Universidad de Ingeniería y Tecnología

FUNDAMENTOS DE ROBÓTICA

Prof. Oscar E. Ramos Ponce

Guía de Laboratorio 1:

Introducción a ROS para Robótica

Lima - Perú

2017 - 1

Laboratorio 1:
Introducción a ROS para Robótica

1. Objetivos

- Conocer de manera práctica los aspectos básicos del funcionamiento de ROS (*Robot Operating System*).
- Mostrar modelos de robots en el visualizador RViz de ROS e interactuar con ellos gráficamente y a través de código.
- Comandar desde ROS las posiciones articulares de un robot en el simulador dinámico Gazebo.

2. Equipos, Materiales y Otros

No	Descripción	Cantidad
1	Computadora con ROS en Linux	1

3. Orientaciones de Seguridad

- Respetar las recomendaciones del docente así como las recomendaciones indicadas en los carteles de señalización.
- Usar los EPP durante el laboratorio.
- Utilizar los equipos de acuerdo con las recomendaciones.
- Respetar las recomendaciones para el uso del ambiente.

4. Aspectos Básicos de ROS

Nota: Responder a las preguntas de la guía en un documento separado utilizando cualquier editor de texto disponible. Cuando sea necesario, incluir el código fuente desarrollado y algunas capturas de pantalla de los resultados obtenidos. Presentar el informe en formato pdf. Para este laboratorio se asumirá que se está usando *ROS Indigo* en *Linux*.

4.1. Creación de un Espacio de Trabajo (*Workspace*)

Para poder trabajar con ROS, se debe primero tener un espacio de trabajo (*workspace*), el cual contendrá todo aquello que será desarrollado. En este laboratorio, se creará un espacio de trabajo llamado `lab_ws` en el directorio base. Para ello, abrir un terminal e ingresar los siguientes comandos:

```
$ cd ~
$ mkdir -p lab_ws/src
```

Lo que estos comandos hacen es ir al directorio base (`cd`) y crear una carpeta (`mkdir`) llamada `lab_ws` que a su vez contiene la carpeta `src`. Una vez creadas estas carpetas, es necesario ir a la carpeta `src`, que será la que contendrá todo el código fuente, e inicializar el nuevo espacio de trabajo (`catkin_init_workspace`). Esto se realiza de la siguiente manera:

```
$ cd lab_ws/src
$ catkin_init_workspace
```

Seguidamente se debe compilar el espacio de trabajo, a pesar de que está (por el momento) vacío, ya que ello creará archivos que son necesarios. Para ello, se va a la carpeta del espacio de trabajo (`lab_ws`) y se usa el comando `catkin_make`:

```
$ cd ~/lab_ws
$ catkin_make
```

Finalmente, se debe hacer que el espacio de trabajo creado esté visible para ROS. Esto se consigue ejecutando el script `setup.bash` que se encuentra en la carpeta `devel` (se ejecuta usando el comando `source`). Para que esta ejecución se realice cada vez que se abre un nuevo terminal, se añade la instrucción al `.bashrc` (que es el programa que se ejecuta al abrir un nuevo terminal) del siguiente modo:

```
$ echo "source ~/lab_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Luego de esto, cerrar el terminal y abrir uno nuevo. Para verificar que todo marcha bien, al escribir el comando `roscd` se debe llegar a la dirección `~/lab_ws/devel`.

4.2. Creación de un Paquete de ROS

Todo el código que se utiliza debe estar contenido en un paquete. Para poder trabajar en este laboratorio se creará un paquete llamado `lab1` de la siguiente forma:

```
$ cd ~/lab_ws/src
$ catkin_create_pkg lab1 std_msgs rospy
```

Este paquete se crea con 2 dependencias: `std_msgs`, utilizado para mensajes estándar, y `rospy`, necesario para crear nodos en Python (si se fuese a crear nodos en C++ sería

necesario añadir la dependencia `roscpp`). El paquete por defecto contiene la carpeta `src`, que es donde se almacenará el código fuente de Python, y los archivos `package.xml` y `CMakeLists.txt`. El archivo `CMakeLists.txt` contiene toda la información necesaria para la compilación del paquete y está basado en el estándar utilizado para *cmake*. Para más información sobre paquetes, ver el apéndice A.5 (opcional).

4.3. Creación de un Nodo en Python

En ROS los programas ejecutables se denominan *nodos* y pueden estar escritos en diversos lenguajes. Un ejemplo de nodo en Python es el siguiente, el cual mostrará un mensaje en pantalla. Dentro de la carpeta `src` del paquete `lab1` crear un archivo llamado `nodo1` y hacerlo ejecutable. Los comandos para ello son:

```
$ roscd lab1
$ cd src
$ touch nodo1
$ chmod a+x nodo1
```

El archivo `nodo1` contendrá el siguiente programa de Python.

nodo1

```
#!/usr/bin/env python

import rospy

if __name__ == "__main__":

    rospy.init_node("nodo1")

    print "Hola UTEC!"
    rospy.loginfo("Mi mensaje normal :)")
    rospy.logwarn("Mi mensaje de advertencia :P")
    rospy.logerr("Mi mensaje de error :(")

    rospy.spin()
```

Este archivo puede ser creado con cualquier editor de texto (kate, sublime, gedit, emacs, vim, etc). Las instrucciones importantes del programa `nodo1` son:

- `rospy`: siempre se debe importar para un nodo de ROS en Python
- `init_node`: inicializa un nodo ROS con el nombre *nodo1*. Es importante notar que el nombre del nodo debe ser único.
- `loginfo, logwarn, logerr`: muestran mensajes con diferentes colores, dependiendo de la severidad del mensaje.
- `spin`: realiza un bucle interno que termina al presionar `Ctrl+c` o cuando el ROS Master es finalizado.

Ejecución. Para ejecutar este nodo primero se debe tener un *ROS Master*, el cual se crea al ejecutar *roscore* en un terminal. Para esto, abrir un nuevo terminal y usar el siguiente comando:

```
$ roscore
```

Luego, desde otro terminal, ejecutar el nodo de la siguiente manera:

```
$ rosrun lab1 nodo1
```

Este comando especifica primero el nombre del paquete (lab1) y luego el nombre del nodo (nodo1). Si todo va bien, se debe ver tres mensajes con distintos colores en el terminal. Para finalizar el programa, se presiona `Ctrl+c` en el respectivo terminal. Para finalizar el *ROS Master* es necesario detener el comando *roscore*, para lo cual se va al terminal donde fue lanzado y se presiona `Ctrl+c`.

Comandos Adicionales para Nodos. Los siguientes comandos proporcionan información útil sobre los nodos.

- `rosnode list`: brinda una lista de todos los nodos activos
- `rosnode info nombre_nodo`: brinda información sobre un nodo específico

Es usual ejecutar estos comandos desde un nuevo terminal (mientras el nodo no haya aún sido detenido).

4.4. Robot en ROS

Los robots en ROS se definen mediante modelos URDF. En esta parte se mostrará el modelo URDF del robot *LBR iiwa* de *KUKA*. Este modelo se encuentra en repositorio provisto por el fabricante `https://github.com/ros-industrial/kuka_experimental`. De aquí solo se usará la carpeta `kuka_lbr_iiwa_support` que es la que contiene el modelo propiamente dicho. Para esto, ejecutar los siguientes comandos:

```
$ cd /tmp
$ git clone https://github.com/ros-industrial/kuka_experimental
$ cd ~/lab_ws/src
$ mkdir kuka
$ cd kuka
$ mv /tmp/kuka_experimental/kuka_lbr_iiwa_support .
```

Visualización en RViz. Dentro del paquete creado anteriormente (`lab1`), crear una carpeta llamada `launch` para mostrar el modelo en RViz (un visualizador de ROS), y una carpeta llamada `config` para almacenar la configuración.

```
$ roscd lab1
$ mkdir launch config
```

Dentro de la carpeta `launch` crear un archivo llamado `display_iiwa.launch` con el siguiente contenido.

display_iiwa.launch

```
<?xml version="1.0"?>
<launch>

  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
      kuka_lbr_iiwa_support)/urdf/lbr_iiwa_14_r820.xacro'" />

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="
      joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher" />

  <arg name="gui" default="True" />
  <param name="use_gui" value="$(arg gui)"/>

  <arg name="config_file" value="$(find lab1)/config/iiwa.rviz"/>
  <node name="rviz" pkg="rviz" type="rviz" respawn="false" output="screen" args
      ="-d $(arg config_file)"/>

</launch>
```

Este archivo de lanzamiento `display_iiwa.launch` tiene las siguientes partes:

- `robot_description`. Es un parámetro que almacena el modelo URDF del robot. Se le debe indicar el camino al modelo del robot (en este caso al modelo xacro, que genera el modelo urdf).
- `joint_state_publisher`. Crea *sliders* para cada articulación y publica los valores articulares en el tópico `joint_state`.
- `robot_state_publisher`. Lee los valores angulares del tópico `joint_state` y los convierte en transformaciones de posición y orientación para cada cuerpo rígido del robot en formato `tf`.
- `gui`. Si es verdadero, se muestra los *sliders*. De lo contrario estos no se muestran.
- `config_file`. Este archivo contendrá la información relacionada con la ventana de RViz de tal modo que la siguiente vez mantenga las mismas propiedades que tenía en la anterior sesión.
- `rviz`. Lanza *rviz* con el nodo `rviz`, y pertenece al paquete del mismo nombre.

Luego, ejecutar el archivo creado (*lanzarlo*) con el siguiente comando:

```
$ roslaunch lab1 display_iiwa.launch
```

Esto abrirá una ventana de *RViz*, un visualizador de ROS, pero por defecto no se muestra nada. Se debe añadir un robot del siguiente modo: Add → RobotModel. Luego, en la parte izquierda, cambiar *Fixed Frame* de `map` a `base`. Utilizar el mouse para llegar a una vista similar a la mostrada en la Figura 1 y guardar la configuración: File → Save Config.

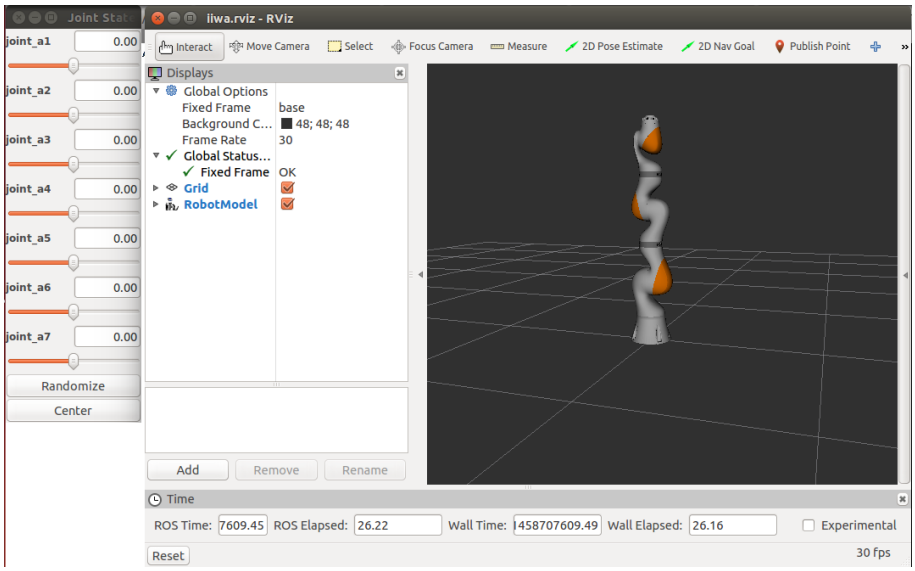


Figura 1: Robot LBR iiwa de KUKA en RViz

Con los *sliders* del costado se puede mover cada una de las articulaciones del robot manipulador. Probar moviendo las articulaciones. Tomar en cuenta que el modelo mostrado es puramente cinemático y no toma en cuenta los pesos o las fuerzas ejercidas. Tampoco toma en cuenta las colisiones. Para cerrar el modelo, presionar `Ctrl+c` en la ventana donde se ejecutó `roslaunch`.

Se puede descargar modelos de diferentes robots manipuladores, móviles, humanoides, aéreos, entre otros, en `https://wiki.ros.org/Robots`. Para cualquier robot, el archivo para lanzarlo es similar al anterior, si el modelo URDF está expresado como xacro.

Sistemas de Coordenadas. Cada una de las partes del robot tiene asociada un sistema de coordenadas fijo. Para observarlo, añadir elementos de tipo *TF*. Para esto, ir a `Add` → `TF`. En el panel izquierdo, dentro de `TF` es recomendable deseleccionar las opciones `Show Names` y `Show Arrows` y solo dejar seleccionada `Show Axes`. Por convención, el orden de ejes se da en formato *RGB*: el eje rojo (red) es x, el eje verde (green) es y y el eje azul (blue) es z. Se puede ver cómo al mover los sliders, estos sistemas también se mueven.

Preguntas.

1. ¿Cuántos grados de libertad tiene el robot LBR iiwa de KUKA?
2. El robot LBR iiwa tiene una configuración antropomórfica que emula un hombro, un codo y una muñeca. ¿Cuántos grados de libertad tiene en el hombro, cuántos en el codo, y cuántos en la muñeca?
3. Con base en el modelo del robot (mostrado en RViz), ¿cuáles son los límites articulares del robot?
4. Con el movimiento de sliders, encontrar alguna configuración en la cual la orientación del sistema asociado al efector final con respecto al sistema de la base esté dado por:

$$R = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Brindar la configuración angular (valores de cada ángulo) así como una captura de pantalla de la configuración.

4.5. Envío de Configuración Angular desde un Nodo

En esta parte se busca enviar los ángulos al robot ya no desde sliders sino desde un programa de Python (un nodo). Dentro de la carpeta *src* del paquete creado (`lab1`) crear un archivo denominado `send_joints` y hacerlo ejecutable. Copiar el siguiente código en dicho archivo:

send_joints

```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import JointState

if __name__ == "__main__":

    rospy.init_node("sendJointsNode")
    pub = rospy.Publisher('joint_states', JointState, queue_size=1000)

    # Nombres de las articulaciones
    jnames = ("joint_a1", "joint_a2", "joint_a3", "joint_a4", "joint_a5",
              "joint_a6", "joint_a7")
    # Configuracion articular deseada (en radianes)
    jvalues = [0, -0.78, 0, 1.57, 0, 0.78, 0]

    # Objeto (mensaje) de tipo JointState
    jstate = JointState()
    # Asignar valores al mensaje
    jstate.header.stamp = rospy.Time.now()
    jstate.name = jnames
    jstate.position = jvalues

    # Frecuencia del envio (en Hz)
    rate = rospy.Rate(100)
    # Bucle de ejecucion continua
    while not rospy.is_shutdown():
        # Tiempo actual (necesario como indicador para ROS)
        jstate.header.stamp = rospy.Time.now()
        # Publicar mensaje
        pub.publish(jstate)
        # Esperar hasta la siguiente iteracion
        rate.sleep()
```

Este nodo crea un mensaje estándar de tipo `JointState` que contiene los nombres de las articulaciones y su valor angular. Este mensaje es luego continuamente publicado en un tópico de nombre `/joint_states`. Para esto, se declara un elemento publicador (de tipo `rospy.Publisher`) con el nombre `pub`. La forma en la que trabaja ROS es a través del envío y recepción de mensajes en un *tópico*: uno o más publicadores (*publisher*) publican un mensaje en un tópico determinado y uno o más suscriptores (*subscriber)* se suscriben al tópico para leer los mensajes. En este caso, el nodo creado publica el valor articular en el tópico `/joint_states` para que RViz obtenga los datos angulares desde dicho tópico.

Para que pueda notarse la utilidad se requiere ejecutar RViz para visualizar el modelo del robot.

Envío a RViz. Es necesario lanzar RViz pero a diferencia de lo que se realizó antes, ahora no se desea tener un `joint_state_publisher` (los sliders) que genere los valores articulares ya que estos estarán dados por el nodo anterior. Así, dentro de la carpeta `launch` del paquete `lab1` crear un archivo llamado `display_iiwa_v2.launch` con el siguiente código:

display_iiwa_v2.launch

```
<?xml version="1.0"?>
<launch>

  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
      kuka_lbr_iiwa_support)/urdf/lbr_iiwa_14_r820.xacro'" />

  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher" />

  <arg name="config_file" value="$(find lab1)/config/iiwa.rviz"/>
  <node name="rviz" pkg="rviz" type="rviz" respawn="false" output="screen" args
      ="-d $(arg config_file)"/>

</launch>
```

Este *launch file* ejecuta un nodo de tipo `robot_state_publisher` que se suscribe automáticamente al tópico `/joint_states` y publica automáticamente las posiciones y orientaciones de los cuerpos rígidos que conforman el robot en el tópico `/tf`. Luego, RViz de manera automática se suscribe al tópico `/tf` para actualizar los valores de cada elemento del robot.

Ejecutar el nodo de ROS junto con RViz a través de los siguientes comandos, cada uno en un terminal diferente (notar que si `roscore` no ha sido lanzado, `roslaunch` lo lanza automáticamente)

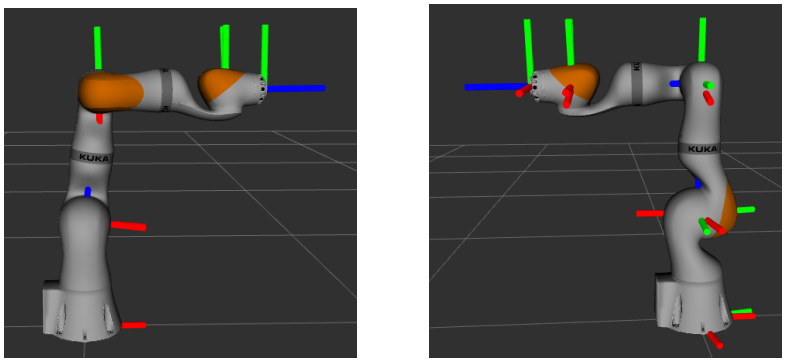
```
$ roslaunch lab1 display_iiwa_v2.launch
$ rosrun lab1 send_joints
```

Observar la ventana de RViz: ahora el brazo del robot tiene una configuración diferente a la mostrada anteriormente. Cambiar valores de `jvalues` en el archivo de Python `send_joints` y observar los cambios en RViz (cada vez que se realiza un cambio se debe volver a ejecutar el nodo).

Pregunta.

1. Para todos los nodos que están ejecutándose (excepto para `/rosout`) especificar todos los tópicos a los cuales están suscritos y en los cuales están publicando.

2. Con base en la información anterior, especificar cómo llega la información de los ángulos desde el nodo creado en Python hasta RViz.



(a) Configuración 1 (b) Configuración 2

Figura 2: Configuraciones deseadas: se desea iniciar con la configuración 1 y terminar en la 2.

Problema. La Figura 2 muestra dos configuraciones diferentes del robot LBR iiwa, una con el efector final hacia la derecha y la otra con el efector final hacia la izquierda.

1. Utilizando los *sliders*, encontrar dos configuraciones articulares (una para cada caso) que asemejen las configuraciones mostradas en la figura. Reportar los valores articulares encontrados y una captura de pantalla en cada caso.

2. Crear un nodo que comience con la configuración 1 y termine en la configuración 2 haciendo una interpolación lineal de cada articulación. Utilizar más de 100 puntos de interpolación para cada articulación (y variar la frecuencia de envío para que el movimiento no sea demasiado lento ni demasiado rápido). El nodo estará en la carpeta `src` del paquete `lab1` y se denominará `send_joints_cont`. En el reporte incluir el programa y capturas de pantalla del movimiento obtenido.

5. Simulación en Gazebo

Gazebo es un simulador dinámico *open source* que permite simular robots en ambientes complejos de interiores y exteriores. En esta parte del laboratorio se realizará la simulación del robot *LBR iiwa de KUKA*. La configuración y descripción dinámica del robot, necesarios para la simulación en Gazebo, son provistos como paquetes de ROS. Para tener acceso a los mismos se clonará el repositorio que los contiene pero solamente se utilizará los paquetes `iiwa_control`, `iiwa_description` e `iiwa_gazebo`, que corresponden directamente a la simulación. Para ello, ejecutar los siguientes comandos:

```
$ cd /tmp
$ git clone https://github.com/SalvoVirga/iiwa_stack
$ cd ~/lab_ws/src/kuka
$ mv /tmp/iiwa_stack/iiwa_control /tmp/iiwa_stack/iiwa_description .
$ mv /tmp/iiwa_stack/iiwa_gazebo .
```

La ejecución de la simulación utiliza archivos de tipo *launch* desde donde se carga las partes necesarias. Sin embargo para el caso de Gazebo la creación de estos archivos es más compleja debido a que se requiere más parámetros de configuración (como, por ejemplo, la especificación de los controladores de cada una de las articulaciones)

dado que la simulación incluye la dinámica del sistema. Además, se requiere la instalación de controladores y paquetes de ROS tales como `ros_control`, `controller_manager`, `gazebo_ros_control`, `gazebo_ros_pkgs`, `position_controllers joint_limits_interface`, `joint_state_controller`, `joint_trajectory_controller` (que se puede hacer usando el comando `apt-get install`). En este laboratorio, los paquetes necesarios ya han sido instalados. Para mayor rapidez, descargar (del aula virtual) el archivo comprimido *iiwa_helpers.zip* y luego:

1. Copiar los archivos `iiwa_gazebo.launch` y `iiwa_world.launch` en la carpeta `launch` del paquete `lab1`.

2. Crear una carpeta llamada `worlds` dentro de `lab1` y copiar el archivo `iiwa.world` dentro de esta carpeta.

Una vez copiados los archivos se puede ejecutar la simulación. Para ello, escribir el siguiente comando

```
$ roslaunch lab1 iiwa_gazebo.launch
```

Se debe observar una escena semejante a la mostrada en la Figura 3. En este caso no hay sliders y no se puede modificar las articulaciones manualmente.

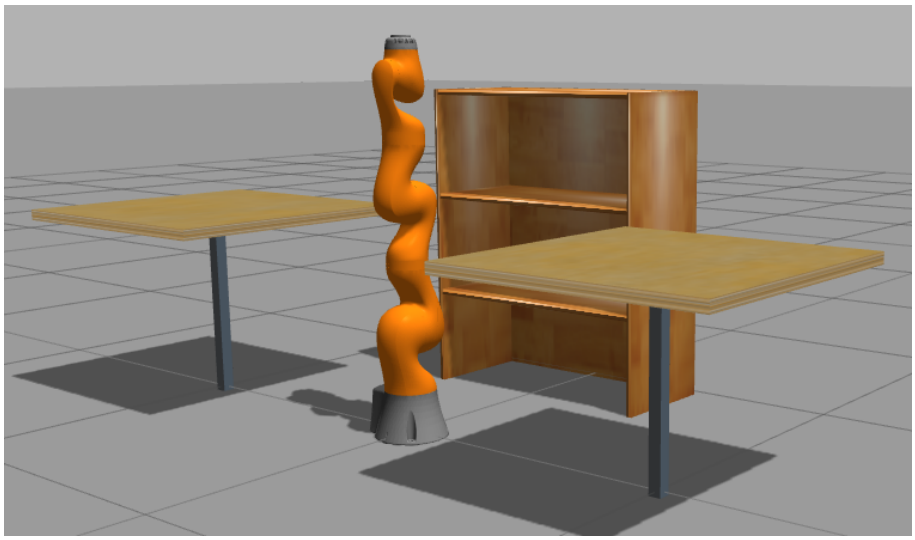


Figura 3: Robot LBR iiwa de KUKA en Gazebo junto a mesas y un armario

El robot en Gazebo asume el comportamiento de un robot real, en este caso controlado por posición. Es decir, se debe enviar posiciones (o una trayectoria) para que el robot comande el bajo nivel de sus motores para alcanzar dicha configuración articular. Las configuraciones articulares se enviarán mediante el siguiente programa de Python que se llamará `send_joints_gz`, y que debe ser colocado en `src` del paquete `lab1`. Recordar que se debe hacer que el script creado sea ejecutable (usando `chmod`).

send_joints_gz

```
#!/usr/bin/env python

import rospy
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint

if __name__ == "__main__":

    rospy.init_node("sendJointsGzNode")
    topic = '/iiwa/PositionJointInterface_trajectory_controller/command'
    pub = rospy.Publisher(topic, JointTrajectory, queue_size=1000, latch=True)

    # Nombres de las articulaciones
    jnames = ('iiwa_joint_1', 'iiwa_joint_2', 'iiwa_joint_3',
              'iiwa_joint_4', 'iiwa_joint_5', 'iiwa_joint_6',
              'iiwa_joint_7')

    # Configuraciones articulares deseadas (3, en este caso)
    jvalues = []
    # Comenzar con articulaciones en cero
    jvalues.append( (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) )
    # Modificar los valores siguientes:
    jvalues.append( (-1.0, 1.0, 2.0, 0.5, 0.5, 0.5, 0.5) )
    jvalues.append( (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) )
    jvalues.append( (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) )
    # Duraciones para cada valor articular
    jtime = (3.0, 6.0, 9.0, 12.0)

    # Objeto (mensaje) de tipo JointState
    jtrajectory = JointTrajectory()
    jtrajectory.header.stamp = rospy.Time.now()
    jtrajectory.joint_names = jnames
    jtrajectory.points = ()

    for i in range(len(jvalues)):
        p = JointTrajectoryPoint()
        p.positions = jvalues[i]
        p.time_from_start = rospy.Duration(jtime[i])
        jtrajectory.points += (p,)

    # Publicar la trayectoria
    pub.publish(jtrajectory)

    rate = rospy.Rate(10)
    while not rospy.is_shutdown():
        rate.sleep()
```

Para ejecutar el programa, mientras se tiene la simulación de gazebo corriendo, desde otro terminal ejecutar:

```
$ rosrun lab1 send_joints_gz
```

Se observará las articulaciones del robot moviéndose a la posición indicada debido a que las posiciones representan valores deseados que son enviados a los controladores de cada articulación del robot.

Problema. Modificar las 3 posiciones articulares deseadas (excepto la primera que se mantendrá siempre en cero) de tal modo que el robot se mueva primero hacia la mesa delantera, luego hacia el armario y finalmente hacia la mesa posterior (en ese orden). Las configuraciones que se obtuvieron para la Figura 2 pueden ser de ayuda. Indicar las configuraciones usadas y algunas capturas de pantalla.

Preguntas.

1. ¿Resulta sencillo obtener posiciones deseadas (o modificar las posiciones deseadas) usando el espacio articular? ¿Por qué cree que pasa eso?

2. ¿Qué se puede hacer para que sea más fácil controlar la posición del robot? Escribir alguna(s) idea(s).

6. Conclusiones

Elaborar algunas conclusiones sobre el laboratorio desarrollado e indicarlas en el informe del laboratorio.

Referencias

- Joseph, Lentin. *Mastering ROS for Robotics Programming*. Packt Publishing Ltd. 2015.

- Quigley, Morgan, Brian Gerkey, and William D. Smart. *Programming Robots with ROS*. O'Reilly. 2015.

- Tutoriales de ROS: `http://wiki.ros.org/ROS/Tutorials`

A. Apéndice: Introducción a ROS

A.1. ¿Qué es ROS?

ROS significa *Robot Operating System* y es una plataforma de desarrollo de aplicaciones en robótica que provee características como comunicación de mensajes, computación distribuida, reuso de código, entre otras. La filosofía es poder elaborar programas versátiles que puedan funcionar con diferentes robots realizando solo cambios menores al código.

El proyecto ROS comenzó en el 2007 como parte del *Stanford AI Robot Project* del laboratorio de Inteligencia Artificial de la Universidad de Stanford. En el 2008 su desarrollo pasó a *Willow Garage*, un instituto e incubadora de investigación en robótica, pero con ayuda continua de reconocidas instituciones (de investigación e industriales) a nivel internacional. Desde el 2013, ROS es mantenido permanentemente por la OSRF cuyas siglas en inglés significan *Open Source Robotics Foundation*.

La comunidad de ROS ha experimentado un crecimiento muy rápido y actualmente cuenta con una gran cantidad de usuarios y desarrolladores a nivel mundial. La mayoría de las compañías y laboratorios de investigación en robótica están ahora portando su software a ROS. Esta tendencia también es visible en robots industriales, donde compañías están paulatinamente migrando de aplicaciones propietarias a ROS. El movimiento llamado *ROS Industrial* se ha incrementado en estos últimos años y su objetivo, básicamente, consiste en extender las capacidades avanzadas de ROS a la manufactura. Las aplicaciones cada vez mayores de ROS pueden generar muchas oportunidades en esta área. Más aún, de acuerdo con las tendencias, el conocimiento de ROS se hace cada vez más importante en el área de robótica tanto en investigación como en industria.

A.2. ¿Por qué utilizar ROS en robótica?

Algunas de las razones para preferir el uso de ROS sobre otras plataformas usadas en robótica (como YARP, Orocos, entre otras) son las siguientes:

- *Paquetes especializados.* Existen muchos paquetes en ROS que brindan diversas capacidades y que son fácilmente integrables con cualquier robot. ROS brinda muchas herramientas para realizar depuración, visualización y simulación.
- *Soporte para sensores y actuadores.* ROS viene con drivers y paquetes de interface para varios sensores y actuadores comunmente usados en robótica. Por ejemplo, lidars, escáneres láser, Kinect, servos Dynamixel, etc. Además, se puede realizar la interfaz de componentes con ROS de manera sencilla.
- *Operatibidad inter-plataforma.* La forma de transmitir mensajes en ROS permite la comunicación de diferentes nodos, los cuales pueden ser programados en cualquier lenguaje que tenga librerías de clientes para ROS (como C/C++, Python, Java, Matlab, entre otros).
- *Modularidad.* Un problema en muchas aplicaciones robóticas es que si un hilo del programa principal colapsa, toda la aplicación se detiene. En ROS, se trabaja de forma paralela y distribuida: si un nodo se detiene, el sistema sigue trabajando.
- *Manejo de recursos concurrentes.* El manejo de recursos de hardware por más de un proceso es normalmente un problema y se puede hacer complicado. La forma en la que ROS funciona, a través de nodos, reduce la complejidad computacional y se incrementa la mantenibilidad del sistema.

- *Comunidad activa.* ROS es una plataforma de código abierto y tiene un soporte continuo de la comunidad en robótica internacional tanto de la industria como de la parte académica (Ver, por ejemplo, `http://answers.ros.org`).

A.3. Filosofía de ROS

En sentido general, se puede decir que ROS sigue la filosofía de Linux/GNU de desarrollo de software en sus aspectos claves. Esto hace que ROS sea "natural" para desarrolladores familiarizados con Linux pero algo "críptico" para quienes están acostumbrados a utilizar ambientes de desarrollo gráfico en Windows o Mac OS X. La filosofía bajo la cual ROS fue creado está basada en los siguientes aspectos principales.

- *Peer to peer.* Los sistemas en ROS están formados por pequeños programas (nodos) interconectados unos con otros a través del intercambio continuo de mensajes.
- *Orientación hacia las herramientas.* Tareas como navegar el código, visualizar las interconexiones del sistema, graficar datos, generar documentación, almacenar datos, etc. son desarrollados por programas separados. Esto motiva la creación de herramientas especializadas.
- *Multilenguaje.* Muchas tareas se realizan con lenguajes script como Python. Sin embargo, para mayor eficiencia, se utiliza lenguajes como C++. También se puede usar Lisp o MATLAB. Más aún, los módulos de ROS pueden correr y comunicarse entre sí independientemente del lenguaje en el que hayan sido escritos. Existen clientes ROS para C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia, entre otros.
- *Pequeño.* Las convenciones de ROS motivan a crear librerías independientes y realizar wrappings para estas librerías de tal modo que puedan enviar y recibir mensaje a través de módulos de ROS. Esto permite mayor reusabilidad del software.
- *Libre y Open Source.* ROS se encuentra bajo la licencia BSD que permite uso comercial y no comercial. ROS pasa datos entre módulos utilizando comunicación interproceso (ICP), de tal modo que los sistemas pueden tener libertad sobre varios componentes.

A.4. Instalación

La plataforma soportada por defecto por ROS es Ubuntu, donde la instalación es muy sencilla (usando apt-get). También es posible su instalación y funcionamiento en otras distribuciones de Linux (como, por ejemplo, Debian) aunque la instalación en este caso es menos trivial. Las diversas versiones de ROS son compatibles con distribuciones específicas; por ejemplo, ROS Indigo es compatible con Ubuntu 14.04. Instrucciones sobre cómo instalar ROS, así como combinaciones de versiones específicas, pueden ser encontradas en `http://wiki.ros.org/indigo/Installation`.

A.5. Paquetes de ROS

Los paquetes constituyen la unidad principal de ROS y son el lugar donde se almacenan programas y librerías específicos. Se encuentran dentro de la carpeta `src` del espacio de trabajo (*workspace*). Por lo general son modulares y contienen funciones específicas, para un fin determinado, de tal modo que puedan fácilmente integrarse con otros paquetes. Son

los paquetes los que brindan la modularidad a ROS. Para crear un paquete se utiliza el comando `catkin_create_pkg` seguido del nombre del paquete y de las dependencias. La sintaxis de este comando es la siguiente:

```
catkin_create_pkg  nombre_paquete  dependencia_1 dependencia_2  ...
```

donde se puede especificar cuantas dependencias sean necesarias. Para poder utilizar este comando se debe estar dentro de la carpeta `src` del espacio de trabajo. Los paquetes tienen diversas dependencias según las herramientas (los otros paquetes) que vayan a utilizar. La idea siempre es reutilizar algo que ya esté hecho.

Los paquetes cuya dependencia es `roscpp` están hechos para C++ y contienen por defecto dos carpetas vacías llamadas `include` y `src`. A su vez, la carpeta `include` contendrá una carpeta con el mismo nombre del paquete. En general, es una buena práctica localizar los archivos de cabecera (`.h, .hh, .hpp, .hxx`) dentro de este directorio. De igual manera, es una buena práctica localizar los archivos fuente (`.c, .cpp, .cc, .cxx`) dentro del directorio `src`.

Además de esto, los paquetes contienen un archivo llamado `package.xml` el cual está escrito usando sintaxis xml. Las líneas que comienzan con `<!--` y terminan con `-->` son comentarios. Es recomendable editar la versión, la descripción del paquete, especificar quién mantiene el paquete y la licencia bajo la cual se provee el paquete (principalmente si el paquete va a ser distribuido a otras personas). En este archivo, las líneas localizadas casi al final, con etiquetas `build_depend`, indican las dependencias en tiempo de compilación; y las líneas con etiquetas `run_depend` indican las dependencias en tiempo de ejecución (enlace). Aquí aparecerán las dependencias que fueron especificadas al crear el paquete. Cuando se desea añadir dependencias adicionales a un paquete, éstas pueden indicarse en esta parte de forma manual. Los paquetes también contienen un archivo de nombre `CMakeLists.txt`, el cual es utilizado por `cmake` para generar un archivo `Makefile`, necesario para la compilación del código. Dentro de cada paquete, el archivo `CMakeLists.txt` se edita de manera adecuada según los archivos de código que contiene el paquete.

Para poder manipular paquetes de manera más fácil y rápida se puede utilizar los siguientes comandos:

- `rospack find nombre_paquete`: encuentra el camino al paquete.
- `rospack depends nombre_paquete`: brinda las dependencias que tiene un paquete.
- `roscd nombre_paquete`: permite acceder directamente al paquete (independientemente del directorio en el que se esté).
- `rosls nombre_paquete`: muestra el contenido del paquete.

A.6. Creación de un Nodo en C++

Como se mencionó en la sección 4.3, los programas ejecutables son denominados *nodos* en ROS. Un ejemplo de nodo en C++ que realiza la misma función que el nodo de Python de la sección 4.3 se muestra a continuación. Dentro de la carpeta `src` del paquete `lab1` (es decir, en `~/lab_ws/src/lab1/src`) crear un archivo llamado `nodo1c.cpp` con el siguiente código:

nodo1c.cpp

```
#include "ros/ros.h"
#include <iostream>

int main(int argc, char **argv)
{
  ros::init(argc, argv,"nodo1c");
  ros::NodeHandle node;

  std::cout << "Hola UTEC" << std::endl;
  ROS_INFO("Mensaje normal");
  ROS_WARN("Mensaje de advertencia");
  ROS_ERROR("Mensaje de error");

  ros::spin();
  return 0;
}
```

Este archivo puede ser creado con cualquier editor de texto (gedit, kate, emacs, vim, sublime, etc) o con algún IDE como QtCreator o Eclipse. Las instrucciones importantes del programa `nodo1c.cpp` son:

- `ros/ros.h`: es la cabecera principal de ROS y siempre se debe incluir cuando se utiliza C++ con ROS
- `ros::init`: inicializa un nodo ROS con el nombre *nodo1*. Es importante notar que el nombre del nodo debe ser único.
- `ros::NodeHandle`: crea un objeto de tipo *Nodehandle*, el cual se utiliza para comunicarse con el sistema ROS.
- `ROS_INFO, ROS_WARN, ROS_ERROR`: muestran mensajes con diferentes colores, dependiendo de la severidad del mensaje.
- `ros::spin`: realiza un bucle interno que termina al presionar `Ctrl+c` o cuando el ROS Master es finalizado.

Compilación. Para compilar el programa se debe modificar el archivo `CMakeLists.txt` localizado dentro del directorio del paquete `lab1`. En este caso este archivo se localiza en `~/lab_ws/src/lab1/CMakeLists.txt`. Al abrir el archivo, se observará que hay contenido por defecto pero la mayor parte son comentarios (comienzan con #). El archivo `CMakeLists.txt` debe modificarse para que quede como se indica a continuación.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(lab1)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)
catkin_package()

include_directories(${catkin_INCLUDE_DIRS})

add_executable(nodo1c src/nodo1c.cpp)
target_link_libraries(nodo1c ${catkin_LIBRARIES})
```

La descripción de los comandos es la siguiente:

- `cmake_minimum_required`: versión mínima de *cmake* necesaria.
- `project`: indica el nombre del proyecto, en este caso, del paquete de ROS.
- `find_package`: carga la configuración de paquetes (indicados como dependencias).
- `include_directories`: indica el camino a los archivos de cabecera. En este caso, `${catkin_INCLUDE_DIRS}` utiliza las cabeceras por defecto.
- `add_executable`: genera los ejecutables. El primer nombre es el nombre del ejecutable; el segundo nombre es el archivo que contiene el código fuente. En este caso, el ejecutable se denominará `nodo1c`.
- `target_link_libraries`: indica las librerías contra las que se enlaza el ejecutable creado. En este caso, `${catkin_ LIBRARIES}` indica las librerías especificadas por defecto al cargar los paquetes.

Una vez modificado el `CMakeLists.txt`, la compilación se ejecuta desde el espacio de trabajo utilizando el comando `catkin_make` del siguiente modo:

```
$ cd ~/lab_ws
$ catkin_make
```

Si no hay errores, la compilación debe finalizar indicando 100 %, y el ejecutable especificado (`nodo1c`) se crea en `~/lab_ws/devel/lib/lab1`.

Ejecución. El ejecutable creado se denomina *nodo* de ROS. Para ejecutarlo se debe primero tener un *ROS Master*, el cual se crea al ejecutar *roscore* en un terminal. Para esto, abrir un nuevo terminal y usar el siguiente comando:

```
$ roscore
```

Luego, desde otro terminal, ejecutar el nodo de la siguiente manera:

```
$ rosrun lab1 nodo1c
```

Este comando especifica primero el nombre del paquete (lab1) y luego el nombre del nodo (nodo1). Si todo va bien, se debe ver tres mensajes con distintos colores en el terminal. Para finalizar el programa, se presiona `Ctrl+c` en el respectivo terminal. Para finalizar el *ROS Master* es necesario detener el comando *roscore*, para lo cual se va al terminal donde fue lanzado y se presiona `Ctrl+c`.

A.7. Mensajes en ROS

ROS tiene diversos tipos de mensajes por defecto, los cuales van desde mensajes simples hasta mensajes más complicados sirviendo para propósito general o propósitos específicos. Por convención, los paquetes que contienen definiciones de mensajes llevan el nombre terminado en *_msgs*. Así, se tiene por ejemplo:

- std_msgs
- sensor_msgs
- geometry_msgs

entre otros. Se puede ver estos mensajes usando `roscd` y localizando la carpeeta `msg`. Por ejemplo, para ver los mensajes estándar:

```
$ roscd std_msgs/msg
```

Se puede encontrar más información sobre mensajes en `http://wiki.ros.org/msg`.