ECAI21.2 - Robótica Móvel (Prática) - Turma 01

Prof. André Chaves Magalhães
Dezembro/2023
Universidade Federal de Itajubá - *Campus* de Itabira

RELATÓRIO - PROJETO 5

Emilly Lara Deiró*, Rodrigo*, Saulo Labiapari*, Vinicius de Sousa David*, Wilgner Luís Pereira Barreto*

* *Universidade Federal de Itajubá - Campus de Itabira*
Rua Irmã Ivone Drumond, 200 - Distrito Industrial II - 35903-087
Itabira, Minas Gerais, Brasil

E-mails: emillydeiro@unifei.edu.br, @unifei.edu.br, saulolabiapari@unifei.edu.br, d2019005631@unifei.edu.br, wilgner.v9@gmail.com

Resumo— Este trabalho propõe a implementação de um algoritmo tipo Tangent BUG, navegando um robô com acionamento diferencial simulado no StageROS em um ambiente com obstáculos. O robô esta equipado com um sensor laser Hokuyo e deve se mover entre duas posições quaisquer, escolhidas pelo usuário em tempo de execução, sem colidir com os obstáculos. Caso não haja caminho entre as posições escolhidas, o robô deve informar isto ao usuário em um tempo finito.

Palavras-chave— Controlador, Robô, Tangentbug, StageROS

1 Introdução

A presente pesquisa tem como propósito criar um robô com acionamento diferencial, instituído por um laser Hokuyo, por meio de um algoritmo do tipo Tangent BUG.

O algoritmo Tangent BUG é efetivo para o planejamento de caminhos robóticos, tendo como condição que o robô possua um objetivo global, com conhecimento apenas sobre as informações locais do ambiente. Este algoritmo, faz uso de um sensor de alcance finito, com o intuito de coletar dados sobre obstáculos existentes no ambiente.

O comportamento do robô é dividido em dois: movimento para o objetivo e seguir borda. O movimento para o objetivo é utilizado enquanto não houver obstáculos bloqueadores ou que o mesmo não aumente a distância heurística. Sendo esta dada pela equação

$$h = d(x, P_t) + d(P_t, q_goal) \quad (1)$$

em que,

- x é a posição atual do robô;
- q_goal é a posição do objetivo;
- P_t é o ponto de tangência mais próximo do obstáculo.

Já o comportamento seguir borda, é comandado quando a distância heurística aumenta. Permanecendo neste comportamento até que a distância percorrida seja menor que a distância seguida.

O Tangent BUG possui diversas vantagens com relação a outros algoritmos de planejamento de caminhos robóticos. Dentre elas, destacam-se:

- Comportamento Simples;
- Caminho Intuitivo;
- Efetividade;
- Adaptação.

2 Objetivos

Este estudo tem como objetivo principal implementar um algoritmo tipo Tangent BUG, a fim de, navegar um robô com acionamento diferencial simulado no StageROS em um ambiente com obstáculos. Considerando que o robô deve estar equipado com um sensor laser Hokuyo e que deve se mover entre duas posições quaisquer, a serem escolhidas pelo usuário em tempo de execução, de forma que não colida com os obstáculos. Ademais, caso não haja caminho entre as posições escolhidas, o robô deve informar isto ao usuário em um tempo finito.

3 Ferramentas

Para a realização deste projeto foram utilizados o Sistema Operacional de Robôs (ROS) e o ambiente de simulação StageROS. O ROS é um framework de software livre amplamente utilizado em robótica, que fornece uma plataforma para desenvolvimento, integração e execução de software em robôs. Ele oferece

uma ampla variedade de bibliotecas, ferramentas e pacotes para programação de robôs, incluindo controle, percepção, planejamento e comunicação. O ROS foi utilizado neste projeto para implementar um robô, equipado com um sensor laser Hokuyo e para visualizar os dados de sua trajetória em tempo real no ambiente de simulação.

O StageROS é um ambiente de simulação 2D para robôs móveis que permite a criação de cenários virtuais para testar e validar algoritmos de controle e navegação. Ele é baseado no simulador Stage, que é um simulador de robôs de código aberto e multiplataforma. O StageROS foi utilizado neste projeto para simular o robô de acionamento diferencial e para testar o controlador desenvolvido pelos alunos. Ele permitiu que os alunos avaliassem o desempenho do algoritmo Tengent BUG em diferentes cenários e ajustassem os parâmetros do controlador para melhorar a precisão e a estabilidade do robô.

4 Desenvolvimento

código define várias variáveis para armazenar informações como velocidades desejadas, parâmetros de controle, posições do robô e do objetivo, informações do sensor laser, entre outros. Essas variáveis são utilizadas ao longo do código para controlar o comportamento do robô.

Seu desenvolvimento começa na implementação de várias funções e variáveis que são utilizadas ao longo do código para realizar cálculos e controlar o comportamento do robô. Algumas das funções e variáveis mais importantes incluem:

- `myAbs` e `sign`: funções que retornam o valor absoluto e o sinal de um número, respectivamente.
- `rad2deg`: função que converte um ângulo em radianos para graus.
- `calculateDistance`: função que calcula a distância euclidiana entre dois pontos.
- `Rx`, `Ry`, `Gx`, `Gy`: variáveis que armazenam as coordenadas do robô e do objetivo.
- `po` e `sp`: vetores que armazenam a posição atual do robô e a posição desejada (setpoint), respectivamente.
- `vl` e `va`: variáveis que armazenam as velocidades linear e angular desejadas do robô.
- `vdmax`, `vamax`, `atoli`, `atole`, `dtoli`, `dtole`: parâmetros de controle para ajustar as velocidades do robô.
- `wid`: variável que armazena a largura do robô.
- `obstacles`: vetor que armazena as posições dos obstáculos detectados pelo sensor laser.
- `samples`, `bmin`, `bmax`, `binc`, `rmin`, `rmax`,
- `ranges_laser`, `angles_laser`: variáveis relacionadas à leitura do sensor laser.
- `detected_front`, `detected_right`, `detected_left`,
- `detected_left_45`, `detected_right_45`: variáveis relacionadas à detecção de obstáculos.
- detected_right_45: variáveis que indicam se o robô detectou obstáculos em diferentes direções.

Essas funções e variáveis são utilizadas em diferentes partes do código para realizar cálculos e tomar decisões de navegação. Por exemplo, a função ‘calculateDistance‘ é usada para calcular a distância entre o robô e o objetivo, enquanto as variáveis ‘vl‘ e ‘va‘ são atualizadas com base nas informações do ambiente e do objetivo para controlar a velocidade do robô. As variáveis relacionadas à leitura do sensor laser são usadas para detectar obstáculos e ajustar a trajetória do robô para evitar colisões. Em resumo, a seção 3 do código define as funções e variáveis necessárias para controlar o comportamento do robô e navegar em um ambiente desconhecido.

A função ‘odomCallback‘ é um callback que lida com os dados de odometria do robô. Ela é acionada sempre que uma nova mensagem de odometria é recebida, atualizando as informações sobre a posição e orientação atual do robô.

A função ‘odomCallback‘ recebe um parâmetro que contém a mensagem de odometria, que inclui a posição e orientação do robô. Dentro da função, os dados da mensagem são extraídos e utilizados para atualizar as variáveis que representam a posição e orientação atual do robô.

A extração dos dados da mensagem de odometria geralmente envolve a obtenção das coordenadas x, y e a orientação (frequentemente representada como um ângulo) do robô a partir da mensagem recebida. Esses dados são então utilizados para atualizar as variáveis que armazenam a posição e orientação atual do robô no ambiente.

Essa função é essencial para manter um acompanhamento em tempo real da posição e orientação do robô, o que é fundamental para a navegação autônoma, pois permite que o robô ajuste sua trajetória com base em sua localização atual.

A função ‘odomCallback‘ é um exemplo de como os callbacks são utilizados em sistemas baseados em ROS para lidar com dados de sensores e atualizar o estado do robô em tempo real.

Ja função ‘UpScan‘ é responsável por processar os dados provenientes do sensor de varredura a laser montado no robô. Ela é acionada sempre que uma nova leitura do sensor é recebida, permitindo que o robô detecte obstáculos e atualize suas informações sobre o ambiente ao seu redor.

Dentro da função ‘UpScan‘, a leitura do sensor a laser é processada para identificar obstáculos e calcular a distância entre o robô e os obstáculos detectados

```cpp
// Callbacks
void odomCallback(const nav_msgs::Odometry::ConstPtr& msg) {

    double x, y, theta; // current position and orientation of robot
    // Get current position of robot
    x = msg->pose.pose.position.x;
    y = msg->pose.pose.position.y;

    // Get current orientation of robot
    tf::Quaternion q(
    msg->pose.pose.orientation.x,
    msg->pose.pose.orientation.y,
    msg->pose.pose.orientation.z,
    msg->pose.pose.orientation.w);
    tf::Matrix3x3 m(q);
    double roll, pitch;
    m.getRPY(roll, pitch, theta); // convert quaternion to Euler angles
    current_position.position.x = x;
    current_position.position.y = y;
    current_position.position.z = theta;

    yaw = theta;
    po[_x_] = x;
    po[_y_] = y;
    po[_a_] = yaw;
}
```

Figura 1: odom Function

```cpp
void UpScan(const sensor_msgs::LaserScan& laserScan)
{
    // Clear previous obstacle positions
    min_range = INF;
    obstacles.clear();
    for(size_t i = 0; i < samples; i++)
    {
        ranges_laser[i] = laserScan.ranges[i];
        if (laserScan.ranges[i] < min_range) {
            min_range = laserScan.ranges[i];
        }
        if (laserScan.ranges[i] < D_MAX) {
            double angle = laserScan.angle_min + i * laserScan.angle_increment;
            double obs_x = std::cos(angle) * laserScan.ranges[i];
            double obs_y = std::sin(angle) * laserScan.ranges[i];

            // Transform the obstacle position to the map frame
            double map_x = po[_x_] + obs_x * std::cos(yaw) - obs_y * std::sin(yaw);
            double map_y = po[_y_] + obs_x * std::sin(yaw) + obs_y * std::cos(yaw);

            // Store obstacle position
            obstacles.push_back(Eigen::Vector3d(map_x, map_y, 0.0));
        }
    }
    // Reading from the sensors
    detected_front = int(ranges_laser[135] < 0.3);
    detected_right = int(ranges_laser[45] < 0.5);
    detected_right_45 = int(ranges_laser[90] < 0.5);
    detected_left = int(ranges_laser[225] < 0.5);
    detected_left_45 = int(ranges_laser[180] < 0.5);
    obstacle_detected = (detected_front || detected_right || detected_left);
    if(obstacle_detected){
        double angle = laserScan.angle_min + 135 * laserScan.angle_increment;
        double obs_x = std::cos(angle) * laserScan.ranges[135];
        double obs_y = std::sin(angle) * laserScan.ranges[135];

        // Transform the obstacle position to the map frame
        double map_x = po[_x_] + obs_x * std::cos(yaw) - obs_y * std::sin(yaw);
        double map_y = po[_y_] + obs_x * std::sin(yaw) + obs_y * std::cos(yaw);
        goal << map_x, map_y;
        d_folowed_at = calculateDistance(map_x, map_y, sp[_x_], sp[_y_]);
    }
}
```

Figura 2: Upscan

```cpp
void MovePotential(){
    Eigen::Vector2d robot_position;
    robot_position << po[_x_], po[_y_];

    Eigen::Vector2d control_input = -kp * potential_gradient(robot_position, goal, obstacles);
    double dt = 1. / 10.;
    double dx = (goal.x() - l_goal.x()) / dt;
    double dy = (goal.y() - l_goal.y()) / dt;
    //Feedback Linearization
    u1 = 0 + control_input[0];
    u2 = 0 + control_input[1];
    l_goal << goal;

    // feedback linearization
    Eigen::Matrix2d A;
    A << cos(yaw), -d*sin(yaw),
         sin(yaw), d*cos(yaw);

    Eigen::Vector2d vw = A.inverse() * Eigen::Vector2d(u1,u2);


    // ROS_INFO("x_d: %f y_d: %f", goal.x(), goal.y());
    // ROS_INFO("x_robot: %f y_robot: %f", po[_x_], po[_y_]);
    u1 = kp1*vw[0];
    u2 = kp1*vw[1];
    Vtot = sqrt(pow(u1, 2) + pow(u2, 2));


    if (Vtot >= Vmax){
        u1 = u1 * Vmax / Vtot;
        u2 = u2 * Vmax / Vtot;
    }

    // Apply inverse dynamics controllers to errors and generate velocity commands for robot
    v1 = u1;
    va = u2;
}
```

Figura 3: Move Potential

A função ‘findDiscontinuityPoint‘ desempenha um papel crucial na identificação de pontos de descontinuidade ou mudanças significativas na leitura do sensor a laser. Esses pontos de descontinuidade podem indicar a presença de obstáculos ou características distintas no ambiente, e são fundamentais para o planejamento de trajetórias seguras e a navegação eficiente do robô.

A função ‘MovePotential‘ utiliza um método baseado em potencial para gerar comandos de controle que levam em consideração tanto o objetivo de navegação quanto a presença de obstáculos no ambiente.

A função ‘MovePotential‘ realiza as seguintes etapas principais:

1. Obtenção da posição do robô e do objetivo: A função obtém a posição atual do robô e a localização do objetivo de navegação no ambiente.

2. Cálculo do campo de potencial: Com base na posição do robô, do objetivo e na presença de obstáculos, a função calcula um campo de potencial que representa as influências positivas (atração em direção ao objetivo) e negativas (repulsão dos obstáculos) sobre o movimento do robô.

3. Geração de comandos de controle: Utilizando o campo de potencial calculado, a função gera comandos de controle que levam em consideração as forças de atração e repulsão, resultando em um movimento suave e seguro em direção ao objetivo, evitando colisões com obstáculos.

4. Aplicação de feedback linearização: A função aplica técnicas de feedback linearização para converter os comandos de controle gerados a partir do campo de potencial em velocidades lineares e angulares que podem ser aplicadas diretamente ao robô.

5. Limitação da velocidade: Antes de enviar os comandos de controle ao robô, a função verifica e limita a velocidade resultante para garantir que o movimento do robô permaneça dentro de limites seguros e operacionais.

O codigo usado no tangent bug, se vale de codigos dos exercicios passados da disciplina, como Campos Potenciais, para desviar dos obstaculos e do Feedback Linearization, para que o mesmo chegue ao seu destino.

A função 'moveFollowingEdgeObstacle' é responsável por controlar o movimento do robô quando ele está seguindo a borda de um obstáculo. Essa função é crucial para permitir que o robô navegue de forma segura e eficiente em ambientes onde a estratégia de seguir a borda de um obstáculo é apropriada.

A função 'moveFollowingEdgeObstacle' realiza as seguintes etapas principais:

1. Cálculo da direção da borda do obstáculo: Com base nas leituras do sensor e na detecção da borda do obstáculo, a função calcula a direção da borda em relação à posição atual do robô.

2. Ajuste da velocidade linear e angular: Com base na direção da borda do obstáculo e nas informações sobre a detecção de obstáculos, a função ajusta a velocidade linear e angular do robô para seguir a borda de forma suave e precisa.

3. Atualização da posição do objetivo: A função atualiza a posição do objetivo de navegação com base na direção da borda do obstáculo, permitindo que o robô ajuste sua trajetória para permanecer próximo à borda.

4. Verificação e ajuste da trajetória: A função verifica continuamente a presença de obstáculos à medida que o robô segue a borda e ajusta a trajetória conforme necessário para evitar colisões e permanecer alinhado com a borda do obstáculo.

5. Controle da velocidade e direção: Com base nas informações sobre a detecção de obstáculos e na direção da borda, a função controla a velocidade e a direção do robô para garantir um movimento suave e seguro ao longo da borda do obstáculo.

A função 'moveToTheTarget' é responsável por controlar o movimento do robô em direção ao objetivo de navegação. Essa função utiliza informações sobre a posição atual do robô e a localização do objetivo para gerar comandos de controle que permitem que o robô navegue de forma autônoma em direção ao objetivo.

A função 'moveToTheTarget' realiza as seguintes etapas principais:

1. Cálculo da distância e direção para o objetivo: A função calcula a distância e a direção para o objetivo de navegação com base na posição atual do robô e na localização do objetivo.

2. Seleção da ação apropriada: Com base nas informações sobre a distância e a direção para o objetivo, a função seleciona a ação apropriada para o robô, que pode ser ir diretamente para o objetivo, seguir a borda de um obstáculo ou seguir uma trajetória tangente em torno de um obstáculo.

3. Geração de comandos de controle: Utilizando as informações sobre a ação selecionada, a função gera comandos de controle que permitem que o robô navegue em direção ao objetivo de forma suave e precisa.

4. Atualização da posição do objetivo: A função atualiza continuamente a posição do objetivo com base na posição atual do robô e em informações sobre a presença de obstáculos, permitindo que o robô ajuste sua trajetória para alcançar o objetivo de forma eficiente.

5. Verificação da detecção de obstáculos: A função verifica continuamente a presença de obstáculos no ambiente e ajusta a trajetória do robô conforme necessário para evitar colisões e garantir a segurança da navegação.

```
void moveToTheTarget(ros::Publisher pub, geometry_msgs::Twist vel) {
    while (ros::ok()) {
        // delta 'dl' and 'da' to goal
        double dx = sp[_x_] - po[_x_];
        double dy = sp[_y_] - po[_y_];
        double dl = hypot(dx,dy);
        double da = atan2(dy,dx) - po[_a_];
        Action action = Action::Stop;
        //d_folowed = dl;

        if(CheckGoalDir(da,dl)){
            action = Action::GoGoal;
            followEdge = 0;
            d_folowed = INF;
        }else{
            action = Action::GoGoal;
            if(obstacle_detected){
                action = Action::GoTangent;
                if(save_d_follow){
                    d_folowed = d_folowed_at;
                    save_d_follow = 0;
                }

                if(detected_front){
                    if(dl > d_folowed){
                        action = Action::GoFollowingEdge;
                        followEdge = 1;
                    }
                } else {
                    action = Action::GoTangent;
                    followEdge = 0;
                }
            }
        }

        if (CheckGoalDir(da,dl)){
            action = Action::GoGoal;
            save_d_follow = 1;
            followEdge = 0;
            d_folowed = INF;
        }
        if (followEdge){
            action = Action::GoFollowingEdge;
            if(dl < d_folowed){
                save_d_follow = 1;
                followEdge = 0;
                action = Action::GoGoal;
            }
        }
```

Figura 4: moveToTheTarget

5 Conclusões

Com base na análise do código e no funcionamento das funções implementadas, é possível concluir que o sistema de controle de movimento demonstra um alto nível de sofisticação e eficiência na navegação autônoma do robô. A utilização de técnicas como feedback linearização, cálculo de campos de potencial e detecção de bordas de obstáculos permite que o robô navegue de forma segura e dinâmica, adaptando-se a ambientes desafiadores e desconhecidos.

Além disso, a capacidade do sistema de reagir a mudanças no ambiente, como a detecção de obstáculos e a identificação de pontos de descontinuidade, demonstra uma abordagem robusta e adaptativa para

a navegação autônoma. Essas características são essenciais para a aplicação bem-sucedida do robô em cenários do mundo real, onde a capacidade de evitar obstáculos e seguir trajetórias complexas é fundamental.

Referências