Integrating ROS and MATLAB

By Peter Corke

The Robot Operating System (ROS) has gained wide currency for the creation of working robotic systems, initially in the laboratory but now also in industry. Despite ongoing evolution, the fundamental principles of publishing and subscribing on topics, application-specific messages, invoking services, and sharing parameters have remained constant. The primary programming environment for those working with ROS is C++ and Python, though using Java is also possible.

MATLAB is a powerful tool for prototyping and simulating control systems and robotics [1], [2], but, until very recently, it has not been easy to integrate with ROS. The need for such integration is evidenced by many solutions that have been developed, including the JavaScript Object Notation-based rosbridge (http://wiki.ros.org/rosbridge_suite), the Java-based ROS-MATLAB bridge package (https://code.google.com/p/mplab-ros-pkg/wiki/java_matlab_bridge), and the ROSlab-IPC bridge (https://alliance.seas.upenn.edu/meam620/wiki/index.php?n=Roslab.IpcBridge), among others. However, none of these have caught on in a big way, perhaps due to installation and usability concerns.

With the recent release of MATLAB 2015a, there is a better option available through the newly introduced Robotics Systems Toolbox $\mathcal{Q}$ (RST). This toolbox has three main areas of functionality: ROS integration, support for pose represented as special Euclidean group (3) homogeneous transformations, and probabilistic-road-map-based path planning. The remainder of this article will introduce the ROS functionality in a tutorial manner. Note that this functionality is an evolution of Mathworks' own ROS input/output package introduced in 2014, and the RST ROS application program interface has some changes with respect to this earlier package.

Assuming the presence of a running ROS system with an ROS master, we initialize the MATLAB ROS subsystem with the IP address and port number of the ROS master, e.g.,

```
rosinit('192.168.1.10',
11311).
```

If no arguments are provided, then MATLAB will create an ROS master and display its URI so that it can be used by other nodes.

Next, we want to publish on a topic; so let us take a simple example from the ROS tutorial. We first create a message object of the standard string type

```
msg = rosmessage('std_
      msgs/String');
```

and then set its value

```
msg.Data = 'hello world';
```

The message is an object, and its properties are hierarchical and match the fields of the message. We can read or write the properties directly without having to use setter or getter methods. All that remains now is to publish it.

```
rospublisher('/MyTopic',
msg);
```

Alternatively, we could create a publisher object and optionally specify the message type

```
pub = rospublisher('/
      MyTopic', 'std_
      msgs/String');
```

and then invoke its send method

```
pub.send(msg);
```

Various options can be configured at construction time for the publisher object.

Receiving a topic is just as easy. We first create a subscriber object for the particular topic and optionally specify the message type

```
sub = rossubscriber('/
      MyTopic', 'std_
      msgs/String').
```

The constructor has various options to control buffer size and whether only the most recent message should be returned. We read the next message on the topic by

```
msg = sub.receive(),
```

which blocks until a message is received, but we could also specify a timeout interval in seconds

```
msg = sub.receive(5)
```

An alternative to polling for messages is to establish a callback

```
sub = rossubscriber('/
      MyTopic', 'std_
      msgs/String',
      @rxcallback)
```

Digital Object Identifier 10.1109/MRA.2015.2418513
Date of publication: 18 June 2015

to the function

```
function rxcallback(src,
msg)
    disp([char(msg.
Data()), sprintf('\n
Message received: %s',
datestr(now))]);
```

which is invoked on every message receipt.

Next, let us look at a more complex message: the velocity twist with time stamp

```
msg =
rosmessage('geometry_
msgs/TwistStamped'),
```

and we can view its definition

```
>> definition(msg)
% A Twist with reference
coordinate frame and
timestamp std_msgs/Head-
er Header Twist Twist
```

or access one of its fields

```
msg.Twist.Linear.X = 0;
```

Custom messages are also possible but beyond the scope of this article. (See http://www.mathworks.com/matlabcentral/fileexchange/49810 for details.) The ROS parameters can be accessed via a `ParameterTree` object returned by

```
ptree = rosparam,
```

and we can use it to set, get, create, or delete parameters in the ROS parameter server.

```
ptree.get('rosversion')
ptree.set('myparameter', 23)
```

and parameters can have integer, logical, char, double, or cell array types.

We can also access and create services in MATLAB code. Inspired by the TwoInts example given in the ROS tutorial, we can easily create a service to add two integers

```
sumserver = rossvcserver('/
            sum', rostype.
            roscpp_tutori-
            als_TwoInts, @
            SumCallback),
```

and the service function is

```
function resp =
 SumCallback(~,req,resp)
  resp.Sum = req.A +
   req.B;
```

and this can now be invoked from any ROS node

```
$ rosservice call /sum2 1
2 sum: 3
```

or from inside MATLAB by first creating a service client

```
sumclient = rossvcclient
            ('/sum'),
```

creating a message with the numbers to be added

```
sumreq = rosmessage
         (sumclient);
         sumreq.A = 2;
         sumreq.B = 1,
```

and then invoking the service

```
sumresp = call(sumclient,
sumreq,'Timeout',3)
>> sumresp.Sum ans = 3.
```

There is also the capability to read and write ROS bag files. First, we open the bag file and list the available topics:

```
bag = rosbag ('quad-2014-
      06-13.bag')bag.
      AvailableTopics.
```

To extract all the images on the topic `/preview`, we use the `select` method to choose the particular topic, use `readMessages` to extract a cell array of 100 messages that happen to be of type `sensormsgs/Image`, and then convert this to a cell array of images that can be displayed

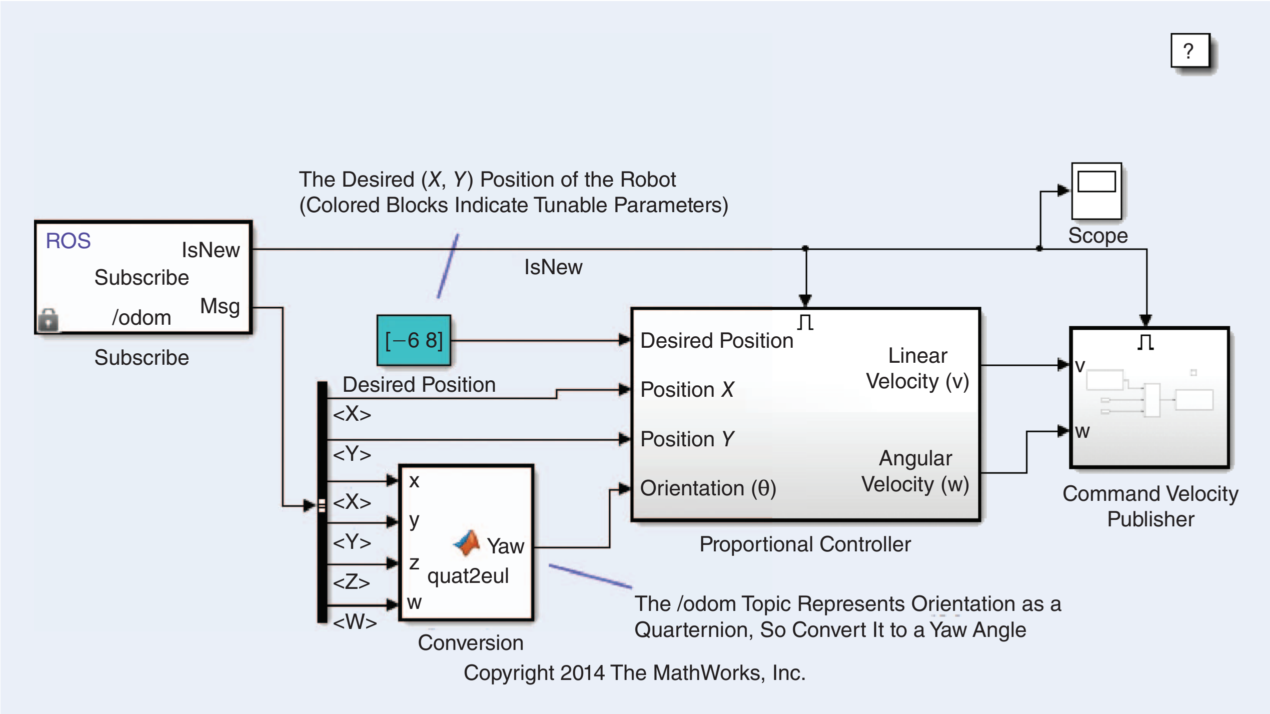

Figure 1. A robot controller implemented in Simulink with ROS. (Image used with permission from MathWorks.)

```
preview = bag.select
          ('Topic', 'pre-
          view')
 subset = preview.readMes-
          sages(500:599)
 images = cellfun(@readIm-
          age, all, 'Uni-
          formOutput',
          false);
```

We could also extract inertial measurement unit (IMU) data and place the x-axis translational and z-axis angular velocity, for example into a `timeseries` object, which automatically picks up the message time stamps and has various methods for analysis and display.

```
imu = bag.select('Topic',
      'fcu/imu');
 ts = imu.timeseries('Linea
      rAcceleration.X',
      'AngularVelocity.Z');
      ts.plot
```

A number of interactive commands are available in the MATLAB environment, which mimics the ROS command line utilities to find messages, nodes, topics, parameters, or services, e.g.,

```
>> rosmsg list
>> rosmsg show geometry_
msgs/TwistStamped
>> rosnode list
>> rostopic list
>> rosparam list
>> rosservice list.
```

This provides all the programmatic tools required to write code in MATLAB that can fully participate in an ROS-based robot control system. A powerful advantage of MATLAB ROS is its platform independence—this code will work on a Mac, Windows, or Linux system. The real-time update rate is, of course, going to depend on the size of the messages, the complexity and efficiency of your MATLAB code, and the performance of your computer, but tens of hertz is feasible.

An alternative to programmatic implementation is to use the Simulink block diagram modeling environment, as shown in Figure 1. An RST provides a palette of blocks that includes `Publish` and `Subscribe`. The `Msg` output of the `Subscribe` block is a bus type, and we can use a Simulink Bus Selector to pull out the particular message fields in which we are interested. We use a triggered subsystem to ensure that a message is published only after a message is received. With the appropriate Simulink settings, this controller can run in "real time" (see the aforementioned caveats), and the `Scope` blocks allow us to conveniently see what is happening. We can, of course, log signals to workspace variables for later analysis and graphical display.

Finally, and most significantly, we can export this diagram as code. Simulink generates a `.tgz` archive file that contains all the code necessary to build a standalone real-time ROS node on a Linux system.

In this short article, we can only skim the surface of the new MATLAB capability for ROS integration. More details can be found at mathworks.com/ros. Over time, the RST will gain additional functionality that will allow students, engineers, and researchers to create robotic systems more quickly and leverage the large and mature ROS code base.

References

[1] P. I. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Berlin Heidelberg, Germany: Springer-Verlag, 2011.

[2] P. Corke. Robotics toolbox for MATLAB. [Online]. Available: http://www.petercorke.com/robot