Objectives

This lab introduces the Mathworks Robotics System Toolbox. Proficiency in Matlab is assumed, please revisit the assignments on Matlab Basics and Simulink in the course Scientific Programming in Matlab in case you need to refresh your knowledge.

The lab Robotics System Toolbox I covers the functions of the robotics toolbox utilized in Matlab scripts or on the command line. It is concerned with the subscriber communication to process range sensor and odometry information from the Turtlebot robot.

The second part Robotics System Toolbox II deals with the publisher communication which enables Matlab to control the motion of Turtlebot. It also introduces the concepts of action clients and you will design your first primitive reactive obstacle avoidance behavior.

The third part Robotics System Toolbox III covers representations of spatial transformations between coordinate frames in the three dimensional Euclidean space. You will learn to inspect the ROS transformation tree and to utilize transforms to map point cloud data between frames.

The three parts are distributed over three weeks of lab sessions, in case you complete an assignment ahead of time, go ahead and continue with the next lab if there is time left.

This tutorial assumes that the turtlebot3 packages are already installed in your workspace. In case they are not installed yet, please revisit the 'TurtleBot3 Installation' section of the ROS Introduction 1a tutorial.

Robotics System Toolbox

The Robotics System Toolbox provides algorithms and connectivity for controlling and monitoring mobile robots either in simulation or in reality. The toolbox provides an interconnectivity between MATLAB and Simulink and the Robot Operating System (ROS). This allows you to develop code in Matlab for ROS-enabled robots such as the Turtlebot and robot simulators such as Gazebo. The toolbox includes functions for map representation, path planning, and path following for differential drive robots. It supports the integration of methods for control design, computer vision and state machines with the ROS core. In particular, it enables message exchange with native ROS nodes via topics and services. Basically functions such as `rostopic` and `rosmsg` mimic their counterparts in ROS. Within Matlab or Simulink you can create publisher, subscribers and service clients servers. Throughout the course, you will utilize Matlab for publishing and subscribing to topics on the ROS network.

ROS Communication via Topics

ROS shares information among nodes by sending and receiving messages via publisher and subscribers. Messages are a simple data structure for sharing data. ROS nodes allow communication among different processes within a robotic application. The runtime

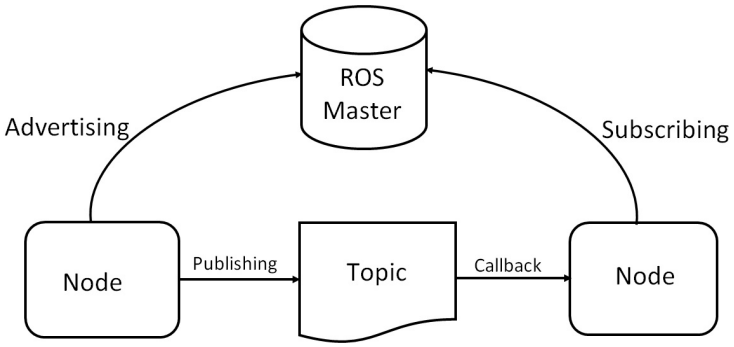

Figure 1: ROS communication architecture. [1]

architecture of ROS is composed of five components: ROS Master, nodes, publishers, subscribers, and topics.

Nodes: Nodes are processes that compute information in a modular fashion. Each node constitutes a running instance of some small ROS program. Robot controllers in ROS are modular comprised of multiple nodes for designated tasks such as processing sensor raw data, actuation of robot motors, localization and path planning.

Master: The ROS Master handles name registration and coordinates the exchange of messages among the components and invokes services within the computation graph. Without the master nodes are unaware of each other. The master also hosts the parameter server which stores information in a central location.

Messages: Nodes communicate with each other by passing messages. A message is a data structure (similar to C/C++ or Matlab structs) organized in typed fields. ROS support primitive data types (integer, float, boolean, etc.) as well as arrays of primitive types.

Topic: Messages in ROS are organized into named topics. The topic is a name that indicates the content of the message. This means that a topic type is defined by the message type published on it.

Publisher and Subscriber: The basic communication mechanism for ROS nodes are messages. Messages are structured into named topics. Any node that shares his information publishes messages on the appropriate topic and those node which are interested in that information subscribe to the topic. This mechanism is similar to the way in which you subscribe to a tweet on twitter.

Figure 1 illustrates the communication between a publisher and a subscriber node. The ROS Master manages the registration of ROS nodes and monitors their operation and communication between publishers and subscribers via topics.

Connect to ROS master from Matlab

[1] source: DesignNews

The Robotics System Toolbox employs the same or similar syntax and commands as ROS itself. In other words functions that you already know from the ROS introductory tutorial such as `rostopic` and `rosnode` on the Matlab command line behave in the same way as their counterparts invoked from an Ubuntu terminal.

In order to interact with ROS from Matlab, you typically follow these steps:

- `rosinit`: Tries to connect to a ROS master running on localhost. To connect to a ROS network, you can create the ROS master in MATLAB or connect to an existing ROS master (initiated via the Terminal). In both cases, MATLAB creates and registers its own ROS node (called the MATLAB global node) with the master.
- Exchange Data: Once connected, MATLAB exchanges data with other ROS nodes through publishers (`rospublisher`), subscribers (`rossubscriber`) and services (`rosservices`).
- `rosshutdown`: Disconnects from the ROS network.

The command

```
rosnode list
```

lists all the nodes that are currently available on the ROS network. Similar

```
rostopic list
```

reports the topics that are currently available on the ROS network. With

```
rostopic info <topic>
```

you obtain specific information about a particular topic.

Tasks 1-4 are supposed to be solved with the ubuntu command line.

1) Start the Turtlebot Gazebo simulation in ROS from the command line:

   ```
   roslaunch turtlebot3_gazebo turtlebot3_world.launch
   ```

2) List the nodes that are present in the ROS network.

   ```
   rosnode list
   ```

3) List the topics that are available in the ROS network.

   ```
   rostopic list
   ```

4) Investigate with

```
rostopic info /scan
```

whether there are nodes that publish to the `/scan` topic.

5) Start Matlab (in this course usually on the Windows host). Connect with

```
rosinit('<YOUR_ROS_MASTER_URI>')
```

to the external ROS master. In your case the ROS master resides on the VM ubuntu. You can specify the address of the master in two ways: by an IP address or by a host name of the computer that runs the master.

Do not forget to shutdown to the current master with

```
rosshutdown
```

before invoking `rosinit` with a different master.

6) The primary mechanism for ROS nodes to exchange data is to send and receive messages. Messages are transmitted on a topic and each topic has a unique name in the ROS network. Ensure that all nodes can communicate with the master and with each other. The individual nodes communicate with the master to register subscribers, publishers, and services. Use `rostopic list` to see which topics are available. Check with `rostopic info` whether the topics `/scan`, `/odom` and `/cmd_vel` are available in the ROS network.

Subscribe to Scan Topic

ROS shares information using messages and services. Messages are a simple data structure for sharing data. Services use a pair of messages for a request-reply interaction.

You subscribe to a topic with `rossubscriber` and receive messages on this topic with `receive`. In MATLAB, you create the subscriber with

```
scanSub = rossubscriber(topicname,msgtype)
```

The message type of the `/scan` topic is `sensor_msgs/LaserScan`. In case of doubt query the message type with `rostopic type /scan` .
Utilize the subscriber to either receive the most recent message with:

```
laserMsg = scanSub.LatestMessage;
```

or wait for the next message with:

```
laserMsg = receive(scanSub);
```

In the first case the program control flow immediately returns to Matlab, in the second case the program waits for the next message to be published on the topic before the programm proceeds. Figure 2 illustrates the publisher and subscriber mechanism between ROS/Gazebo and Matlab.

[2]source: RST

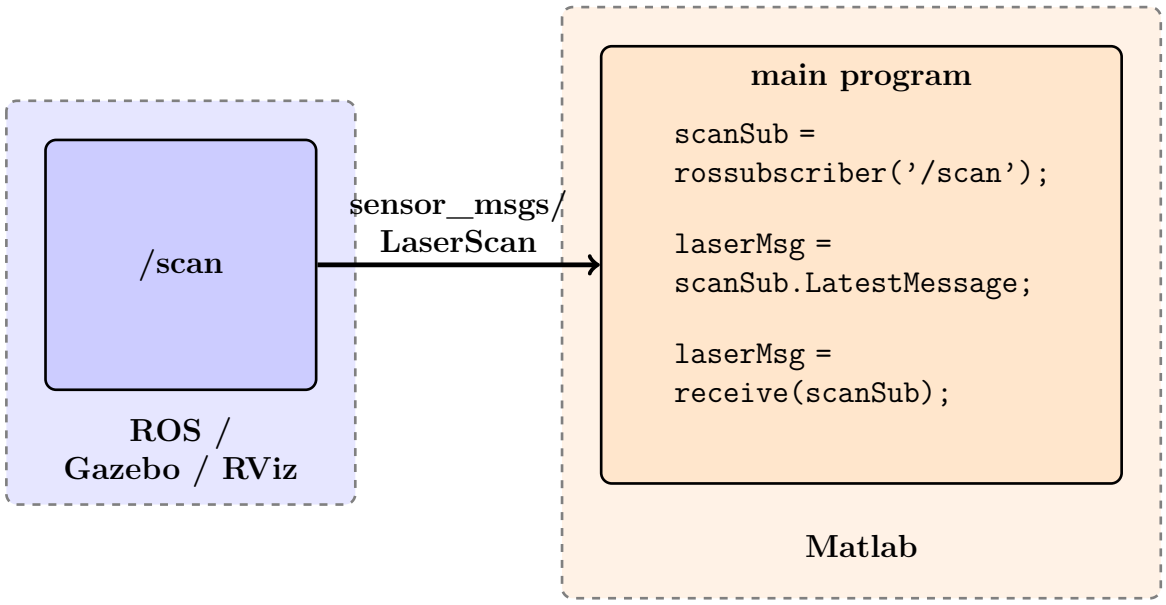

Figure 2: Matlab subscriber for ROS topic `/scan` [2]

7) Use `rossubscriber` to subscribe to the `/scan` topic. `rossubscriber` detects its message type automatically.

```
scanSub = rossubscriber('/scan')
```

Use `receive` to wait for a new message (the second argument is a time-out in seconds). Assign the scan message to `laserMsg` in order to store the message including the laser scan data.

```
laserMsg = receive(scanSub,10);
```

For some message types such as `Laserscan`, Matlab provides convenient visualizers. For the LaserScan message, the function `plot` visualizes the range data. The MaximumRange name-value pair specifies the maximum plot range.

```
figure;
plot(laserMsg,'MaximumRange',7);
```

The `robotics.Rate` object achieves the execution of `for` or `while` loops in Matlab at a fixed cycle rate. The command `waitfor(rateObj)` synchronizes the loop execution time with the ROS time. It pauses the execution of the loop until the cycle time for the current iteration expired. With `rateObj.reset` you reset the clock of the rate object `rateObj`. With `rateObj.TotalElapsedTime` denotes the total time that elapsed either since the instantiation of `rateObj` or the last `rateObj.reset`.

```
rateObj = robotics.Rate(rate);
rateObj.reset;  % reset time at the beginning of the loop
while (...)
  ...
  waitfor(rateObj); % delay loop to match control rate
end
```

8) Reutilize the subscriber `scanSub` to the `/scan` topic Write a script with an infinite loop that receives and visualizes the scan data

```
laserMsg = scanSub.LatestMessage;
```

at a rate of 5 Hz using a rate object.

Superimpose the collected point cloud in a 2D `plot` (using `hold on`) as the robot moves through the environment. The laser scanner moves with the robot and collects range data with respect to the laser frame associated with the robocentric frame. Therefore the point cloud data of consecutive scans is not aligned w.r.t. a static global frame.

The basic infinite loop for receiving and processing messages on a topic looks like this. First, define the parameters at which rate the loop processes data and after the time out after which it terminates. The while loop terminates once the `maxTime` elapses.

```
rate = 5; % Loop Rate in Hz number of iterations / sec
maxTime = 40; % Maximum loop time (loop terminate after maxTime sec)
rateObj=robotics.Rate(rate);
rateObj.reset;  % reset time at the beginning of the loop
while rateObj.TotalElapsedTime < maxTime
  % Receive laser scan message and plot range data
  ...
  waitfor(rateObj);
end
```

9) Launch the navigation stack for the Turtlebot3 and RViz for visualization with:

```
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
```

In case you observe an offset between sensor readings and the actual map in RViz, set the inital pose `2D Pose Estimate` button in RViz roughly to the actual position of the Turtlebot3 in the map in Gazebo. In RViz command your robot to navigate to a remote goal location using the `2D NavGoal` button. Run the Matlab script and observe the superposition of laser scan data as the robot moves through the environment.

Subscribe to Odometry Topic

ROS publishes the TurtleBot pose in Gazebo via the topic `/odom`. The `nav_msgs/Odometry` represents an estimate of a position and velocity in free space. The field `Pose` of type

`geometry_msgs/PoseWithCovariance` contains the robot 3D pose with uncertainty. The field `Pose` contains fields `Position` and `Orientation` that denote the robots position and orientation.

Notice that the field `Pose` of `nav_msgs/Odometry` describes the orientation of the robot frame in terms of a quaternion $\mathbf{q} = [\eta\ \epsilon_x\ \epsilon_y\ \epsilon_z]$. The position part of `nav_msgs/Odometry` describes the robot position in terms of an ordinary vector (x, y, z).

From the message position and orientation part it is possible to calculate the planar robot pose (x_r, y_r, θ_r) w.r.t. the static global frame X_w, Y_w according to figure 3. The vector (x_r, y_r) denotes the origin of the robocentric frame X_r, Y_r w.r.t. the global frame and θ_r denotes the angle between the robots x-axis X_r and the global x-axis X_w.

The estimated pose (x_r, y_r, θ_r) is relative to the robots initial pose $(0, 0, 0)$ at the start of the simulation in Gazebo. In general the pose is estimated by tracking the robot motion from odometry and correcting the estimate by matching the range data with the objects in a global map. The pose estimation is achieved with adaptive Monte Carlo localisation (`amcl`).

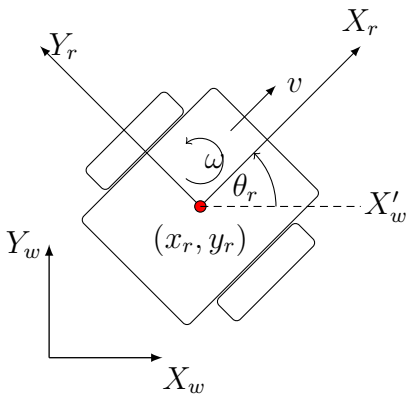

Figure 3: Robot pose w.r.t. global frame X_w, Y_w and robocentric frame X_r, Y_r.[3]

Your Matlab code subscribes to the topic `odom` in order to keep track of the robot pose extracted from the message `nav_msgs/Odometry` as shown in figure 4.

10) Create a subscriber `odomSub` that subscribes to the topic `/odom`. Obtain the robots current pose.

11) Query the robots current pose with `receive` and inspect the data structure of the `odom` message type in the workspace window, in particular the structure of the `Pose` field and its substructures.

[3]source: RST
[4]source: RST

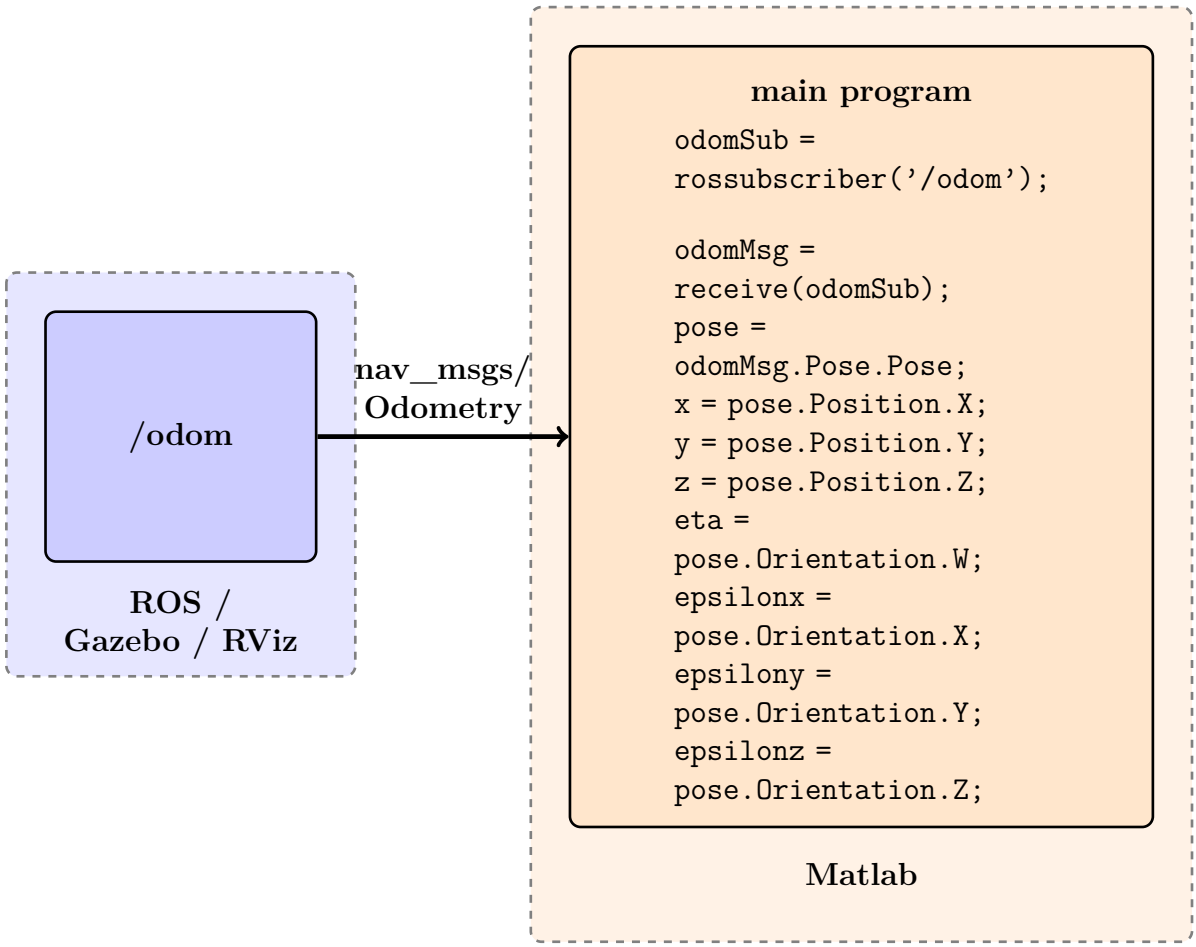

Figure 4: Matlab subscriber for ROS topic `/odom` [4]

12) Augment your script with the infinite loop and rate object. Within the loop receive messages on the /odom topic and visualize the robots planar path as a sequence of x, y-positions in an ordinary `plot`. For that purpose inspect the structure of the pose message format and extract the x, y-components of the position part from the field `Position`.

```
odomMsg = odomSub.LatestMessage;
pose = odomMsg.Pose.Pose;
x = pose.Position.X;
y = pose.Position.Y;
z = pose.Position.Z;
```

13) **optional:** The robots orientation is stored as a quaternion in the field `Orientation` of the field `Pose`.

```
odomMsg = odomSub.LatestMessage;
pose = odomMsg.Pose.Pose;
eta = pose.Orientation.W;
epsilonx = pose.Orientation.X;
```

```
epsilony = pose.Orientation.Y;
epsilonz = pose.Orientation.Z;
```

The function `quat2eul` converts the quaternion vector $[\eta, \epsilon_x, \epsilon_y, \epsilon_z]$ into the Euler angle representation. Extract the current robot orientation θ of the robot along the z-axis in degrees from the first Euler angle. Augment your script such that it plots the robot planar position and orientation at location x, y with a unit vector in direction $[\cos(\theta), \sin(\theta)]$ with `quiver`:

```
quiver(x,y,u,v,scale);
```

has a scale parameter to adjust the size of the vectors in the plot.

14) Command your robot with RViz to navigate to a remote goal location with the `2D NavGoal` button. Start your Matlab script and observe the evolution of the robot path in the Matlab figure.

15) **optional:** Write a Matlab function:

```
function [ x, y, theta ] = OdometryMsg2Pose( odomMsg )
```

that maps an `odom` message of type `nav_msgs/Odometry` into the components $[x,\ y,\ \theta]$ of the planar robot pose vector.

Such a function is convenient for mapping ROS odometry messages into a planar mobile robot pose. The function is provided for future assignments in the Moodle workspace.

Asynchronous Programming

Mastering asynchronous programming is essential for using ROS with distributed nodes and asynchronous communication. Figure 5 shows a typical synchronous call. The caller invokes a method on a callee. The callee handles the request, performs the necessary computations and returns the result to the caller. In the example Matlab is the caller, which queries ROS for a new message on the `odom` topic.

```
odomSub = rossubscriber('/odom');
...
odomMsg = receive(odomSub);
```

The caller (Matlab) is blocked while the callee (ROS) is busy to recompute the robot pose from odometry and compose the message. This type of blocking is less critical if calls are answered in a timely manner and the caller might be able to perform other computations. However, in a distributed system such as ROS a synchronous invocation scheme might result in performance loss. Moreover, synchronous operation might cause a deadlock situation in case the calls end up in a cycle in which multiple nodes mutually wait for each other to publish novel messages.

[5] source: Jadex Active Components User Guide

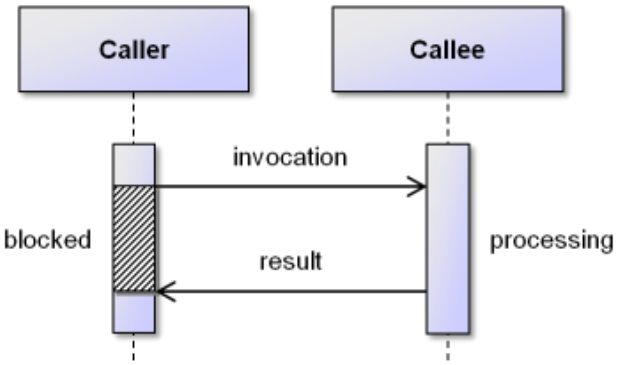

Figure 5: Synchronous call. [5]

Asynchronous calls do not block the caller while the callee is handling the request. The result of an asynchronous call is assigned to a so called future object, which serves as a temporary container to store the actual result of the call.

The caller passes a reference to the future object to the caller and later retrieves the result from the callee from the container. It is assumed that the caller is able to check whether the result has become available. In the left part of figure 6 depicts a scheme named waiting by necessity, which means the caller continues with its code until it can no longer proceed without the result from the callee. The right side of figure 6 illustrates a genuinely non-blocking invocation. In this case the callee explicitly notifies the caller once the result has become available. The caller registers a result listener (callback subscriber) on the future.

```
callbackSub = rossubscriber(topic,@callbackFunction);
```

The callback function

```
function callbackFunction(src,msg)
```

of the callback subscriber (result listener) is invoked in the moment the callee publishes the result. In the context of ROS the result is constituted by the message published by the callee node.

Rather than a subscriber that actively queries novel messages with `receive`, it is more elegant to register a callback function for a subscriber. The callback function is automatically invoked, whenever a new message is published on the topic of the subscriber. That way other MATLAB code is executed while the callback subscriber is dormant and waits for new messages. Callback subscribers are able to handle the type of asynchronous communication of publisher subscriber message passing.

Callbacks are essential if your program operates with multiple subscribers as otherwise the `receive` commands of these subscribers might block each other in a sequential code.

[6] source: Jadex Active Components User Guide

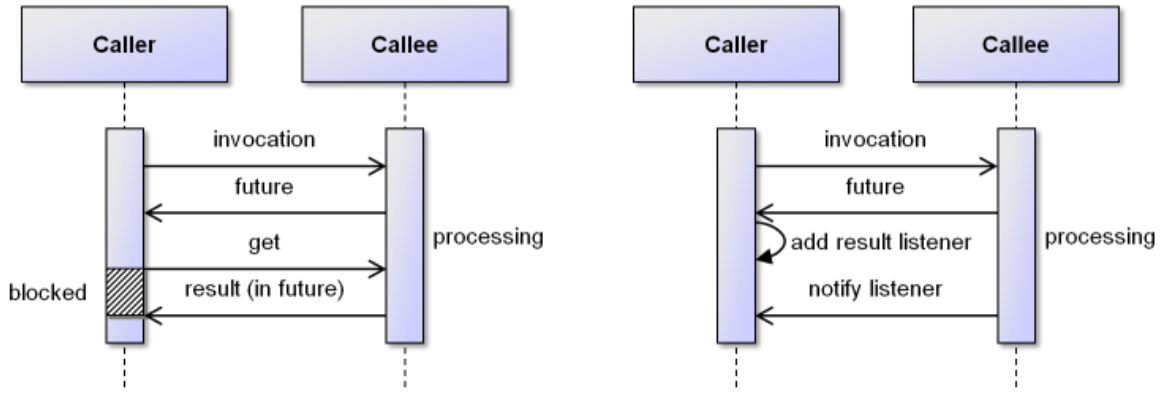

Figure 6: Asynchronous call. [6]

The problem of mutual blocking might be partially resolved by using `LatestMessage` rather than `receive`.

A callback subscriber is registered with:

```
callbackSub = rossubscriber(topic,@callbackFunction);
```

The callback subscriber is stopped by clearing the subscriber variable:

```
clear callbackSub;
```

The subscriber callback function demands at least two input arguments. The first argument, `src`, is the associated subscriber object. The second argument, `msg`, is the received message object. The function header for the callback is:

```
function callbackFunction(src,msg)
```

Notice, that a callback function has no return arguments. If the callback function is supposed to return data it either has to utilize global variables or pass a handle object as additional argument to the callback function.

It is possible to pass additional parameters `userdata` to the callback function by including both the callback function and the parameters as elements of a cell array:

```
callbackSub = rossubscriber(topic,{@callbackFunction, userdata});
```

In that case the function header for the callback becomes:

```
function callbackFunction(src,msg,userdata)
```

16) The task is to plot the robot path using a callback function rather than the main loop as in the assignment 12. Write a callback function for the `odom` topic:

[7] source: RST

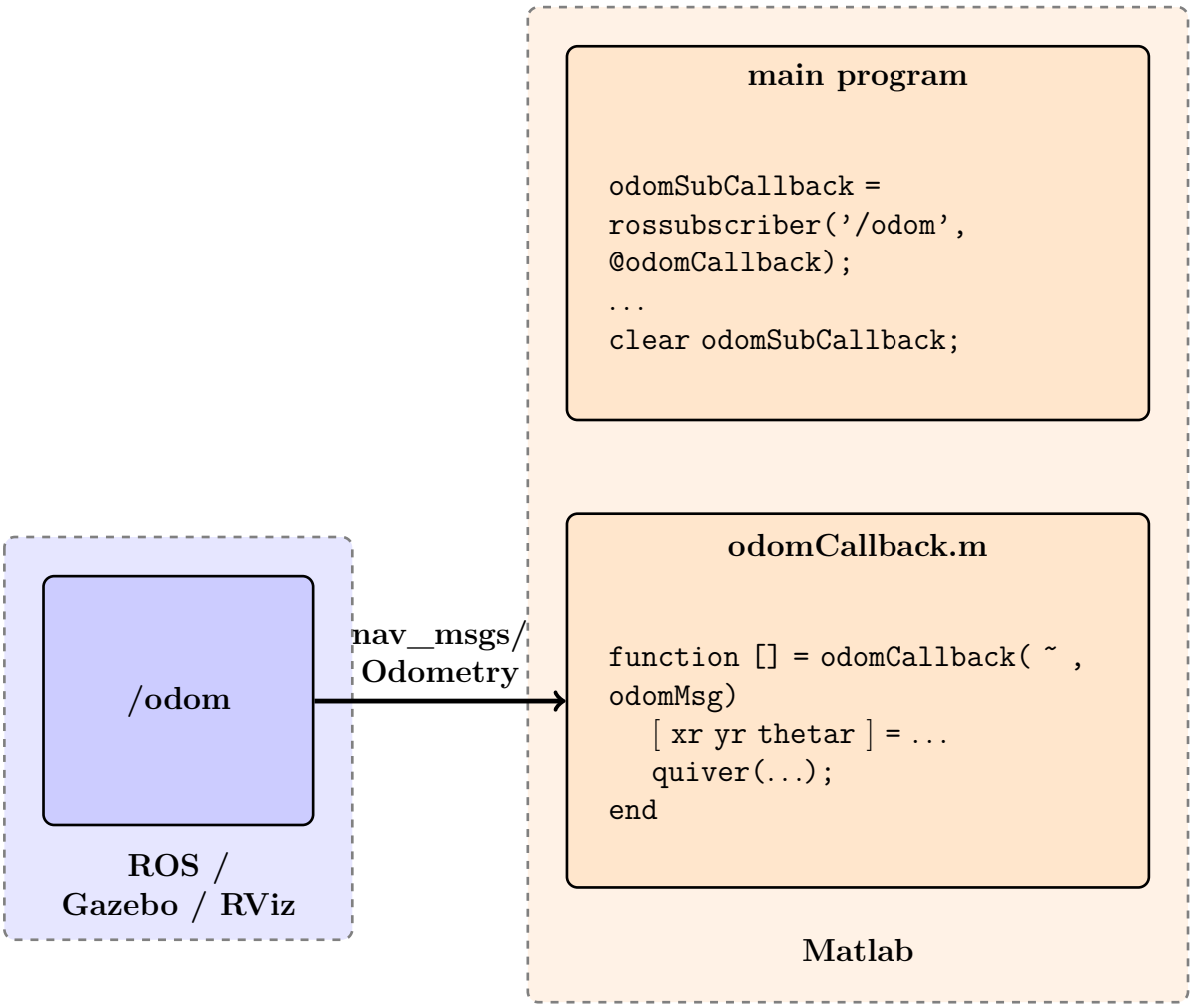

Figure 7: Matlab callback subscriber for ROS topic `/odom` [7]

```
function [] = odomCallback( ~, odomMsg)
```

which converts the `odomMsg` into pose format with `OdometryMsg2Pose` and plot the pose either as markers (`plot`) or heading vectors (`quiver`). Figure 7 illustrates the callback subscriber structure divided among the main program and the callback function.

17) Instantiate a callback subscriber `odomSubCallback` that subscribes to the `/odom` topic and refers to your callback function `odomCallback`. Move the robot and observe the evolution of the path. Clear the callback subscriber `odomSubCallback` to stop plotting.

18) Global variables provide a means to share data between the callback function and the workspace of the main program. Declare global variables `xr yr thetar` in the main program to denote the robot pose $[x_r, y_r, \theta_r]$ in the world frame.

Declare the same global variables in the callback function `odomCallback` and assign the pose information extracted from `odomMsg`.

The most current pose information in terms of the global variables can be accessed from the main program. Record pose data within the while loop by storing pose data and current time in arrays. Plot the robot pose components versus time.

```
global xr yr thetar
i = 1;
while rateObj.TotalElapsedTime < maxTime
  xdata(i) = xr;
  ydata(i) = yr;
  thetadata(i) = thetar;
  t(i) = rateObj.TotalElapsedTime;
  i = i+1;
  waitfor(rateObj);
end
plot(...);
```

Rather than to extract information from callback functions by global variables it is more flexible and transparent to pass a handle object as an additional input argument to the callback function.

An ordinary value class constructor returns an object that is associated with the variable to which it is assigned. If you reassign this variable it creates an independent copy of the original object. If you pass the copy as an argument to a function the original variable is not modified unless the function explicitly returns the modified object as an output argument and you reassign it to the original object.

```
[x, ...] = function myfun (x, ...)
```

This approach is non-feasible for callback functions as they do not allow output arguments.

In contrast a handle object provides a reference to the object created. It allows you to assign the handle object to multiple variables or pass it to functions without copying the original object. Therefore a function that modifies a handle object passed as an input argument does not need to return the object.

Instances of classes that derive from the `handle` class are references to the underlying object data. If you pass a handle object as an argument to a function it does not copy the data but the argument refers to the same object as the original handle. Any changes to the object within the function apply to the original object.

All handle classes are derived from the abstract `handle` class. A handle class is defined by inheriting from the `handle` class and storing the class definition in a separate m-file:

```
classdef MyHandleClass < handle
  properties
  ...
```

```
  end

  methods
  ...
  end
end
```

Properties contains the object data. Classes define the same properties for all objects, but each object has its own unique data values.

The property specification block defines property names and default values. Property names must be listed on separate lines. The following code defines properties (fields) `x, y, theta` with default values of zero:

```
properties
  x = 0
  y = 0
  theta = 0
end
```

The syntax to access a property is identical to the structure field syntax (see Scientific Programming in Matlab). Assume, `obj` is an object of a class, then you access the numerical value of a property by referencing the property name `PropertyName`.

```
val = obj.PropertyName
```

In order to assign a value to a property, the property reference occurs on the left side of the equal sign in the assignment.

```
obj.PropertyName = val
```

In the following you are supposed to modify your callback function `odomCallback` to operate with an object handle for the pose rather than global variables. Figure 8 illustrates the main code with the `PoseHandle` object `pose`, the callback function `odomCallback` with the additional input argument `pose` as a handle object and the file `PoseHandle.m` with the class definition of class `PoseHandle`.

19) Define a handle class `PoseHandle` with properties `x, y, theta` to describe the planar robot pose. Select the button `New` in the Matlab Editor menu and select *class* to obtain a template code for class definitions. Provide the class name, the inheritance from the handle class, the property names and their default values. In case of doubt retrieve the file `PoseHandle.m` from the Moodle workspace.

20) Augment your callback function `odomCallback` by an additional input parameter `pose` as a handle object of class `PoseHandle`. Assign the pose information extracted from `odomMsg` to the properties `x, y, theta` of the `pose` object.

[8] source: RST

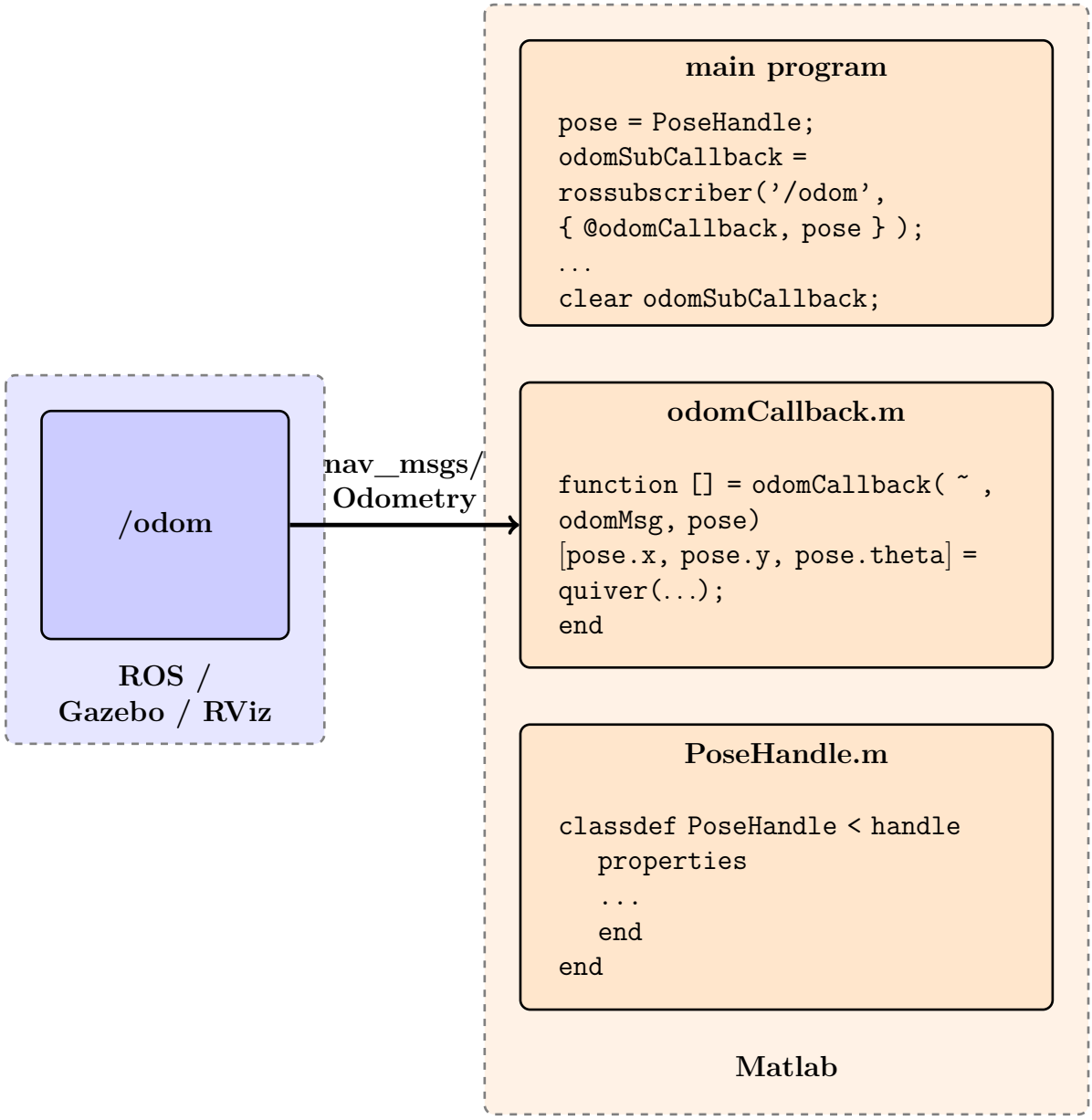

Figure 8: Matlab callback subscriber for ROS topic `/odom` with handle object[8]

21) In your main program instantiate a handle object `pose` from the class `PoseHandle` and pass it as an additional argument to the callback function `odomCallback` in the instantiation of the `odomSubCallback` subscriber. Replace the assignment to the data recording arrays `xdata, ydata, thetadata` in the while loop from assignment (18). Replace the global variables in the right hand side of the assignment by the corresponding `pose` properties.

22) Augment your callback function by an additional input parameter `fig` to denote the figure in which to plot the robot pose. Notice, that figure objects are already handle

objects, that means the `plot` commands in `odomCallback` refer to the original figure object `fig` in the main code.

```
function [] = odomCallback( ~, odomMsg, pose, fig)
  figure(fig); % switch current figure
  ...
end
```

Instantiate your callback subscriber `odomSubCallback` with the additional input argument `fig`.

```
  fig = figure(3);
  odomSubCallback = rossubscriber('/odom',{@odomCallback,pose,fig});
```

Literatur

[1] ROS Tutorials, `http://wiki.ros.org/ROS/Tutorials`, 2016

[2] Jason M. O'Kane, A Gentle Introduction to ROS, `https://cse.sc.edu/~jokane/agitr/`, 2015

[3] K. Waldron, J. Schmiedeler, Kinematics, Springer Handbook of Robotics, Springer, `http://link.springer.com/referenceworkentry/10.1007/978-3-540-30301-5_2`, 2015