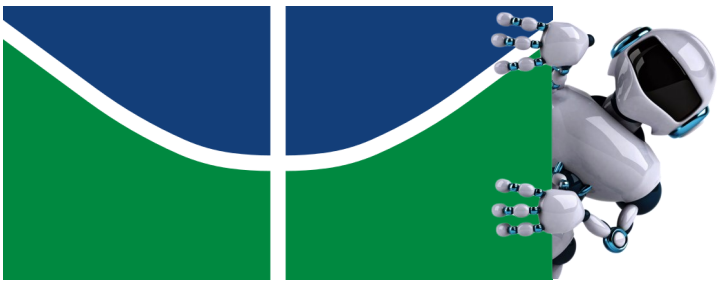
UNIVERSIDADE DE BRASÍLIA



RELATÓRIO DO SISTEMA IMPLEMENTADO

FUNDAMENTOS COMPUTACIONAIS DE ROBÓTICA – TRABALHO 2
NAVEGAÇÃO DO ROBÔ PIONEER SOBRE MAPA TOPOLÓGICO E CRIAÇÃO DE GRADE DE OCUPAÇÃO

Gabriel Rocha Fontenele
15/0126760

(200379)
turma A

2018/2

Entrada / Saída

O início da execução se caracteriza pela escolha de prioridade durante a criação das grades de ocupação. O usuário deve escolher entre 0 para melhorar a precisão ou 1 para aumentar a velocidade. É altamente recomendado o uso da opção 0. A opção 1 fica como alternativa para testes mais informais, visto que as grades de ocupação obtidas neste modo têm nível de fidelidade muito baixa em relação ao ambiente STAGE, em que se baseia.

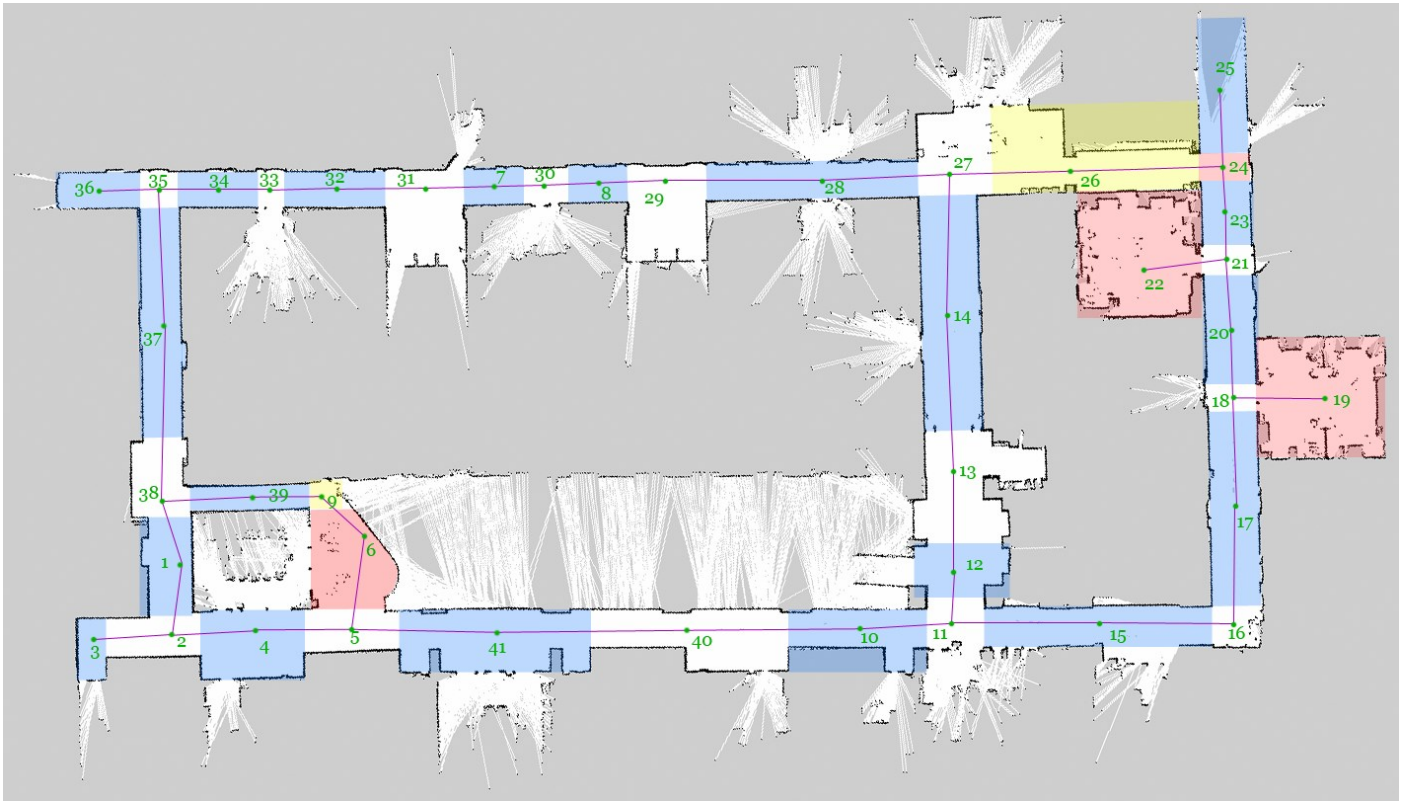
```
Escolha um modo de execucao:
    0 Precisao
    1 Velocidade
Digite: <input>
```

Em seguida, o programa iniciará sua execução. No primeiro passo, é informado em que ponto o pioneer foi iniciado, juntamente ao ambiente simulado, e pedido ao usuário a inserção do nó final.

```
O Pioneer esta no ponto 1 de 41
A qual ponto deseja move-lo? <input>
```

ATENÇÃO! As informações a seguir são de suma importância para o funcionamento correto do programa. O início do código fonte traz na descrição de variáveis globais os valores de *corx* e *cory*. "Cor" significa correção. Essas variáveis são responsáveis pelo correto funcionamento da odometria e localização no grafo do mapa topológico utilizado. O mapa topológico foi feito inteiramente sobre a base do ponto inicial (0.0; 0.0). Isso significa que, ao abrir o ROS, as coordenadas do Pioneer devem estar zeradas. Entretanto em alguns computadores pode aconter que o pioneer inicie nas coordenadas (-55.0; -60.0). Após aberto o ROS e o pioneer já iniciado em um ambiente de simulação, a exemplo do Stage, o programa poderá ser executado com o pioneer estando em um nó inicial diferente do que marca (0.0; 0.0); isto é, a configuração das variáveis cor agem assim que o ROS e o ambiente de simulação são abertos, mas o nó inicial do pioneer pode ser modificado antes do início da execução do código deste projeto. O usuário deve observar atentamente as coordenadas do pioneer na abertura do ambiente de simulação e modificar, se necessário, os valores de correção no código fonte. Veja como se encontram as variáveis em questão:

```
# correcao da odometria (precisa ser modificado com coordenadas do nó 1 no stage)
corx = -55.0
cory = -60.0
```



A figura acima mostra como está configurado o mapa topológico sobre o mapa do ambiente de simulação Stage. O nó o qual foi referido no último paragrafo está indicado pelo número 1.

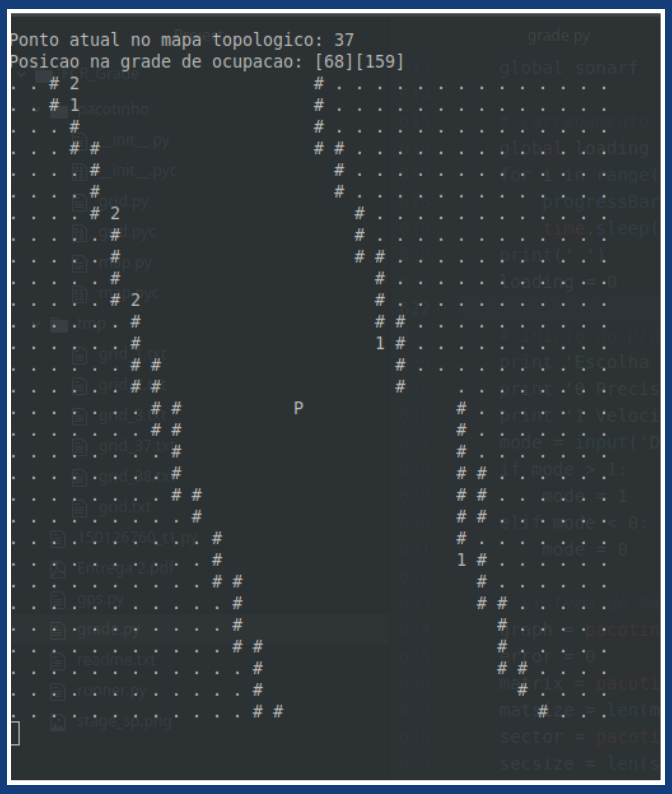
Voltando ao último passo, na interface de usuário, é pedido que seja inserido um novo nó, ao qual o pioneer deverá ir. Caso o usuário insira 0, o programa é encerrado. Caso contrário o usuário pode escolher qualquer nó de acordo com o que foi indicado na imagem.

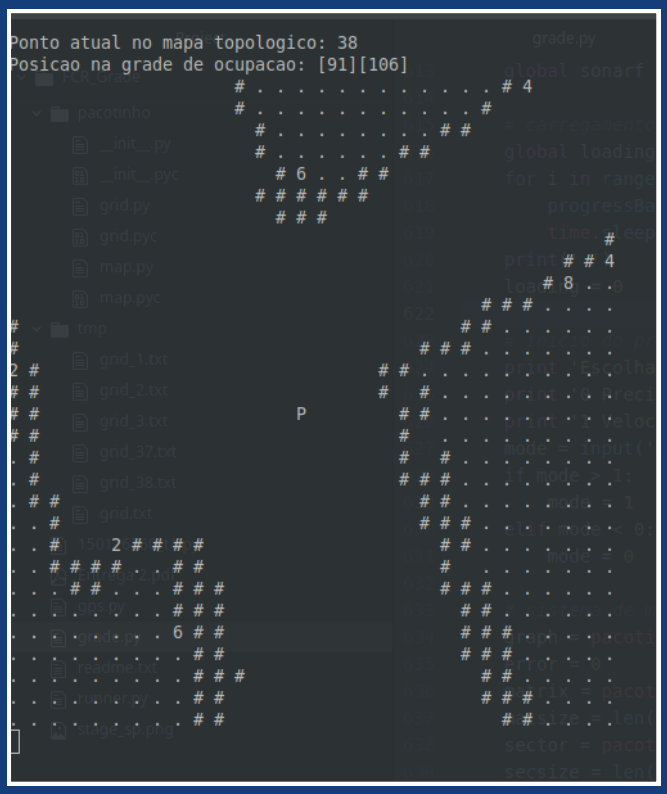
Obs.: a especificação do projeto requeria apenas 10 nós, mas nesta versão do código você tem a liberdade de escolher para que nó quer ir, ou seja, o caminho pode ser maior ou menor que 10 nós de acordo com a sua escolha.

Em seguida, a saída especificará as atividades que o pioneer estiver fazendo. A primeira informação na tela será uma lista com o menor caminho possível entre o nó inicial e o final. Essa lista é criada a partir do algoritmo de Dijkstra. Depois, a saída informará ao usuário por qual ponto da lista o pioneer já passou e se está montando a grade de ocupação de algum nó, também informado. Após o termino do caminho, uma mensagem será mostrada informando que o desenho da grade foi concluído, e seguiremos assim para a segunda parte do programa.

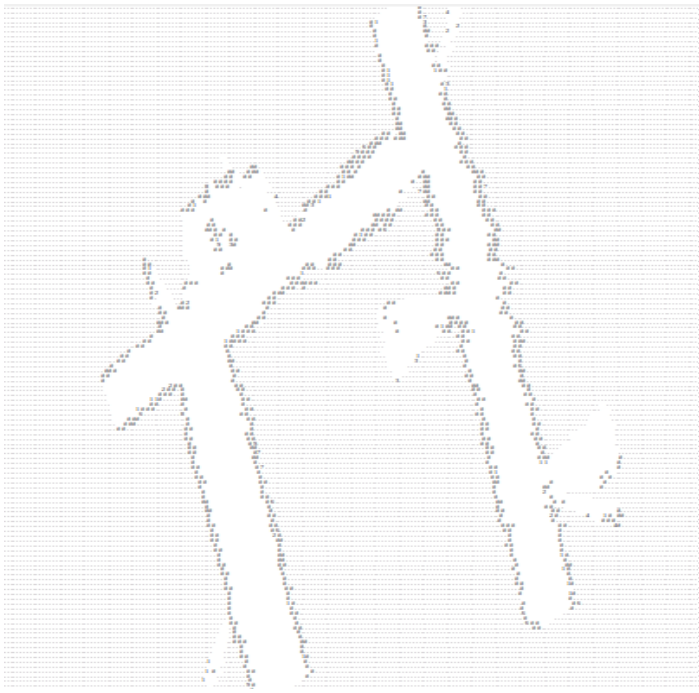
```
Explore um ponto interno
Coordenada X: <input>
Coordenada Y: <input>
```

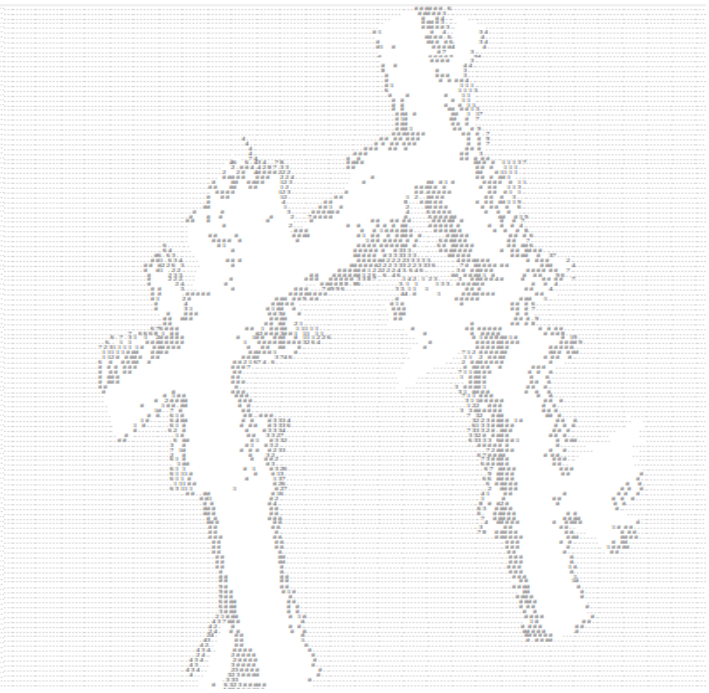
Nesta tela, o usuário deve informar coordenadas de pontos pertencentes à grade feita recentemente. O pioneer irá então usar as informações recém-obtidas para chegar até esse ponto, e será mostrado na tela toda a trajetório do pioneer na grade de ocupação, assim como sua posição no grafo e na matriz que a compõe.





Acima, alguns exemplos da segunda parte da execução. Abaixo, um comparativo entre as grades feitas pelo modo Precisão e pelo modo Velocidade - apenas para informação.





Estruturas

A primeira estrutura representa o mapa topológico do ambiente de simulação Stage. Ela é descrita como um grafo, cujos nós são uma subestrutura identificada pela classe *Map*. Essa subestrutura guarda alguns valores que serão descritos a seguir.

```
class Map:
    center = None
    near = None
    area = None
    value = None
    prev = None
    read = None

graph = [0]*len(nodes)
graph[0] = Map()
graph[0].center = ()
graph[0].near = []
graph[0].area = []
graph[0].value = 999999999.0
graph[0].prev = 0
graph[0].read = 0

def define_graph():
    return graph
```

center guarda as coordenadas principais do nó, que estão relativamente próximas ao centro deste nó. Serão essas coordenadas os objetivos intermediários durante a navegação de exploração do pioneer, enquanto ele escaneia o ambiente em monta a grade de ocupação.

near é uma lista contendo o índice de todos os nós adjacentes ao nó que a possui. É necessária para possibilitar a movimentação do pioneer durante a exploração e para definir o menor caminho do nó atual até um nó final definido pelo usuário.

area guarda as coordenadas da extremidade do nó. Serve para fins de localização, ou seja, para saber onde o pioneer está e para onde ele poderá ir após a montagem da grade de ocupação.

As variáveis *value*, *prev* e *read* serão usadas apenas pela função dijkstra() para determinar o caminho mais curto, e consequentemente melhor caminho, de um nó a outro definidos previamente. *value* guardará a distância do nó inicial até o nó atual pelo caminho mais curto, portanto é plausível que o seu valor inicial seja próximo de infinito ou pelo menos um valor consideravelmente maior que qualquer distância existente no mapa. *prev* guardará o índice do nó anterior. Assim sabendo de onde você veio, é possível definir os passos necessários para planejar o menor caminho. *read* é uma variável booleana, que apenas informa se o nó já foi totalmente explorado durante a ordenação dijkstra.

Temos ainda um segundo tipo de estruturas, destinado à criação das grades de ocupação. Serão matrizes quadradas simples, com três tamanhos diferentes: 240 para a matriz da grade geral, 60 para a grade de um nó no mapa topológico e 30 para exibição na interface de usuário durante a execução da segunda parte do código. Essas matrizes se chamam respectivamente de *matrix*, *gridno* e *sector*.

Algoritmo

A versão mais básica do algoritmo principal do trabalho, em pseudocódigo pode ser descrita da seguinte maneira:

```
Funcao Principal:
   Enquanto robô_está_ligado:
        ponto_inicial = ponto_ao_ligar_robo()
        ponto_final = <input>
        caminho = dijkstra(ponto_inicial, ponto_final)
        Para i de 0 até tamanho(caminho):
             ajustar_angulo()
             navegar_ate(caminho[i])
             grade = desenhar_grade(caminho[i])
        coordenada_x = <input>
        coordenada_y = <input>
        procurar_coordenadas(grade)
        ir_a_pontos(coordenada_x, coordenada_y)
   Fim enquanto
Fim Principal

Funcao ir_a_pontos(coordenada_x, coordenada_y):
   Enquanto x_atual != coordenada_x e y_atual != coordenada_y:
        mostrar_grade()
        ajustar_angulo()
        acelerar()
        Se detectar_obstaculo():
             desviar()
        Fim se
   Fim enquanto
Fim ir_a_pontos
```

Funções

A função progressBar() faz apenas um breve carregamento, que garante que todos os sensores e os dados iniciais da odometria foram definidos. Ela funciona basicamente com um contador e gera uma barra de carregamento simples na saída do terminal. [(1)]

```
def progressBar(value, endvalue, bar_length):
        percent = float(value) / endvalue
        arrow = '-' * int(round(percent * bar_length)-1) + '>'
        spaces = ' ' * (bar_length - len(arrow))

        sys.stdout.write("\rPercent: [{0}] {1}%".format(arrow + spaces, int(round
(percent * 100))))
        sys.stdout.flush()
```

As funções odom_call(data), laser_call(data) e sonarf_call(data) são chamadas apenas para realização de leitura da odometria, sensor laser e sonar frontal, respectivamente. [2]

```
def odom_call(data):
    global x, y, z, corx, cory, loading, tmex

    x = data.pose.pose.position.x - corx
    y = data.pose.pose.position.y - cory
    quaternion = (
        data.pose.pose.orientation.x,
        data.pose.pose.orientation.y,
        data.pose.pose.orientation.z,
        data.pose.pose.orientation.w)
    euler = tf.transformations.euler_from_quaternion(quaternion)
    roll = euler[0]
    pitch = euler[1]
    yaw = euler[2]
    z = yaw

def laser_call(data):
    global laser
    laser = data.ranges

def sonarf_call(data):
    global sonarf
      sonarf = data.ranges
```

A função inside_polygon() é uma função auxiliar. O objetivo dela é retornar sim ou não para coordenadas x e y que estão ou não inseridas em uma área cujas extremidades formam uma lista de outras coordenadas, chamada *points*. [3]

```
def inside_polygon(x, y, points):
    n = len(points)
    inside = False
    p1x, p1y = points[0]
    for i in range(1, n + 1):
        p2x, p2y = points[i % n]
        if y > min(p1y, p2y):
            if y <= max(p1y, p2y):
                if x <= max(p1x, p2x):
                    if p1y != p2y:
                        xinters = (y - p1y) * (p2x - p1x) / (p2y - p1y) + p1x
                    if p1x == p2x or x <= xinters:
                        inside = not inside
        p1x, p1y = p2x, p2y
    return inside
```

As funções placenode() e placegrid() definem que tipo de coordenadas estão sendo avaliadas pela função inside_polygon().

A primeira define *points* como a área de um nó no grafo do mapa topológico, e x e y por sua vez serão as coordenadas atuais do pioneer. Ou seja, placenode() procura o nó do mapa topológico em que o pioneer se encontra atualmente.

A segunda define *points* como a área de um nó cuja grade de ocupação já foi desenhada, e posx e posy (as variáveis que entrarão em inside_polygon() como x e y), representam coordenadas quais se desejam que estejam no interior dessa área. Em resumo, procura-se saber se um ponto já foi desenhado na grade de ocupação (ponto esse para o qual o pioneer deverá se deslocar).

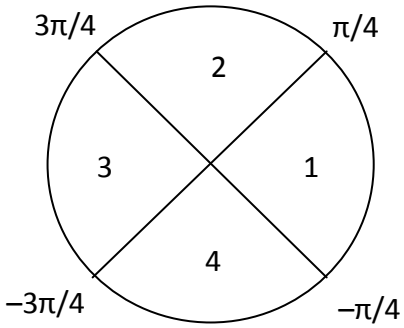
pos_quad() é uma função que define para qual quadrante o pioneer está posicionado em relação ao seu próprio eixo. Para isso, essa função utiliza como base a leitura do ângulo Z atual do Pioneer.

```
def pos_quad():
    global z

    if z == 0:
        quad = 0
    elif (math.pi)/2 >= z and z > 0:
        quad = 1
    elif math.pi >= z and z > (math.pi)/2:
        quad = 2
    elif 0 > z and z > -(math.pi)/2:
        quad = 3
    else:
        quad = 4

    return quad
```

pos_diag() tem basicamente o mesmo propósito de pos_quad() com diferença de –π/4 no início de cada quadrante. Isso será importante para definir qual caminho o Pioneer deve escolher caso sofra desvio por um obstáculo.



calc_ang(posx, posy) recebe as cordenadas finais para o pioneer e calcula o ângulo que ele precisa virar através do arcotangente.

```
def calc_ang(posx, posy):
    global x, y

    tanx = posx - x
    tany = posy - y
    angz = math.atan2(tany, tanx)

    return angz
```

A função adjust_angle, utiliza como base as informações de quadrante e ângulo calculado por cacl_ang e procura o lado para o menor trajeto de rotação em relação ao ângulo desejado. Para isso, ele é auxiliado por duas subfunções semelhantes entre si: adjust_angle_left e adjust_angle_right.

```
def adjust_angle_left(vel, vel_msg, angz):
    global z

    while ((angz + 0.009) < z or z < (angz - 0.009)):
        vel_msg.angular.z = abs(angz - z)
        vel.publish(vel_msg)
    vel_msg.angular.z = 0
    vel.publish(vel_msg)
```

Definido o melhor lado a se virar, a função publica uma velocidade angular, positiva para virar à esquerda e negativa para virar à direita.

adjust_dl e adjust_dr também são semelhantes às subfunções anteriormente apresentadas. A diferença se dá por estas utilizarem os dados de pos_diag ao invés dos quadrantes e serem utilizadas procurar o melhor lado para desvio de obstáculos.

Devido a extensão das demais funções, serão explicadas de forma sucinta as lógicas utilizadas em cada uma. Os respectivos algoritmos estão expressos no código fonte do trabalho, pasta src.

Navegação e Mapeamento

showme() é a função com o objetivo de mostrar a matriz *sector* na saída, ou seja a grade de ocupação pelo qual o pioneer estará se deslocando. Para isso, é fornecida a matriz geral *matrix* e os pontos x e y da odometria passam por um processo de conversão para representarem os índices gx e gy (linha e coluna) da matriz. Assumindo que nessa grade, que será mostrada na tela, o pioneer estará sempre no meio, serão delimitados também o índice inicial e final de acordo com a área que se deseja abranger. No caso da interface de usuário, serão mostrados cerca de 15 metros do raio de abrangência do sensor laser. Em seguida alguns valores da matriz serão convertidos na saída para melhor compreensão do usuário (isso também vale para os arquivos txt criados para análise). Os valores serão:

5 = ‘.’ (valores nunca vistos serão representados por pontos)

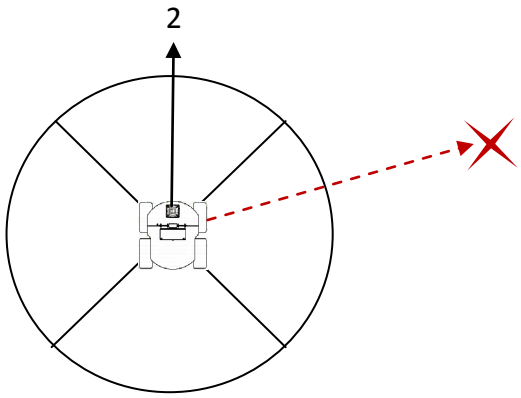
10 = ‘#’ (representa obstáculos sólidos no caminho ou paredes que pode impedir a locomoção ou visão do segmento posterior ao obstáculo)

0 = ‘ ‘ (se uma posição chegar ao valor zero, então está posição está livre para locomoção)

-1 = ‘P’ (valor definido previamente, representa a posição atual do pioneer na grade ou a posição da qual escaneou o ambiente durante a gravação de um arquivo txt contendo a grade de certo nó)

runner(): é a função mais importante no quesito tratamento anti-colisões. A idéia inicial seria de utilizar a grade para andar apenas por ambientes conhecidos. Entretanto devido a complicações durante o desenvolvimento do código, a solução encontrada foi utilizar uma forma de validação na entrada do usuário para inserir apenas um ponto presente na grade de ocupação montada e reutilizar o código do trabalho 1 para tratamento de colisões até chegar neste ponto, com as devidas adaptações que mostram o deslocamento do pioneer por sobre a grade desenhada.

A função em questão, funciona da seguinte maneira: ela publica uma velocidade linear constante até que o pioneer chegue nessas coordenadas. Entretanto, caso o sonar frontal detecte obstáculos à frente (representados na grade por valores iguais ou maiores que 10, além de evitar possíveis obstáculos cujos valores diferem de 0), o pioneer para e decide qual o melhor lado para desviar com base em pos_diag(), e no valor *angz* (que é o ângulo que direciona o pioneer à posição final de modo linear). Por exemplo, se o pioneer estiver virado para *diag* = 2, e a posição final estiver no primeiro ou quarto quadrante em relação à posição atual do pioneer, a melhor decisão será virar à direita.



O segundo passo após um desvio recente, é procurar por uma passagem que leve o Pioneer ao local desejado de forma mais rápida. Suponha que ele acabou de desviar de uma parede e agora está andando lado a lado com ela. Caso os sonares laterais não detectem mais sua presença, é bem provável que haja uma passagem livre ali. Caso isso se confirme, o pioneer recalcula o ângulo em relação às cordenadas finais e repete todo o processo da função runner() até chegar ao seu destino.

```
def walk(vel, vel_msg, posx, posy):
    global x, y

    while (posx+0.3 < x or x < posx-0.3) or (posy+0.3 < y or y < posy-0.3):
        square = (posx - x)**2 + (posy - y)**2
        dist = square**0.5
        vel_msg.linear.x = dist
        vel.publish(vel_msg)
    vel_msg.linear.x = 0
    vel.publish(vel_msg)
```

A função walk() é uma função simples, que apenas publica velocidade para locomoção em linha reta pelo pioneer. Ela é usada durante as viajens entre nós no mapa topológico, sejam para serem explorados ou apenas servirem de passagem para a criação de novas grades em nós adjacentes.

A função dijkstra() segue fielmente às definições do algoritmo de Dijkstra para solucionar o problema do caminho mais curto, neste caso em um grafo não dirijido. Sendo assim não se faz necessária uma explicação formal do funcionamento dessa função. Apenas será deixado explícito que o grafo em questão se refere ao mapa topológico de nome *graph*, com nó inicial sendo posição atual do pioneer e nó final definido pelo usuário. O caminho mais curto é salvo e retornado pela função através da lista *way*. Os valores modificados do grafo são resetados ao final da ordenação.

drawtxt() utiliza a conversão de valores, citada anteriormente, para salvar uma grade de ocupação em um arquivo txt.

drawsec() faz a leitura da matriz da grade geral e delimita sua área para salvar a grade do nó atual em um arquivo txt. Para isso, se utiliza da função de criação geral de grades em txt, drawtxt().

gridmaker() é a função que inicia o processo de escaneamento do ambiente pelo sensor laser para o transformar em uma grade de ocupação. Utiliza-se também de funções auxiliares, portanto prioritariamente tem função de realizar um movimento de rotação de 360° do pioneer para varrer toda a área do nó em que se encontra, estando o pioneer no ponto central desse nó.

maker(), função auxiliar de gridmaker, coleta os dados fornecidos pelo sensor laser e, com auxílio do ângulo do pioneer e do ângulo do laser em relação ao eixo do pioner, transforma as distâncias encontradas em posições proporcionais em uma determinada matriz que conterá uma grade de ocupação. O tamanho padrão definido pelo programador para uma posição na matriz é de 0,5 metros no ambiente de simulação Stage. Às posições obtidas pelos sensores na matriz são somadas o valor de 1 (para cada sensor que resultar nestas posições). Como descrito anteriormente, valores iguais ou maiores que 10 são considerados paredes ou obstáculos.

A partir da posição captada pelo sensor e gravada na matriz como possível obstáculo, toda a distância anterior do sensor laser também será convertida em posições, por sua vez mais próximas a posição do pioneer. À essas posições serão subtraídas o valor de 1, até que cheguem a 0 e representem áreas livres para locomoção do pioneer. Para obter tal resultado, a função maker() chama a função recursiva eraser(), que realiza esta operação.

treat_node é mais uma função simples, que apenas concatena serviços interligados na ordem em que devem ser executados: converte-se as coordenadas do pioneer para uma posição na matriz, chama a função gridmaker() - desenha a grade de ocupação -, chama a função drawsec() - salva o desenho da grade em um arquivo txt - e insere o nó desenhado na lista *drawn* de nós com grades já existentes.

Chegamos então à função run(), principal função do programa e resumida em pseudocódigo na seção Algoritmo. Essa função envolve a interação de entrada e saída para com o usuário e chama todas as demais funções que realizarão o procedimento proposto pelo roteiro do trabalho. Além disso, os principais comandos estão dentro de um loop, permitindo ao usuário continuar explorando o mapa, mesmo após a conclusão de uma grade de ocupação sobre o caminho definido pelos nós anteriormente.

Referências

[1] **progressBar()**

Replace console output in Python - https://stackoverflow.com/a/37630397

[2] **odom_call()**

Função fornecida no fórum de discussões da disciplina FCR

[3] **inside_polygon()**

Testing if a Point is Inside a Polygon in Python - http://okomestudio.net/biboroku/?p=986

Algoritmo de Dijkstra (referência usada apenas para leitura)

https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra

FIM