Trabalho Prático 1 - Robótica Móvel (DCC042)
Planejamento e Navegação

Hugo Araújo de Sousa

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais

hugosousa@dcc.ufmg.br

1. Introdução

Em Robótica Móvel, dois dos conceitos mais fundamentais para desenvolvimento de robôs móveis são o controle do robô e o planejamento de caminhos. O primeiro refere-se à tarefa de fornecer velocidades a um robô de forma que ele siga um determinado caminho, já o segundo refere-se à tarefa de encontrar caminhos em um ambiente que levam o robô de uma posição inicial a uma posição final desejada [Siegwart et al. 2011]. Além disso, o controlador do robô pode ser realimentado constantemente, de acordo com o que é lido sobre o ambiente, através dos sensores do robô. Esse esquema de realimentação é chamado de malha fechada. Neste trabalho, dois algoritmos de planejamento de caminho foram implementados: *Bug* 2, para robôs holonômicos; e Campos Potenciais, para robôs diferenciais, ambos utilizando o esquema de controle em malha fechada. Para isso, foi utilizado o *framework* ROS (*Robot Operating System*) [Quigley et al. 2009].

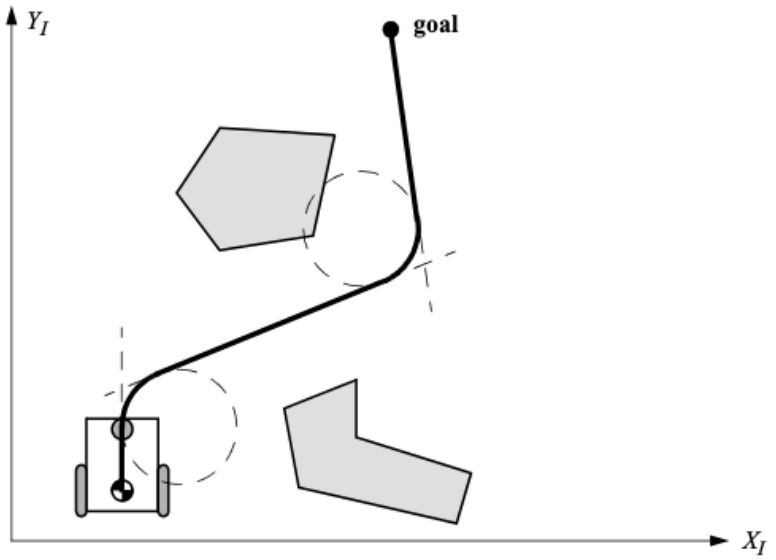

Figura 1. Exemplo de controle de um robô diferencial seguindo um caminho até uma posição alvo. O caminho seguido leva em consideração a presença de obstáculos no ambiente. [Siegwart et al. 2011]

1.1. Robôs holonômicos e o algoritmo Bug 2

A pose ($\mathbf{q}$) de um robô é definida como a tupla de variáveis que definem sua posição e orientação no espaço. O número de variáveis que compõem a pose é chamado de grau

de liberdade (DoF - *degree of freedom*, do inglês). Se um robô consegue atuar em todas as variáveis de sua pose de forma independente entre si, ou seja, o seu *DoF* é igual às suas velocidades atuáveis, ele é chamado de holonômico. Neste trabalho, foi escolhido o algoritmo Bug 2 para planejar caminhos para robôs holonômicos. O funcionamento desse algoritmo é mostrado no pseudocódigo abaixo.

- **Algoritmo Bug 2**
  - Definir posição alvo do robô: $\mathbf{g}$.
  - De acordo com a posição inicial $\mathbf{p_0}$ do robô, traçar a reta $\mathbf{m}$ que liga $\mathbf{p_0}$ e $\mathbf{g}$.
  - Enquanto o robô não atingiu a posição alvo, faça:
    - Seguindo $\mathbf{m}$, vá em direção a $\mathbf{g}$.
    - Caso um obstáculo seja encontrado, contorne-o, até encontrar novamente a reta $\mathbf{m}$.

Para este trabalho, uma pequena melhoria foi implementada no algoritmo: ao contornar obstáculos, o robô deve procurar pela reta **m** em algum ponto que o leve a uma distância menor até a posição alvo do que a distância até essa posição que tinha quando encontrou o obstáculo. Para controlar o robô holonômico com o algoritmo Bug 2, foi utilizado o seguinte esquema de controle:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_\theta \end{bmatrix} \begin{bmatrix} (g_x - x) \\ (g_y - y) \\ (g_\theta - \alpha) \end{bmatrix}$$

Onde $\dot{x}, \dot{y}$ e $\dot{\theta}$ representam a velocidade linear do robô no eixo x, y e velocidade angular em torno do eixo z, respectivamente. Além disso, k_x, k_y e k_θ são constantes de ganho para cada uma das velocidades citadas; g_x, g_y e g_θ são as coordenadas x e y da posição alvo e o ângulo formado pela reta que liga o robô à posição alvo e o eixo x, respectivamente; e x, y e α são as coordenadas x e y do robô e seu ângulo com o eixo x, respectivamente.

1.2. Robôs diferenciais e o algoritmo dos Campos Potenciais

Ao contrário dos robôs holonômicos, os robôs diferenciais têm velocidades atuáveis inferiores ao seu *DoF*. Dessa forma, não conseguem atuar em todos os elementos de sua pose de forma independente. Por isso, o seu controle é diferente do controle de robôs holonômicos. Para este trabalho, foi usado o seguinte esquema para controle de robôs diferenciais:

$$\begin{aligned} v_x &= \dot{x} * \cos\alpha + \dot{y} * \sin\alpha \\ v_y &= \dot{y} * \cos\alpha - \dot{x} * \sin\alpha \\ w_z &= k_t * (\theta - \alpha) \end{aligned}$$

Onde v_x, v_y e w_z são as velocidades lineares (já convertidas para o *frame* do mundo) no eixo x, y e angular em torno do eixo z, respectivamente; $\dot{x}, \dot{y}$ são as velocidades lineares em x e y calculadas pelo algoritmo de campos potenciais. Esse algoritmo

funciona como uma analogia à física e o que ocorre com partículas de sinais opostos, de carga elétrica, por exemplo. Dessa forma, a posição alvo atua exercendo uma força de atração no robô, que por sua vez é repelido por forças de repulsão geradas pelos obstáculos ao seu redor. Essas forças de atração e repulsão formam uma força resultante que guia o robô no ambiente. O pseudocódigo do algoritmo é mostrado abaixo.

- **Algoritmo de Campos Potenciais**
  - Definir posição alvo do robô: g.
  - Enquanto o robô não atingiu a posição alvo, faça:
    - Calcular a força de atração à posição alvo.
    - Calcular as forças de repulsão dos obstáculos ao redor do robô.
    - Calcular a força resultante das forças de atração e repulsão.
    - Separar as componentes f_x e f_y da força resultante, que atuam no eixo x e y, respectivamente (essas forças são as variáveis $\dot{x}$ e $\dot{y}$ mostradas acima).
    - Fornecer ao robô suas velocidades com base nessas forças (e o esquema de controle acima).

2. Implementação

A implementação dos algoritmos de controle e planejamento foram feitas no ROS utilizando a linguagem Python 2. Para os dois algoritmos, foram utilizadas mensagem de odometria (`nav_msgs.msg.Odometry`) para determinar a posição do robô no mundo e mensagens do sensor de *laser* (`sensor_msgs.msg.LaserScan`) para determinar a distância dos objetos ao redor do robô no mundo.

2.1. Leitura do *Laser*

O robô holonômico utilizado possui um *laser* que cobre em $180°$ a área a sua frente. No ROS, esse *laser* contém, dentre outras informações, uma lista com os valores lidos de distância dos objetos detectados pelo laser até o robô. Para o robô holonômico, essa lista possui 361 posições, de forma que a posição de índice i informa a distância do objeto ao ângulo de $(\frac{i}{2})$ °a partir da direita do objeto. No caso do robô diferencial, o seu *laser* cobre $360°$ ao seu redor e a sua lista de distâncias possui 721 posições, de forma que, similarmente ao robô holonômico, a posição de índice i na lista de distâncias informa a distância do objeto ao ângulo de $(\frac{i}{2})°$ a partir da posição imediatamente atrás do robô. Em ambos os casos, o ângulo aumenta no sentido anti-horário. As Figuras 2 e 3 ilustram os esquemas mencionados.

2.2. Bug 2

Como mencionado anteriormente, o algoritmo Bug 2 traça uma reta da posição inicial do robô até a posição alvo. Sempre que possível (quando não há obstáculos ou consegue contorná-los, caso eles existam), o robô procura por essa reta e a segue em direção ao alvo. Para calcular a reta, foi utilizado o método `set_goal_line` que calcula a equação geral da reta (formato $ax + by + c = 0$). Assim, a qualquer momento, podemos calcular a distância do robô até essa reta (através do método `get_distance_to_line`) e verificar quando ele está suficientemente próximo dela para continuar seguindo sua direção até a posição alvo. O movimento na direção da reta, rumo à posição alvo é feito através do

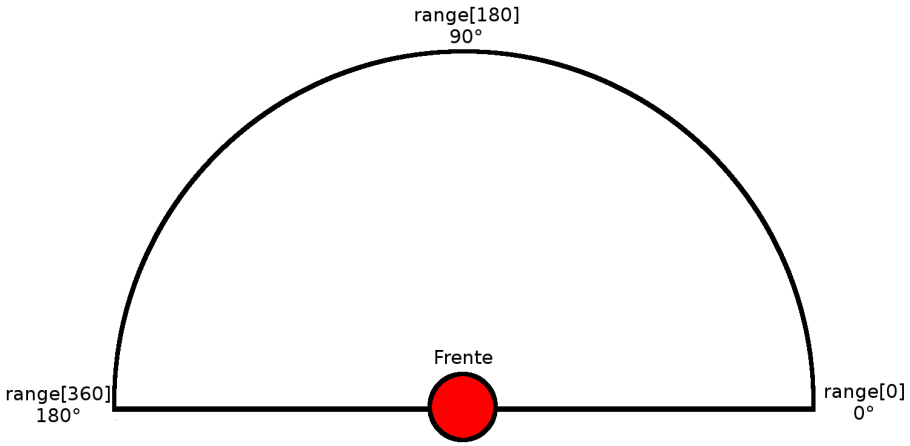

Figura 2. Esquema de visão do robô holonômico e como é feito o acesso às leituras do *laser* no ROS.

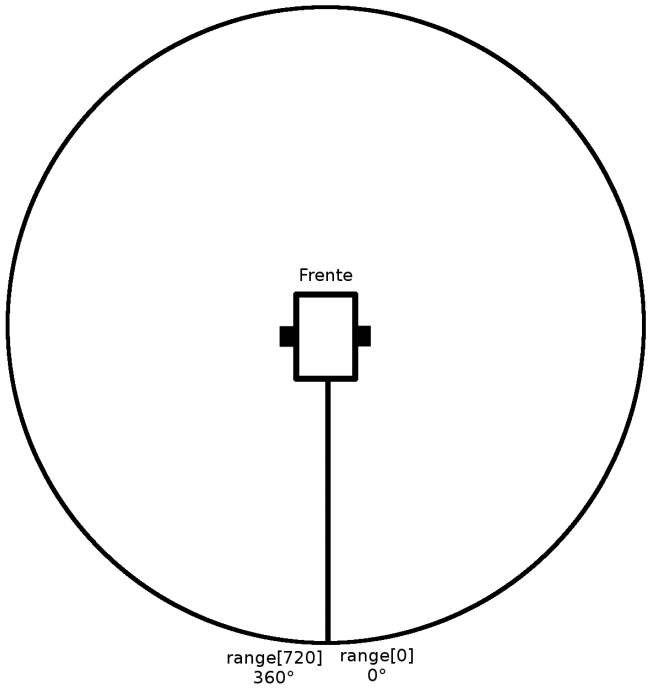

Figura 3. Esquema de visão do robô diferencial e como é feito o acesso às leituras do *laser* no ROS.

controlador implementado no método `follow_line`. Nele, o robô é controlado pelo esquema mostrado na Seção 1.1, e podemos checar se ele encontrou um obstáculo através do método `is_blocked`, que verifica a leitura do *laser* à frente do robô (seguindo o esquema da Figura 2, a frente do robô foi definida pelo intervalo entre os ângulos $120°$ e $60°$). Ao detectar um obstáculo à frente, o robô entra no estado em que deve contorná-lo, ação realizada pelo método `outline_obstacle`, que gira o robô em sentido anti-horário para começar a contornar o obstáculo, procurando pela reta definida inicialmente.

2.3. Campos Potenciais

O algoritmo de Campos Potenciais tem uma visão geral mais simples do que o algoritmo Bug 2, já que não possui tantos estados possíveis quanto o segundo. No algoritmo Bug 2, existe o estado de seguir em direção à posição alvo e o estado de contornar um objeto. Para o algoritmo de Campos Potenciais, o estado é sempre o mesmo: calcular as forças resultantes e seguir em direção à força resultante. Dessa forma, foi implementado o método `get_force_vector` para realizar o cálculo da força resultante que atua no robô. Esse método utiliza os métodos `get_f_att` e `get_f_rep` para calcular as forças de atração e repulsão, respectivamente, e, então, as soma para obter a resultante. O primeiro calcula força a força de atração como proporcional à distância entre a posição alvo e a posição atual do robô. Já o segundo calcula a força de repulsão através das leituras do sensor de *laser* do robô, que informam a distância dos obstáculos. Assim, quando menor a distância até um obstáculo, maior a força de repulsão exercida no robô. Além disso, também define-se uma distância mínima de influência, de forma que somente obstáculos com distância menor que essa distância mínima exercem forças de repulsão.

Durante os testes realizados com esse algoritmo, notou-se que na maioria dos casos, o robô encontrava um mínimo local e não conseguia sair de tal posição, uma vez que a força resultante se tornava zero. Para contornar esse problema, uma modificação foi feita em relação às forças de repulsão, que passar a ser aplicadas em um ângulo $\theta = \theta_0 + 90^\circ$.

2.4. Execução e Estrutura do Projeto

Os algoritmos foram criados sob um pacote ROS de nome `tp1`. Além, disso, ambos os programas dos algoritmos recebem obrigatoriamente como parâmetro de linha de comando as coordenadas x e y da posição alvo do robô. Dessa forma, podemos executar os programas da seguinte forma (após a execução dos nodos `roscore` e `stage_ros`).

```
rosrun tp1 bug2.py <x> <y>
rosrun tp1 potential_fields.py <x> <y>
```

Além disso, para executar todos os nodos necessários de uma só vez e rodar os algoritmos, podemos usar os arquivos `launch` do ROS, da seguinte forma.

```
roslaunch tp1 bug2.launch x:=<x> y:=<y>
roslaunch tp1 pfields.launch x:=<x> y:=<y>
```

A fim de garantir o funcionamento correto do projeto, é importante que a estrutura do pacote no *workspace* ROS seja similar à mostrada na Figura 4.

3. Testes

Para verificar a corretude da implementação dos algoritmos e como eles são eficazes em diferentes cenários, essa Seção ilustra alguns dos testes realizados. Dessa forma, além de utilizar mapas diferentes, foram utilizadas posições iniciais e alvo diferentes, de forma que os algoritmos devem achar caminhos em diferentes situações. Para cada teste também foi coletado o tempo de execução do algoritmo (para os casos onde um caminho até a posição alvo é encontrado).

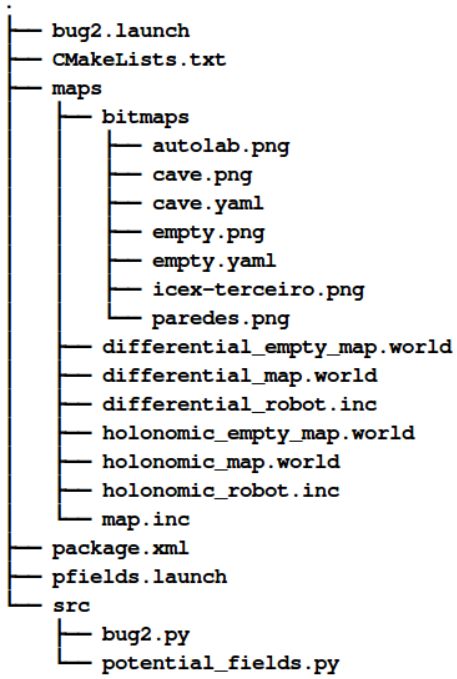

Figura 4. Árvore do diretório do pacote do projeto no *workspace* do ROS.

Para o algoritmo de Campos Potenciais, notou-se que a escolha de parâmetros (as constantes de repulsão e atração e a distância mínima de influência dos obstáculos) afeta o desempenho de forma extremamente significativa. Por isso, foi necessário, para cada teste, executá-lo várias vezes, alterando esses parâmetros, para que o robô ao menos conseguisse chegar à posição alvo sem bater em obstáculos, se manter preso ao redor de obstáculos (devido a forças de repulsão tangentes muito fortes) ou parar em mínimos locais. Nas descrições dos testes abaixo, a notação **k = (...)** indica os parâmetros escolhidos para as constantes de atração, repulsão e distância mínima de influência dos obstáculos, nessa ordem. Três dos principais testes executados estão listados abaixo.

1. **Teste 1: Mapa cave.png, posição de início (-7, -7) e posição alvo (-3, 0)**. Cenário: obstáculos simples.
   - **Bug 2:** Alvo encontrado sem problemas. **Tempo de execução**: 21.31s.
   - **Campos Potenciais:** Alvo encontrado sem problemas. **k = (10, 5, 2)**. **Tempo de execução:** 18.77s.
2. **Teste 2: Mapa paredes.png, posição de início (6, -6) e posição alvo (-5.5, 6)**. Cenário: caminhos estreitos.
   - **Bug 2:** Alvo encontrado sem problemas. **Tempo de execução**: 88.67s.
   - **Campos Potenciais:** Alvo encontrado sem problemas. **Tempo de execução:** 38.84s.
3. **Teste 3: Mapa paredes.png, posição de início (4, 6) e posição alvo (-5, -4.5)**. Cenário: cercado de obstáculos.
   - **Bug 2:** Alvo encontrado sem problemas, poderia ter saído pela direita da cerca de obstáculos, porém considerou que estava fechado. **Tempo de execução**: 63.11s.
   - **Campos Potenciais:** Alvo não encontrado: mínimo local e batida.

4. Conclusão

Nestre trabalho foram implementados dois algoritmos de planejamento de caminhos: Bug 2 para robôs holonômicos e Campos Potenciais para robôs diferenciais. Para isso, além dos algoritmos de planejamento de caminhos, foi necessário implementar o controle de cada robô, para que pudessem navegar nos caminhos gerados no ambiente.

A maior dificuldade encontrada se refere à curva de aprendizado de utilização do *framework* ROS, cujo funcionamento não é trivial. Não existe muita documentação disponível sobre as mensagens trocadas pelos nós utilizados pelo *framework*, de forma que foi necessário realizar vários testes iniciais para verificar o funcionamento básico de cada um dos componentes necessários para a implementação dos algoritmos.

Uma outra dificuldade foi a de relacionar os diferentes *frames*, isto é, sistemas de coordenadas envolvidos na manipulação dos robôs. É necessário estar atento para converter os pontos e velocidades de um sistema para o outro, tendo sempre o sistema de coordenadas do mundo (ambiente) como referencial principal.

De maneira geral, creio que o aprendizado foi grande, não só referente ao funcionamento e uso do *framework* ROS, mas também em relação aos algoritmos de controle e planejamento de caminhos. Acredito que as aulas da disciplina foram importantes, entretanto, não foram suficientes para a implementação do trabalho, de forma que várias fontes externas [Craig 2005, Yoonseok Pyo 2017, Choset et al. 2015] foram consultadas, principalmente o livro texto. Os testes executados e seus resultados serviram para mostrar as vantagens e desvantagens de cada algoritmo e o tipo de situações onde cada um se sai melhor.

Referências

Choset, H., Hager, G., and Dodds, Z. (2015). Robotic motion planning: Bug algorithms. *Lecture Notes, Carnegie Melon University, http://www. cs. cmu. edu/˜ motionplanning/lecture/Chap2-Bug-Alg_howie. pdf*.

Craig, J. J. (2005). *Introduction to robotics: mechanics and control*, volume 3. Pearson/Prentice Hall Upper Saddle River, NJ, USA:.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

Siegwart, R., Nourbakhsh, I. R., Scaramuzza, D., and Arkin, R. C. (2011). *Introduction to autonomous mobile robots*. MIT press.

Yoonseok Pyo, Hancheol Cho, L. J. D. L. (2017). *ROS Robot Programming (English)*. ROBOTIS.