Reading Instructions

Please carefully study the section 35.8 and 35.9 in the chapter on motion planning and obstacle avoidance in the Springer Handbook of Robotics [1]. Furthermore study the paper on the vector field histogram method for obstacle avoidance [2, 3].

Motion Planning and Obstacle Avoidance

Motion planning is concerned with the computation of a collision-free trajectory to the target configuration that complies with the vehicle constraints. These techniques rely upon an accurate map of the environment and solve the navigation problem in a complete and global manner, in other words, they find a collision free path to the goal if it exists.

However, the actual environment might differ from the map as objects such as furniture might be moved over time and dynamic obstacles such as humans do not appear in the map. In case the environment is dynamic precise path planning tends to fail as it does not account for dynamic objects. Path planning and trajectory optimization are covered in the following labs.

Obstacle avoidance constitutes a simpler approach to the robot navigation problem in partially unknown and dynamic environments. Its objective is to guide the robot towards a goal while avoiding collisions with obstacles detected by robot centric range sensors, such as a laser. Such a reactive obstacle avoidance utilizes range sensor information as feedback for motion control and adapts the original plan in lieu of local corrections to the map.

Many architectures including the ROS navigation stack combine the global path planning with a local obstacle avoidance scheme. The global path provides intermediate way-points that constitute temporal targets for the local sensor based obstacle avoidance.

The obstacle avoidance problem consists of computing a robot motion that avoids a collision with the local obstacles detected by the range sensors as shown in figure 1. At the same time, the control is supposed to bring the robot closer to the goal pose. This type of feedback control generates a sequence of motions that steer the robot towards the target on a collision-free path. Let $\mathbf{q}_{target}$ denote a target pose and $\mathbf{q}_{t_i}$ the current pose. In the case of a mobile robot moving in a plane, the pose is defined by the vector $\mathbf{q}_{t_i} = (x_{t_i}, y_{t_i}, \theta_{t_i})$. At time t_i the robot A is at q_{t_i} and perceives the local obstacles $O(\mathbf{q}_{t_i})$ by means of a range reading $S(\mathbf{q}_{t_i})$. The objective is to compute a motion command u_i such that:

- the generated trajectory from pose $\mathbf{q}_{t_i}$ towards the new pose $\mathbf{q}_{t_{i+1}}$ is collision free
- and the new pose is closer to the target $F(\mathbf{q}_{t_i}, \mathbf{q}_{target}) < F(\mathbf{q}_{t_i+T}, \mathbf{q}_{target})$, in which the scalar function F measures the progress of the current pose towards a target pose.

The result is a sequence of motion controls $\{u_1, \ldots, u_n\}$ computed online that avoids the obstacles while each intermediate configuration $\{\mathbf{q}_{t_1}, \ldots, \mathbf{q}_{target}\}$ brings the robot closer to the target. Obstacle avoidance methods are inherently local and iterative and therefore provide no guarantee to eventually reach the target even if a feasible path exists. The drawback of the risk that the robot gets trapped due to the lack of global planning is compensated by the advantage of incorporating sensor information into the motion control problem.

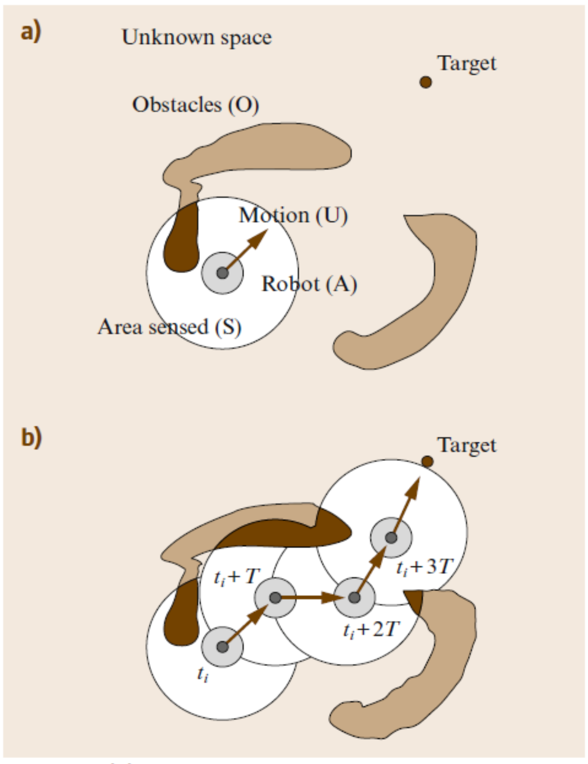


Figure 1: Obstacle avoidance problem [1]

Vector Field Histogram Method

The methods of subset of controls first determine a candidate set of feasible motion controls. The optimal control is selected from the candidate set according to some optimality criterion, usually related to the target heading. There are two types of methods

- Methods that compute a subset of motion directions in terms of desired robot heading. Examples of these methods are the vector field histogram [2] and the obstacle restriction method [4]. The desired heading is then mapped onto a motion command.

[1]source: Springer Handbook of Robotics, Motion Planning and Obstacle Avoidance [1]

- Methods that directly compute a subset of velocity controls such as the dynamic window approach [6] and trajectory roll out [5].

Vector field histogram first computes a set of obstacle-free candidate directions and then selects the direction closest to the target heading. It calculates the steering direction based on the laser scan data and then converts the steering direction to an appropriate linear and an angular velocity (v, ω). If there is no feasible steering direction, the robot stops and scans for a free direction by rotating in place.

First, the local environment is partitioned into equidistant angular sectors w.r.t the robot base frame. Each sector corresponds to a possible direction of robot steering. A polar histogram is constructed in which each bin represents the obstacle polar density in the corresponding sector as shown in figure 2. For that purpose, those laser scan readings r_i, θ_i that belong to the k-th sector $\Omega_k = \{\theta_i \in [\theta_k, \theta_{k+1}]\}$ are weighted by the inverse distance to the robot center r_i and are accumulated according to

$$h_k = \sum_{\{i\,:\,\theta_i \in \Omega_k,\ r_i \in [0, r_{max}]\}} (1 - \frac{r_i}{r_{max}})^{\alpha} \tag{1}$$

The obstacle density h_k counts obstacles in the segment and weights that count by the factor $(1 - \frac{r_i}{r_{max}})^{\alpha}$ that increases as the distance to the obstacle r_i decreases. The rate of decrease is determined by the cut off distance r_{max} and the exponential α.

Due to the discrete nature of the histogram grid, the mapping from range readings to polar histogram may appear ragged and cause errors in the selection of the steering direction. Therefore, the histogram is smoothed with a spatial low pass filter according to

$$h'_k = \frac{1}{2l+1} \sum_{i=-l}^{l} (l - |i| + 1) h_{k+i} \tag{2}$$

In general a window size of $l = 3$ (filter width $2l + 1 = 7$) yields good results.

The resulting histogram has peaks (directions with a high density of nearby obstacles) and valleys (directions with a low density of remote obstacles). The set of candidate directions denoted by the term candidate valley is determined as the set of adjacent sectors S which histogram components are below a threshold.

$$S = \{k : h_k \leq h_{max}\} \tag{3}$$

and which are closest to the component k_{target} that contains the target direction. Determine a value for the threshold h_{max} by either plotting h and looking at the figures or calculate it by setting r_i to a minimal radius and calculate h as shown in equation 1

The optimal direction is selected from the candidate valley according to the goal sector and the following heuristics. Four distinctive cases are investigated in sequence and the respective solution sector k_{sol} is chosen:

[2]source: Springer Handbook of Robotics, Motion Planning and Obstacle Avoidance [1]

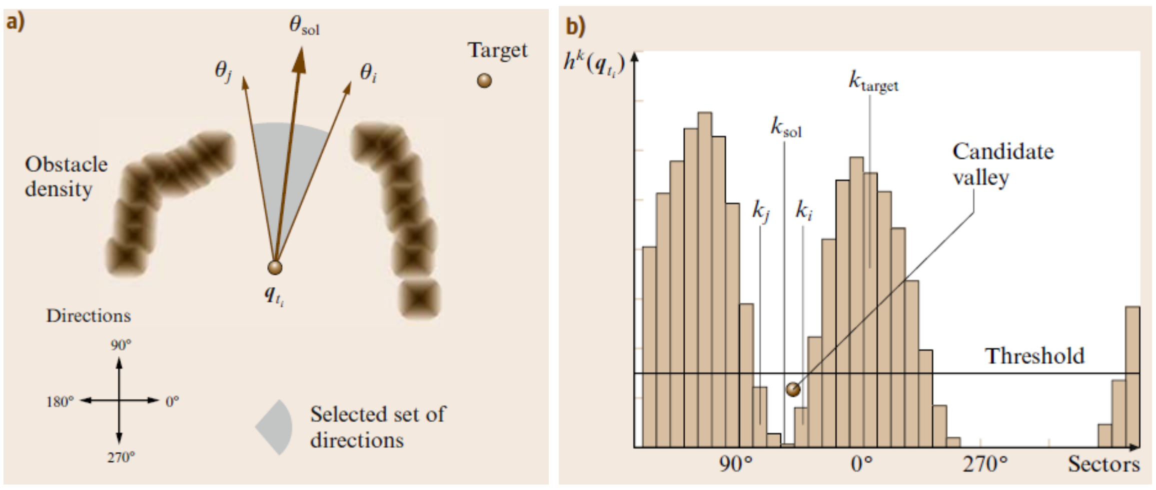


Figure 2: obstacle distribution (left), candidate valley of feasible directions [2]

Case 1: The goal sector k_{target} lies within the selected valley. Solution: $k_{sol} = k_{target}$, where k_{target} is the sector that contains the goal location.

Case 2: The goal sector is not in the selected valley and the number of sectors that form the candidate valley is greater than m. Hence, the candidate valley is non-narrow. Solution: $k_{sol} = k_i \pm m/2$, where m is a fixed number of sectors and k_i the sector of the valley closer to the target sector k_{target}.

Case 3: The goal sector is not in the selected valley and the number of sectors of the valley is lower or equal to m. Hence, the candidate valley is narrow. Solution: $k_{sol} = \frac{k_i+k_j}{2}$ is the bisector of k_i and k_j which denote the extreme sectors of the valley. This is the case depicted in figure 2

Case 4: The candidate valley $S = \emptyset$ is empty. The robot rotates in place and scans its environment until an obstacle free direction is perceived.

In the following tasks, you will implement a function `vfh` that first calculates the vector field histogram from scan data and then determines the appropriate steering direction. At first test your function on static data and later implement a reactive obstacle avoidance behavior based on the steering direction computed with the Vector Field Histogram (VFH) algorithm.

Obstacle Avoidance with Basic Vector Field Histogram Method

1) The static range data for testing your `vfh` function is provided in the uploaded files `rangedata_caseX.mat` in Moodle, where the X represents the case to test. Each

[4]source: RST

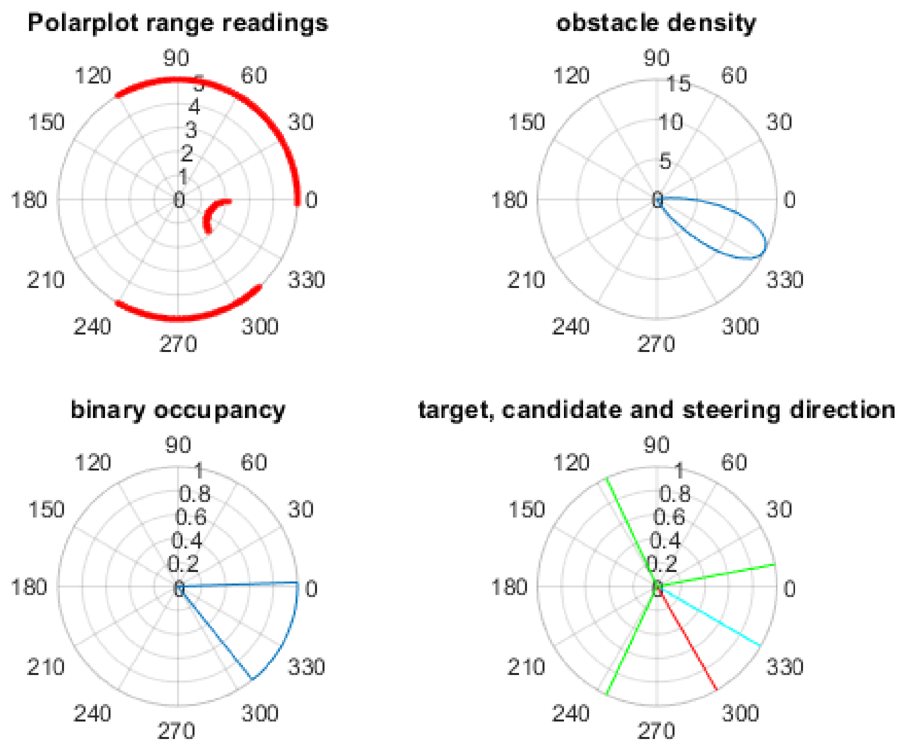


Figure 3: polar plots of range data (upper left), polar obstacle density (upper right), occupied sectors (lower left) and candidate directions (blue and green), target direction (cyan) and steering direction (red).[4]

data file contains an array of 683 range readings from a laser range sensor with a 240° field of view in the vectors ranges and angles as well as a target direction in `targetdir`. Load each of the four `rangedata_caseX.mat` files, create new figures and call the `vfh` function that you will implement in the next few tasks with each of the datasets. The goal is to create a figure with four polarplots similar to the example plot in figure 3 for each of the four cases.

2) Design a Matlab function

```
function steeringdirection = vfh(ranges, angles, targetdir)
```

that determines the steering direction of the robot according to the vector field histogram method. The input parameters `ranges` and `angles` denote the laser range data extracted from the `/scan` message. The input parameter `targetdir` denotes the heading towards the target in robocentric coordinates obtained from transforming the navigation goal message from the map frame to the base link frame. The output parameter `steeringdirection` is the obstacle-free direction $\theta_{k_{sol}}$ in robocentric coordinates based on the vector field histogram algorithm. Angles are specified in radians w.r.t. the robots current heading.

Code and implement the function in a step by step manner following the instructions in the next tasks. At first the function merely plots the range data with `polarplot` as illustrated in the upper left figure 3. The output argument is set to

```
steeringdirection = NaN;
```

to indicate that no feasible steering direction is computed yet.

3) Convert the range data into a polar obstacle density histogram according to equation (1) from the laser range reading. Define internal parameters for `numsectors = 90`, `rmax = 3`, `alpha = 1.5` and `narrow = 0.52`. Plot the histogram in a separate subfigure with `polarplot`.

   - `numsectors`: Number of angular sectors in histogram. This parameter defines the number of bins in the histogram. 64 bins is a reasonable partition.
   - `alpha`: This corresponds to the parameter α in the exponential in equation (1). Reasonable numbers are $\alpha \in [1.0, 2.0]$.
   - `rmax`: This corresponds to the parameter r_{max} in equation (1) and denotes the maximal view range of the sensor. Reasonable numbers are $r_{max} \in [0.5, 5.0]$.
   - `narrow`: Threshold in radians to distinguish between a non-narrow candidate valley (case 2) and a narrow candidate valley (case 3). 0.52 corresponds to 30 degrees.
   - `hmax`: Sectors with an obstacleDensity exceeding this threshold will be considered as occupied.

   The code for histogram calculation is mostly given in the assignment. Step through the following code and try to understand the steps. You might want to try different parameters.

   For histogram calculation proceed as follows:

   - determine the `sectorincrement` as the angular separation of sectors and generate the array of `sectormidpoints` with `linspace` at equally spaced angular intervals and the `sector_edges` as the lower und upper bounds of the sector intervals.

```
ranges = max(0,ranges);
minangle = min(angles);
maxangle = max(angles);
sectorincrement = (maxangle-minangle)/numsectors;
sectormidpoints = linspace(minangle, maxangle, numsectors);
sector_edges = [sectormidpoints - sectorincrement/2,
                sectormidpoints(end) + sectorincrement/2];
```

   - filter the range data by selecting only the range data with $r_i < r_{max}$ using `find`.

```
valididx = find(ranges<rmax);
validranges = ranges(valididx);
validangles = angles(valididx);
```

- compute the vector of `weightedranges` from the `validranges`, `rmax` and `alpha` according to

$$h_k = \sum_{\{i \,:\, \theta_i \in \Omega_k,\ r_i \in [0, r_{max}]\}} (1 - \frac{r_i}{r_{max}})^{\alpha}. \tag{4}$$

```
weightedRanges = (1.0- validranges/rmax).^alpha;
```

- determine the association of `angles` to the sectors (`sector_edges`) with the function `histcounts`. The function

```
[N,edges,bin] = histcounts(X,edges)
```

sorts the data `X` into bins with the bin edges specified by the `edges`. The output argument `bin` is an array of the same size as `X` whose elements are the bin indices for the corresponding elements in `X`. In other words `bin(i)` denotes the index of the bin to which element `X(i)` belongs.

```
[~,edges,bin] = histcounts(validangles, sector_edges);
```

- accumulate the vector of `weightedranges` in the polar density histogram `obstacledensity` of length `numsectors` according to (1) based on the array of bin indices `bin`.

```
obstacleDensity = zeros(1, numsectors);
for i=1:length(bin)
  obstacleDensity(1, bin(i)) = ...
    obstacleDensity(1, bin(i)) + weightedRanges(i);
end
```

- Plot the polar obstacle density in a `polarplot` as shown in figure 3 for the sample range data `ranges` and `angles`.

4) Smooth the polar obstacle density histogram `obstacledensity` with the Matlab function `filtfilt` according to equation (2) with a filter window size $l = 2$ and filter width of $2l + 1 = 5$. For a window size of $l = 2$ the filter coefficients become $b = \frac{1}{7}[1, 2, 3, 2, 1]$ and $a = 1$.

```
y = filtfilt(b,a,x)
```

filters the input data `x` using a rational transfer function defined by the numerator and denominator coefficients `b` and `a`.

Return the first element of the obstacleDensity vector as a second output variable `hc` of the `vfh` function. `hc` corresponds to the `obstacleDensity` in driving direction and will be used later for computing the commanded velocity.

Plot the smoothed `obstacledensity` histogram.

5) Determine the logical array of occupied sectors `occupiedsectors` (non candidate valleys) with a polar density above the threshold $h > h_{max}$ according to equation (3). Plot the occupied sectors for the sample range data `ranges` and `angles` in a separate subfigure with `polarplot` as shown in figure 3.

```
sectoroccupied = zeros(1,numsectors);  % boolean
sectoroccupied(obstacleDensity > hmax) = true;
```

6) Determine the upper and lower indices of the nonoccupied sectors `~occupiedsector` with the function `diff` that detects transitions from occupied to nonoccupied sector.

```
changes = diff([0 ~sectoroccupied 0]);
foundSectors = find(changes);
sectors = reshape(foundSectors, 2, []);
sectors(2,1:end) = sectors(2,1:end) - ones(1, size(sectors, 2));
```

Afterward, `sectors(1,:)` contains the lower indices and `sectors(2,:)` the upper indices of free sectors.

7) Determine target direction according to case 1:

Check whether the target direction `targetdir` is within the boundaries of any of the non occupied sectors (`~sectoroccupied`) with intervals computed according to the `sectormidpoints` and the upper and lower indices in `sectors`. In that case return `targetdir` as `steeringdirection`. Test your function for the sample range data `rangedata_case1.mat`. The target direction given in the data is in a free sector such that the resulting steering direction should be the target direction. As an additional test you can vary the input target direction to an occupied direction and test if your function returns `NaN`.

8) Determine target direction according to case 2:

Check which of the valleys are narrow sectors by estimating the valley width utilizing sector indices (`sectors`) and `sectormidpoints` and compare the width to the `narrow` parameter. For the non-narrow valleys generate one candidate direction at each boundary of the valley minus half of the narrow-threshold `narrow` to avoid that the `steeringdirection` is too close to a valley boundary. In the end select the candidate direction which is next to the target direction as final steering direction. Test your implementation with the sample range data `rangedata_case2.mat`. The result should consist of two non-narrow valleys, where the target direction is in an occupied sector. Hence four candidate directions should be estimated where the one next to the target direction is the output one.

9) Determine target direction according to case 3:

For all valleys which are narrow only one candidate direction is generated. Each narrow candidate valley proposes a single candidate direction according to the `sectormidpoints` of the central sector of that valley. Again the final steering direction is the candidate direction that is closest to `targetdir`. Plot the candidate directions, target direction and steering direction as shown in figure 3 for the sample range data `ranges` and `angles` and for `targetdir` inside or outside free sectors. Test your implementation with the sample range data `rangedata_case3.mat`. The result should consist of two non-narrow valleys on the most right and most left side and one narrow valley slightly right of the robot. In total five candidate directions should be generated in this scenario: Two for each non-narrow valley and one for the narrow valley. The final steering direction should be the one in the center of the narrow valley.

10) Handle case 4:

If the set of candidate directions is empty, the `vfh` function returns the output parameter `steeringdirection = NaN;` To test this utilize `rangedata_case4.mat` where the robot is placed in a corner.

Reactive Obstacle Avoidance with the VFH Algorithm

In the following we implement a reactive obstacle avoidance behavior based on the vector field histogram and determine an appropriate turn rate ω that turns towards the desired steering direction and a safe linear velocity.

Technical Prerequisites

For this lab a custom launch file is created. Since the permissions only allow editing files in the user workspace, clone the turtlebot3 repositories to your workspace if the package is not present yet (under `~/catkin_ws/src/`):

```
cd ~/catkin_ws/src
git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
```

This lab is concerned with obstacle avoidance. The assignments for reactive obstacle avoidance simulate the robot with the house map (cf. figure 4). Start the gazebo simulation with the house map:

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

For using the *amcl* localization, a map of the environment is necessary. The map of the house environment is uploaded in Moodle. Download the map files (.yaml and .pgf), place them in `/catkin_ws/src/turtlebot3/turtlebot3_navigation/maps/` and start the *amcl* Localization and *rviz*:

```
roslaunch turtlebot3_navigation turtlebot3_amcl.launch map_file:=/home/<YOUR_USER>/
  catkin_ws/src/turtlebot3/turtlebot3_navigation/maps/house_map.yaml
```

In case the robot gets stuck because it crashes into the environment, you can usually free the robot by moving the robot to a free location in the vicinity with the Translation Mode (shortcut T) in Gazebo. You need to restart the simulation, if you observe a wrong localization of the robot in rviz.

The navigation stack plans a new global and local path in response to a new goal pose. In order to block the navigation stack from controlling the robot, the `move_base` package must be deactivated for the course of this lab. Copy the `turtlebot3_navigation.launch` file (usually located in `~/catkin_ws/src/turtlebot3/turtlebot3_navigation/launch`) to a new file `turtlebot3_amcl.launch` and comment or remove the `move_base` section in the launch file.

The `amcl` node is required for the robot to localize himself in the map which is published by the `map_server`. The `turtlebot3_amcl.launch` file is also provided in the moodle workspace and should be placed in the previously named folder for launch files in the `turtlebot3_navigation` package. Start the `amcl`, `map_server`, `robot_state_publisher` and `rviz` nodes with the launch file:

```
roslaunch turtlebot3_navigation turtlebot3_amcl.launch
```

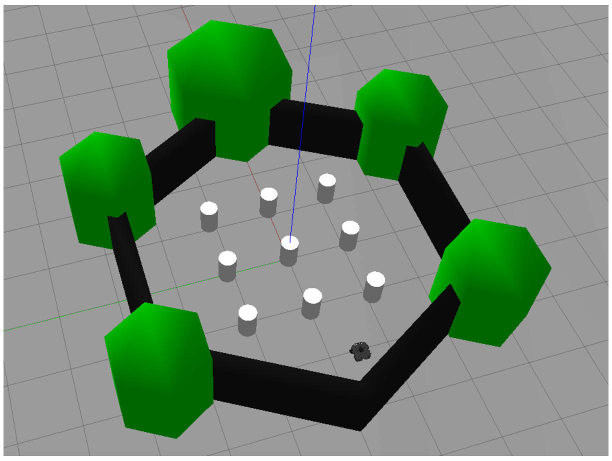


Figure 4: Obstacle map

Set the `2D Pose Estimate` in rviz to a location near the actual location visible in gazebo. The target pose is specified in rviz with the `2D Nav Goal` button and is published on the topic `/move_base_simple/goal`. Unlike in some prior tasks, the robot will not move when sending the goal due to the deactivated `move_base` node. In this lab, you will use the Vector Field Histogram algorithm to guide the robot towards the goal pose. Notice that the goal pose is published w.r.t. the `map` frame. Hence, you will use a transform to map the target pose into robot centric coordinates w.r.t. frame `base_link`.

In cases 1-3 the robot moves at a constant forward velocity $v = v_0$ unless the presence of obstacles demands safe motion at a reduced linear velocity. The regulation of the linear velocity follows the same reasoning as the one for the primitive obstacle avoidance behavior in the lab Robotics Introduction Toolbox II.

Let h'_c denote the smoothed polar obstacle density in the current direction of travel. Large values of h'_c indicate that either a large obstacle lies ahead or a smaller obstacle is close to the robot. Either case demands a sharp turn of the robot. This is achieved by a reduction of the linear velocity in order to allow completion of the turn towards the novel direction. The linear velocity is determined according to

$$v = v_0 \max(0, (1 - h'_c/h_m)) \tag{5}$$

and h_m is an empirical parameter that achieves the desired reduction in speed.

The above control law reduces the linear velocity of the robot in anticipation of an steering maneuver. One might further consider that steering directions that deviate from the current robots heading require an ongoing turning. Thus, the linear velocity v is reduced to v_{red} in proportion to the turn rate

$$v_{red} = v(1 - |\omega|/\omega_{max}) + v_{min} \tag{6}$$

in which v denotes the linear velocity according to (5), ω_{max} denotes the maximum admissible turn rate and v_{min} is a small non-zero velocity offset.

The steering direction $_{k_{sol}}$ is mapped onto the turn rate

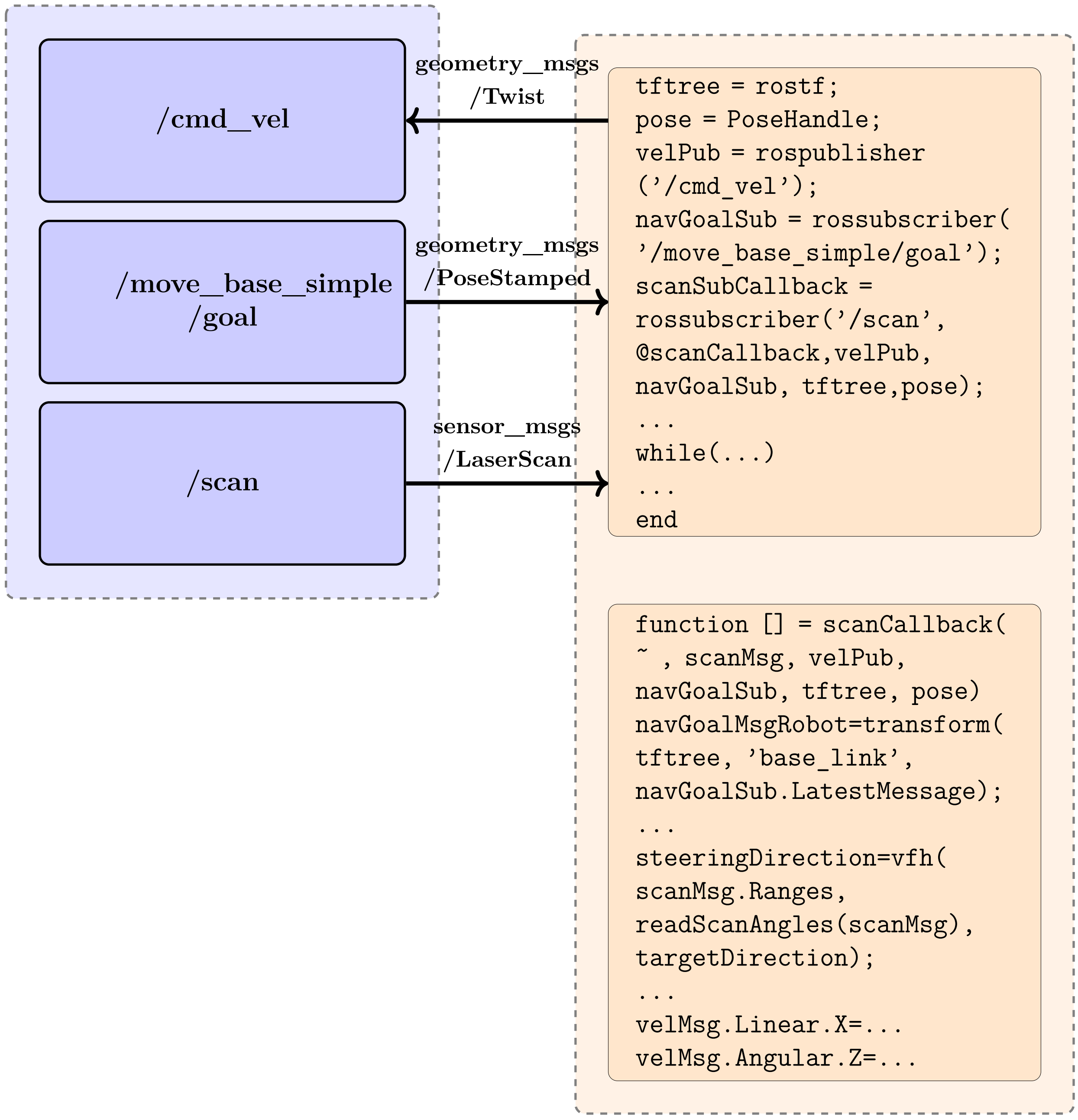
$$\omega = \text{sat}(k_\omega \theta_{k_{sol}}) \tag{7}$$

in which $\theta_{k_{sol}}$ denotes the direction of the bisector of the solution sector k_{sol} in the robocentric frame. In case 4 the robot stops and rotates in place $v = 0$, $\omega = \omega_{max}$.

Design an obstacle avoidance scheme that implements the basic vector field histogram methods in terms of a mapping between a laser range sensor (topic `/scan`). The Turtlebot3 features the 360° laser distance sensor LDS-01. A sensor with a broad field of view has the advantage that oscillatory behaviours during wall following are avoided.

The goal pose in robocentric coordinates is obtained by mapping the navigation goal pose (topic `/move_base_simple/goal`) into the robocentric frame `base_link` using the appropriate transform. The motion commands are published on the topic `/cmd_vel`.

For the regulation of the linear velocity reutilize the Matlab code for the simple reactive obstacle avoidance scheme from the assignment in the lab on Robotics Toolbox Introduction II. Figure 5 illustrates the publisher, subscriber architecture with a callback subscriber on the topic `/scan`. The callback function determines the steering direction from the scan message by means of `vfh`. The steering direction is mapped onto the corresponding velocity command to be published on the topic `/cmd_vel`. The callback function obtains a reference to a `PoseHandle` object by which it reports the navigation goal pose in the robocentric frame back to the main program. The main loop governed by a rate object merely monitors progress towards the goal and terminates once the navigation goal is reached.



```
tftree = rostf;
pose = PoseHandle;
velPub = rospublisher
('/cmd_vel');
navGoalSub = rossubscriber(
'/move_base_simple/goal');
scanSubCallback =
rossubscriber('/scan',
@scanCallback,velPub,
navGoalSub, tftree,pose);
...
while(...)
...
end
```

```
function [] = scanCallback(
~ , scanMsg, velPub,
navGoalSub, tftree, pose)
navGoalMsgRobot=transform(
tftree, 'base_link',
navGoalSub.LatestMessage);
...
steeringDirection=vfh(
scanMsg.Ranges,
readScanAngles(scanMsg),
targetDirection);
...
velMsg.Linear.X=...
velMsg.Angular.Z=...
```

Figure 5: Publisher subscriber structure [5]

11) Set the ROS_IP and ROS_MASTER_URI variables both in Matlab and Ubuntu for a more reliable transmission of messages.

```
% MATLAB:
setenv('ROS_IP','<YOUR_MATLAB_IP>')
setenv('ROS_MASTER_URI','http://<YOUR_UBUNTU_IP>:11311')
```

[5]source: RST

```
% Ubuntu:
export ROS_IP=<YOUR_UBUNTU_IP>
export ROS_MASTER_URI=http://<YOUR_UBUNTU_IP>:11311
```

Initialize ROS in Matlab, initialize the ROS transformation tree to later access the transform from `map` to `base_link` frame. Initialize a publisher `velPub` for motion commands and a subscriber `navGoalSub` to monitor the navigation goal in RViz.

```
rosinit('<YOUR_UBUNTU_IP>');
tftree = rostf;
pause(1.0);

velPub = rospublisher('/cmd_vel');
navGoalSub = rossubscriber('/move_base_simple/goal');
disp('select navgoal in RVIZ to start obstacle avoidance');
% Matlab will wait for the receive of a goal message
navGoalMsg = receive(navGoalSub);
```

12) Reutilize the class `PoseHandle` in `PoseHandle.m` from the assignment Robotics System Toolbox I. The class is also uploaded to Moodle. The pose handle object has the properties x, y, and θ and should be instantiated as `goalPose`.

```
% PoseHandle.m
classdef PoseHandle < handle
% Handle to store the robots position
  properties
    x = 0;
    y = 0;
    theta = 0;
  end

  methods
  end
end
```

Instantiate a Subscriber for the `\scan` Topic that triggers a function called `scanCallback` whenever a message is received. Use the following variables as arguments for the `scanCallback` function:

- `velPub` (for publishing computed `vfh` velocity commands)
- `navGoalSub` (for getting the latest `navGoal` messages)
- `tftree` (for transforming the latest `navGoal` messages to the `base_link` frame)
- `goalPose` to the callback function (for storing the `goalPose` in robocentric coordinates for plotting in the main loop later)

13) Implement a callback function `scanCallback` that is called whenever a message on the `/scan` topic is received. The callback function determines the steering direction with help of the `vfh` algorithm. Steering direction and obstacle distance are mapped onto the corresponding motion command in terms of velocity and turn rate.

Define maximum and minimum ranges for the turnrate ω, linear velocity v, a goal radius r_{goal}, and a controller gain k_ω for ω. Furthermore instantiate a blank velocity message.

```
function [] = scanCallback( ~, scanMsg, velPub, navGoalSub, tftree, pose)
  komega = 2.0;  % gain for turn rate
  omegamin = -1.5;
  omegamax = 1.5;
  vsafe = 0.5;
  rgoal = 0.5;
```

14) Get the latest `navGoalMsg` and map it into the robocentric frame `base_link` with help of the corresponding transform. Extract the target pose $x^{(r)}, y^{(r)}, \theta^{(r)}$ in robocentric with the helper function `poseStampedMsg2Pose.m` (given in template file) and assign it to the pose handle object `pose`. Convert the Cartesian target location $x^{(r)}, y^{(r)}$ into polar coordinates `targetDirection` and `rho` with `cart2pol`:

```
  navGoalMsgRobot = transform(tftree,'base_link', navGoalSub.LatestMessage);

  [goalPose.x, goalPose.y, goalPose.theta] = PoseStampedMsg2Pose(
    navGoalMsgRobot);
  [targetDirection, rho] = cart2pol(goalPose.x, goalPose.y);
```

15) Determine the robots `steeringdirection` for the current `targetDirection` and range data in `scanMsg` with help of the previously implemented function `vfh`.

16) The turn rate `omega` is determined from the designated steering direction $\theta_{k_{sol}}$ according to

$$\omega = \text{sat}(k_\omega \theta_{k_{sol}}) \tag{8}$$

in which sat is the saturation function that limits the turnrate to the interval $\omega \in [-\omega_{max}, \omega_{max}]$

17) To safely avoid collisions, the translational velocity is reduced with the proximity of obstacles.

Calculate the speed `v` according to equations 5 and 6.

The robot can be stopped when approaching the goal pose, if the distance to the goal pose ρ is smaller than a predefined parameter r_{goal}:

$$v_{cmd} = v_{red}\frac{\rho}{r_{goal}} \qquad \forall \rho < r_{goal} \tag{9}$$

18) Instantiate an empty velMsg from the velPub, compose and publish the `velMsg` from linear velocity `v` and turn rate `omega`

```
velMsg = rosmessage(velPub);
velMsg.Linear.X = v;
velMsg.Angular.Z = omega;
send(velPub,velMsg);
```

19) Test the VFH obstacle avoidance in the house environment and plot the path of the robot in a rated loop.

```
    roslaunch turtlebot3_gazebo turtlebot3_house.launch

```

What are the limitations of VFH in terms of global navigation?

Tune the parameters α, r_{max}, h_{max} of the VFH algorithm such that the robot safely traverses the gaps between the obstacles. The plots generated within the `vfh` function are a handy tool for tuning the parameters. For tuning, you might want to stop sending the `/cmd_vel` message and place the robot at different locations with the Translation Mode (Shortcut T) in Gazebo.

Vector Field Histogram Method+

The VFH+ method is an improved version of the orginal vector field histogram method [3]. The basic structure of the VFH+ method mimics the original implementation and is based on a polar histogram of obstacle density. The computation of the primary polar histogram according to equation 1 remains the same. The original VFH method does not explicitly take into account the width of the robot. Instead, it uses an empirically determined low-pass filter to compensate for the robot width and to smooth the polar histogram.

The VFH+ method uses a theoretically determined low-pass filter to compensate for the width of the robot. Obstacle cells in the map are enlarged by the robot radius r_r, which is defined as the distance from the robot center to its furthest perimeter point. For further safety, the obstacle cells are actually enlarged by a radius $r_{r+s} = r_r + d_s$ where d_s is the minimum distance between the robot and an obstacle. This width compensation method is implemented very efficiently by enlarging the obstacles while building the primary polar histogram. Instead of updating only one histogram sector for each cell as done in the original VFH method, all histogram sectors that correspond to the enlarged cell are updated.

For each range reading r_i, the enlargement angle γ_i is defined by:

$$\gamma_i = \arcsin \frac{r_{r+s}}{r_i} \tag{10}$$

For each sector, the polar obstacle density is then calculated by:

$$h_k = \sum_{\{i:\theta_i \in \Omega_k^*, r_i \in [0, r_{max}]\}} (1 - \frac{r_i}{r_{max}})^\alpha \tag{11}$$

with the sector $\Omega_k^* = \{\theta_i \in [\theta_k - \gamma_i, \theta_{k+1} + \gamma_i]\}$ boundaries enlarged by γ_i. The result of this process is a polar histogram that takes into account the width of the robot.

A smooth trajectory is achieved by avoiding oscillations in the commanded turn rate. In most scenarios the original VFH method does not exhibit oscillations. However, the fixed threshold of the VFH method might cause oscillations in the case of narrow passages, as the resulting histogram switches between a blocked and open segment in the direction of the gap. This causes the commanded turn rate to alternate with no definite decision whether to traverse or circumnavigate the opening.

The VFH+ method is already implemented in the Matlab Navigation toolbox. The vector field histogram (VFH) class enables the robot to avoid obstacles based on range sensor data. Given a range sensor reading in terms of ranges and angles, and a target direction in the robocentric frame, the VFH+ method determines an obstacle-free steering direction. The algorithm takes the ranges and angles as a lidarScan object from range sensor data and builds a polar histogram for obstacle locations. Then, it uses the input histogram thresholds to calculate a binary histogram that indicates occupied and free directions. Finally, the algorithm computes a masked histogram, which is computed from the binary histogram based on the minimum turning radius of the robot.

The algorithm selects multiple steering directions based on the open space and possible driving directions. A cost function, with weights corresponding to the previous, current, and target directions, calculates the cost of different possible directions. The algorithm then returns an obstacle-free direction with minimal cost.

Obstacle Avoidance with Advanced Vector Field Histogram Method

20) (**Optional**) The Matlab Navigation toolbox provides an implementation of the advanced vector field histogram method in an object `controllerVFH`. Instantiate a vector field histogram object `vhfplus` and enable the direct use of lidarScan objects with

```
vfhplus = controllerVFH;
vfhplus.UseLidarScan = true;
```

A lidarScan object can be directly constructed from incoming `/scan` messages:

```
scan = lidarScan(scanMsg)
```

The vfhplus object is associated with a function by the same name

```
vfhplus(scan, targetDirection);
```

that determines the steering direction. It has a similar signature as the function `vfh` that you implemented. Thus, you have to substitute the `vfh` function and the `ranges` and `angles` arguments by `vfhplus` and the `scan` object in your code. The `controllerVFH` object has similar parameters and properties as the basic vector field histogram approach.

- `NumAngularSectors` number of angular sectors in the histogram.
- `HistogramThresholds` thresholds for binary histogram computation specified as a two element vector. The two thresholds to determine the mapping from polar obstacle density to binary histogram. Polar obstacle densities higher than the upper threshold are represented as occupied space in the binary histogram. Densities smaller than the lower threshold are represented as free space. Densities that fall between the limits are set to the values in the previous binary histogram.

The VFH+ object has the following additional VHF+ specific parameters

- `DistanceLimits` upper and lower bounds for laser range readings. The range readings specified in equation 1 are considered only if they fall within the distance limits and ignored otherwise.
- `RobotRadius` : radius of the robot in meters
- `SafetyDistance` : safety distance around the robot. This is a safety distance to leave around the robot position in addition to `RobotRadius`. The robot radius and safety distance determine the obstacle-free direction in terms of r_{max} in equation 1.
- `MinTurningRadius` : minimum radius the robots drives as a curve

Inside your callback function `scanCallback`, instantiate an object `vhfplus` and setup its properties. Replace the call to your `vfh` function by `vfhplus`

```
vfhplus = controllerVFH;
vfhplus.DistanceLimits = [0.05 1];
vfhplus.RobotRadius = 0.2;
vfhplus.MinTurningRadius = 0.2;
vfhplus.SafetyDistance = 0.1;
```

21) (**Optional**) The current velocity control (Task 17) of the robot relies on the computed `obstacleDensity` in driving direction, which is not available for Matlab's `vfhplus` implementation. Use Matlab's Helper Function (provided at the bottom of the template) `exampleHelperComputeAngularVelocity` for computing the velocities commanded to the robot. Use the same `bool` as in the previous task for switching between the current velocity control and the velocity control for `vfhplus`.

```
% Calculate velocities
if ~isnan(steeringDirection) % If steering direction is valid
  v = 0.2;
  omega = exampleHelperComputeAngularVelocity(steeringDirection, 1);
else % Stop and search for valid direction
  v = 0.0;
  omega = 0.5;
end
```

22) (**Optional**) Evaluate the VFH+ approach for obstacle avoidance within the `House` environment. Tune the parameters of VFH+ and compare the performance of VFH+ with your VFH implementation.

References

[1] Javier Minguez, Florent Lamiraux, Jean-Paul Laumond, Motion Planning and Obstacle Avoidance, Springer Handbook of Robotics

[2] J. Borenstein, Y. Koren, The vector field histogram - fast obstacle avoidance for mobile robots, IEEE Trans. Robot. Autom. 7, 278-288, (1991)

[3] Iwan Ulrich and Johann Borenstein, VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots Proc. of the 1998 IEEE International Conference on Robotics and Automation, 1572 - 1577, (1998)

[4] Javier Minguez, The obstacle restriction method (ORM): obstacle avoidance in difficult scenarios, IEEE Int. Conf. Intell. Robot Syst. (2005)

[5] Brian P. Gerkey and Kurt Konolige, Planning and Control in Unstructured Terrain

[6] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance