias abordadas utiliza diferentes técnicas. No primeiro caso, implementou-se um algoritmo de planejamento de caminhos conhecido como roadmap em um robô holonômico. Essa abordagem envolve a representação prévia do ambiente, onde o espaço é dividido em células válidas e inválidas. A partir das células válidas, construiu-se um grafo de nós e arestas para encontrar o melhor caminho entre o ponto inicial e o ponto final.

Já no segundo caso, aplicou-se o algoritmo de campos potenciais em um robô diferencial. Essa abordagem foi utilizada de maneira reativa (sense/act) e não requer conhecimento prévio do mapa. Utilizando os sensores disponíveis no robô diferencial e influências de campos vetoriais calculados entre forças de repulsão e atração, o robô é guiado até o objetivo. As cargas de sinais opostos entre o robô e o objetivo representam atração, enquanto as cargas de mesmo sinal entre o robô e os obstáculos representam repulsão. Sendo assim, o robô é atraído diretamente ao (goal) e repelido pelos obstáculos identificados pelo sensor.

2 - Implementação

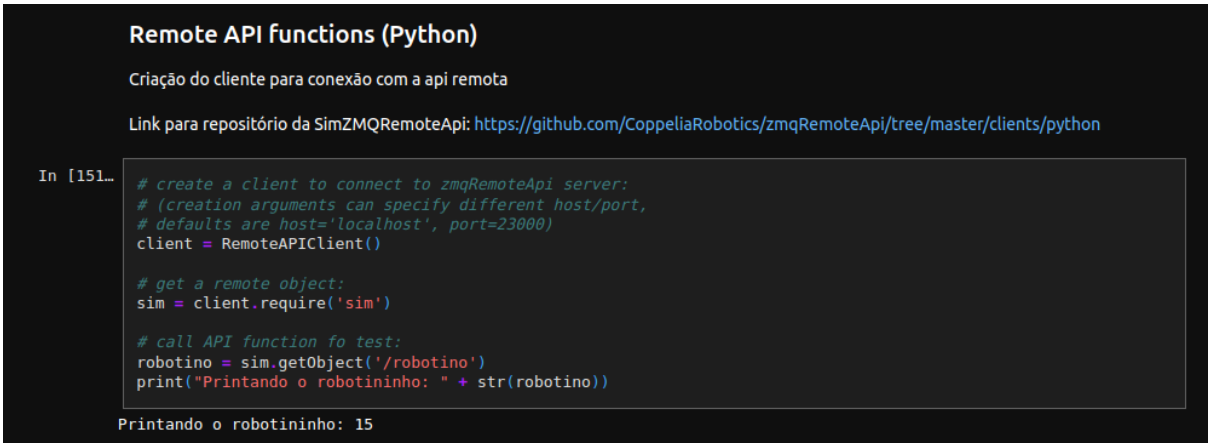
2.1 - Execuções básicas

Primeiramente, o trabalho foi realizado utilizando notebooks para melhor organização e visualização das postagens. Portanto, é necessário executar as cédulas em ordem para que não ocorram problemas de funções utilizadas em outras cédulas.

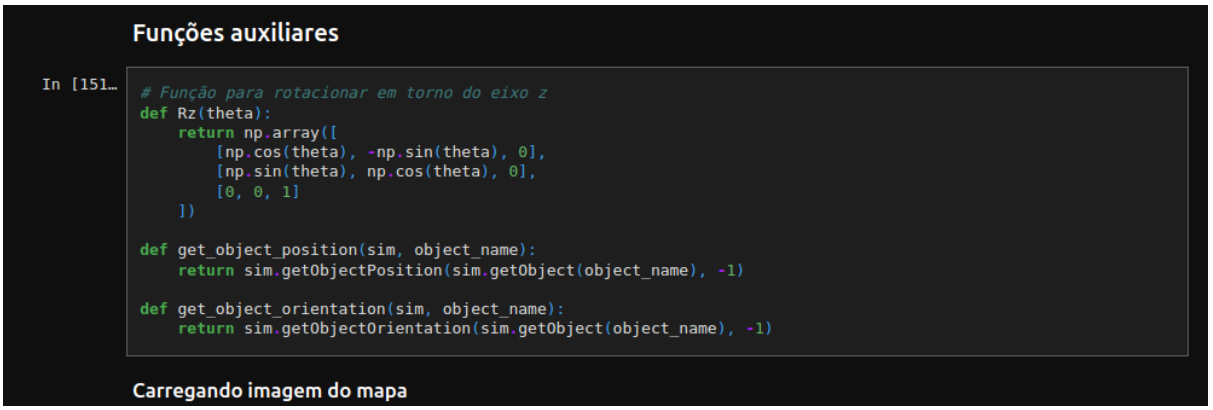
Para que possamos executar o código da forma correta, temos que seguir alguns passos:

Inicialmente é necessário baixar as bibliotecas necessárias para utilização do Simulador, CoppeliaSim através do seguinte código: `pip install coppeliasim-zmqremoteapi-client`

As bibliotecas necessárias para que o projeto rode estão todas destacadas nas primeiras cédulas dos arquivos de “camposDiferenciais” e “roadmap”. Após isso é conectada a API remota e é executada as funções auxiliares de ambos os arquivos.



Conexão com a API Remota



Funções auxiliares do código relativo ao roadmap

2.2 - Roadmap

Assim como os exemplos vistos em sala de aula, no modelo roadmap é necessário ter conhecimento prévio do mapa. Para isso, utilizamos como ponto de partida o exemplo de mapa fornecido em sala de aula, a fim de gerar a grade com as células binarizadas. O

mapa foi configurado de forma quadrada, com dimensões definidas como 40x40. Além disso, é importante destacar que o tamanho da célula foi definido como 2 metros, proporcionando ao robô uma folga mais do que suficiente para passar entre os caminhos.

Carregando imagem do mapa

In [151…

```
fig = plt.figure(figsize=(8,8), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

# Invertendo os valores para visualização (Branco - 0, Preto - 1)
img = 1 - mpimg.imread('cave.png')

# Apenas para garantir que só teremos esses dois valores
threshold = 0.5
img[img > threshold] = 1
img[img<= threshold] = 0

ax.imshow(img, cmap='Greys', origin='upper')
```

Out[151… `<matplotlib.image.AxesImage at 0x2870cf9a4d0>`

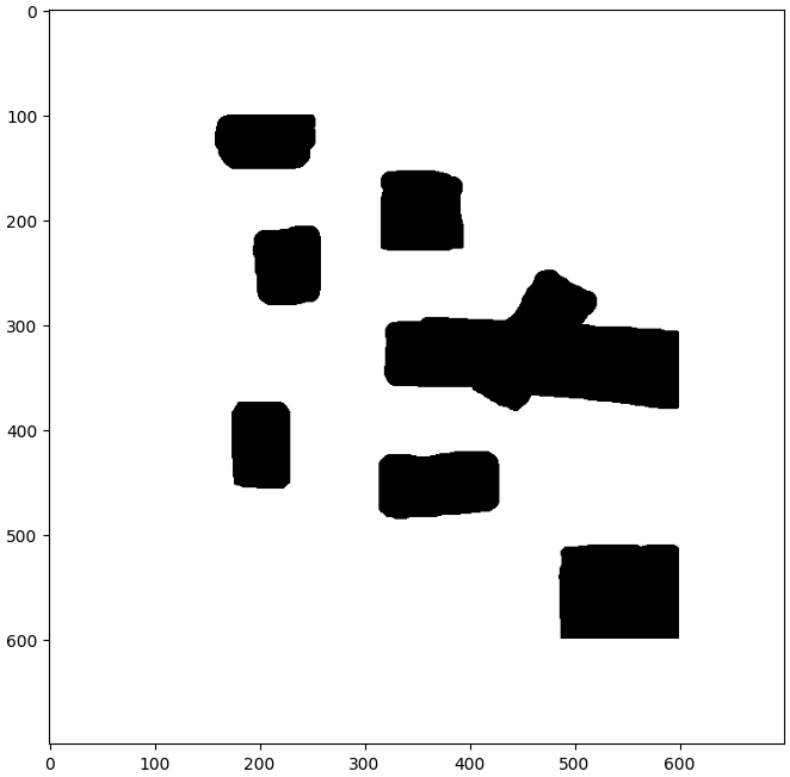
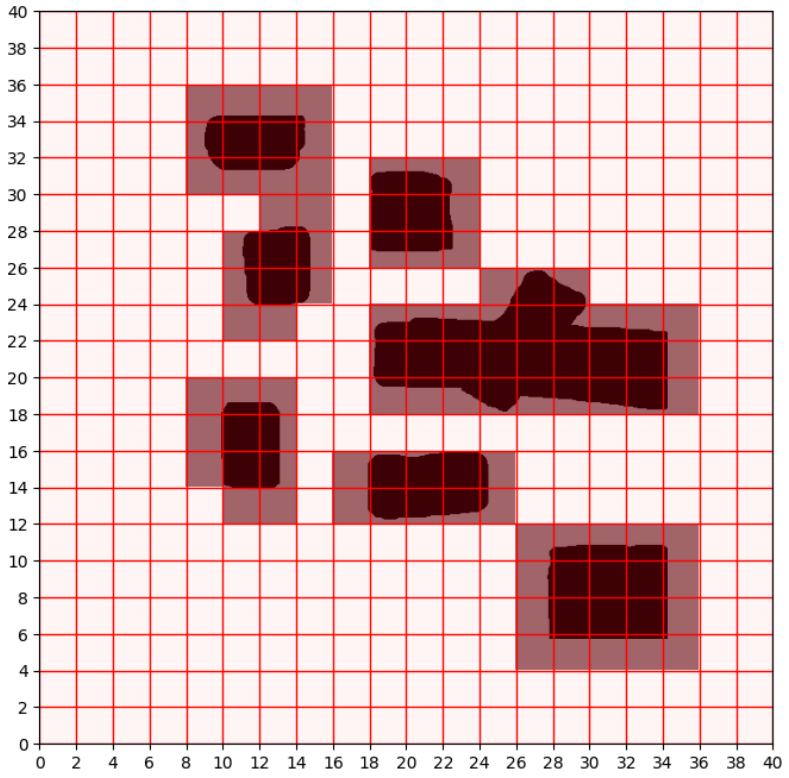


Imagem Carregada

Criando o grid com as células binarizadas

In [151…

```
# Dimensões do mapa informado em metros (X, Y)
map_dims = np.array([40, 40]) # Cave
#map_dims = np.array([22, 43])  # Maze

# Escala Pixel/Metro
sy, sx = img.shape / map_dims

# Tamanho da célula do nosso Grid (em metros)
cell_size = 2

rows, cols = (map_dims / cell_size).astype(int)
grid = np.zeros((rows, cols))

# Preenchendo o Grid
# Cada célula recebe o somatório dos valores dos Pixels
for r in range(rows):
    for c in range(cols):

        xi = int(c*cell_size*sx)
        xf = int(xi + cell_size*sx)

        yi = int(r*cell_size*sy)
        yf = int(yi + cell_size*sy)

        grid[r, c] = np.sum(img[yi:yf,xi:xf])

# Binarizando as células como Ocupadas (1) ou Não-ocupadas (0)
grid[grid > threshold] = 1
grid[grid<= threshold] = 0

fig = plt.figure(figsize=(8,8), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

# Plotando Mapa e Células
obj = ax.imshow(img, cmap='Greys', extent=(0, map_dims[1], 0, map_dims[0]), origin='upper')
obj = ax.imshow(grid, cmap='Reds', extent=(0, map_dims[1], 0, map_dims[0]), alpha=.6)

# Plotando as linhas do grid para facilitar a visualização
ax.grid(which='major', axis='both', linestyle='-', color='r', linewidth=1)
ax.set_xticks(np.arange(0, map_dims[1]+1, cell_size))
ax.set_yticks(np.arange(0, map_dims[0]+1, cell_size))
```

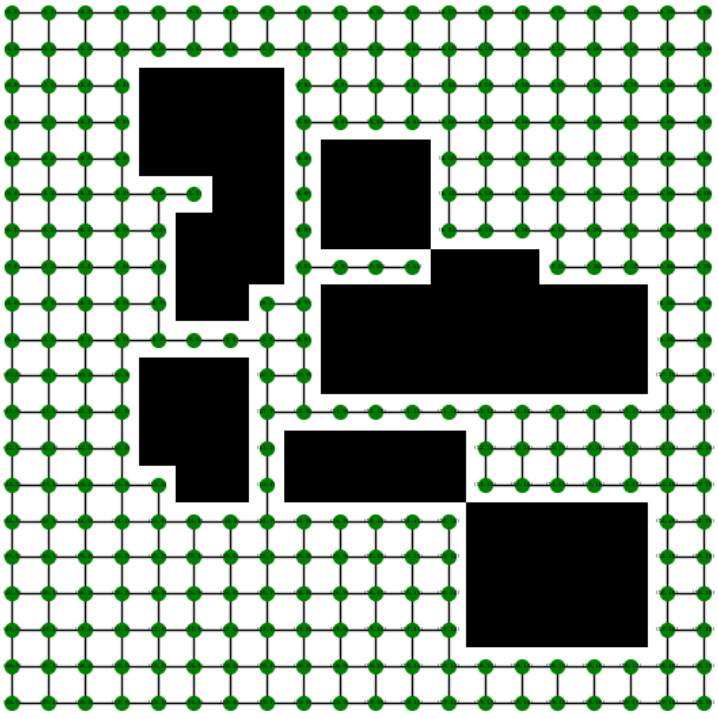


Células Binarizadas Geradas

Criando o Grafo para o Grid feito

```
151…
# Criando vértices em todas as células
G = nx.grid_2d_graph(rows, cols)

# Removendo células que estão em células marcas com obstáculos
for r in range(rows):
    for c in range(cols):
        if grid[r][c] == 1:
            G.remove_node((r,c))


fig = plt.figure(figsize=(8,8), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

# Grid
obj = ax.imshow(grid, cmap='Greys', extent=(0, map_dims[1], 0, map_dims[0]))

ax.grid(which='major', axis='both', linestyle='-', color='r', linewidth=1)
ax.set_xticks(np.arange(0, map_dims[1]+1, cell_size))
ax.set_yticks(np.arange(0, map_dims[0]+1, cell_size))

# Os vértices serão plotados no centro da célula
pos = {node:(node[1]*cell_size+cell_size/2, map_dims[0]-node[0]*cell_size-cell_size/2) for node in G.nodes()}
nx.draw(G, pos, font_size=3, with_labels=True, node_size=50, node_color="g", ax=ax)

# Vértices de início e fim
start = (0,0)
end = (rows-1, cols-1)
print("Start:", start)
print("End:", end)
```

```
Start: (0, 0)
End: (19, 19)
```



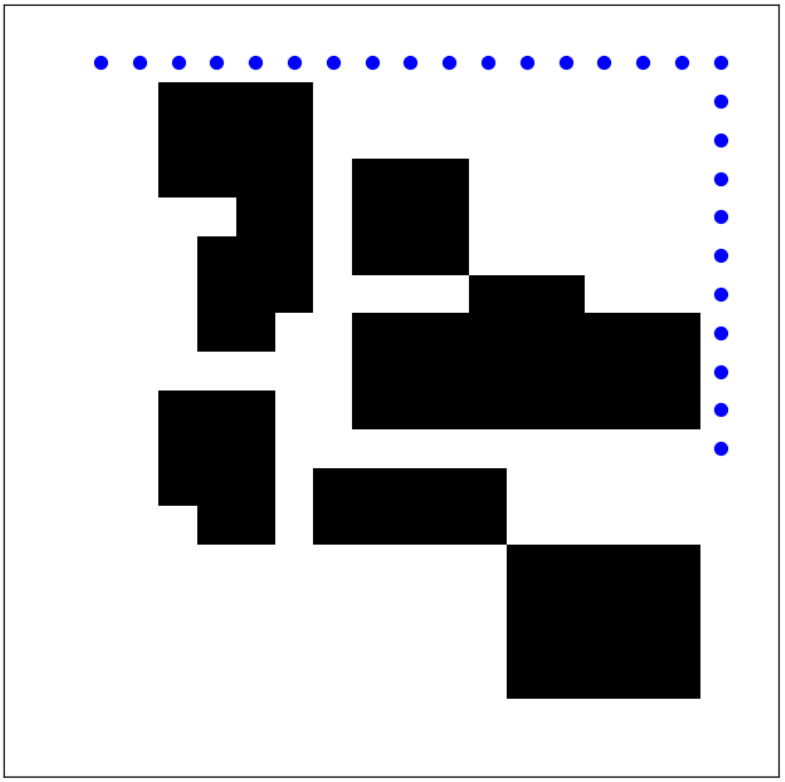
Grafo com as ligações entre os pontos disponíveis no mapa

Semelhante ao feito em sala de aula, após a criação do grafo é definido um nó final dentre os nós disponíveis de maneira aleatória

```python
# Cave
start_node =  (1, 2)
# gerar um valor aleatório para o end_node que esteja dentro do grid e não seja um obstáculo
if grid[r][c] != 1:
    end_node = (random.randint(0, rows-1), random.randint(0, cols-1))
print("End node:", end_node)

# se o nó final for um obstáculo, gerar um novo nó final
while grid[end_node[0]][end_node[1]] == 1:
    end_node = (random.randint(0, rows-1), random.randint(0, cols-1))

#end_node = (17, 18) # Goal 1

fig = plt.figure(figsize=(8,8), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

# Mapa
obj = ax.imshow(grid, cmap='Greys', extent=(0, map_dims[1], 0,map_dims[0]))

# Caminho
path = nx.shortest_path(G, source=start_node, target=end_node)
nx.draw_networkx_nodes(G, pos, nodelist=path, node_size=50, node_color='b')

print("Path:", path)
#prntar o número de posições no path
print("Path length:", len(path))
```

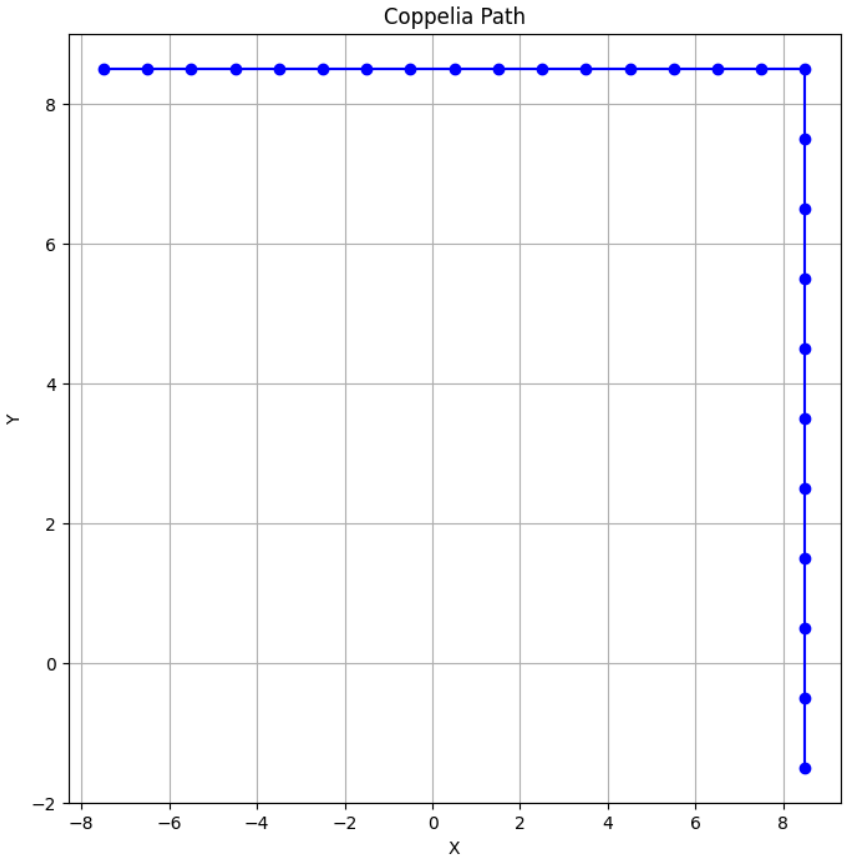
✓ 0.2s Python

```
End node: (11, 18)
Path: [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (1, 11), (1, 12), (1, 13), (1, 14), (1, 15), (1, 16), (1, 17), (
Path length: 27
```

Caminho gerado a partir no nó final aleatório

Após essa etapa, são realizados ajustes no caminho encontrado anteriormente para garantir que os eixos estejam na orientação correta e que as células estejam representadas em metros. Adicionalmente, é feita a conversão do caminho do mapa para o formato adequado de plotagem no CoppeliaSim. Esse caminho é representado de forma mais realista, e apesar de manter o mesmo formato e número de pontos, é possível observar distorções nas coordenadas entre os dois mapas.



Caminho convertido ao referencial do mapa do Coppelia

Por fim, com o caminho definido para ser utilizado no mapa, é possível aplicar os conhecimentos adquiridos em movimentação e controle para que nosso robô, holonômico nesse caso, percorra o caminho armazenado em CoppeliaPath. Para isso seguimos um código também semelhante ao mostrado na aula 7, Locomoção e Modelos Cinemáticos.

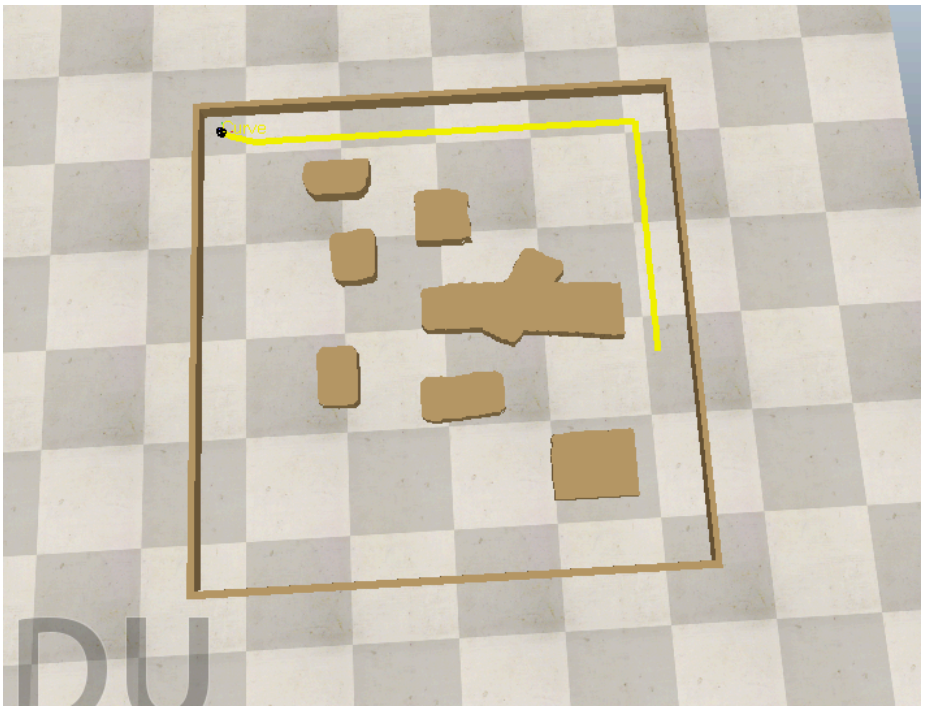
Definimos os parâmetros de cinemática direta e o ganho do controlador para garantir um movimento suave e preciso do robô. No loop principal, iteramos sobre os pontos do caminho, atualizando continuamente a posição e a orientação do robô enquanto ele se move em direção ao próximo ponto do caminho. Dentro do loop, calculamos o erro entre a posição atual do robô e o próximo ponto do caminho. Se o erro estiver abaixo de um limite pré-definido, consideramos que o objetivo foi alcançado e encerramos o loop. Caso contrário, calculamos o vetor de controle com base no ganho e no erro e determinamos as velocidades lineares e angulares necessárias para alcançar o próximo ponto do caminho.

```python
# Conectando-se ao CoppeliaSim
# Run a simulation in asynchronous mode:
clientID = sim.startSimulation()


if clientID != -1:
    print("Connected to remote API server")

    # Get the robot's handle
    robotino = sim.getObjectHandle('/robotino')

    # Get the robot's wheels
    wheel1 = sim.getObjectHandle('wheel0_joint')
    wheel2 = sim.getObjectHandle('wheel1_joint')
    wheel3 = sim.getObjectHandle('wheel2_joint')

    # Retirado da aula 07 de navegação
    # Robotino
    L = 0.135   # Metros
    r = 0.040   # Metros

    # Cinemática Direta
    Mdir = np.array([[-r/np.sqrt(3), 0, r/np.sqrt(3)], [r/3, (-2*r)/3, r/3], [r/(3*L), r/(3*L), r/(3*L)]])

    ganho = np.array([[0.1, 0, 0], [0, 0.1, 0], [0, 0, 0.1]]) # Ganho do controlador

    coppeliaPath = np.column_stack((coppeliaPath, np.zeros(len(coppeliaPath))))
```

```python
    for i in range(len(coppeliaPath)):
        # Posição do Robotino
        print("Goal: ", coppeliaPath[-1])

        while True:
            # Posição inicial do Robotino em cena
            robotinoPos = get_object_position(sim, '/robotino')  #start
            robotinoOri = get_object_orientation(sim, '/robotino')

            q_robot = np.array([robotinoPos[0], robotinoPos[1], robotinoOri[2]])

            # Calcule o erro entre a posição atual e o próximo ponto do caminho
            error = coppeliaPath[i] - q_robot
            errorNorm = np.linalg.norm(error[:2])

            print("Coppelia: ", q_robot)

            if errorNorm < 0.05:
                print("Goal reached")
                break

            # Calcule o vetor de controle usando o ganho e o erro
            qdot = ganho @ error

            # Calculando a velocidade linear e angular
            Minv = np.linalg.inv(Rz(q_robot[2]) @ Mdir)
            u = Minv @ qdot

            # Enviando velocidades para as rodas
            sim.setJointTargetVelocity(wheel1, 10 * u[0])
            sim.setJointTargetVelocity(wheel2, 10 * u[1])
            sim.setJointTargetVelocity(wheel3, 10 * u[2])

            # Atualizando a posição e orientação do robô
            robotinoPos = sim.getObjectPosition(robotino, -1)
            robotinoOri = sim.getObjectOrientation(robotino, -1)

        # Parando as rodas
        sim.setJointTargetVelocity(wheel1, 0)
        sim.setJointTargetVelocity(wheel2, 0)
        sim.setJointTargetVelocity(wheel3, 0)

    pos = sim.getObjectPosition(robotino, -1)
    print(pos)
```

Trecho responsável pelo controle e movimentação do código

Por fim, no CoppeliaSim, o robô exibe o caminho através de rastro que foi utilizado semelhante aos arquivos de aula disponíveis.



Estratégia de roadmap executada no software CoppeliaSim

Obs: no início do caminho demonstrado na imagem anterior, o robô se desloca levemente do ponto inicial do caminho. Isso se deve justamente por essa não ter sido a posição inicial do robô no mapa ao início da execução do programa. O comportamento nesse caso é do robô seguir o menor caminho possível até este primeiro ponto (uma linha reta). Por isso é sempre recomendado colocar o robotino próximo ao ponto inicial definido ao início do programa.

2.3 - Campos Potenciais

Para melhor visualização do código em ação, foram feitas duas versões do código de campos potenciais. Uma com as funções mais diretas para utilizar os conceitos dentro do simulador e outra com os plots dos gráficos de acordo com os objetos em cena. Para essa explicação, será utilizada a segunda versão. As imagens do código relativo a plotagem serão descritas como “Código de plote” já a versão final para rodar a simulação é definida como “Código do Coppelia”.

Primeiramente, assim como no TP anterior, foi utilizado o código para leitura dos sinais captados pelo sensor. A função recebe os handles do sensor (hokuyo) interpreta os sinais e verifica se os dados obtidos estão corretos, feito isso ela retorna os pontos de range e a orientação destes.

```python
def readSensorData(range_data_signal_id="hokuyo_range_data",
                   angle_data_signal_id="hokuyo_angle_data"):

    string_range_data = sim.getStringSignal(range_data_signal_id)

    string_angle_data = sim.getStringSignal(angle_data_signal_id)

    # verifique se ambos os dados foram obtidos corretamente
    if string_range_data != None and string_angle_data != None:
        # descompacte dados de mensagens de alcance e sensor
        raw_range_data = sim.unpackFloatTable(string_range_data)
        raw_angle_data = sim.unpackFloatTable(string_angle_data)

        return raw_range_data, raw_angle_data

    return None
```

Feito isso, foi criado uma função para facilitar as transformações que viram ser utilizadas mais pra frente do código. Essa função tenta de maneira genérica utilizar as posições e orientação de um determinado ponto para realizar uma transformação homogênea para o referencial das variáveis do parâmetro.

```python
# Utiliza as posições e orientações de de um ponto em determinado referencial global e realiza a transformação homogênia
# para o referencial da posição e orientação de posA e  oriA

def transformacao(posA, oriA):
    # Criação da matriz de transformação homogênea
    RWA = Rz(oriA[2])

    TWA = np.array([[posA[0]], [posA[1]], [posA[2]]]) # Transforma a posição em um array

    # Adiciona uma linha [0, 0, 0, 1] à direita da matriz RWA
    HWA = np.column_stack((RWA, TWA))
    HWA = np.row_stack((HWA, [0, 0, 0, 1]))  # Adiciona a linha [0, 0, 0, 1] ao final da matriz HWA

    return HWA
```

✓ 0.0s

De maneira semelhante aos códigos da aula de Campos Potenciais, é criada a função att_force para retornar a força de atração para um ponto definido como nosso ponto destino (goal). O plote é feito para visualização da seguinte maneira.

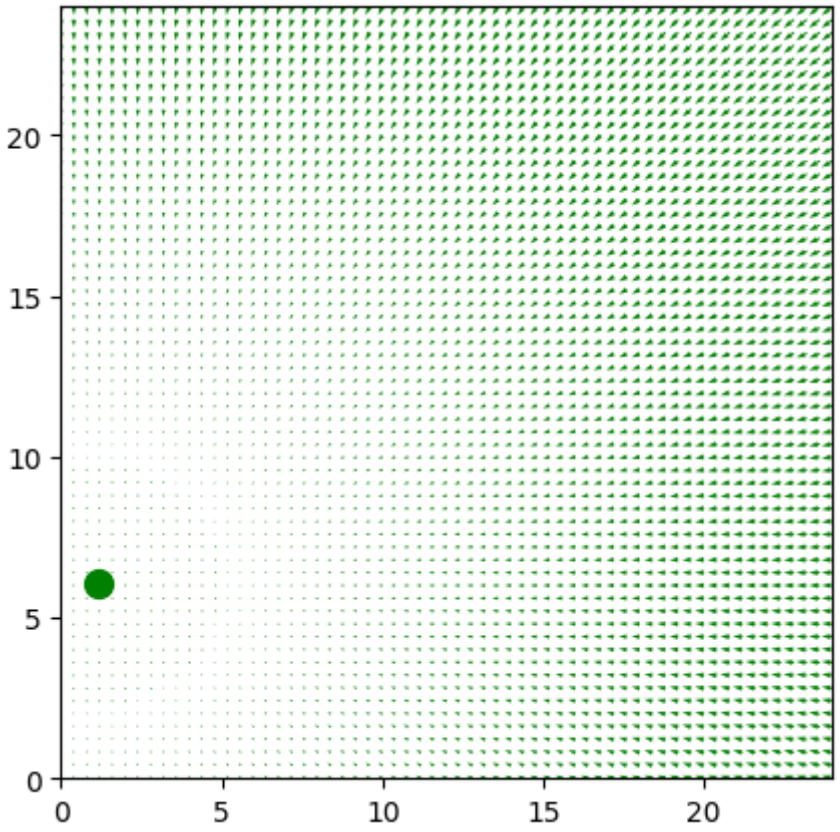
```python
def att_force(q, goal, katt= 0.1):
    Fatt = katt *(goal - q)
    return Fatt
```

✓ 0.0s

(Código Coppelia) Função para calcular força de atração do código final

```python
def att_force(q, goal, katt=.01):
    return katt*(np.array(goal) - np.array(q))

#goal = np.array([8, 2])
goal = get_object_position(sim, '/tree')
print("Goal", goal)

# reshape goal to 2 element array
goal = np.array(goal[:2])


fig = plt.figure(figsize=(8,5), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

XX, YY = np.meshgrid(np.arange(0, WORLDX+.4, .4), np.arange(0, WORLDY+.4, .4))
XY = np.dstack([XX, YY]).reshape(-1, 2)
print(XY)

Fatt = att_force(XY, goal)
Fatt_x = Fatt[:,0]
Fatt_y = Fatt[:,1]

ax.quiver(XX, YY, Fatt_x, Fatt_y, color='g')

plt.plot(goal[0], goal[1], 'og', markersize=10)

ax.set_xlim(0, WORLDX)
ax.set_ylim(0, WORLDY)
```

(Código de plote) Função para calcular força de atração para o goal



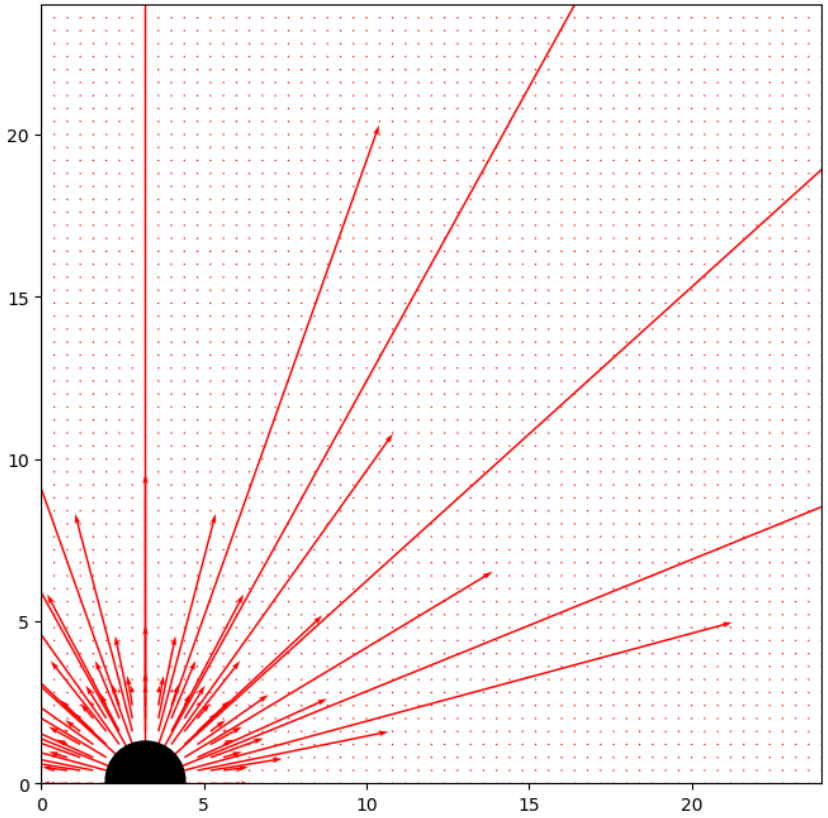
Calculamos a força de repulsão e realizamos o plote de maneira semelhante ao visto em aula.

```python
def rep_force(q, obs, R=3, krep=.1):

    # Obstáculo: (x, y, r)
    v = np.array(q) - np.array(obs[0:2])
    d = np.linalg.norm(v, axis=1) - obs[2]
    #d = np.linalg.norm(v, axis=1)
    # se for necessário fazer reshape
    print("len(d) -> ", len(d))
    if len(d.shape) >= 1:
        d = d.reshape((len(v) ,1))

    # Adicionando uma pequena constante ao raio para evitar divisão por zero
    obs_radius = obs[2] if obs[2] > 0 else 0.001
    rep = (1/d**2)*((1/d)-(1/R))*(v/d)

    invalid = np.squeeze(d > R)
    rep[invalid, :] = 0

    return np.array(krep)*np.array(rep)

# Obstáculo: (x, y, r)
#obs = np.array([3, 4, .5])
obs = get_object_position(sim, '/muro')
obs = np.array(obs[:2])
obs_ori = get_object_orientation(sim, '/muro')
print("ori -> ", obs_ori)
# juntar os vetores obs e terceira_posicao
#obs = np.hstack((obs, obs_ori[2]))
obs = np.concatenate((obs, obs_ori))#[2:3]
print("Obstacle", np.array(obs))

fig = plt.figure(figsize=(8,8), dpi=100)
ax = fig.add_subplot(111, aspect='equal')

Frep = rep_force(XY, np.array(obs))
print("Frep -> ", Frep)
Frep_x = np.copy(Frep[:,0])  # Cuidado com as referências
Frep_y = np.copy(Frep[:,1])  # Cuidado com as referências
```

(Código de plote) Função de repulsão

```
# Threshold para visualização
fmax = .15
Fm = np.linalg.norm(Frep, axis=1)
Frep_x[Fm > fmax], Frep_y[Fm > fmax] = 0, 0

ax.quiver(XX, YY, Frep_x, Frep_y, color='r')
ax.add_patch(patches.Circle((obs[0], obs[1]), obs[4], color='k'))

ax.set_xlim(0, WORLDX)
ax.set_ylim(0, WORLDY)
```

(Código de plote) Função de repulsão



Plotagem da força de repulsão definida na cena do Coppelia

Para o cálculo da força de repulsão utilizando as informações do laser, percorremos os obstáculos e se a distância for menor que o raio de influência, nós calculamos a força repulsiva.

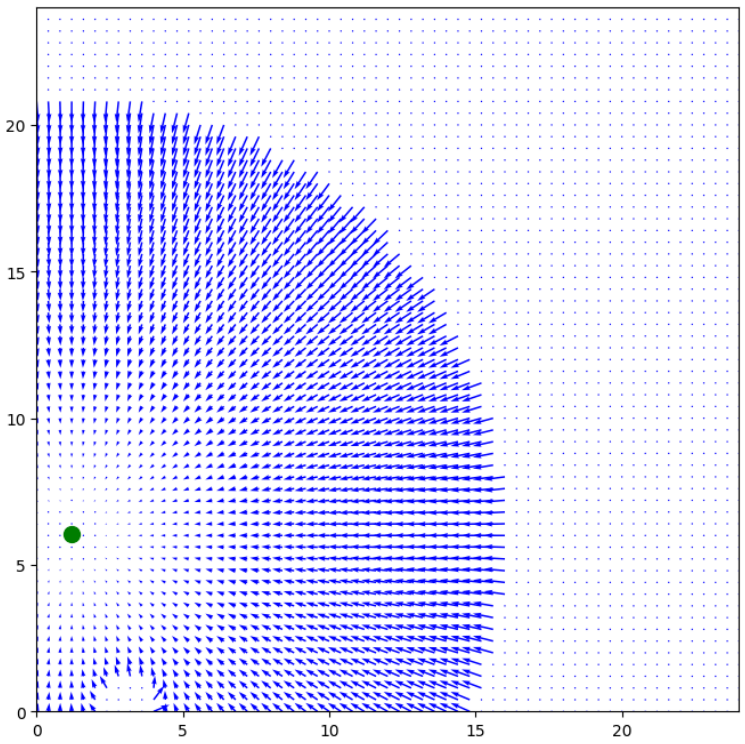
```python
def rep_force(q, obs, R=2, krep=.1):
    Frep = np.zeros(2)
    for obstacle in obs:
        v = q[0:2] - obstacle
        d = np.linalg.norm(v)

        # Se a distância for menor que o raio de influência calcular a força repulsiva
        if (d < R):
            rep = (1/d**2)*((1/d)-(1/R))*(v/d)
            Frep += rep

    return krep*Frep
```

✓ 0.0s

(Código Coppelia) Função de repulsão

```python
def tt_force(q,goal,laser_data,obs,obs_pts,HWL, max_sensor_range = 5):
    Frep = np.zeros(2)
    for i in range(len(laser_data)):
        ang, dist = laser_data[i] #pega os valores de angulo e distância

        if (max_sensor_range - dist) > 0.1:
            x = dist * np.cos(ang) #meu x
            y = dist * np.sin(ang) #meu y
            point = np.array([x,y,0,1])

            if len(HWL) != 0:
                point = HWL @ point
                obs.append(point[0:2])
                obs_pts.append(point)
    Frep = rep_force(q, obs)
    Fatt= att_force(q, goal)
    Ft = Fatt + Frep
    print("Ft new -> ",Ft)

    return Ft
```

Por fim, acima temos a função que nos retorna a força total sendo a soma das duas forças. Nela também há uma pequena adaptação para converter a posições dos pontos do laser para o referencial do robô.



Plotagem da força total

Obs: O local em vazio é justamente onde temos o obstáculo no mapa

Por fim, definimos os handles e variáveis que serão utilizadas no loop principal onde realizamos a leitura do laser, pegamos esses dados, realizamos a transformação homogênea do referencial do laser para o mundo, definimos um goal e calculamos a força total.

```python
# Conectando-se ao CoppeliaSim
# Run a simulation in asynchronous mode:
clientID = sim.startSimulation()

if clientID != -1:
    print("Connected to remote API server")

    # Handle para o ROBÔ
    laser_robot = sim.getObject('/Pioneer_p3dx')

    #Handle para o LASER
    laser = sim.getObject('/Pioneer_p3dx/fastHokuyo')
    print("laser handle -> ", laser)

    # Handle para as juntas das RODAS
    motorLeft = sim.getObject('/Pioneer_p3dx_leftMotor')
    motorRight = sim.getObject('/Pioneer_p3dx_rightMotor')

    L = 0.381 # Distância entre as rodas
    r = 0.0975 # Raio da roda

    maxv = 1 # para limitar a velocidade linear
    maxw = np.deg2rad(45) # para limitar a velocidade angular

    Ft_x = 0
    Ft_y = 0
```

```python
while True:

    laser_robot_position = get_object_position(sim, '/Pioneer_p3dx')
    print("Posição do robô: ", laser_robot_position)

    laser_robot_orientation = get_object_orientation(sim, '/Pioneer_p3dx')

    # Handle para os dados do LASER
    laser_range_data = "hokuyo_range_data"
    laser_angle_data = "hokuyo_angle_data"

    if laser_range_data is not None and laser_angle_data is not None:
        raw_range_data, raw_angle_data = readSensorData()

    laser_data = np.array([raw_angle_data, raw_range_data]).T
    laser_pos = get_object_position(sim, '/Pioneer_p3dx/fastHokuyo')
    laser_ori = get_object_orientation(sim, '/Pioneer_p3dx/fastHokuyo')
    #print("Laser pos -> ", laser_pos)
    #print("Laser ori -> ", laser_ori)

    Hlw = transformacao(laser_pos, laser_ori) # Matriz de transformação do laser para o mun
```

```
v = 0.4 # Velocidade linear
w = 0 # Velocidade angular

q = [laser_robot_position[0], laser_robot_position[1]] # Posição do robô
#goal = goal[:2] # Posição do goal
#goal = [0.5, 4.75] # Goal primeiro mapa
goal = [-3.6, 1.5] # Goal segundo mapa
laser_robot_position2d = laser_robot_position[:2] # Posição do robô/laser em 2D
obs = [] # Lista de obstáculos
obs_pts = [] # Lista de pontos dos obstáculos

Ft = tt_force(np.array(laser_robot_position2d), np.array(goal), laser_data, obs, obs_pts, Hlw)

print("Força total -> ", Ft)

Ft_x = Ft[0]
Ft_y = Ft[1]

# Constantes
kv = 1
kw = 2
```


Para a controladora, utilizamos o modelo de De Luca e Oriolo, no qual a força de repulsão é convertida em velocidades lineares e angulares, determinando assim as velocidades das rodas do robô. Definimos também um condicional para que se o robô estiver de costas para o objetivo ele gira e vai até ele.

3 - Testes

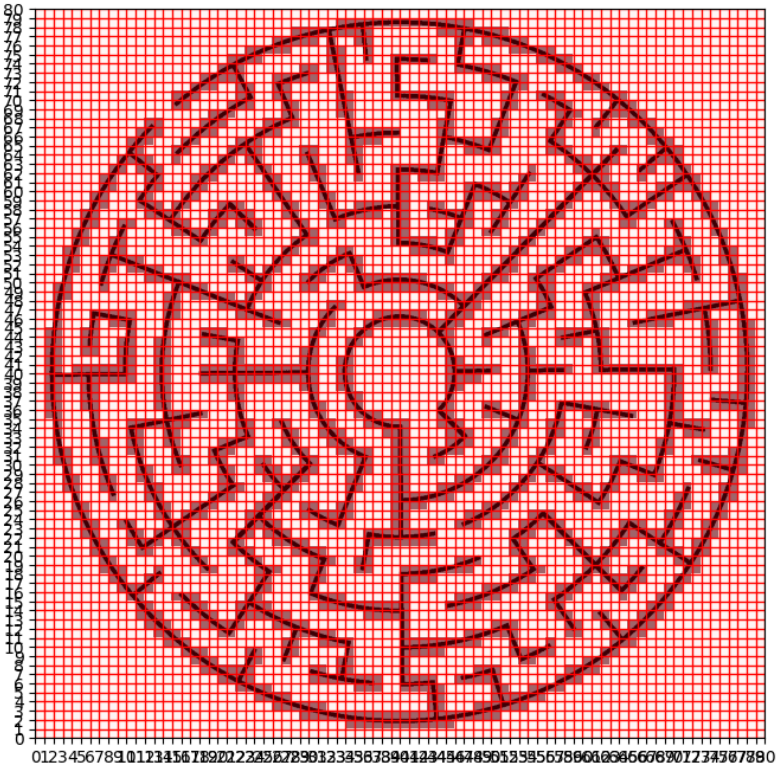
3.1 - Roadmap

No primeiro caso, o código respondeu de maneira bem-sucedida em diferentes posições do mapa, como o ponto mostrado no tópico 2.2 para o mapa cave.png (disponível junto aos arquivos do trabalho). No entanto, para verificar o comportamento do código, foram realizados testes também em outro tipo de mapa. O outro tipo de mapa utilizado foi o circular_maze, também disponível nos exemplos de aula. Para este, foi necessária uma pequena adaptação para uma melhor divisão de células binarizadas.

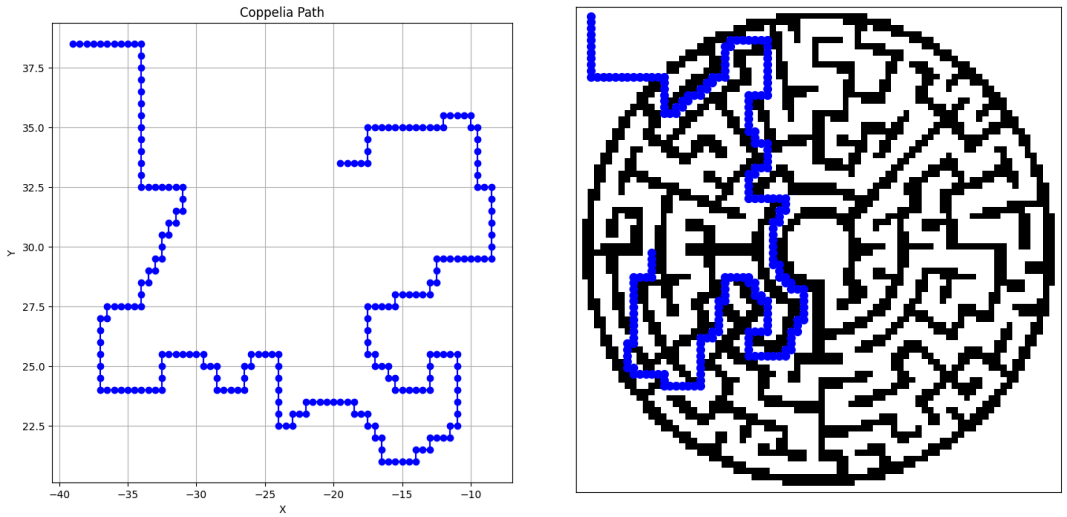
```
# Dimensões do mapa informado em metros (X, Y)
#map_dims = np.array([40, 40]) # Cave
map_dims = np.array([80, 80])  # Maze Circle

# Escala Pixel/Metro
sy, sx = img.shape / map_dims

# Tamanho da célula do nosso Grid (em metros)
# cell_size = 2 # Cave
cell_size = 1 # Maze Circle

rows, cols = (map_dims / cell_size).astype(int)
grid = np.zeros((rows, cols))
```

E outra adaptação para convertemos da maneira correta para os pontos na cena do CoppeliaSim. Todas essas estão indicadas por comentários ao lado.

```python
# Convertendo para os referenciais da cena
path = np.asarray(path) # Convertendo para array numpy
#print("Path cru: ",path)
path[:, [1, 0]] = path[:, [0, 1]] # invertendo os eixos porque no mapa utilizamos os nós invertidos.
path = path * cell_size + cell_size/2 # Convertendo para metros e adicionando o deslocamento para o centro da célula


#coppeliaPath = [20, 20] - path # Cave
coppeliaPath = [80, 80] - path # Maze Circle
for i in range(len(coppeliaPath)):
    coppeliaPath[i, 0] = (coppeliaPath[i, 0] * (-1))/2
    coppeliaPath[i, 1] = (coppeliaPath[i, 1])/2
```



Binarização da grid utilizando o mapa maze_circle.png



Comparação entre o caminho convertido para o referencial no Coppelia e o calculado anteriormente

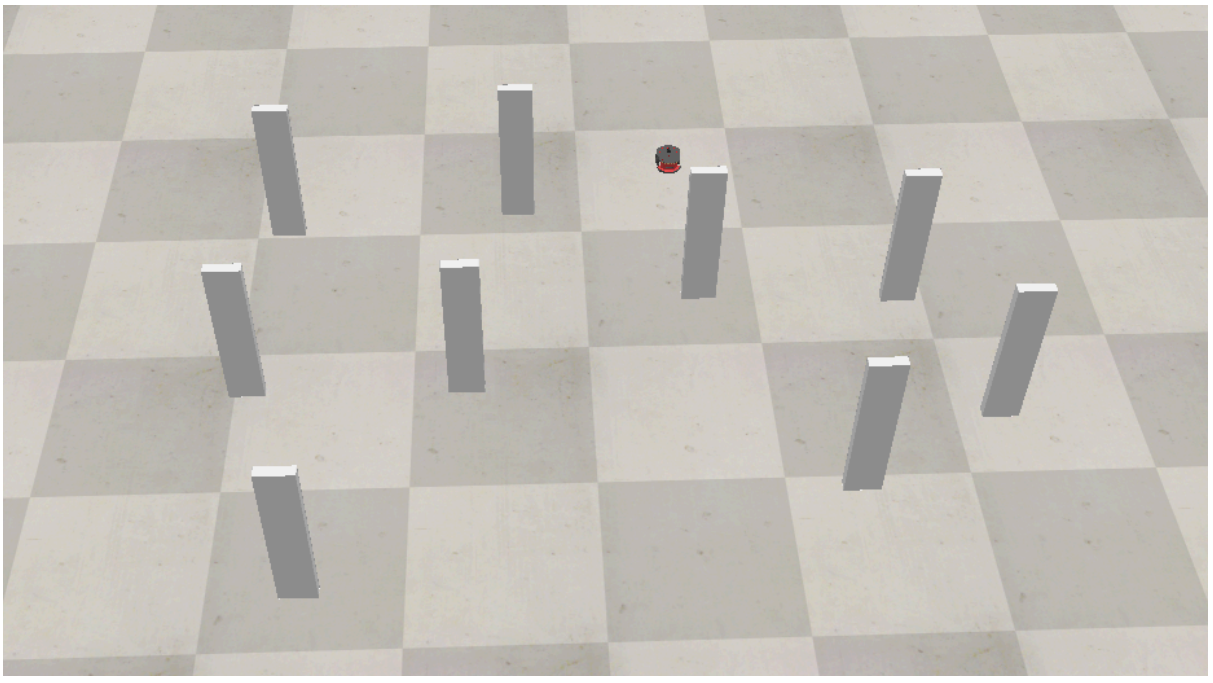
No segundo caso, o robô enfrentou diversos problemas devido à complexidade do mapa, não conseguindo efetivamente alcançar o objetivo. Isso pode ter sido influenciado pela quantidade de células e sua relação com a escala do mapa. Em mapas mais simples, ele conseguiu completar a trajetória com sucesso, embora tenha sido necessário ajustar os parâmetros mencionados anteriormente.

3.2 - Campos Potenciais

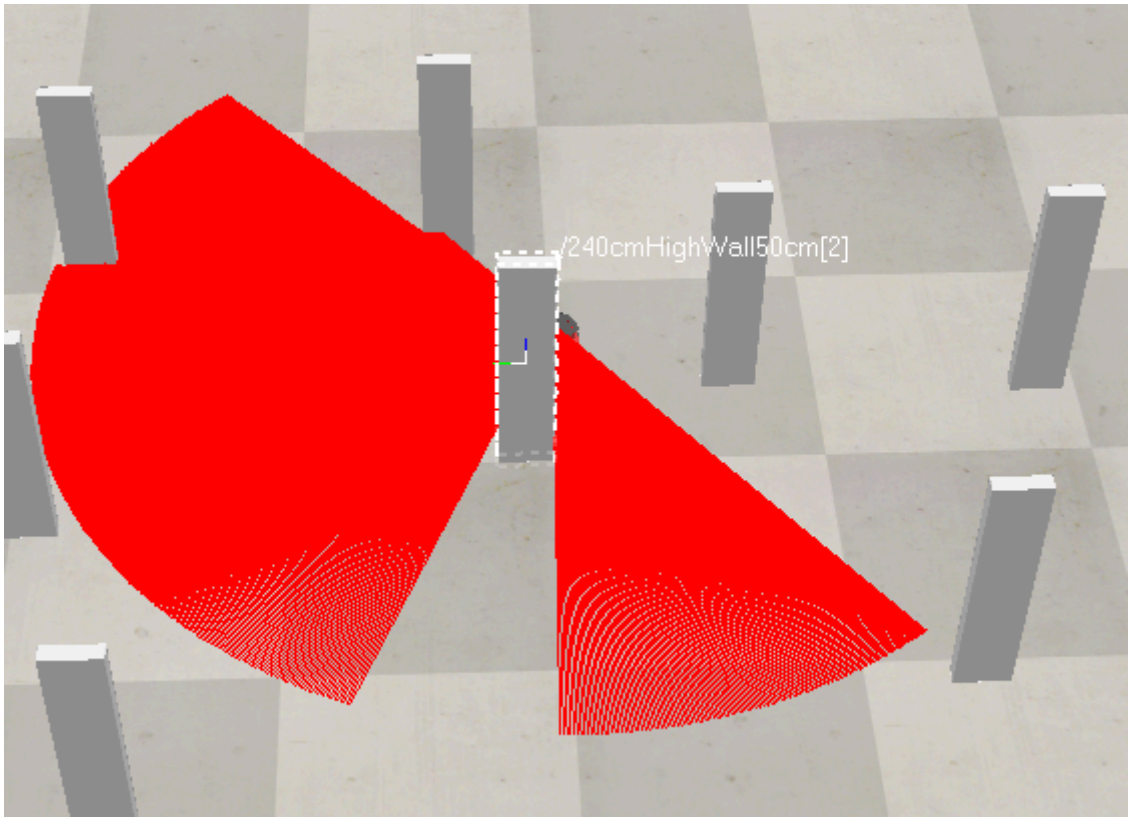
A abordagem de campos potenciais foi feita em dois cenários diferentes. Um mais simples, somente com dois obstáculos e um com uma maior quantidade.

No primeiro caso, o experimento foi conduzido em um ambiente com poucos obstáculos, a fim de avaliar o comportamento básico do algoritmo. O robô foi posicionado em um ponto inicial e o objetivo foi definido em uma posição distante, com somente um obstáculo evidenciado no tópico 2.3. O algoritmo foi capaz de guiar o robô até o objetivo de forma eficaz, demonstrando sua capacidade de navegação em ambientes simples.

No segundo experimento, o ambiente foi configurado com múltiplos obstáculos Primeiro com espaços mais afastados e depois com espaços estreitos. O objetivo era avaliar a capacidade do algoritmo de contornar obstáculos e encontrar um caminho até o objetivo. Embora o algoritmo tenha demonstrado sucesso na navegação em ambientes com obstáculos mais afastados, observamos que em casos com mais de um obstáculo sendo captado pelo sensor ao mesmo tempo, o robô ocasionalmente ficava preso. Em conclusão, os experimentos realizados confirmam a utilidade do algoritmo de Campos Potenciais principalmente quando não existe um conhecimento prévio do mapa.



Segundo cenário (obstáculos mais espaçados)



Segundo cenário (obstáculos menos espaçados) - Robô preso

4 - Conclusão

Para concluir, este trabalho prático foi uma oportunidade para aplicar os conhecimentos teóricos adquiridos em sala de aula em cenários práticos e desafiadores. A implementação de duas estratégias distintas de planejamento e navegação - o algoritmo de roadmap e os campos potenciais - ofereceu uma compreensão mais profunda das complexidades envolvidas na movimentação autônoma de robôs em ambientes simulados.

Ao explorar o algoritmo de roadmap, foi possível entender a importância da representação prévia do ambiente e da construção de um grafo para encontrar o caminho ótimo entre o ponto inicial e final. Por outro lado, a abordagem dos campos potenciais destacou a capacidade de navegação reativa do robô, utilizando influências de campos vetoriais para evitar obstáculos e alcançar o objetivo sem conhecimento prévio do mapa.

Em resumo, este trabalho não apenas consolidou nossos conhecimentos em planejamento e navegação, mas também nos proporcionou uma base sólida para explorar abordagens mais avançadas em robótica móvel. Esperamos que as lições aprendidas aqui sirvam como ponto de partida para futuras pesquisas e projetos neste campo fascinante e em constante evolução.

5 - Bibliografia

Prof. Douglas G. Macharet - Robótica Móvel - Ferramental

Documentação zmqRemoteApi - regular API reference
regular API reference
Prof. Douglas G. Macharet - Robótica Móvel - Descrição espacial e Transformações rígidas

Prof. Douglas G. Macharet - Locomoção – Modelos cinemáticos

Prof. Douglas G. Macharet - Controle – Cinemático

Prof. Douglas G. Macharet - Locomoção – Modelos cinemáticos

Prof. Douglas G. Macharet - Paradigmas Robóticos

Prof. Douglas G. Macharet - Campos Potenciais

Prof. Douglas G. Macharet - Roadmaps

Robótica con Python - Robótica con Python y CoppeliaSim - YouTube