Trabalho 2 – Campos Potenciais

Silas Franco dos Reis Alves

1 Introdução

Dado o mapa do ambiente esboçado pela Figura 1, o objetivo é fazer com que o robô, inicialmente em $(-19, -8)$, navegue para três posições conhecidas do mapa representando três objetos: lixeira $(-19{,}2)$, objeto 1 $(2{,}4)$ e objeto 2 $(9, -7)$.

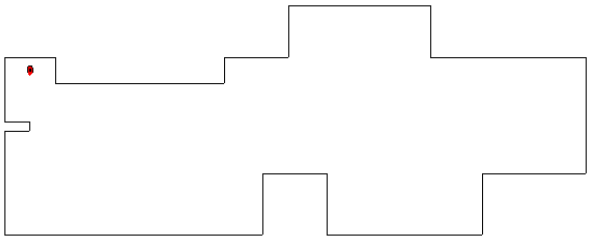

Figura 1 – Mapa do ambiente.

Para esse fim, foram desenvolvidas duas abordagens. Na primeira, o robô primeiro levanta o mapa do ambiente e depois, com o ambiente já conhecido, aplica a técnica de campos potenciais para encontrar os três pontos meta. Na segunda, o robô aplica a técnica de campos potenciais no mapa desconhecido para cumprir sua missão ao passo que mapeia o ambiente.

O robô móvel utilizado para este trabalho é o robô Pioneer, que conta com 16 sonares com 5,0 m de alcance e um sensor de distância SICK que faz uma varredura de 180 pontos numa faixa de 180° (resolução de 1°) com alcance de 8,0 m.

2 Exploração

Para permitir o robô explorar o ambiente em questão, foi desenvolvido um algoritmo "seguidor-de-parede", que, conforme o nome implica, faz o robô navegar livremente pelo ambiente enquanto se mantem alinhado à parede mais próxima. Obviamente, uma restrição para este método de exploração é que todo o campo possa ser visto pelos sensores do robô quando este está próximo às paredes. Por exemplo, um ambiente quadrado demasiado grande poderá ficar com sua região central inexplorada caso os sensores de distância não sejam capazes de alcançar seu centro. Este não é um problema para este ambiente, cujas dimensões, em metros, são 40 m de largura por 20 m de comprimento, e cujo sensor SICK do robô consegue medir distâncias de até 8 m.

O algoritmo para seguir paredes é consiste de apenas 3 condições. Na primeira condição, ilustrada pela Figura 2-(a), o robô está alinhado a uma parede e, portanto, segue com uma velocidade linear v_x positiva e utiliza a velocidade angular v_ω para manter-se alinhado a parede. Nessa condição, v_x é proporcional ao mínimo entre as distâncias lidas pelos sonares 3 e 4, multiplicadas por uma constante k_{v_x}, e v_ω é proporcional à diferença entre as leituras dos sonares 7 e 8 multiplicada a uma constate k_{v_ω}. A segunda condição ocorre quando o robô

encontra uma parede à sua frente, que é quando o valor mínimo entre os sonares 3 e 4 é inferior à 3 m. Nesta condição, o robô para, tomando $v_x = 0$, e gira no sentido anti-horário com v_ω constante até que os sensores 8 e 7 se alinhem novamente a parede, permitindo-o voltar à primeira condição. Finalmente, a terceira condição ocorre quando os sonares 3, 4 e 6 não detectam uma parede à frente, o que significa que o robô deve fazer uma curva no sentido horário enquanto segue em frente. Nesta condição, v_x é proporcional ao mínimo entre as distâncias lidas pelos sonares 3 e 4, multiplicadas por uma constante k_{v_x}, e v_ω constante em sentido horário até que o sonar 6 encontre uma parede, retomando o robô para a primeira condição.

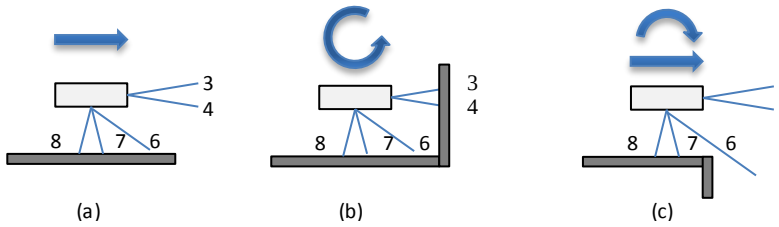

Figura 2 – Três condições do robô: (a) seguindo a parede alinhado a ela; (b) alinhando-se a uma nova parede à frente; (c) alinhando-se a uma nova parede ao lado.

2.1 Implementação em Python

A primeira parte do software trata de inicializar algumas constantes como o tamanho do mapa nos eixos das abscissas ($mapa_{dim,x}$) e das ordenadas($mapa_{dim,y}$), e também o tamanho do robô ($robo_{dim}$), conforme o Algoritmo 1.

```
from pylab import *
from playerc import *
from time import time

# Constantes.
ROBO_DIM = 0.5
MAPA_DIM_X = 40.0
MAPA_DIM_Y = 20.0
```

Algoritmo 1 – Configuração do ambiente.

Como o mapa será representado por um mapa de grade de ocupação com dimensão m linhas por n colunas, onde $m = mapa_{dim,y}/robo_{dim}$ e $n = mapa_{dim,x}/robo_{dim}$, é necessário criar uma função que converta as coordenadas do mapa em coordenadas da grade de ocupação, conforme mostra o Algoritmo 2. Para simplificar a criação de mapas, foram implementadas duas rotinas que atribuem o valor 1 a determinada grade do mapa (ou seja, torna-a ocupada) e o valor 0 (ou seja, torna-a vazia). Estas duas rotinas também são descritas pelo Algoritmo 2.

```
# Converte um ponto nas coordenadas do mapa para as coordenadas da grade
# de ocupação
def ponto_mapa(mapa, (x, y)):
    global ROBO_DIM
    y = mapa.shape[0] - (y / ROBO_DIM + mapa.shape[0] / 2.0)
    if y <= -1:
        y = 0
    elif y >= mapa.shape[0]:
        y = mapa.shape[0] - 1

    x = x / ROBO_DIM + mapa.shape[1] / 2.0
    if x <= -1:
        x = 0;
    elif x >= mapa.shape[1]:
        x = mapa.shape[1] - 1
```

```
    return (y, x)

# Seta determinado ponto na grade de ocupação como obstáculo.
def ponto_obstaculo(mapa, (x, y)):
    mapa[ponto_mapa(mapa, (x, y))] = 1

# Seta determinado ponto na grade de ocupação como vazio.
def ponto_vazio(mapa, (x, y)):
    mapa[ponto_mapa(mapa, (x, y))] = 0
```

Algoritmo 2 – Função que converte as coordenadas do mapa em coordenadas da grade.

Também é necessária uma função que capture os dados do sensor SICK, arranje-os num vetor de 180 elementos, e mapeie a região observada pelos sensores. Para cada leitura i do sensor SICK, cujo ângulo em relação ao lado direito do robô é dado por $\theta + \alpha$, é calculada a posição l do obstáculo (ou ponto vazio, caso não haja nenhum obstáculo e o sensor alcance seu limite máximo de 8,0 m) l detectado a r metros de distância do sensor em o, conforme mostra a Figura 3. Se o ponto l for decorrente de um obstáculo, a grade correspondente recebe o valor 1, caso contrário (vazia), recebe o valor 0. Além disso, todas as grades entre o e l são consideradas vazias. Esse processamento é realizado pelo Algoritmo 3

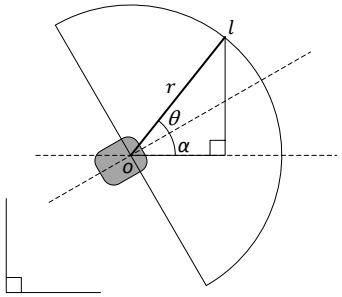

Figura 3 – Diagrama de uma leitura do sensor SICK.

```
# Mapeia a região vista pelo sensor SICK.
def mapeia_regiao (c, p, l, mapa):
    # Atualiza a leitura do sensor.
    c.read()

    # Cria um array que receberá as leituras de distância
    dist = np.zeros(180)

    # Mapeia a região vista por cada uma das leituras de distância.
    for i in xrange(180):
        # Recupera o ponto (lx, ly) visto pelo sensor.
        beta = p.pa + radians(i - 90)
        dist[i] = l.ranges[i]
        ly = p.py + sin(beta) * l.ranges[i] #y
        lx = p.px + cos(beta) * l.ranges[i] #x

        # Detecta se o ponto é um obstáculo ou não e o marca no mapa.
        if l.ranges[i] == 8.0:
            mapa[ponto_mapa(mapa, (lx, ly))] = 0
        else:
            mapa[ponto_mapa(mapa, (lx, ly))] = 1

        # Converte a posição atual do robô e o ponto (lx, ly) em coordenadas
        # da grade de ocupação (valores inteiros).
        (oy, ox) = ponto_mapa(mapa, (p.px, p.py))
        (ly, lx) = ponto_mapa(mapa, (lx, ly))
        oy, ox, ly, lx = int(oy), int(ox), int(ly), int(lx)

        # Todas as grades entre a grade onde está o robô e a grande vista
        # pelo sensor (lx, ly) estão vazias, conforme o algoritmo descrito
```

```
        # em http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
        steep = abs(ly - oy) > abs(lx - ox)
        if steep:
            (ox, oy) = (oy, ox)
            (lx, ly) = (ly, lx)
        if ox > lx:
            (ox, lx) = (lx, ox)
            (oy, ly) = (ly, oy)
        dx = lx - ox
        dy = abs(ly - oy)
        erro = dx / 2
        passo_y = -1
        y = oy
        if oy < ly:
            passo_y = 1
        for x in xrange(ox, lx):
            if steep:
                if mapa[x, y] == 0.5:
                    mapa[x, y] = 0
            else:
                if mapa[y, x] == 0.5:
                    mapa[y, x] = 0
            erro -= dy
            if erro < 0:
                y += passo_y
                erro += dx

    # Retorna as distâncias lidas.
    return dist
```

Algoritmo 3 – Função que adquire os dados do sensor SICK e mapeia a região por ele vista.

Para fins de comodidade, foi escrita uma função que organiza convenientemente as leituras dos sonares em um vetor, conforme o Algoritmo 4.

```
# Retorna a leitura dos 16 sonares em torno do robô.
def sonares(c, s):
    # Atualiza os dados do robô.
    c.read()

    # Cria um array e nele grava todas as leituras de distância.
    dist = np.zeros(s.ranges_count)
    for i in xrange(s.ranges_count):
        dist[i] = s.ranges[i]

    # Retorna o array com os valores lidos.
    return dist
```

Algoritmo 4 – Função que organiza os dados dos sonares.

Com as funções dos Algoritmos 1 à 4, é possível escrever a rotina que controla o movimento do robô e que lhe confere o comportamento de seguir parede. As regras dessa rotina são descritas no início da Seção 2, e seu algoritmo é mostrado pelo Algoritmo 5. Nesta implementação, o algoritmo opera por um tempo determinado pelo usuário.

```
def segue_parede_e_mapeia(c, p, l, s, mapa, tempo_max=600):
    # Inicia as constantes e variáveis.
    K_x = 0.5
    K_a1 = 10
    K_a2 = radians(10)
    dist = mapeia_regiao(c, p, l, mapa)
    inicio = time() # Recupera o tempo em que o algoritmo é iniciado.

    # Segue a parede e mapeia o ambiente enquanto tempo_max não foi alcançado.
    while time() - inicio < tempo_max:
        # Atualiza a leitura do SICK, mapeando o ambiente, e dos sonares.
        mapeia_regiao(c, p, l, mapa)
        sonar = sonares(c, s)

        #Atualiza o gráfico.
```

```
        imshow(mapa, interpolation='nearest')
        pause(0.001)

        # Segue a parede, mantendo a velocidade linear constante e usando
        # a velocidade angular para manter o robô alinhado à parede.
        vx = K_x
        va = (sonar[7] - sonar[8]) * -K_a1

        # Se o robô estiver muito próximo de uma parede à sua frente, ele
        # gira até se alinhar a essa nova parede.
        if min(sonar[3], sonar[4]) < 4:
            va = K_a2
            p.set_cmd_vel(0, 0, va, 1)
            while abs(sonar[7]-sonar[8]) > 0.1 and min(sonar[3], sonar[4]) < 3:
                mapeia_regiao(c, p, l, mapa)
                sonar = sonares(c, s)
        # Senão, se a parede que ele estava seguindo acabar, ele começa a
        # fazer uma curva para alcançar essa nova parede.
        elif sonar[6] == 5:
            va = -K_a2

        # Garante que a velocidade angular não será maior que o esperado.
        if va < 0 and va < -K_a2:
            va = -K_a2
        elif va > 0 and va > K_a2:
            va = K_a2

        # Envia os comandos ao robô
        p.set_cmd_vel(vx, 0, va, 1)

    # Para o movimento do robô.
    p.set_cmd_vel(0, 0, 0, 1)
```

Algoritmo 5 – Controle do seguidor de parede.

Finalmente, é escrita a rotina principal, que se conecta aos componentes do robô, cria o mapa e executa o controlador. Essa rotina é mostrada pelo Algoritmo 6.

```
# Rotina principal que inicializa o programa e roda o algoritmo de mapeamento.
def main():
    # Conecta ao robô.
    c = playerc_client(None, 'localhost', 6665)
    if c.connect() != 0:
        print playerc_error_str()

    # Conecta à interface que permite mover e recuperar a localização.
    p = playerc_position2d(c,0)
    if p.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    s = playerc_ranger(c, 0)
    if s.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Conecta ao laser.
    l = playerc_ranger(c, 1)
    if l.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Atualiza os dados dos componentes do robô (position2D, sonar e SICK).
    p.get_geom()
    s.get_geom()
    l.get_geom()
    c.read()

    # Cria um mapa de grade de ocupação e inicializa todas as grades com 0.5
    mapa = ones((MAPA_DIM_Y / ROBO_DIM , MAPA_DIM_X / ROBO_DIM )) / 2.0
    grid(True)
    imshow(mapa, interpolation='nearest')
    savefig('mapa_exp00.png')
    pause(0.001)

    # Segue a parede por 10 minutos (600 segundos) para mapear o ambiente.
    segue_parede_e_mapeia(c, p, l, s, mapa, 300)
```

```
    # Salva o mapa levantado pelo robô.
    savefig('mapa_exp01.png')
    pause(0.001)
    save('mapa', mapa)
    savetxt('mapa.txt', mapa)

    #Encerra a comunicação com o robô.
    p.unsubscribe()
    l.unsubscribe()
    s.unsubscribe()
    c.disconnect()


# Executa a função main().
if __name__ == '__main__':
    main()
```

Algoritmo 6 – Rotina principal do seguidor de linha.

2.2 Resultados

Após 3 minutos executando, o robô foi capaz de gerar o mapa mostrado pela Figura 4. Conforme mostra a figura, as regiões verdes não são exploradas, as azuis são vazias e as vermelhas são os obstáculos ou paredes. O robô foi capaz de mapear o ambiente satisfatoriamente e, após 10 minutos, toda a extensão da parede foi coberta pelo robô.

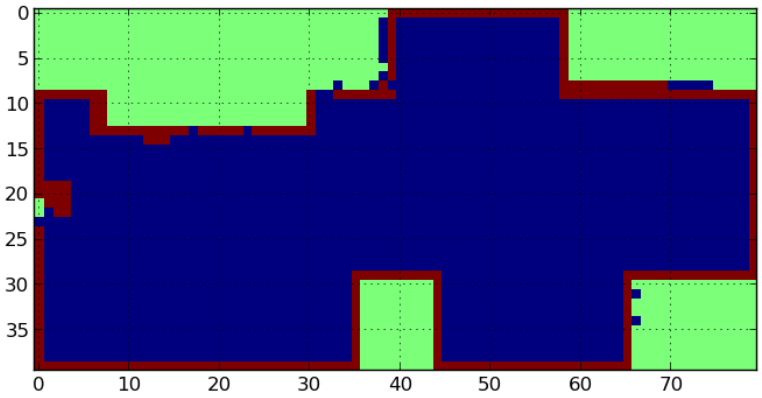

Figura 4 – Mapa levantado pelo robô móvel.

3 Algoritmo Clássico de Campos Potenciais

Com o mapa levantado pelo algoritmo do seguidor de paredes, foi possível utilizar o método de campos potenciais para levar o robô a três posições conhecidas do campo: lixeira, objeto 1 e objeto 2. O algoritmo de campos potenciais segue o método apresentado em (Silva, 2011),e é descrito pelo Algoritmo 7.

```
# Calcula o campo potencial em um ponto do mapa.
def campo_potencial_ponto(campo, meta, ponto, C=1, Q=1):
    # Inicia os valores.
    meta = ponto_mapa(campo, meta)
    (y, x) = ponto_mapa(campo, ponto)
    cp_f = None
    cp_t = None
    obs_y, obs_x = np.where(campo > 0.9)

    # Se o ponto escolhido não for obstáculo ou meta, calcula seu potencial
    if campo[y, x] < 0.9 and (y, x) != meta:
        # Calcula a Fa da meta.
        dx = meta[1] - x
        dy = meta[0] - y
```

```
        d = ((dx)**2+(dy)**2)**0.5
        k = C / d
        fx = dx * k
        fy = dy * k

        # Calcula a Fr de todos os obstáculos
        for i in xrange(obs_x.shape[0]):
            dx = x - obs_x[i]
            dy = y - obs_y[i]
            d = ((dx)**2+(dy)**2)**0.5
            fr = Q / d**2
            k = fr / d
            fx += dx * k
            fy += dy * k

        #Calcula F e theta
        f = ((fx)**2+(fy)**2)**0.5
        theta = arctan2(-fy, fx)
        cp_f = f
        cp_t = theta

    # Retorna F e theta calculados.
    return cp_f, cp_t
```

Algoritmo 7 – Código de campos potenciais.

Para facilitar o processo de depuração da rotina, e também para permitir a geração de gráficos dos potenciais, foi criada uma rotina que calcula os potenciais de todos os pontos vazios do campo. Conforme mostra o Algoritmo 8, esta rotina é uma especialização do Algoritmo 7.

```
# Calcula o campo potencial em todos os pontos do mapa.
def campo_potencial_completo(campo, meta, figura=None, C=1, Q=1):
    # Se foi fornecido o nome de uma figura, prepara a criação do gráfico.
    if figura != None:
        X, Y = meshgrid(linspace(0, campo.shape[1]-1, campo.shape[1]),
                        linspace(0, campo.shape[0]-1, campo.shape[0]))
        U = np.zeros(campo.size)
        V = np.zeros(campo.size)

    # Inicia os valores.
    meta = ponto_mapa(mapa, meta)    # Converte a meta para coordenadas da grade
    cp_f = copy(campo)               # Matriz com os valores de F
    cp_t = zeros(campo.shape)        # Matriz com os valores de theta
    obs_y, obs_x = np.where(campo > 0.9)     # Lista de todos os obstáculos

    # Calcula F e theta para todos os elementos da grade.
    for y in xrange(campo.shape[0]):
        for x in xrange(campo.shape[1]):
            # Se a célula for vazia, calcula F e theta correspondentes
            if campo[y, x] < 0.5 and (y, x) != meta:
                # Calcula a Fa da meta
                dx = meta[1] - x
                dy = meta[0] - y
                d = ((dx)**2+(dy)**2)**0.5
                k = C / d   # Constante usada para garantir |F| = C
                fx = dx * k
                fy = dy * k

                # Calcula a Fr de todos os obstáculos.
                for i in xrange(obs_x.shape[0]):
                    dx = x - obs_x[i]
                    dy = y - obs_y[i]
                    d = ((dx)**2+(dy)**2)**0.5
                    fr = Q / d**2
                    k = fr / d
                    fx += dx * k
                    fy += dy * k

                # Atualiza os dados do gráfico
                if figura != None:
                    U[x+(campo.shape[0] - y)*campo.shape[1]] = fx
                    V[x+(campo.shape[0] - y)*campo.shape[1]] = -fy
```

```
                # Calcula F e theta
                f = ((fx)**2+(fy)**2)**0.5
                theta = arctan2(-fy, fx)
                cp_f[y, x] = f
                cp_t[y, x] = theta

    # Mostra a figura com o gráfico de vetores.
    if figura != None:
        quiver(X, Y, U, V)
        savefig(figura)
        show()

    # Retorna F e theta calculados
    return cp_f, cp_t
```

Algoritmo 8 – Cálculo do campo potencial para todo o mapa.

Com a função do Algoritmo 7 foi possível criar uma rotina de controle de navegação para o robô móvel. A rotina de controle emite sinais de acionamento para os motores do robô enquanto ele se encontrar a uma distância d_{parada} do ponto de $meta$. A cada iteração, o controlador calcula o potencial $\vec{F}$, descritos por seu módulo $|F|$ e ângulo ω, da localidade onde se encontra o robô e a partir deles determina a velocidade linear v_x e angular v_ω do robô. Esse controlador é descrito pelo Algoritmo 9.

```
# Calcula a distância euclidiana entre dois pontos.
def distancia((x0, y0), (x1, y1)):
    return ((x1 - x0) ** 2 + (y1 - y0) ** 2) ** 0.5

# Ajusta os ângulos para que fiquem no intervalo [-pi, pi]
def ajusta_angulo(alfa):
    if alfa > pi:
        alfa -= 2 * pi
    elif alfa < -pi:
        alfa += 2 * pi
    return alfa

# Controla o robô usando os campos potenciais.
def controle(c, p, mapa, meta, d_parada=0.01, k_d = 1.0):
    # Ganhos do controlador
    k_v = 0.5
    k_om = 1.2

    # Calcula a distância inicial â meta.
    c.read()
    d = distancia((p.px, p.py), meta)

    # Enquanto a distância do robô ao objetivo for maior que o crit. de parada
    while d > d_parada:
        # Atualiza os dados do robô.
        c.read()
        robo, th = (p.px, p.py), p.pa

        # Calcula a distância e determina a velocidade em X
        d = distancia(robo, meta)

        # Recupera a força e o ângulo do potencial.
        f, om = campo_potencial_ponto(mapa, meta, robo)

        # Calcula o erro do robô ao ângulo do potencial e a velocidade angular.
        th_err = ajusta_angulo(ajusta_angulo(om) - ajusta_angulo(th))
        v_th = th_err * k_om

        # Determina a velocidade linear quando o erro angular é menor que
        # 10 graus, reduzindo a velocidade conforme se aproxima da meta.
        if abs(th_err) < degrees(10):
            if d > k_d:
                v_x = f * k_v
            else:
                v_x = d / k_d * k_v
        else:
            v_x = 0
```

```
        # Envia as novas velocidades ao robô.
        p.set_cmd_vel(v_x, 0.0, v_th, 1)

    # Para o robô.
    p.set_cmd_vel(0.0, 0.0, 0.0, 1)
```

Algoritmo 9 – Controle de navegação do robô por campos potenciais.

Finalmente, foi criada a rotina principal que carrega o mapa criado pelo seguidor de parede, conecta aos componentes do robô e aciona o controlador de navegação por campos potenciais, conforme o Algoritmo 10.

```
def main():
    #Carrega o mapa conhecido
    mapa = loadtxt('mapa.txt')

    #Posição dos objetos conhecidos
    lixeira = (-19, 2)
    objeto1 = (2, 4)
    objeto2 = (9, -7)

    # Conecta ao robô.
    c = playerc_client(None, 'localhost', 6665)
    if c.connect() != 0:
        print playerc_error_str()

    # Conecta à interface que permite mover e recuperar a localização.
    p = playerc_position2d(c,0)
    if p.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Conecta ao laser.
    r = playerc_ranger(c, 1)
    if r.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Obstáculos = lixeira, objeto2; Meta = objeto1
    print 'Buscando pelo objeto 1'
    ponto_obstaculo(mapa, lixeira)
    ponto_obstaculo(mapa, objeto2)
    #campo_potencial_completo(mapa, objeto1, '00.png')
    controle(c, p, mapa, objeto1)

    # Obstáculos = objeto2; Meta = lixeira
    print 'Buscando pela lixeira'
    ponto_vazio(mapa, lixeira)
    #campo_potencial_completo(mapa, lixeira, '01.png')
    controle(c, p, mapa, lixeira)

    # Obstáculos = lixeira; Meta = objeto2
    print 'Buscando pelo objeto 2'
    ponto_obstaculo(mapa, lixeira)
    ponto_vazio(mapa, objeto2)
    #campo_potencial_completo(mapa, objeto2, '02.png')
    controle(c, p, mapa, objeto2)

    # Meta = lixeira
    print 'Buscando pela lixeira'
    ponto_vazio(mapa, lixeira)
    #campo_potencial_completo(mapa, lixeira, '03.png')
    controle(c, p, mapa, lixeira)

    # Encerra a comunicação com o robô.
    p.unsubscribe()
    r.unsubscribe()
    c.disconnect()
```

```
# Executa a função main().
if __name__ == '__main__':
    main()
```

Algoritmo 10 – Rotina principal para o controle de navegação por campos potenciais.

3.1 Resultados

O algoritmo de campos potenciais permitiu que o robô navegasse satisfatoriamente por todo o ambiente, sem se chocar a obstáculos e apresentando uma rota suave. A Figura 5 mostra os campos potenciais gerados nas 4 fases da missão, que consistiam em buscar o objeto 1, buscar a lixeira pela primeira vez, buscar o objeto 2, e buscar a lixeira pela segunda vez.

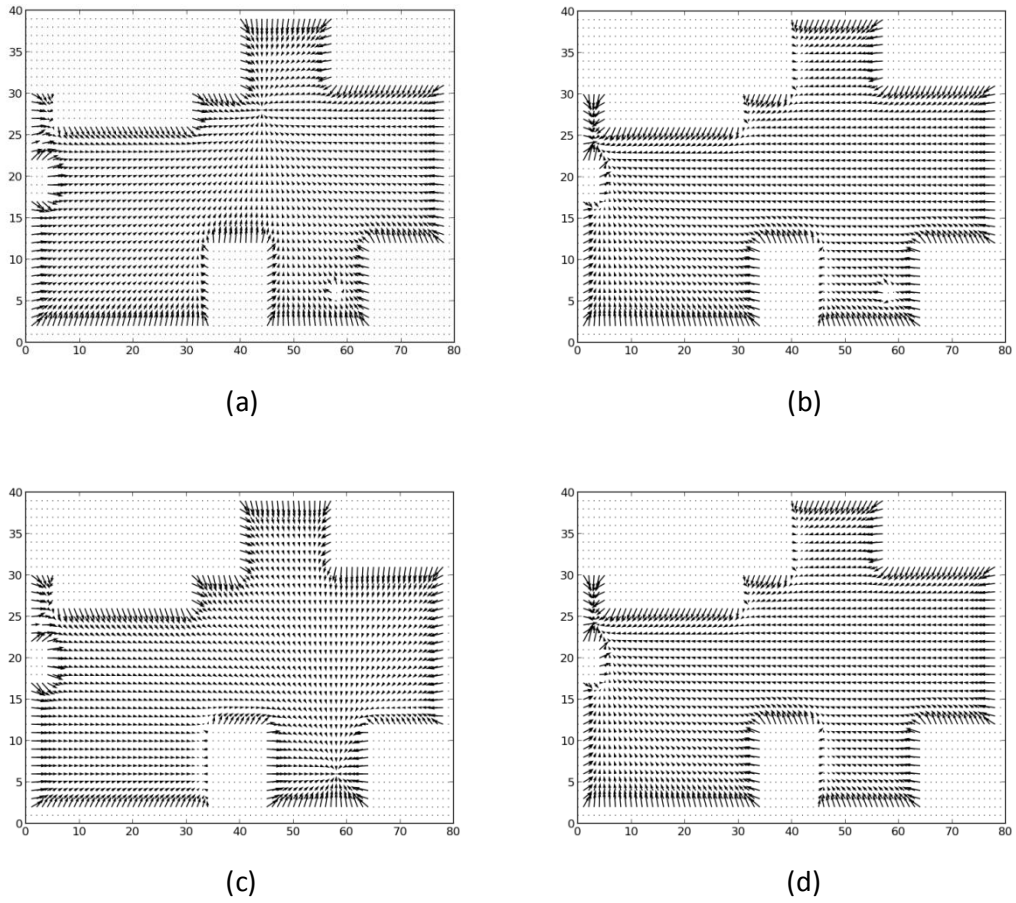

Figura 5 – Campos potenciais nas 4 fases de execução da missão: (a) busca do objeto 1, (b) primeira busca a lixeira, (c) busca ao objeto 2, e (d) segunda busca a lixeira.

4 Exploração com Campos Potenciais

Modificando o Algoritmo 9, adicionando o mapeamento do Algoritmo 3, foi possível escrever uma versão do controlador de navegação por campos potenciais que utiliza um mapa de um ambiente desconhecido e o mapeia em tempo de execução. Isso é possível pois ao iniciar o programa, o mapa está completamente vazio e, portanto, o potencial de todas as grades apontam para o ponto de meta. Conforme o robô navega, os obstáculos são adicionados ao mapa e seus potenciais de repulsão passam a guiar o robô seguramente em direção à meta, mas de forma a evitar os obstáculos descobertos. Esse algoritmo é mostrado pelo Algoritmo 11.

```
# Controla o robô usando os campos potenciais.
def controle(c, p, l, mapa, meta, d_parada=0.01, k_d = 1.0):
    # Ganhos do controlador
    k_v = 0.5
    k_om = 1.2

    # Calcula a distância inicial â meta.
    c.read()
    d = distancia((p.px, p.py), meta)

    # Mostra a grade de ocupação.
    grid(True)
    imshow(mapa, interpolation='nearest')
    pause(0.001)

    # Enquanto a distância do robô ao objetivo for maior que o crit. de parada
    while d > d_parada:
        # Atualiza os dados do robô.
        c.read()
        robo, th = (p.px, p.py), p.pa

        # Mapeia a região em busca de obstáculos.
        mapeia_regiao(c, p, l, mapa)

        # Calcula a distância e determina a velocidade em X
        d = distancia(robo, meta)

        # Recupera a força e o ângulo do potencial.
        f, om = campo_potencial_ponto(mapa, meta, robo)

        # Calcula o erro do robô ao ângulo do potencial e a velocidade angular.
        th_err = ajusta_angulo(ajusta_angulo(om) - ajusta_angulo(th))
        v_th = th_err * k_om

        # Determina a velocidade linear quando o erro angular é menor que
        # 10 graus, reduzindo a velocidade conforme se aproxima da meta.
        if abs(th_err) < degrees(10):
            if d > k_d:
                v_x = f * k_v
            else:
                v_x = d / k_d * k_v
        else:
            v_x = 0

        # Envia as novas velocidades ao robô.
        p.set_cmd_vel(v_x, 0.0, v_th, 1)

        # Atualiza a vizualização da grade.
        imshow(mapa, interpolation='nearest')
        pause(0.001)

    # Para o robô.
    p.set_cmd_vel(0.0, 0.0, 0.0, 1)
```

Algoritmo 11 – Controle de navegação por campos potenciais e com capacidade de mapeamento.

Finalmente, algumas modificações foram realizadas no Algoritmo 10 para utilizar esse controlador, conforme mostra o Algoritmo 12.

```
# Função principal de exploração.
def main():
    global ROBO_DIM, MAPA_DIM_X, MAPA_DIM_Y

    #Carrega o mapa conhecido
    mapa = np.ones((MAPA_DIM_Y / ROBO_DIM , MAPA_DIM_X / ROBO_DIM )) / 2.0

    #Posição dos objetos conhecidos
    lixeira = (-19, 2)
    objeto1 = (2, 4)
    objeto2 = (9, -7)

    # Conecta ao robô.
    c = playerc client(None, 'localhost', 6665)
    if c.connect() != 0:
```

```
        print playerc_error_str()

    # Conecta à interface que permite mover e recuperar a localização.
    p = playerc_position2d(c,0)
    if p.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Conecta ao laser.
    l = playerc_ranger(c, 1)
    if l.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Obstáculos = lixeira, objeto2; Meta = objeto1
    print 'Buscando pelo objeto 1'
    ponto_obstaculo(mapa, lixeira)
    ponto_obstaculo(mapa, objeto2)
    controle(c, p, l, mapa, objeto1)

    savefig('explora00.png')
    pause(0.001)

    # Obstáculos = lixeira, objeto2; Meta = lixeira
    print 'Buscando pela lixeira'
    ponto vazio(mapa, lixeira)
    controle(c, p, l, mapa, lixeira)

    savefig('explora01.png')
    pause(0.001)

    # Obstáculos = lixeira; Meta = objeto2
    print 'Buscando pelo objeto 2'
    ponto_vazio(mapa, objeto2)
    ponto obstaculo(mapa, lixeira)
    controle(c, p, l, mapa, objeto2)

    savefig('explora02.png')
    pause(0.001)

    # Meta = lixeira
    print 'Buscando pela lixeira'
    ponto_vazio(mapa, lixeira)
    controle(c, p, l, mapa, lixeira)

    savefig('explora03.png')
    pause(0.001)

    # Encerra a comunicação com o robô.
    p.unsubscribe()
    l.unsubscribe()
    c.disconnect()

# Executa a função main().
if __name__ == '__main__':
    main()
```

Algoritmo 12 - Rotina principal para o controle de navegação por campos potenciais com capacidade de exploração.

4.1 Resultados

O algoritmo de campos potenciais permitiu que o robô navegasse satisfatoriamente por todo o ambiente, sem se chocar a obstáculos e apresentando uma rota suave, mesmo sem conhecer o mapa de antemão. A Figura 6 mostra os mapas de grade de ocupação gerados no final de cada uma das 4 fases da missão, que consistiam em buscar o objeto 1, buscar a lixeira pela primeira vez, buscar o objeto 2, e buscar a lixeira pela segunda vez.

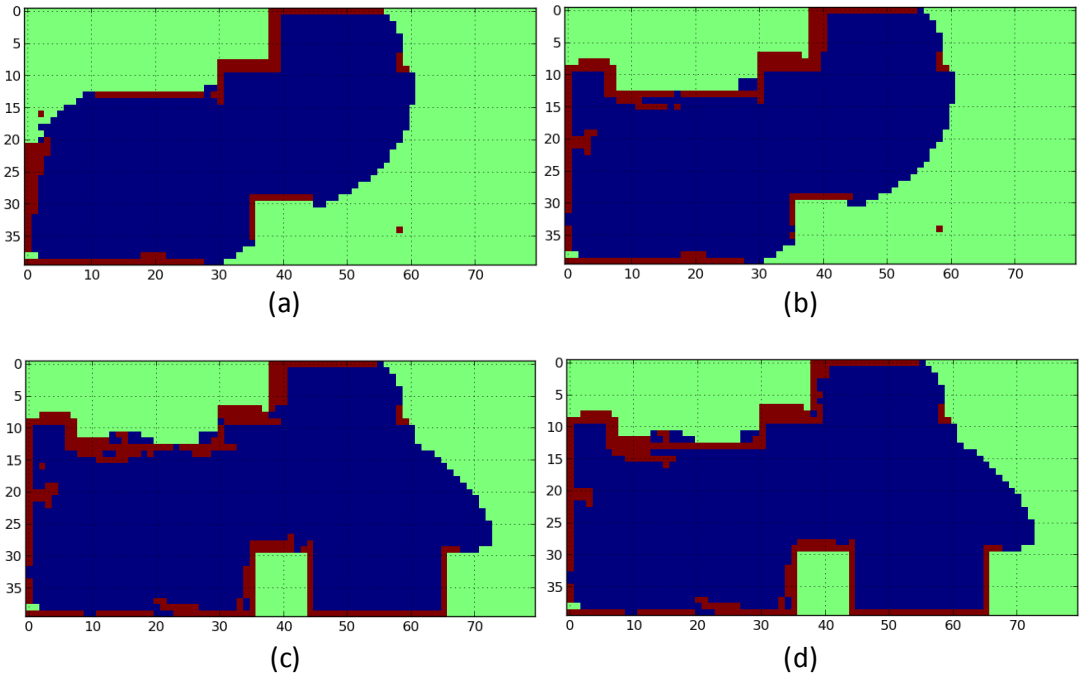

Figura 6 – Mapa de grade de ocupação no final das 4 fases de execução da missão: (a) busca do objeto 1, (b) primeira busca a lixeira, (c) busca ao objeto 2, e (d) segunda busca a lixeira.

5 Navegação com Campos Potenciais Harmônicos e Orientados

Outros dois métodos de navegação foram testados: campos potenciais harmônicos e campos potenciais orientados. O método dos campos potenciais baseia-se na aplicação da equação de Laplace sobre um mapa discretizado, que gera uma função potencial que não sofre com mínimos locais. A solução harmônica, porém, é um caso particular de outra classe de solução que permite orientar os vetores que chegam ao objetivo – daí o nome campos potenciais orientados. Ambos são muito semelhantes, conforme se pode ver pelos Algoritmos 13 e 14.

```
# Aplica o método de Campos Potenciais Harmonicos, dado o campo, a meta,
# o erro máximo e quantidade máxima de iterações
def campos_potenciais_harmonicos(campo, meta, e=1e-3, iteracoes=1000):
    cph = array(campo, copy=True, dtype=double)
    meta = ponto_mapa(campo, meta)
    cph[meta] = 0
    campo[meta] = True

    for k in xrange(iteracoes):
        erro_quadratico = 0
        for y in xrange(1, campo.shape[0]-1):
            for x in xrange(1, campo.shape[1]-1):
                if campo[y, x] == False:
                    p_velho = cph[y, x]
                    p_acima, p_abaixo, p_esquerda, p_direita = cph[y-1, x], cph[y+1, x],
cph[y, x-1], cph[y, x+1]
                    p_novo = p_velho + 1.8 / 4.0 * (p_direita + p_esquerda + p_acima +
p_abaixo - 4.0 * p_velho)
                    erro_quadratico += (p_novo - p_velho) ** 2
                    cph[y, x] = p_novo

        if erro_quadratico < e:
            campo[meta] = False
            return (cph, k, erro_quadratico)
```

Algoritmo 13 – Método de Campos Potenciais Harmônicos.

```
# Aplica o método de Campos Potenciais Orientados, dado o campo, a meta, um
# vetor, uma constante epsilon, o erro máximo e quantidade máxima de iterações
def campos_potenciais_orientados(campo, meta, v, e=0.1, erro_maximo=1e-3,
iteracoes=1000):
    meta = ponto_mapa(campo, meta)
    campo[meta] = True
    cph = array(campo, copy=True, dtype=double)
    cph[meta] = 0
    l = e / 2.0

    for k in xrange(iteracoes):
        erro_quadratico = 0
        for y in xrange(1, campo.shape[0]-1):
            for x in xrange(1, campo.shape[1]-1):
                if campo[y, x] == False:
                    p_velho = cph[y, x]
                    p_acima, p_abaixo, p_esquerda, p_direita = cph[y-1, x], cph[y+1, x],
cph[y, x-1], cph[y, x+1]
                    p_novo = ((1 + l * v[1]) * p_direita + (1 - l * v[1]) * p_esquerda +
(1 + l * v[0]) * p_acima + (1 - l * v[0]) * p_abaixo) / 4.0
                    erro_quadratico += (p_novo - p_velho) ** 2
                    cph[y, x] = p_novo

        if erro_quadratico < erro_maximo:
            campo[meta] = False
            return (cph, k, erro_quadratico)
```

Algoritmo 14 – Método de Campos Potenciais Orientados.

Os Algoritmos 13 e 14 retornam uma matriz com o potencial de cada grade, logo o controlador – mostrado pelo Algoritmo 15 – precisa calcular o ângulo do vetor de cada célula a fim de guiar o robô até a meta.

```
# Controla o robô usando os campos potenciais.
def controle(c, p, cph, meta, d_parada=0.5, k_d = 1.0):
    # Ganhos do controlador
    k_v = 5
    k_om = 0.5

    # Calcula a distância inicial â meta.
    c.read()
    d = distancia((p.px, p.py), meta)

    # Enquanto a distância do robô ao objetivo for maior que o crit. de parada
    while d > d_parada:
        # Atualiza os dados do robô.
        c.read()
        robo, th = (p.px, p.py), p.pa
        robo_mapa = ponto_mapa(cph, robo)

        # Calcula a distância e determina a velocidade em X
        d = distancia(robo, meta)

        # Recupera a força e o ângulo do potencial.
        (y, x) = robo_mapa
        p_acima, p_abaixo, p_esquerda, p_direita = cph[y-1, x], cph[y+1, x], cph[y, x-
1], cph[y, x+1]
        om = atan2(p_abaixo - p_acima, p_esquerda - p_direita)

        # Calcula o erro do robô ao ângulo do potencial e a velocidade angular.
        th_err = ajusta_angulo(ajusta_angulo(om) - ajusta_angulo(th))
        v_th = th_err * k_om

        # Determina a velocidade linear proporcionalmente a distância,
        # reduzindo a velocidade conforme se aproxima da meta.
        v_x = d / k_d * k_v
        if v_x > 1.0:
            v_x = 1.0
        elif v_x < 0.2:
            v_x = 0.2

        # Envia as novas velocidades ao robô.
        p.set_cmd_vel(v_x, 0.0, v_th, 1)
```

```
    # Para o robô.
    p.set_cmd_vel(0.0, 0.0, 0.0, 1)
```

Algoritmo 15 – Controlador de Navegação.

Para utilizar os campos potenciais harmônicos, estipulou-se como critérios de parada uma quantidade máxima de 1000 iterações ou o erro quadrático máximo de 1×10^{-8}, conforme mostra o Algoritmo 16. Para os campos potenciais orientados, adotou-se ainda o valor de $\epsilon = 0.5$, conforme mostra o Algoritmo 17.

```
def main():
    # Conecta ao robô.
    c = playerc_client(None, 'localhost', 6665)
    if c.connect() != 0:
        print playerc_error_str()

    # Conecta à interface que permite mover e recuperar a localização.
    p = playerc position2d(c,0)
    if p.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Conecta ao laser.
    r = playerc_ranger(c, 1)
    if r.subscribe(PLAYERC_OPEN_MODE) != 0:
        print playerc_error_str()

    # Carrega o mapa do ambiente.
    mapa = loadtxt('mapa.txt')
    mapa = where(mapa > 0, 1, mapa)
    mapa = array(mapa, dtype=bool)

    #Posição dos objetos conhecidos
    lixeira = (-19, 2)
    objeto1 = (2, 4)
    objeto2 = (9, -7)

    # Navega até o objeto 1
    cph, k, e = campos_potenciais_harmonicos(mapa, objeto1, 1e-8)
    print k, e
    desenha_cph_v(cph)
    controle(c, p, cph, objeto1)

    # Navega até a lixeira
    cph, k, e = campos_potenciais_harmonicos(mapa, lixeira, 1e-8)
    print k, e
    desenha_cph_v(cph)
    controle(c, p, cph, lixeira)

    # Navega até o objeto 2
    cph, k, e = campos_potenciais_harmonicos(mapa, objeto2, 1e-8)
    print k, e
    desenha_cph_v(cph)
    controle(c, p, cph, objeto2)

    # Navega até a lixeira
    cph, k, e = campos_potenciais_harmonicos(mapa, lixeira, 1e-8)
    print k, e
    desenha_cph_v(cph)
    controle(c, p, cph, lixeira)
```

Algoritmo 16 – Programa principal para a navegação com campos potenciais harmônicos.

```
def main():
    # Conecta ao robô.
    c = playerc_client(None, 'localhost', 6665)
    if c.connect() != 0:
        print playerc_error_str()
```

```
# Conecta à interface que permite mover e recuperar a localização.
p = playerc_position2d(c,0)
if p.subscribe(PLAYERC_OPEN_MODE) != 0:
    print playerc_error_str()

# Conecta ao laser.
r = playerc_ranger(c, 1)
if r.subscribe(PLAYERC_OPEN_MODE) != 0:
    print playerc_error_str()

# Carrega o mapa do ambiente.
mapa = loadtxt('mapa.txt')
mapa = where(mapa > 0, 1, mapa)
mapa = array(mapa, dtype=bool)

#Posição dos objetos conhecidos
lixeira = (-19, 2)
objeto1 = (2, 4)
objeto2 = (9, -7)

# Navega até o objeto 1
cpo, k, e = campos_potenciais_orientados(mapa, objeto1, (0,-1), 0.5, 1e-8)
desenha cph_v(cpo)
print k, e
controle(c, p, cpo, objeto1)

# Navega até a lixeira
cpo, k, e = campos_potenciais_orientados(mapa, lixeira, (0,-1), 0.5, 1e-8)
desenha cph_v(cpo)
print k, e
controle(c, p, cpo, lixeira)

# Navega até o objeto 2
cpo, k, e = campos_potenciais_orientados(mapa, objeto2, (-1,0), 0.5, 1e-8)
desenha_cph_v(cpo)
print k, e
controle(c, p, cpo, objeto2)

# Navega até a lixeira
cpo, k, e = campos_potenciais_orientados(mapa, lixeira, (0,-1), 0.5, 1e-8)
desenha_cph_v(cpo)
print k, e
controle(c, p, cpo, lixeira)
```

Algoritmo 17 – Programa principal para a navegação com campos potenciais orientados.

5.1 Resultados

Os campos potenciais harmônicos, como pode ser visto na Figura 7, geram um campo potencial que leva ao objetivo com curvas mais suaves em direção ao objetivo, quando comparado aos campos potenciais clássicos. O tempo de processamento foi maior, na ordem de segundos, o que é atribuído ao fato da linguagem Python não ser eficiente para cálculo numérico. Nas três situações da Figura 7, a quantidade de iterações necessárias para resolver o problema foram (a) 114, (b) 104 e (c) 111, com erro máximo de 1×10^{-8}. Já os campos potenciais orientados, como mostra a Figura 8, distorcem o campo para forçar o robô a chegar no objetivo com uma orientação em particular. Seu desempenho também é menor que os campos potenciais clássicos. Para calcular os potenciais da Figura 8, o algoritmo precisou de (a) 424, (b) 419 e (c) 323 iterações, com erro máximo de 1×10^{-8}.

Ambos os métodos foram capazes de levar o robô para a meta independente do ponto de partida, sem o problema de mínimos locais. A diferença de desempenho entre ambas está na quantidade de iterações necessárias para convergir: a versão orientada, nos testes, levou cerca de 3 a 4 vezes mais iterações que a versão harmônica. Por outro lado, a versão orientada deforma os vetores para forçar o robô chegar à meta com um ângulo predefinido.

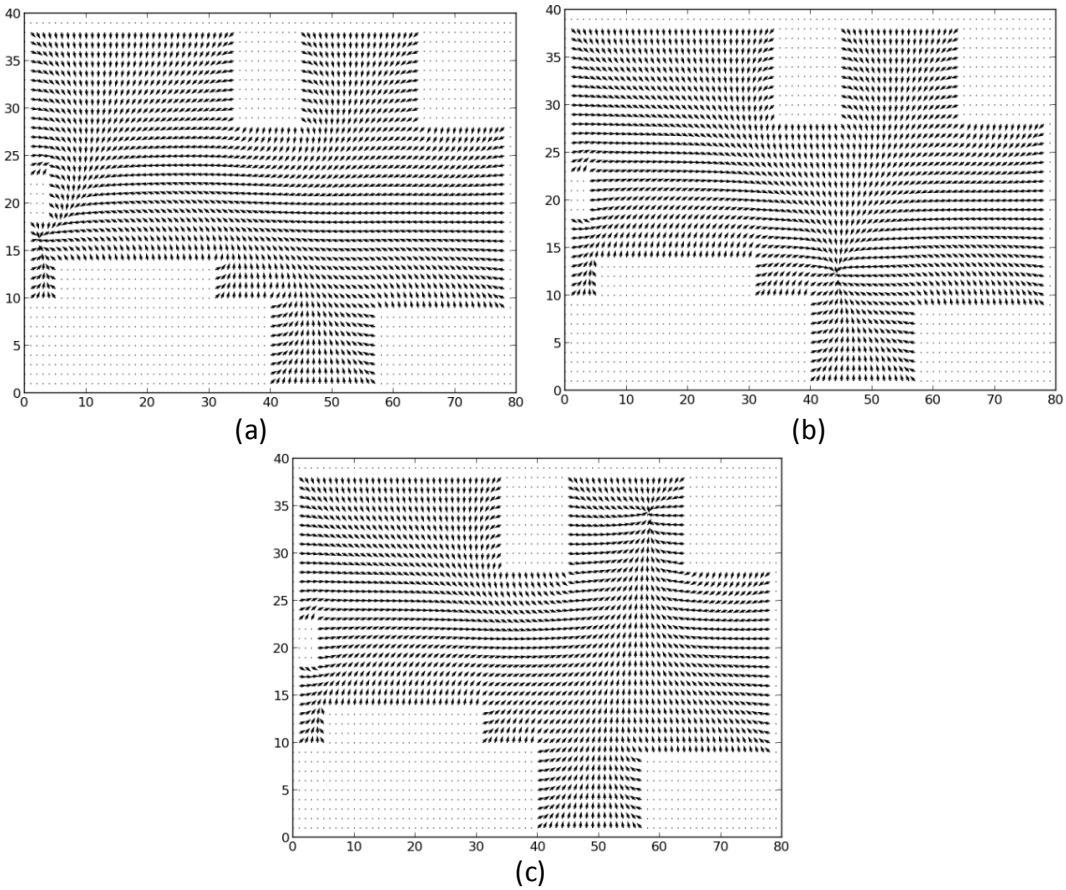

Figura 7 – Campos potenciais que levam até: (a) lixeira, (b) objeto 1 e (c) objeto 2.

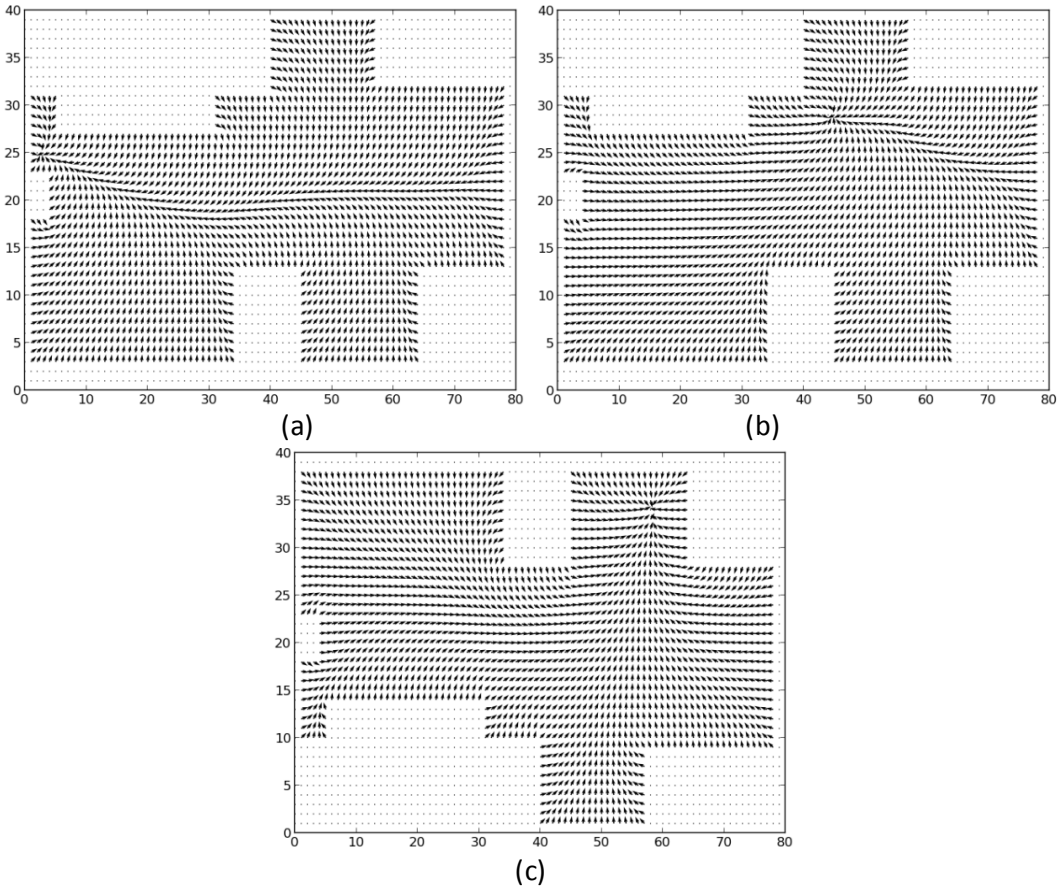

Figura 8 – Campos potenciais que levam até: (a) lixeira, (b) objeto 1 e (c) objeto 2.