Objectives

This lab introduces the Mathworks Robotics System Toolbox. Proficiency in Matlab is assumed, please revisit the assignments on Matlab Basics in the course Scientific Programming in Matlab in case you need to refresh your knowledge. The Robotics System toolbox provides an interface between MATLAB and Simulink and the Robot Operating System (ROS) that enables you to test and verify applications on ROS-enabled robots and robot simulators such as Gazebo or Stage.

The second part Robotics System Toolbox II introduces the concept of a publisher to control the motion of the robot in the Gazebo simulator by commanding velocities. For ongoing tasks, such as navigating the robot to a remote goal direct motion control of the robot puts a burden on the client. The service request is neither approriate as the client is blocked while its wait for the server to complete the mission. Action clients and servers are suitable for ongoing goals, the action client requests a goal but proceeds with its own code. The action server handles the request and notifies the client only once the goal has been completed. You will implement a simple reactive obstacle avoidance behavior that maps laser range reading onto motor commands such that the Turtlebot avoids collisions with objects in the environment. Please study the first two sections *Robot Control Approaches* and *Basic Principles of Behavior-Based Systems* of the book chapter on *Behavior Based Robotics* prior to the lab [1].

ROS Publisher

Messages in ROS are routed via a communication channel with publish and subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is the name that identifies the content of the message. Other nodes that are interested in that information subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic. A single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others existence.

In the same way in which the Robotics Toolbox employs `rossubscriber` to subscribe to topics such as scan and odometry information it uses `rospublisher` to publish information, for example velocity commands for the simulated mobile robot in Gazebo.

```
pub = rospublisher(topicname);
```

instantiates a publisher `pub` for the topic `topicname`. It is assumed that the topic already exists on the ROS master topic list. The publisher retrieves the topic message type from the topic list on the ROS master. Whenever the Matlab node publishes messages on that topic, other ROS nodes that subscribe to that topic receive those messages.

```
pub = rospublisher(topicname,msgtype);
```

instantiates a publisher for a topic and adds that topic to the ROS master topic list. If the ROS master topic list already contains a matching topic, the ROS master merely adds the Matlab global node to the list of publishers for that topic.

```
msg = rosmessage(pub);
```

creates an empty message `msg` determined by the topic published by `pub`. The empty message with default values provides the structure with fields to which you assign the information before you publish the message.

It is more convenient to combine the instantiation of the publisher with the generation of the empty message in a single assignment.

```
[pub, msg] = rospublisher(topicname);
```

returns a message `msg` that you complete with your data in order to send it the publisher `pub`. The message is initialized with default values.

You publish the message `msg` with

```
send(pub,msg);
```

to the topic specified by the publisher `pub`. All subscribers of that topic in the ROS network receive the message.

The motion of the simulated or real robot is controlled by the topic `/cmd_vel`. The associated message `geometry_msgs/Twist` defines the robots linear and angular velocity

$\mathbf{v}, \omega$. The message has fields `linear` and `angular` that denote the linear and angular velocities as ordinary 3D vector.

The field `linear` corresponds to the translational motion, the field `angular` to the angular velocity. The Turtlebot only possesses two local degrees of freedom, namely linear velocity v along the x-axis which coincides with the direction of the robots current heading and turnrate ω in terms of angular velocity along the z-axis as illustrated in figure 1.

v is specified by `msg.Linear.X` whereas ω is specified in terms of `msg.Angular.Z`. The kinematics of a differential drive robot w.r.t. to its global pose $[x, y, \theta]$ is described by:

$$\begin{aligned} \dot{x} &= v \cos\theta \\ \dot{y} &= v \sin\theta \\ \dot{\theta} &= \omega \end{aligned}$$

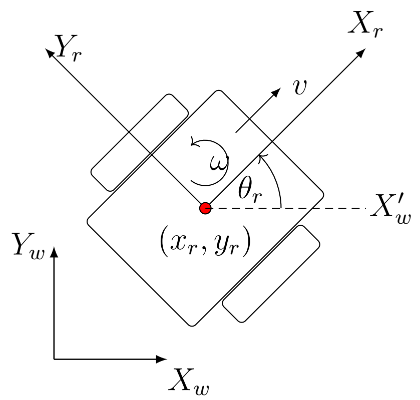

Figure 1: Robot motion according to linear velocity v along current axis X_r and angular velocity ω along Z_r.[2]

In the first part of assignment Matlab sends basic motion commands to the simulated robot in ROS-Gazebo. Figure 2 illustrates the publisher-subscriber communication structure between ROS and Matlab. In Matlab the publisher `velPub` publishes messages on the topic `/cmd_vel` with `send`. In ROS-Gazebo the Gazebo simulation node receives these motion command messages and moves the simulated robot according to the commanded velocities. The subscriber interface abstracts the internals of the simulation from the publisher.

1) Launch the Turtlebot Gazebo simulation in ROS from the command line with the world file. Spawn the turtlebot at a position outside of the box to move the robot in free space without collisions.

[2]source: RST
[3]source: RST

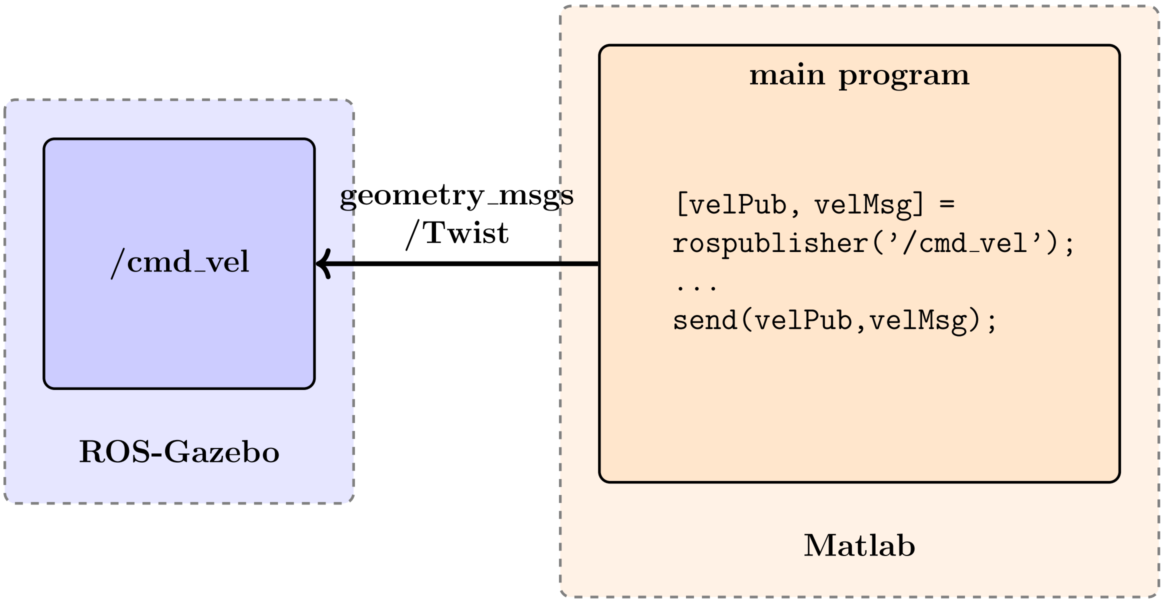

Figure 2: Matlab publisher for ROS topic `/cmd_vel` [3]

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch x_pos:=5 y_pos:=5
```

2) Shutdown the current connection to the ROS master in Matlab `rosshutdown` in case the current master is still running. Reconnect with `rosinit` to the new master.

3) Create a publisher `velPub` for the `/cmd_vel` topic. Create an empty message `velMsg` for that topic either with `rosmessage` or directly with publisher. Set the X-component of linear velocity field of the `velMsg.Linear.X` to $0.2 m/s$. Publish the message `velMsg` with the command `send`. The robot should start moving in a straight line. Stop the robot by sending a new message with zero linear velocity.

4) Write a function `forward` that moves the robot in a straight line for a specified `distance` at a specified linear velocity `velocity`.

```
function [] = forward( velPub, distance, velocity )
```

The input argument `velPub` denotes the publisher.

The distance traveled by the robot is controlled in an open loop fashion by dead reckoning as product of commanded velocity and time. The forward motion with the command velocity executes for a time interval $t = d/v$ seconds. Utilize a rate object with a rate that is inverse proportional to that time interval.

```
rateObj = robotics.Rate(...);
send(...);
waitfor(rateObj);
send(...);
```

5) Write a similar function `turn` that turns the robot on the spot for an `angle` θ at a `turnrate` ω.

```
function [] = turn( velPub, angle, turnrate )
```

The angle θ traveled by the robot is controlled in an open loop fashion by dead reckoning. The turn motion lasts for $t = |\theta|/\omega$ seconds. Ensure the correct sign of the turn rate `turnrate` for clockwise and counterclockwise rotations.

This approach of controlling the robot motion has the obvious drawback that the program in Matlab has to wait until the desired longitudinal or rotational movement has completed. That blocks Matlab from processing other information and it does not allow the program to abort or interrupt the movement command.

ROS Actions

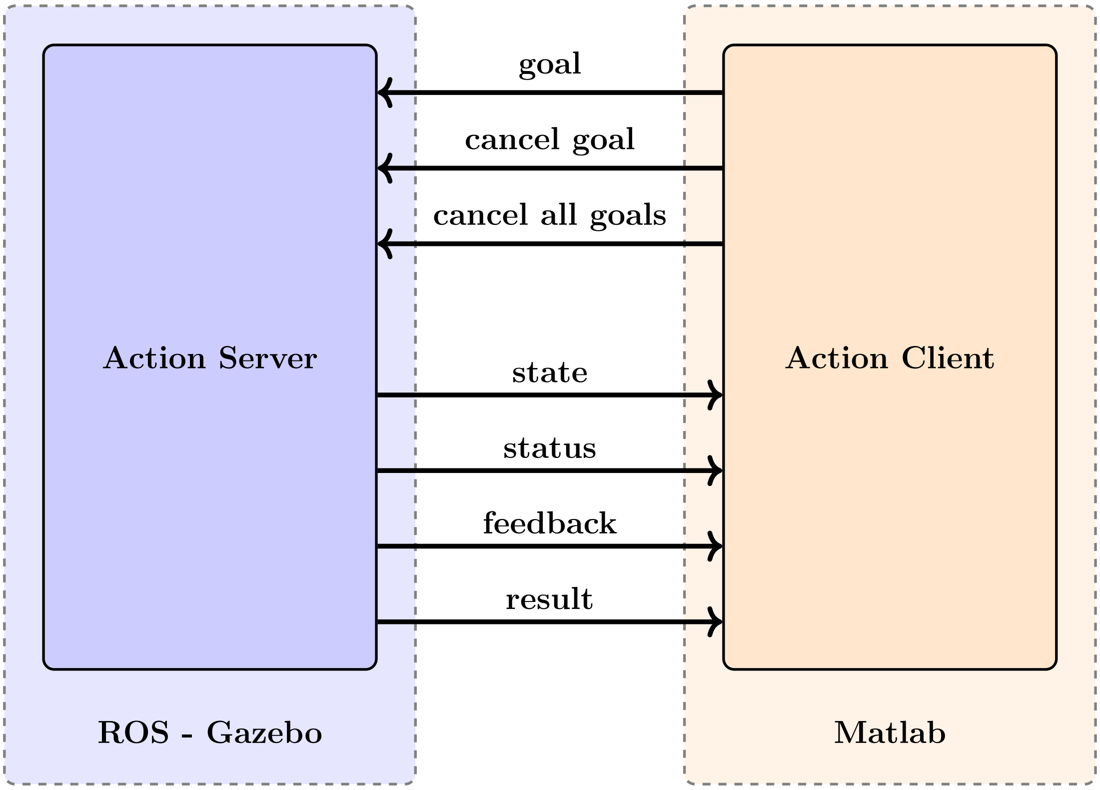

Figure 3: ROS action client server communication structure [4]

ROS Actions implement a client to server communication with adhering to specific protocol. The actions utilize ROS topics to send goal messages from a client to the server as shown in figure 3. It is possible to cancel an ongoing goal using the action client. After receiving a goal, the server processes it and returns information about the progress back

[4]source: RST

to the client. This information includes the status of the server, the state of the current goal, feedback on that goal during operation, and eventually a result message when the goal is complete.

The `sendGoal` function to send goals to the server. Send the goal and wait for it to complete using `sendGoalAndWait`. This function allows you to return the result message, final state of the goal and status of the server. While the server is executing a goal, the callback function `FeedbackFcn`, is called to provide data relevant to that goal. Cancel the current goal using `cancelGoal` or all goals on server using `cancelAllGoals`.

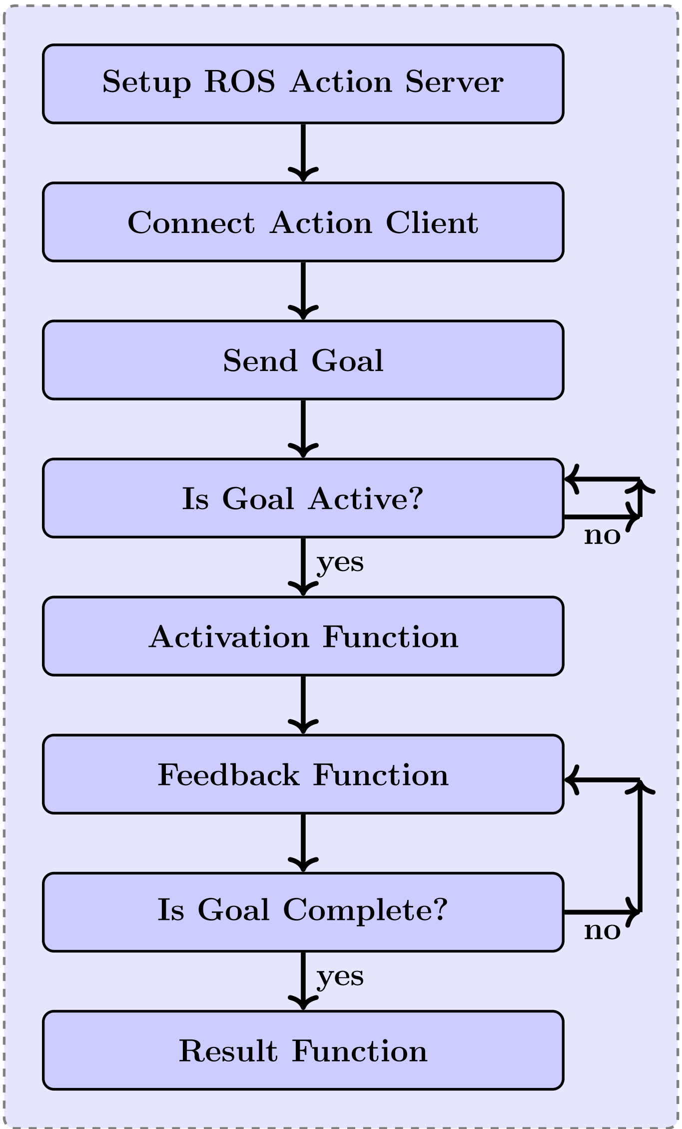

Figure 4: ROS action setup and control flow [5]

[5] source: RST

The control flow of an action client is illustrated in figure 4 composed of the following steps

- Setup ROS action server. With `rosaction list` inspect which actions are available on the ROS network.
- Create an action client and connect it to the server with `rosactionclient` with an action type available on the ROS network. Retrieve a blank `goalMsg` from `rosactionclient`. Use `waitForServer` to wait for the action client to connect to the server.
- Send a goal using `sendGoal`. Specify the `goalMsg` that corresponds to the action type. Modify the blank message `goalMsg` with your desired parameters.
- When a goal status becomes `'active'`, the action is executed and the `ActivationFcn` callback function is called.
- While the goal status remains `'active'`, the server continues to execute the goal. The feedback callback function processes information about this goals execution periodically whenever a new feedback message is received. Use the `FeedbackFcn` to access or process the message data sent from the ROS server.
- When the goal is achieved, the server returns a result message and status. Use the `ResultFcn` callback to access or process the result message and status.

```
[actClient,goalMsg] = rosactionclient(actionname);
```

returns a goal message `goalMsg` to send the action client. The goal message is initialized with default values for that message. If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

The command

```
waitForServer(actClient);
```

waits for the action client to connect to server upon which it can send action to the server with

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,timeout)
```

The Matlab sends the goal to the action server and waits for its completion. The parameter `timeout` specifies a maximum time to complete the action. That behavior is similar to the previous implementation of `forward` in which Matlab waits for the goal completion.

In order to send a goal message to the action server and not to interrupt the program utilize

```
sendGoal(actClient,goalMsg)
```

The specified action client tracks this goal. The function does not wait for the goal to be executed and returns immediately. If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks of the client are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

6) The `'/move_base'` action is provided by the Navigation Stack. Launch the Navigation stack in a second ubuntu terminal with:

   ```
   roslaunch turtlebot3_navigation turtlebot3_navigation.launch
   ```

   Display the current list of actions on the ROS network. Confirm that there is an action `'/move_base'`.

7) Instantiate an action client `actClient` and an empty goal message `goalMsg` for the action `'/move_base'`.

8) Inspect the structure of the `goalMsg` which is of type `'move_base_msgs/MoveBaseGoal'`. Modify the goal position, orientation and map frame id.

   ```
   goalMsg.TargetPose.Pose.Position.X = 7.0;
   goalMsg.TargetPose.Pose.Position.Y = 7.0;
   goalMsg.TargetPose.Pose.Orientation.W = 1.0;
   goalMsg.TargetPose.Header.FrameId = 'map';
   ```

9) Send the modified goal message `goalMsg` with the action client `actClient` and observe the command line output of the activation, feedback and result callback functions.

10) Send a modified goal message `goalMsg` with a remote goal pose. Cancel the goal while the goal has not completed yet.

11) Write a function `squarepath` which moves the robot along a square of side length `a` by a sequence of four straight line motions and right angle turns as shown in figure 5. Implement the function with an action client rather than direct velocity commands. In that case you are supposed to use `sendGoalAndWait` in order not to overwrite a goal that is still active. Since the commanded movement of `squarepath` is relative to the current pose, first determine the current pose by subscribing to the topic `/odom`. Be aware that there is an offset of $[\Delta x, \Delta y] = [2, 2]$ between the odom and map frame which you have to add to determine the goal pose from the odom pose. Calculate the intermediate poses according to the side length of the square. Notice that there eight intermediate goals (see figure 5), four for the forward motion and four for the right angle turn. Your function sends the corresponding eight goal messages in that order to the action server. For the sake of convenience the function

    ```
    function [ goalMsg ] = Pose2GoalMsg( x, y, theta )
    ```

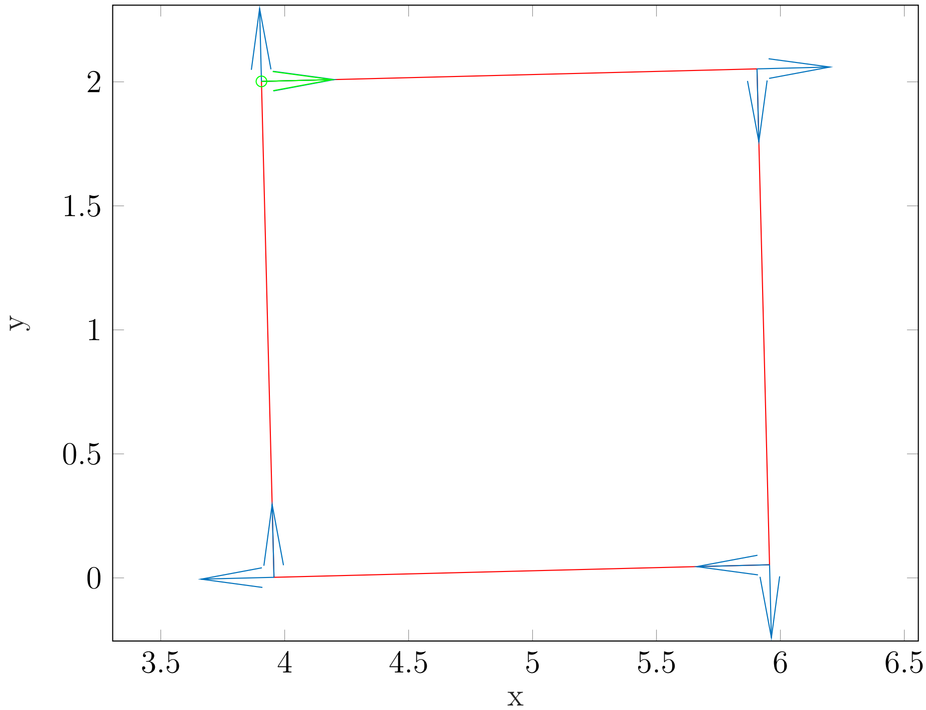

Figure 5: Square path of side length $a = 2$ with intermediate poses [6]

is provided which converts a pose into a message of type `move_base_msgs/MoveBaseGoal`.

[6] source: RST

Callback Subscriber for Laser Scan Messages

This assignment is concerned with subscribing to the `/scan` topic and processing the laser scan message with a Callback function. The `sensor_msgs/LaserScan.msg` is mapped onto a handle object of type ScanHandle which is composed of the properties ranges and angles. Determine the distance and heading to the most imminent obstacle with which the robot might collide if it does not alter its direction of motion. This obstacle representation provides the basis for a reactive obstacle avoidance behavior.

12) Restart the Turtlebot Gazebo simulation in ROS from the command line. Launch the Turtlebot navigation and Rviz.

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
```

Shutdown and restart the ROS master in Matlab.

13) Define a handle class `scanHandle` in a new file named `scanHandle.m`. A convenient way to do so is by utilizing Matlabs class template (*New* → *Class*). The class `scanHandle` has no `methods` and the `properties` resemble the main properties of the message `sensor_msgs/LaserScan.msg` which is published on the `/scan` topic:

- ranges : array of range readings in m
- angles : array of angles in rad
- rmin : minimum range reading
- phimin : angle of minimum range reading

The properties `rmin` and `phimin` become later relevant in the context behavior based obstacle avoidance.

14) Implement a Callback function for the `/scan` topic,

```
function [] = scanCallback( ~, LaserScanMsg, laserScan, beta, robotradius)
```

which converts the LaserScanMsg into to the `ScanHandle` object `laserScan`. The callback function extract the range reading and corresponding angles (`readScanAngles()`) from the `sensor_msgs/LaserScan.msg` message and assigns them to the properties `ranges, angles`.

15) Instantiate a Callback subscriber `scanSubCallback` that subscribes to the `/scan` topic and refers to your Callback function `scanCallback`.

Scaled Obstacle Distance

The objective it to design a simple obstacle avoidance behavior that maps laser range readings (subscriber `/scan`) onto motor actions (publisher `/cmd_vel`). The behavior relies on two principles, reduce velocity in the vicinity of obstacles and turn away from nearby obstacles. Proximity to an obstacle is defined by

The scaled obstacle distance $\hat{r}_{min}$ accounts for the observation that obstacles located along the robots current direction of travel X_r require more caution than obstacles in lateral locations.

$$\hat{r}_{min} = r_{min}(1.0 - \beta cos(\phi_{min})) \tag{1}$$

Effectively, the scaling in Eq. (6) replaces the original circular safe, turn and stop regions of range readings by ellipsoids, with the minor axis oriented along direction of travel X_r and the major axis in lateral direction as shown in figure 6. If the range reading is reduced according to 1 the safety range scales inverse proportional.

$$\hat{r}_{safe} = \frac{r_{safe}}{(1.0 - \beta cos(\phi_{min}))} \tag{2}$$

The ratio of the minor and major axis is determined by β. For $\beta = 0$ the unsafe regions are circular. The unsafe region is elongated in longitudinal direction X_r with increasing β.

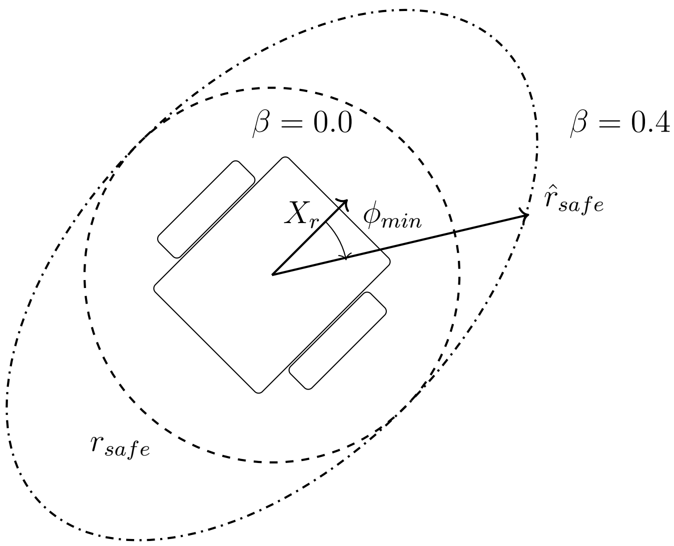

Figure 6: Circular unsafe region for $\beta = 0.0$ (dashed) and scaled unsafe region $\hat{r}_{safe}$ with $\beta = 0.4$ (dash-dot). [8]

16) Augment the callback subscriber function `scanCallback` such that it determines the distance `rmin` and heading `phimin` towards the most relevant obstacle. The relevance of an obstacle depends on its proximity to the robot but also to the

[8] source: RST

relative heading. Nearby obstacles in longitudinal direction are far more relevant for obstacle avoidance than remote obstacles or obstacles in lateral direction. Thus the scaled obstacle distance $\hat{r}_i$ of the scan r_i, ϕ_i in polar coordinates is given by

$$\hat{r}_i = (r_i - r_{robot})(1.0 - \beta cos(\phi_i)) \tag{3}$$

in which $\beta \in [0, 1]$ determines the shrinkage of the absolute distance r_i along the longitudinal distance as illustrated in figure 6. The robot radius r_{robot} is subtracted from the range readings r_i as those measure the distance to the center of the robot rather than the distance to the robots perimeter. `beta, robotradius` are additional input parameters to the callback function. Determine the scan with minimal scaled obstacle distance $\hat{r}_i$

$$j = \text{argmin}_i \, \hat{r}_i \tag{4}$$

and assign $r_{min} = r_j - r_{robot}$ and $\phi_{min} = \phi_j$ to the `scanHandle` properties `rmin, phimin` The original laser range readings r_i are w.r.t. the center of the robot. For a safe obstacle avoidance it is important to consider the robots diameter by subtracting r_{robot} from r_j. That way obstacle avoidance behavior inputs `rmin, phimin` reflect the true spatial separation between the robots circular perimeter and the obstacle.

17) Implement a loop in which the heading and the distance to the nearest object are plotted as an arrow starting at the origin with `quiver`. The arrow should point from the origin to the nearest obstacle. Utilize the previously created `scanHandle` which is filled in the `scanSubCallback` function.

Behavior Based Robotics

Behavior-based robotics is an approach that claims that robots are able to exhibit complex-appearing behaviors with no or little internal states or models of their immediate environment [1]. Their actions mostly emerge from sensory-motor links. These mappings from sensor perceptions to motor actions are described by a functional relationship.

The basic principles of behavior-based control are:

- Behaviors achieve or maintain particular goals. A homing behavior achieves the goal of guiding the robot towards a goal location. A wall-following behavior maintains the goal of following a wall.
- Behaviors are implemented as simple control laws designed for a particular task or scenario (obstacle avoidance, wall following).
- Behaviors map robot perceptions (range readings, camera images, touch sensor) to motor commands. In our case the laser scan is mapped onto a translational and angular velocity of the Turtlebot.

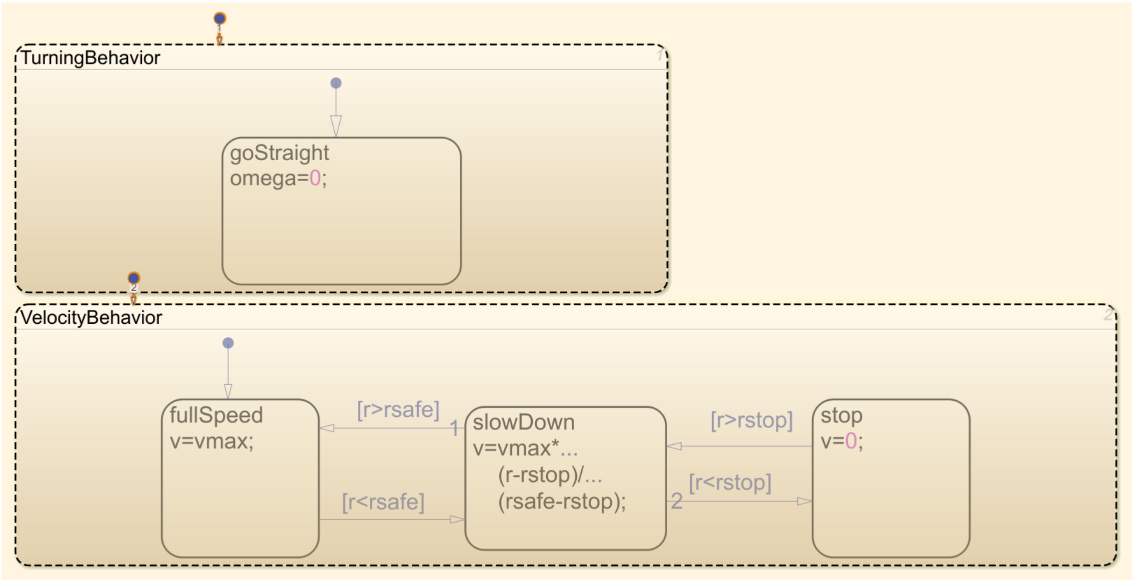

Figure 7: State Machine Template for Reactive Obstacle Avoidance [9]

- Multiple behaviors may process the same inputs and control the same outputs. This paradigm of modularized, decentralized control matches with the ROS concept of nodes that communicate via publishers and subscribers.
- The design methodology is incremental, behaviors are added one at a time and are not aware of each other. At first *survival behaviors* such as obstacle avoidance are designed. Later more complex behaviors such as wall following or door passing are added. This approach is somewhat similar to an object oriented design in which the internal operations of an object remain hidden to the outside observer.
- Behaviors operate concurrently, not sequentially.
- Behaviors interact among each other and with the environment, typically coordinated by an arbitration or command fusion scheme.

Reactive Obstacle Avoidance with State Machine

State machines are representations of dynamic systems that transition from one mode of operation (state) to another. State machines focus on the modes of operation and the conditions required to transition from one state to the next. They facilitate the design of models and automations that remain clear and concise even as model complexity increases. The design of complex control systems relies on state machines to handle complex logic.

The objective it to design a simple obstacle avoidance behavior that maps laser range readings (subscriber `/scan`) onto motor actions (publisher `/cmd_vel`). Reutilize the `scanHandle` class and the subscriber Callback from the previous assignments.

[9]Source: RST

The behavior relies on two fundamental relationships, the velocity behavior maps range readings to linear velocity, the turning behavior maps mapping laser range readings to angular velocities. The state machine displayed in figure 7 is composed of two subcharts `VelocityBehavior` and `TurningBehavior` that run and operate in parallel (indicated by the dashed subchart border lines). The so-called parallel decomposition denoted by AND means that both subcharts are active at the same time. The states inside the subcharts operate in exclusive OR decomposition (indicated by solid state border lines), which requires that only one state at a time is active. The subchart for the `TurningBehavior` is incomplete as it has only a single state `goStraight` and the output variable turn rate ω is always zero. In the assignment you are going to augment the `TurningBehavior` with additional states, transitions and output calculations.

The underlying relationship of the `VelocityBehavior` between the scaled minimum range reading $\hat{r}_{min}$ reduces the linear velocity v with decreasing range readings and eventually stops the robot at a stopping distance r_{stop} (see figure (8)). The `VelocityBehavior` is given in the template.

The `VelocityBehavior` is composed of three modes of operation (states) corresponding to the regimes shown in figure 8. In the state `fullSpeed` the scaled obstacle distance is large $\hat{r}_{min} > r_{safe}$ and the robot moves at full speed $v = v_{max}$. In the state `slowDown` as the distance becomes smaller $r_{stop} < \hat{r}_{min} < r_{safe}$ the robot slows down. Eventually for the state `stop` as the distance becomes critical $\hat{r}_{min} < r_{stop}$ the robot stops. The robot moves at maximum velocity v_0 if the obstacle distance exceeds a safe range r_{safe}. In between r_{safe} and r_{stop} the robots translational velocity decreases linearly from $v = v_{max}$ to $v = 0$.

$$v = \begin{cases} 0, & \text{for } \hat{r}_{min} < r_{stop} \\ v_{max}\frac{\hat{r}_{min}-r_{stop}}{r_{safe}-r_{stop}}, & \text{for } r_{stop} \leq \hat{r}_{min} \leq r_{safe} \\ v_{max}, & \text{for } \hat{r}_{min} > r_{safe} \end{cases} \tag{5}$$

The state machine representation is more intuitive than `if...then...else` code as it separates the transition conditions between states from the state specific calculations of the output variable v. The parameters $r_{stop}, r_{turn}, v_{max}$ are defined as constants within the state machine. The input variable r and the output variable v are defined as local variables in the state machines data definition.

To operate the state machine in your program code create a StateFlow object in the workspace and call its step function.

```
sfObject = sfAvoidObstacle('-warningOnUninitializedData', false);
```

The StateFlow object `sfObject` operates the state machine defined in `sfAvoidObstacle.sfx`. The parameter `'-warningOnUninitializedData'=false` suppresses warning messages for uninitialized variables in the state machine.

[11]source: RST

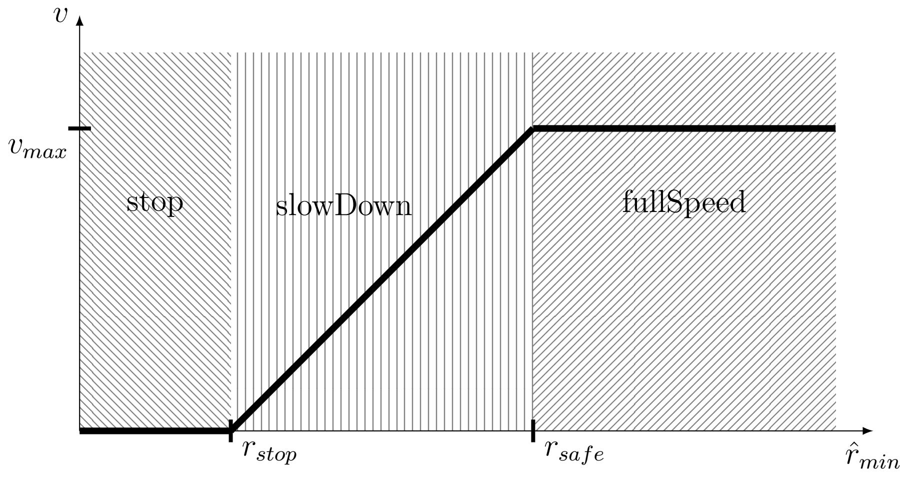

Figure 8: Mapping of scaled minimum range reading $\hat{r}_{min}$ to translational velocity v. The partition of input domains into states `stop, slowDown, fullSpeed` is shown. [11]

The state machine is invoked by the repeated call of the step function passing the input variables (local variables) `r,phi` as arguments. The StateFlow object contains as properties the output variables `sfObject.v, sfObject.omega` of the state machine and the list of the active state(s), whose updated values are queried after the step function call. The following code (provided in the LiveEditor template file) records and plots the states and outputs of the state machine for a sequence of decreasing and increasing $r \in [0.1,\ 0.6]$ m.

```
sfObject = sfAvoidObstacle('-warningOnUninitializedData', false); % instantiate
    stateflow object
statesTurn = {'TurningBehavior.goStraight', 'TurningBehavior.turnLeft', ...
              'TurningBehavior.turnRight', 'TurningBehavior.completeLeftTurn', ...
              'TurningBehavior.completeRightTurn'};
statesVelocity = {'VelocityBehavior.fullSpeed', 'VelocityBehavior.slowDown', ...
          'VelocityBehavior.stop'};

r = [0.6:-0.025:0.1, 0.1:0.025:0.6];
phi = zeros(1,length(r));
phi(2:2:length(r)) = -0.2; % alternate phi
phi(1:2:length(r)) = 0.2;  % alternate phi

for i = 1:length(r)
  step(sfObject, 'r', r(i), 'phi', phi(i));
  v(i) = sfObject.v;
  omega(i) = sfObject.omega;
  stateSeriesTurn(i) = find(strcmp(statesTurn, sfObject.getActiveStates{2}));
  stateSeriesVelocity(i) = find(strcmp(statesVelocity, sfObject.getActiveStates{4}));
end

clear sfObject;

rsafe = 0.5;
```

```
rturn = 0.25;
rstop = 0.15;

subplot(311);
plot(1:length(r), r, 1:length(r), rsafe*ones(1,length(r)), '--', ...
     1:length(r), rturn*ones(1,length(r)), '--', ...
     1:length(r), rstop*ones(1,length(r)), '--', 'LineWidth', 2);
xlabel('steps');
ylabel('$r$', 'Interpreter', "latex");
legend({'$r$', '$r_{safe}$', '$r_{turn}$', '$r_{stop}$'}, 'Interpreter', "latex");

subplot(312);
plot(1:length(r), v, 1:length(r), omega, 'LineWidth', 2);
xlabel('steps');
ylabel('$v,\omega$', 'Interpreter', "latex");
legend({'$v$', '$\omega$'}, 'Interpreter', "latex");

subplot(313);
plot(1:length(r), stateSeriesTurn, 1:length(r), stateSeriesVelocity, 'LineWidth', 2);
xlabel('steps');
ylabel('$s$', 'Interpreter', "latex");
legend({'$s_\omega$', '$s_v$'}, 'Interpreter', "latex");
```

The state machine is shutdown by deleting the the StateFlow object from the workspace.

```
clear sfObject;
```

18) Test the behavior of the state machine for the input descreasing, increasing sequence $r \in \{0.6 \to 0.1 \to 0.6\}$. Since the `TurningBehavior` only possesses the single state `fullSpeed` the output ω is not affected.

The fundamental relationship between scaled minimum range reading and turn rate ω is such that the magnitude of the turn rate increases with decreasing range reading (see figure (10)). If the scaled obstacle distance $\hat{r}_{min}$ is above safe distance r_{safe} the robot moves straight $\omega = 0$ at cruising velocity $v = v_{max}$. The robot is supposed to turn at maximum turn rate $\omega = \omega_{max}$ if the scaled obstacle distance $\hat{r}_{min}$ is short of a distance r_{turn}. The robot moves straight $\omega = 0$ if the scaled obstacle distance exceeds a safe range r_{safe}. In between r_{safe} and r_{turn} the robots turn rate increases linearly from $\omega = 0$ to $\omega = \omega_{max}$.

$$|\omega| = \begin{cases} \omega_{max} & \text{for } \hat{r}_{min} < r_{turn} \\ \omega_{max}\frac{r_{safe}-\hat{r}_{min}}{r_{safe}-r_{turn}} & \text{for } r_{turn} \leq \hat{r}_{min} \leq r_{safe} \\ 0 & \text{for } \hat{r}_{min} > r_{safe} \end{cases} \tag{6}$$

The sign of the turn rate depends on the obstacle direction ϕ_{min} under which the nearest obstacle emerges w.r.t. the robocentric frame. The robot is supposed to turn away from the obstacle. Thus the sign of the turn rate ω is opposite to the sign of the obstacle direction ϕ_{min}.

$$\omega = -sgn(\phi_{min})|\omega| \tag{7}$$

Eqs. (6) and (7) establish a purely reactive, memoryless behavior in which controls only depend on the current perception r_{min}, ϕ_{min}.

19) Augment the subchart of the `TurningBehavior` in the `sfAvoidObstacle.sfx` state machine by two states `turnLeft, turnRight` that correspond to the cases $r_{min} < r_{safe}$ and $\text{sgn}(\phi_{min}) = \pm$. In both states calculate the turn rate ω according to Equations (6) and (7).

20) Add the conditions for the transitions between states:

    - `goStraight` transits to either `turnLeft, turnRight` (depending on sign of ϕ) once $r < r_{safe}$.
    - Vice versa `turnLeft, turnRight` return to state `goStraight` once $r > r_{safe}$.
    - A transition between `turnLeft, turnRight` occurs whenever ϕ changes its sign.

21) Test the behavior of the augmented state machine for the the same sequence $r \in \{0.6 \to 0.1 \to 0.6\}$ and an alternating sequence of $\phi = \pm 0.1$. Observe the behavior of the turn rate ω.

The lack of memory causes the robot to get stuck in corners by turning back and forth as ϕ_{min} switches sign for the nearest obstacle to the left and right. Memorizing the sign of the last turn rate avoids inconsistent turns. The current turning direction is maintained until the robot clears itself from the obstacle once $r_{min} > r_{turn}$. In that case the robot turns with maximum turn rate $\omega = \pm\omega_{max}$ either left or right according to the last turning direction irrespective of the sign of ϕ_{min},

Figure 10 illustrates the domains, boundaries and transition conditions between the states.

22) Augment the subchart of the `TurningBehavior` by two states `completeLeftTurn` and `completeRightTurn` that correspond to the case $r_{min} < r_{turn}$. The turn rate is constant $\omega = \omega_{max}$ for `completeLeftTurn` rsp. $\omega = -\omega_{max}$ for `completeLeftTurn`.

23) Add the conditions for the transitions between states, `turnLeft` $\leftrightarrow$ `completeLeftTurn` and `turnRight` $\leftrightarrow$ `completeRightTurn` depending on $r < r_{turn}$ or $r > r_{turn}$. Notice, that the state machine has no direct transition between `completeLeftTurn, completeRightTurn` to avoid an erratic behavior in corners.

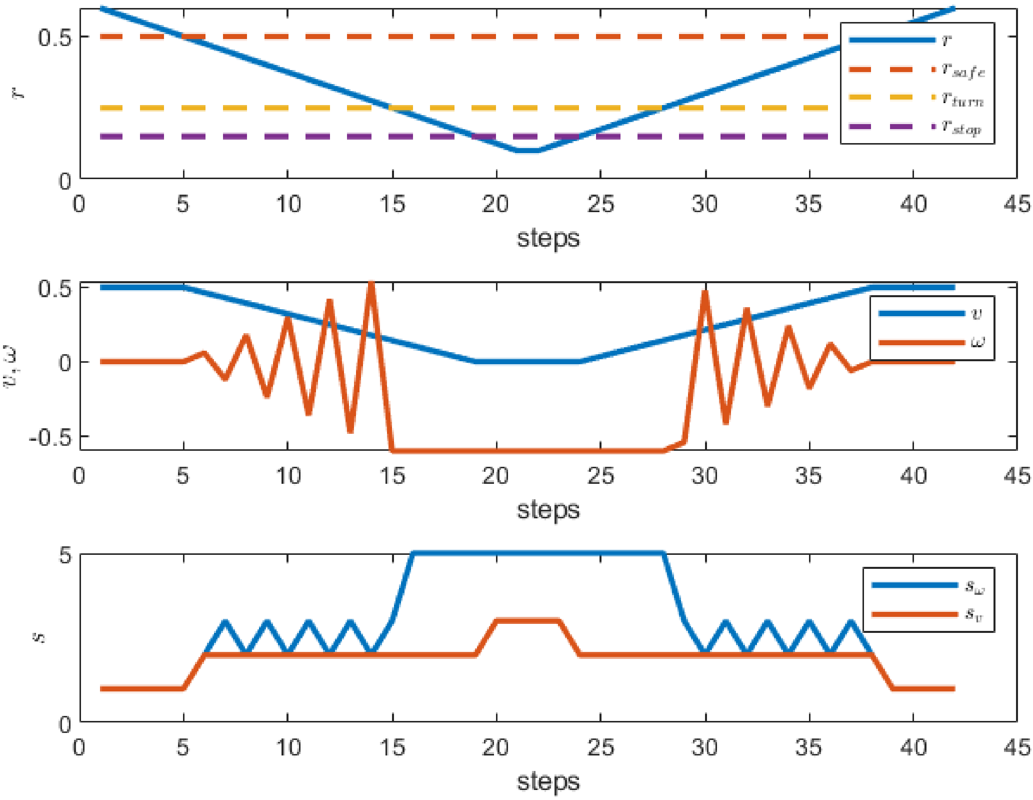

Figure 9: State machine states s_v, s_ω and outputs v, ω for input sequence $r \in \{0.6 \rightarrow 0.1 \rightarrow 0.6\}$ and $\phi = \pm 0.1$ [13]

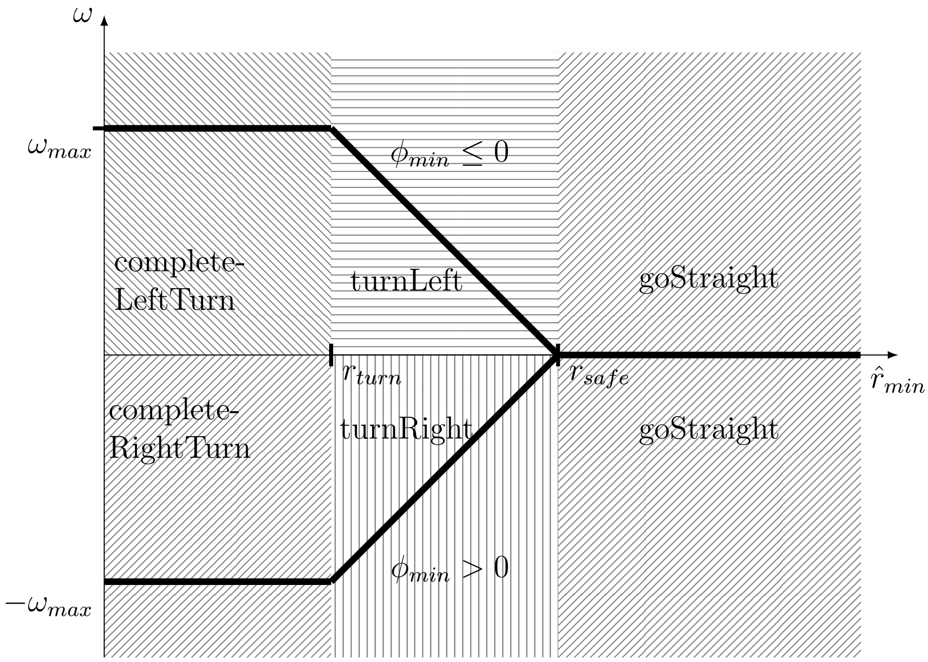

Figure 10: Mapping of scaled minimum range reading $\hat{r}_{min}$ to turn rate ω. The upper low and branch are selected according to the sign of ϕ_{min}. The shaded areas correspond to the states `goStraight, turnLeft, turnRight, completeLeftTurn, completeRightTurn` of the state machine. [15]

24) Test the behavior of the augmented state machine for the the same sequence $r \in \{0.6 \rightarrow 0.1 \rightarrow 0.6\}$ and an alternating sequence of $\phi = \pm 0.1$. Observe the behavior of the turn rate ω and the state sequence. If you implemented the state machine correctly you should observe the behavior shown in figure 9.

25) Operate the state machine `sfAvoidObstacle` within the rated while loop of your program passing the properties `laserScan.rmin, laserScan.phimin` as name-value pairs to the `step` function. Retrieve the translational velocity `SfObject.v` and turn rate `SfObject.omega` from the state machine object and compose the meesage `geometry_msg/Twist` with these values. Publish the message on the topic `/cmd_vel`. The publisher subscriber structure for obstacle avoidance is shown in figure 11. Test your obstacle avoidance in a the Turtlebot simulation. Tune the parameters `rturn, rsafe, rstop` of the state machine such that the robots avoids collisions but traverses free areas at reasonable speed.

26) Test the ability of your obstacle avoidance behavior and state machine to navigate the robot out of corners.

[13]source: RST
[15]source: RST
[17]source: RST

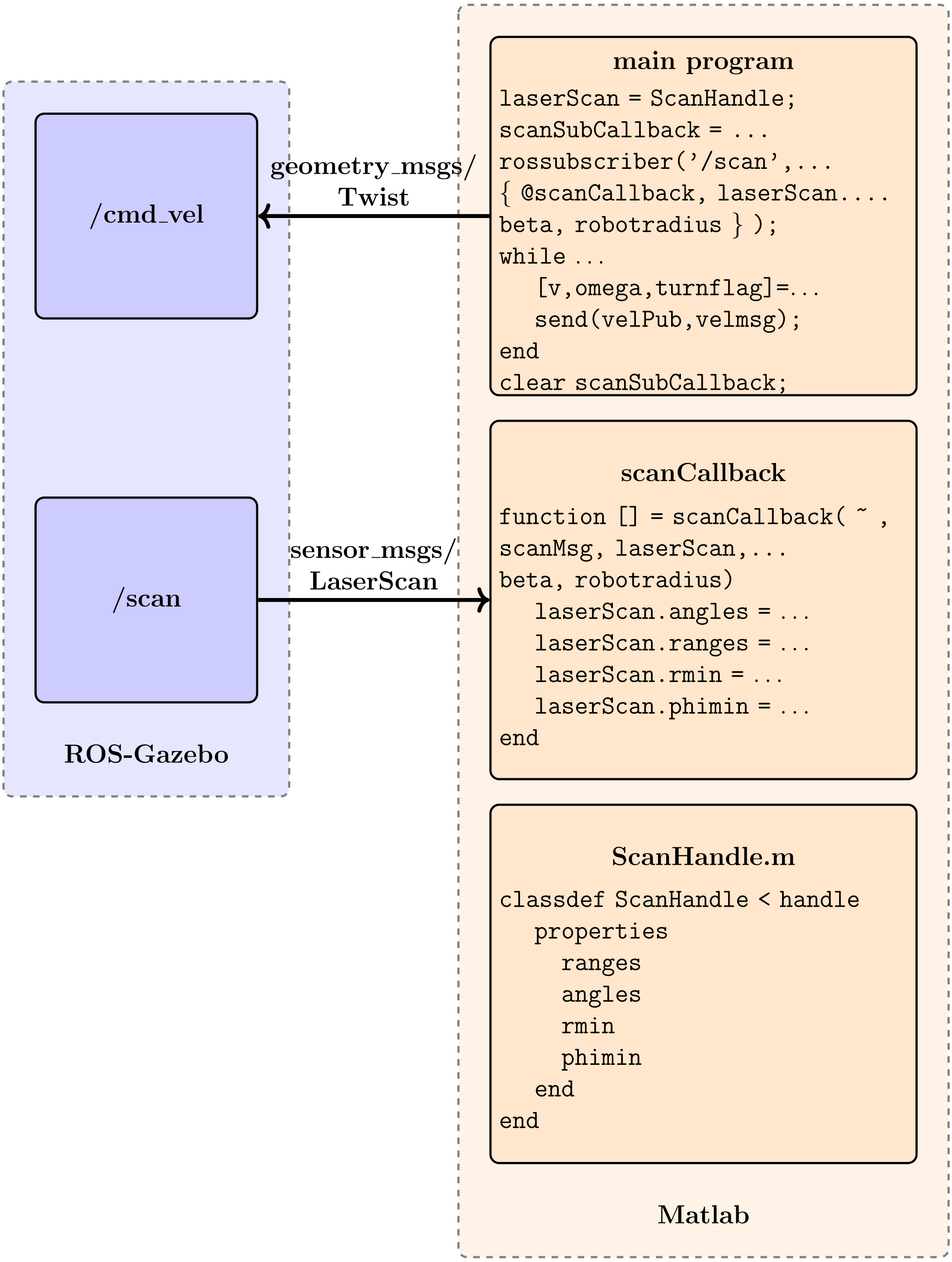

Figure 11: Subscriber, publsiher structure for behavior based obstacle avoidance. Callback subscriber for ROS topic `/scan` with handle object `ScanHandle` and publisher for topic `/cmd_vel`. [17]

References

[1] Maja J. Mataric, Francois Michaud, Behavior-Based Robotics, Springer Handbook of Robotics, Springer, http://link.springer.com/referenceworkentry/10.1007/978-3-540-30301-5_2, 2015