Faculty of Power and Aeronautical Engineering
Department of Robotics

Mobile Robot Report

Task 5

Jafar Jafar,
Pratama Aji Nur Rochman

supervised by
dr. Wojciech Szynkiewicz, & Dawid Seredyński

Contents

1 Introduction

The goal of this task is to introduce navigation realized by Bug2 algorithm. The basic idea is to move along the line connecting the target and initial point and in the case of being near to the obstacle follow its contour and thus circumnavigate it. The program should work as follows:

1. One Find the line l connecting initial and target position.
2. Two Rotate towards the goal.
3. Three Move towards the goal along the l line until reaching an obstacle or the goal. If the goal is reached, then stop.
4. four When the robot reaches the obstacle, save the distance to the goal d.
5. five Use “moving along the walls” controller to avoid the obstacle (Task 4).
6. six Depart immediately when the robot is on the l line again and the distance to the goal is lower than d.
7. seven Go to the step 2

Implement a control callback function solution5 that realizes the Bug2 navigation task. Use the input variables position and orientation to determine the line l and to calculate the distance d. The goal point should be provided as an additional input argument to the function solution5, so running the simualtion would be: run_simulation(@solution5, false, [goal_x, goal_y])

2 Task requirements

- Analyse the LIDAR readings to check if the robot is close to obstacle.
- Use the Finite State Machine (FSM) for switching between different states of Bug2 algorithm (e.g. heading towards the goal, moving along the walls).
- Use the “moving along the walls” controller to circumnavigate the obstacles.
- Use proportional controllers with limited output to move the robot along the line l.
- Use proportional controllers with limited output to move the robot along the line l.
- If the robot reaches the goal, stop the controller.

3 Solution

3.1 Code

The callback function for this task was declared as:

```
function [forwBackVel, leftRightVel, rotVel, finish] = solution5(pts, contacts, position, orientation, varargin)

    if length(varargin) ~= 1
        error('Wrong number of additional arguments: %d\n', length(varargin));
    end

    d = varargin{1};
    goal_x = d(1);
    goal_y = d(2);


    % State Machine (FSM)
    persistent state;
    if isempty(state)
        % the initial state of the FSM is 'init'
        state = 'init';
    end

    % initialize function return variables
    forwBackVel = 0;
    leftRightVel = 0;
    rotVel = 0;
    finish = 0;

    obs_dist = 1;                % obstacle distance
    limit = 5;

    % propotional gains
    paral = 1.5;                 % parallel
    perp = 2.0;                  % perpendicular
```

```matlab
    orient = 5.5;               % orientation

    % limits
    par_limit = 2;
    perp_limit = 1;
    orient_limit = 10;

    % laser coordinates
    points = [pts(1,contacts)' pts(2,contacts)'];

    % distances calculation
    distances = (pts(1,contacts)'.^2 + pts(2,contacts)'.^2).^0.5;

    % get the closest point
    [min_value, min_index] = min(distances);

    % persistent variables used in FSM

    persistent goal_dist;
    persistent p_limit;
    persistent c;
    persistent m;               % gradient

    % manage the states of FSM
    if strcmp(state, 'init')

        % calculate line parameters
        x0 = position(1);
        y0 = position(2);

        m = (goal_y - y0)/(goal_x - x0);
        c = y0 - m * x0;

        goal_dist = sqrt((goal_x - x0) ^ 2 + (goal_y - y0) ^ 2);
        fprintf('changing FSM state to %s\n', state);
```

```

        if abs(m) < 1
            p_limit = [limit limit * m];
        else
            p_limit = [limit/m limit];
        end

        state = 'rotation';

    elseif strcmp(state, 'rotation')

        phi = orientation(3);
        goal_orient = atan2(goal_x - position(1), position(2) - goal_y);
        fprintf('changing FSM state to %s\n', state);

        if abs(phi - goal_orient) < 3 * pi/180
            state = 'move';
        end

        % change orientation to one that is needed, if needed
        error = goal_orient - phi;
        rotVel = orient * error;
        if rotVel > orient_limit
            rotVel = orient_limit;
        elseif rotVel < -orient_limit
            rotVel = -orient_limit;
        end

    elseif strcmp(state, 'move')
        fprintf('changing FSM state to %s\n', state);

        if abs(position(1) - goal_x) < 0.02 && ...
               abs(position(2) - goal_y) < 0.02
            disp(Finish!)
            finish = 1;
```

```
            return;
        end

        if min_value ^ 0.5 < obs_dist
            goal_dist = sqrt((goal_x - position(1)) ^ 2 + (goal_y - position(2)) ^ 2);
            if goal_dist >= obs_dist
                state = 'along_wall';
            end
        end

        % regulators for movement
        u = zeros(1, 2);
        goals = [goal_x goal_y];

        for i = 1: 2
            error = goals(i) - position(i);
            u(i) = 3 * error;
            if u(i) > p_limit(i)
                u(i) = p_limit(i);
            elseif u(i) < -p_limit(i)
                u(i) = -p_limit(i);
            end
        end

        % changing global velocities to local
        phi = orientation(3);
        speed_x = cos(phi) * u(1) + sin(phi) * u(2);
        speed_y = -sin(phi) * u(1) + cos(phi) * u(2);

        % setting speeds
        forwBackVel = speed_y;
        leftRightVel = speed_x;
        rotVel = 0;

    elseif strcmp(state, 'along_wall')
```

```


        % if point on line
        y_line = m * position(1) + c;
        if abs(y_line - position(2)) < 0.02
            disp(Point on line);

            % if new distance to goal less than previously saved distance
            dist = sqrt((goal_x - position(1)) ^ 2 + (goal_y - position(2)) ^ 2);
            if dist <= goal_dist
                state = 'rotation';
            end
        end
        fprintf('changing FSM state to %s\n', state);

        % calculate vector with min distance in global coordinates
        phi = orientation(3);
        vec_wall = points(min_index, :);

        % sensor coordinates to global
        x = cos(phi) * vec_wall(1) - sin(phi) * vec_wall(2);
        y = sin(phi) * vec_wall(1) + cos(phi) * vec_wall(2);
        vec_wall = [x y];

        % calculate perpendicular vector for regulator to maintain distance
        if vec_wall(1) == 0
            vec = [1 0];
        else
            vec = [-vec_wall(2)/vec_wall(1) 1];
            vec = vec/norm(vec);
        end

        speed_x = cos(phi) * vec(1) + sin(phi) * vec(2);
        if speed_x > 0
            vec = -vec;
```

```
        end

        % parallel regulator
        paral_reg = paral * vec;
        paral_reg(paral_reg > par_limit) = par_limit;
        paral_reg(paral_reg < -par_limit) = -par_limit;

        % perpendicular regulator
        perp_reg = (norm(vec_wall) - obs_dist) * vec_wall/norm(vec_wall);
        perp_reg = perp * perp_reg;
        perp_reg(perp_reg > perp_limit) = perp_limit;
        perp_reg(perp_reg < -perp_limit) = -perp_limit;

        % final global speed vector is an aggregate of the ouput of the
        % parallel and perpendicular regulators
        vec = paral_reg + perp_reg;

        % orientation regulator
        % calculation of desired orientation (phi)
        goal_orient = atan2(vec_wall(1), -vec_wall(2));

        % avoid robot oscillation when goal orientation switches between
        % positive and negative values near pi (e.g. 7pi/8 -> -7pi/8)
        if abs(goal_orient - phi) > pi
            if goal_orient < 0
                goal_orient = goal_orient + 2 * pi;
            else
                goal_orient = goal_orient - 2 * pi;
            end
        end

        % compute rotational velocity
        rot = orient * (goal_orient - phi);
        if rot > orient_limit
            rot = orient_limit;
```

```
        elseif rot < -orient_limit
            rot = -orient_limit;
        end

        % global to local velocity conversion
        speed_x = cos(phi) * vec(1) + sin(phi) * vec(2);
        speed_y = -sin(phi) * vec(1) + cos(phi) * vec(2);

        % set function returns, the run is infite
        forwBackVel = speed_y;
        leftRightVel = speed_x;
        rotVel = rot;
    end
end
```

4 Result

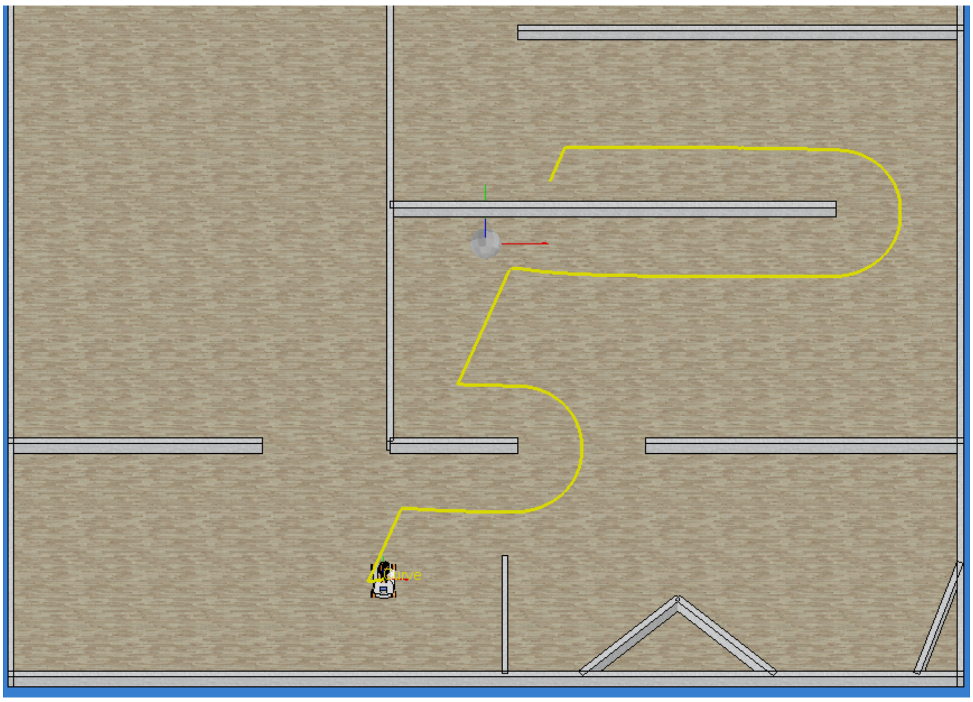

Figure 1: goal x = 1, goal y = 1