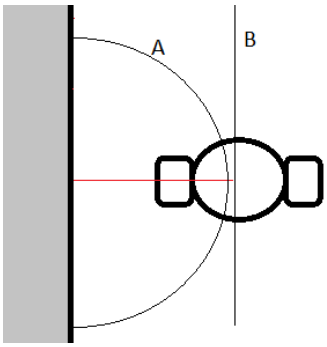
Lab 4: Obstacle avoidance, map generation and path planning for mobile robots

Exercises

Task 1

A few approaches were tested before arriving at one that actually works. A quick overview is now provided:

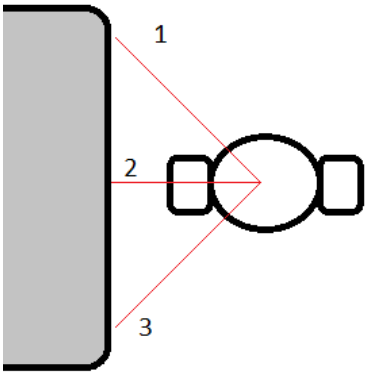
First, it was experienced that using a single lidar measurement at the left of the robot won't cut it, not even for following the wall: both path A and B achieve a constant measurement for this measurement in the picture below. In practice, the robot starts moving until it gets stuck at some point.



Then, two lidar measurements to the left of the robot were used to keep its yaw parallel to the wall. One of these would be slightly ahead of the robot and the other one slightly behind (rays 1 and 3 in the picture below). With trigonometry, the orientation of the robot can be calculated. The distance of the robot to the wall could be calculated from these measurements too, so ray2 would not be needed. The angle different between the two first measurements must be chosen under compromise:

- If they're too open, the robot will react well to interior closed turns to the right because it will see ahead of it. However, it may miss very closed exterior turns to the left because the laser doesn't touch the wall at all.
- If they're too close, the opposite happens.

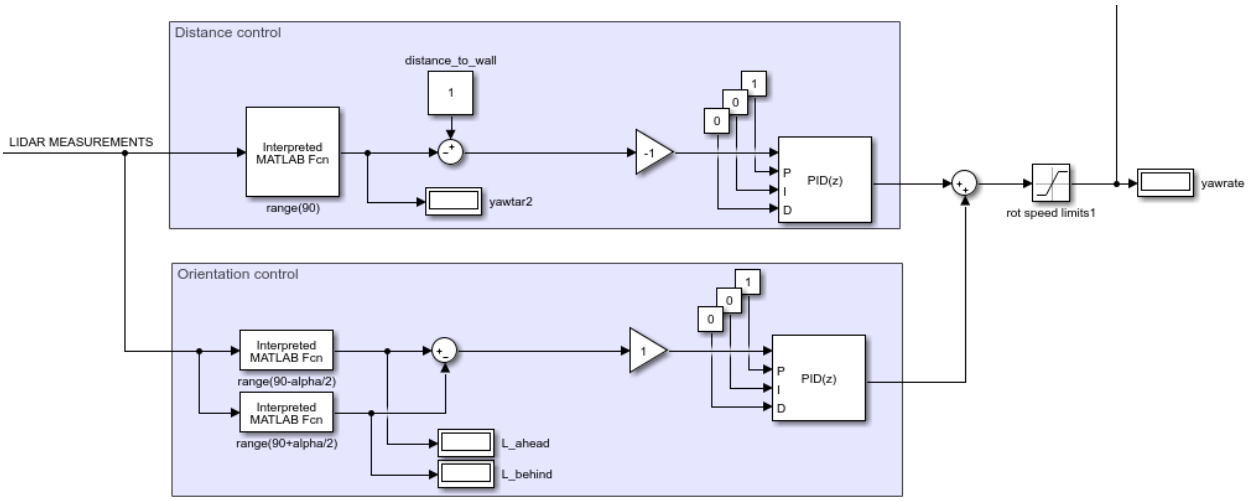
In general, this approach failed when a close turn to the right was required.



After some experimenting, a much simpler approach consisting on 3 measurements was implemented, such as shown in the previous picture. Instead of using rays 1 and 3 to **calculate** orientation, a regulation loop was placed to try and keep them **equal**. Then another control loop regulates the distance to the wall using ray 2.

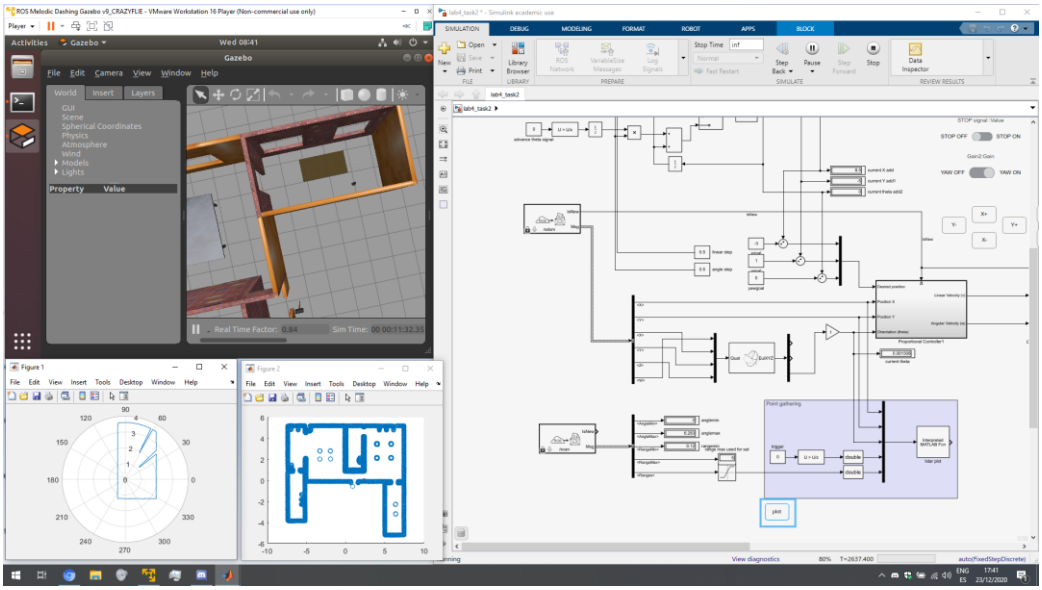
This approach is particularly effective because when the robot cannot see what is ahead (picture the robot overpassing the obstacle drawn) the range measured by the ray 1 will tend to infinite (a saturation value in practice). Then, as it is the case in normal conditions when ray1 > ray2, the robot will simply turn to the left, despite not being able to sense any wall there yet. Same happens if a perpendicular wall were to be met: ray1 range would decrease and the command would be to turn right. The angle between 1 and 3 was set to 90º to achieve a good compromise between seeing ahead & not failing to small wall segments. In the end, because measurements were available, instead of ray2 the minimum range between ray 1 and ray3 was used as control signal.

The control loop used for regulating the yaw-rate of the turtlebot looks as shown in the following picture. Speed was set to a constant during the whole run.



Task 2

For constructing the map the manual driving mode was used, since it was already implemented. To avoid catching too much data a button was added so that lidar measurements are only recorded when it is pressed. Below, a picture of how the workspace looks like while recording the map.



The points will be stored in the workspace variable ‘points’, which should be reset each time a fresh map is desired. It may be needed to define it as empty before starting the simulation.

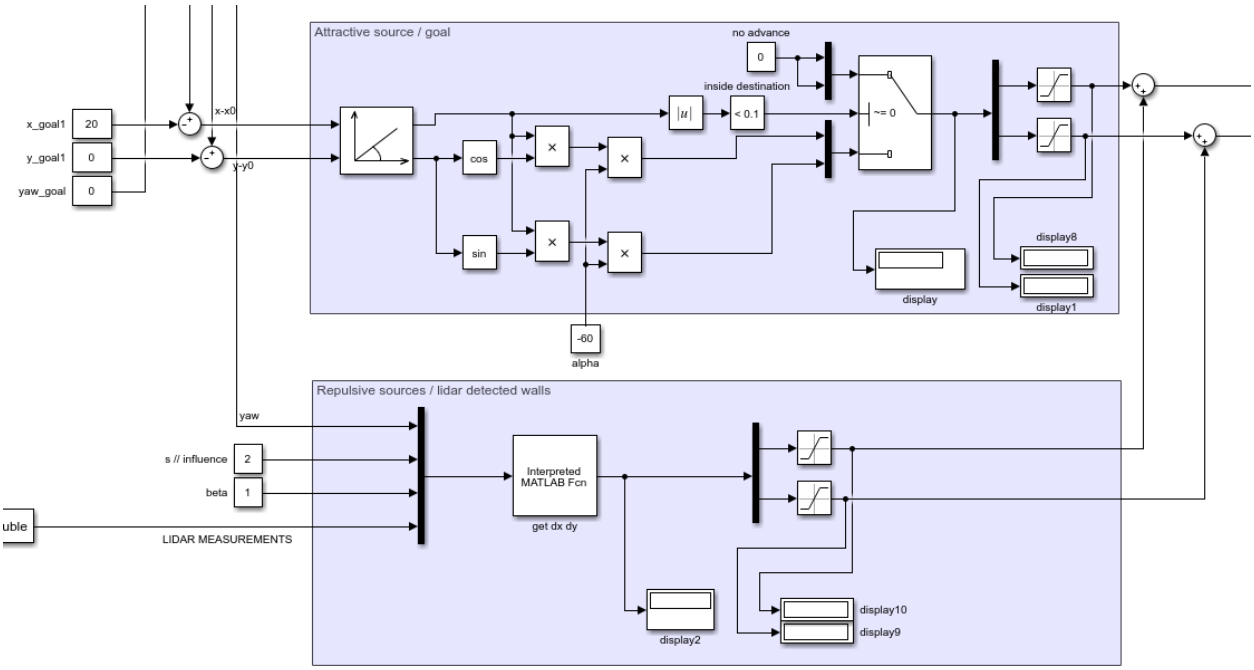
The code for recording the map and displaying the lidar is shown in the next page.

```
function returnval = gather_points(curr_x, curr_y, current_yaw, trigger, ranges)
    %%% use this to retrieve a point cloud for some world
    %%% REMEMBER TO CLEAR THE VARIABLE POINTS=[] before starting!!!
    global points
    figure(1); hold off
    ang = linspace(0, 2*pi, numel(ranges))
    polarplot(ang, ranges)

    if trigger ==1
        % Select finite range points & frame them w.r a CF square to axes
        ii_pablo = (0.2 < ranges) & (ranges < 3.4)
        ranges = ranges(ii_pablo)
        ang = ang(ii_pablo) + current_yaw

        % convert to cartesian
        [x_pablo, y_pablo] = pol2cart(ang(:), ranges(:))

        % append/create points
        points = [points; x_pablo(:)+curr_x y_pablo(:)+curr_y]

        % plot
        figure(2); hold off
        scatter(points(:,1), points(:,2))

    end

    returnval = 0
end
```

Task 3

The potential field was implemented as shown in the course's slides. The field sources trigger a change in the position of the robot. This stimulus is added to the current position of turtlebot in each iteration to obtain a new goal position. This goal position is the used as input for the controller to go from pose A to B that has been used in all other tasks.

An overview on the implementation of the potential fields is shown below.

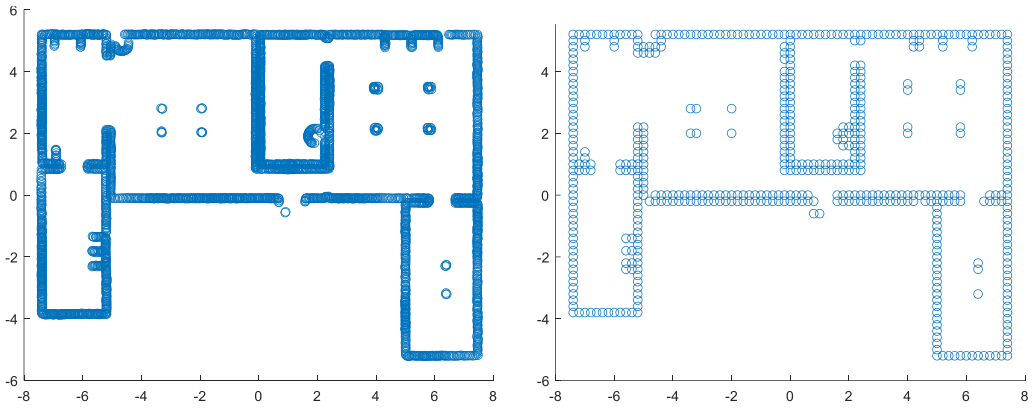


Code for computing the repulsive wall force is shown below.

```
function  yay = live_repulsion(curr_yaw, s, beta, ranges)
    ang = linspace(0, 2*pi, numel(ranges));
    clf
    polarplot(ang, ranges);

    d = ranges;
    theta = ang + curr_yaw;

    dx = - beta  * (s-d(:)) .* cos(theta(:)) .*(s>d(:));
    dy = - beta  * (s-d(:)) .* sin(theta(:)) .*(s>d(:));

    yay = [sum(dx) sum(dy)]

end
```

Task 4

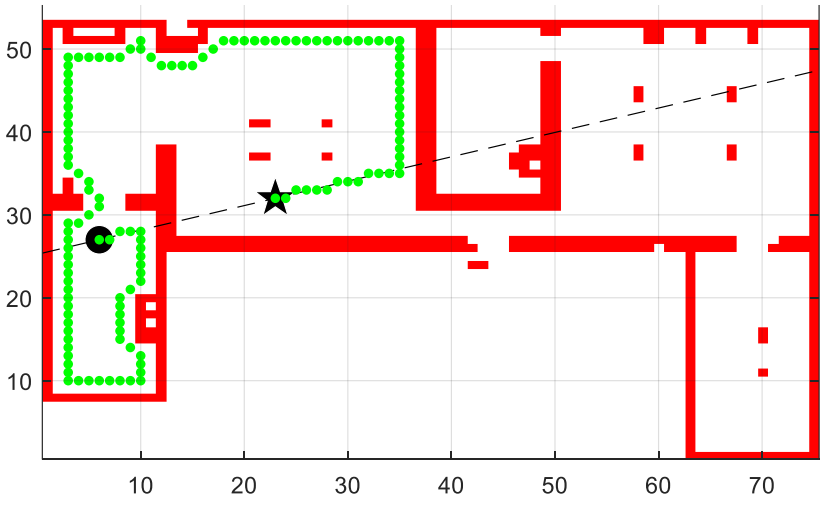
For navigating the house the algorithm Bug2 was selected. The map is first reduced to a resolution of 0.2m and then a binary mask is calculated from the XY coordinates that are available from the measurements, since Bug2 from the toolbox requires a binary mask as input map.



For the thing to work properly, it was necessary:

- To manually cover a few holes in the wall: a more regular point of clouds could have been taken. Because this was done manually (slightly irregularly) the downsampling by rounding to 0.2 left a few holes in some places.
- To cover the bottom of the shelves manually. This has to do with poor sensor selection for the deployment environment: the lidar cannot see the bottom of the shelves but can indeed bump into them and get stuck.

The algorithm is then run on the map and the waypoints are sequentially inputted into the simulink model for following trajectories. Code can be found in the next page.



```
close all
%%% RUN THE FILE 'house_map.m' TO IMPORT VARIABLE 'points'

map_p = points
figure();
scatter(points(:,1), points(:,2))

% reduce map noise & weight
map_p2 = round(map_p*5) /5
figure();
scatter(map_p2(:,1), map_p2(:,2))

% build binary grid
xmin = min(map_p2(:,1)); xmax = max(map_p2(:,1))
ymin = min(map_p2(:,2)); ymax = max(map_p2(:,2))

grid = zeros([(xmax-xmin)/0.2 (ymax-ymin)/0.2])

offset = (map_p2 - [xmin ymin]) ./ 0.2 + [1 1];
offset = cast(offset, 'uint8');

for i = 1:length(offset)
    grid(offset(i,2), offset(i,1)) = 1;
end

% heal map holes
% some holes on wall
grid(27, 14) = 1; grid(26, 13) = 1
for i = 48:52
    grid(i,37) = 1
end
% shelves
for i = 4:7
    grid(51,i) = 1
end
for i = 17:19
    grid(i,10) = 1
end


% Bug w/ animation
figure();
bug = Bug2(grid, 'inflate', 1);
bug.plot()

% WAY THERE
start = round(([-3    1] - [xmin ymin]) ./ 0.2 + [1 1]);
goal =  round(([-6.4    0] - [xmin ymin]) ./ 0.2 + [1 1])
waypoints_1 = bug.query(start, goal, 'animate'); bug.display

% WAY BACK
start = round(([-6.4    0] - [xmin ymin]) ./ 0.2 + [1 1]);
goal =  round(([-3    1] - [xmin ymin]) ./ 0.2 + [1 1])
waypoints_2 = bug.query(start, goal, 'animate'); bug.display

%%% POINTS -> 'waypoints'
waypoints = [waypoints_1(2:end); waypoints_2(2:end)]
waypoints = (waypoints - [1 1]).*0.2 + [xmin ymin]
xx = waypoints(:,1)
yy = waypoints(:,2)
tt = zeros(size(xx))
lim = length(xx)
```