Robot Operating System (ROS)

Lab 2: ROS Publishers and Subscribers

Haitham El-Hussieny, PhD

October 31, 2022

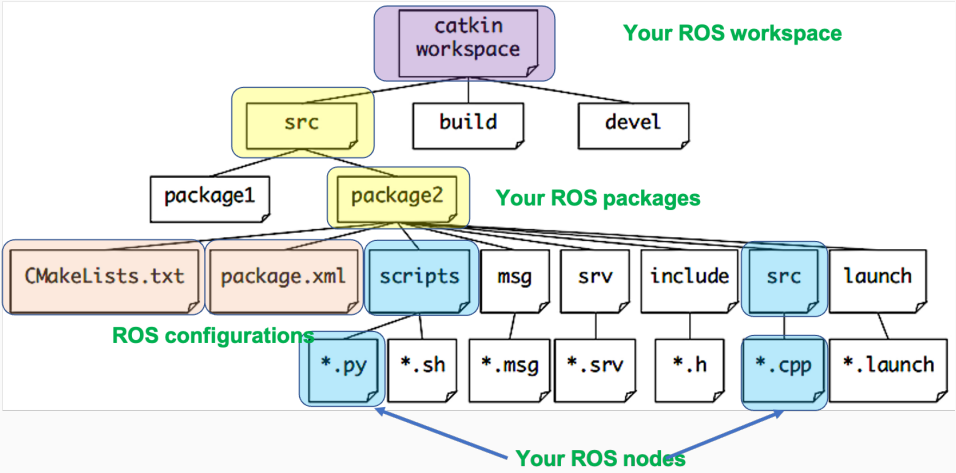
Department of Mechatronics and Robotics Engineering
Egypt-Japan University of Science and Technology (E-JUST)
Alexandria, Egypt.

Outline

ROS Workspace and Packages.

ROS Workspace and Packages.

How ROS workspace is organized?



CREATE A ROS WORKSPACE.

1. In your home directory:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

2. Check the crearted folders:

```
$ cd ~/catkin_ws/
$ ls
build  devel  src
```

3. Sourcing the setup.bash files:

```
$ source devel/setup.bash
```

Catkin:

catkin is the official build system of ROS and the successor to the original ROS build system, rosbuild. catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow.

CREATE A ROS PACKAGE.

1. Change directory to the **catkin_ws/src** folder:

```
$ cd ~/catkin_ws/src
```

2. Create a ROS package:

catkin_create_pkg <package_name> [depend1] [depend2] [depend3]

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

Dependendies:

- **std_msgs**: contains wrappers for ROS primitive types: int8, int16, bool, String, etc.
- **rospy**: a client API to enable Python programmers to quickly interface with ROS.
- **roscpp**: Enables C++ programmers to quickly interface with ROS.

2. Build the ROS package(s):

```
$ cd ~/catkin_ws
$ catkin_make
```

ROS Publisher/Subscriber Nodes in C++.

WRITING A C++ ROS PUBLISHER

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

  ros::Rate loop_rate(10);
  int count = 0;

  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

C++ Publisher Node

- This C++ node is created as a publisher named (”talker”) which will continually broadcast a **String** message.
- Drag the created package folder into VSCode to start writing your first ROS node inside **beginner_tutorial/src** folder.

C++ Publisher Exaplained.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

ros/ros.h includes all the headers necessary to use ROS system. **std_msgs/String** message enables to use String ROS messages.

C++ Publisher Exaplained.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

ros/ros.h includes all the headers necessary to use ROS system. **std_msgs/String** message enables to use String ROS messages.

```
ros::init(argc, argv, "talker");
```

Initialize ROS and specify the name of our node. Node names must be unique in a running system.

C++ Publisher Exaplained.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

ros/ros.h includes all the headers necessary to use ROS system. **std_msgs/String** message enables to use String ROS messages.

```
ros::init(argc, argv, "talker");
```

Initialize ROS and specify the name of our node. Node names must be unique in a running system.

```
ros::NodeHandle n;
```

Create a handle to this process' node.

C++ Publisher Exaplained.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Tell the master that we are going to be publishing a message of type **std_msgs/String** on the topic ***chatter***. The size of our publishing queue is 1000 messages before throwing them away if not recieved.

C++ Publisher Exaplained.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Tell the master that we are going to be publishing a message of type **std_msgs/String** on the topic ***chatter***. The size of our publishing queue is 1000 messages before throwing them away if not recieved.

```
ros::Rate loop_rate(10);
```

Specify the frequency of 10 Hz that you would like to loop at to send the message.

C++ Publisher Exaplained.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Tell the master that we are going to be publishing a message of type **std_msgs/String** on the topic ***chatter***. The size of our publishing queue is 1000 messages before throwing them away if not recieved.

```
ros::Rate loop_rate(10);
```

Specify the frequency of 10 Hz that you would like to loop at to send the message.

```
int count = 0;
while (ros::ok())
{
```

Allow interrupting the node by pressing CTRL-C key.

C++ Publisher Exaplained.

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

Broadcast a message on ROS using a message-adapted String class.

C++ Publisher Exaplained.

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

Broadcast a message on ROS using a message-adapted String class.

```
chatter_pub.publish(msg);
```

Broadcast the message through the topic to anyone who is connected.

C++ Publisher Exaplained.

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

Broadcast a message on ROS using a message-adapted String class.

```
chatter_pub.publish(msg);
```

Broadcast the message through the topic to anyone who is connected.

```
ROS_INFO("%s", msg.data.c_str());
```

The ROS version of *printf/cout* in C++.

C++ Publisher Exaplained.

```
ros::spinOnce();
```

It is necessary if the node is a publisher and subscriber at the same time. It allows executing callbacks for subscribers.

C++ Publisher Exaplained.

```
ros::spinOnce();
```

It is necessary if the node is a publisher and subscriber at the same time. It allows executing callbacks for subscribers.

```
loop_rate.sleep();
```

Sleep for the time remaining to have 10Hz publish rate.

WRITING A C++ ROS SUBSCRIBER

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"


void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "listener");

  ros::NodeHandle n;

  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

C++ Subscriber Node

- This C++ node is created as a subscriber named (”listener”) which will continually recieve the **String** message sent by the talker publisher.
- The subscriber node is created inside **beginner_tutorial/src** folder.

C++ Subscriber Exaplained.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
        ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

This is the callback function that will get called when a new message has arrived on the ***chatter*** topic.

C++ Subscriber Exaplained.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
        ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

This is the callback function that will get called when a new message has arrived on the ***chatter*** topic.

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Subscribe to the chatter topic with the master. ROS will call the chatterCallback() function whenever a new message arrives. The 2nd argument is the queue size, in case we are not able to process messages fast enough. In this case, if the queue reaches 1000 messages, we will start throwing away old messages as new ones arrive.

C++ Subscriber Exaplained.

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
        ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

This is the callback function that will get called when a new message has arrived on the ***chatter*** topic.

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Subscribe to the chatter topic with the master. ROS will call the chatterCallback() function whenever a new message arrives. The 2nd argument is the queue size, in case we are not able to process messages fast enough. In this case, if the queue reaches 1000 messages, we will start throwing away old messages as new ones arrive.

```
ros::spin();
```

ros::spin() enters a loop, calling message callbacks as fast as possible.

BUILDING THE C++ NODES.

1. Modification of the **package_name/CMakeLists.txt** file

1. Adding Dependencies in CMakeLists.txt file.
The default dependencies are added automatically when we created the ros package. Later, you need to add any additional packages.

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
)
```

BUILDING THE C++ NODES.

1. Modification of the **package_name/CMakeLists.txt** file

2. Defining executable nodes in CMakeLists.txt file.

This enables the creation of the executable files for the ROS nodes and link them to the ROS libraries.

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

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

BUILDING THE C++ NODES.

2. Modification of the **package_name/package.xml** file

Adding dependencies in package.xml file.
Defines the library requirements of catkin_make

```
<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>

<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

BUILDING THE C++ NODES.

1. Compile the ROS workspace:

```
$ cd ~/catkin_ws/
$ catkin_make
```

2. Run the **talker** node

```
$ rosrun beginner_tutorials talker
```

3. Run the **listener** node

```
$ rosrun beginner_tutorials listener
```

talker node

```
user:~$ rosrun beginner_tutorials talker
[ INFO] [1653086875.724053564]: hello world 0
[ INFO] [1653086875.824110956]: hello world 1
[ INFO] [1653086875.924118490]: hello world 2
[ INFO] [1653086876.024108687]: hello world 3
[ INFO] [1653086876.124106271]: hello world 4
[ INFO] [1653086876.224098654]: hello world 5
[ INFO] [1653086876.324073493]: hello world 6
[ INFO] [1653086876.424107418]: hello world 7
[ INFO] [1653086876.524116772]: hello world 8
[ INFO] [1653086876.624111979]: hello world 9
[ INFO] [1653086876.724093738]: hello world 10
[ INFO] [1653086876.824107015]: hello world 11
```

listener node

```
user:~$ rosrun beginner_tutorials listener
[ INFO] [1653086957.482505895]: I heard: [hello world 161]
[ INFO] [1653086957.582674158]: I heard: [hello world 162]
[ INFO] [1653086957.682317204]: I heard: [hello world 163]
[ INFO] [1653086957.782292262]: I heard: [hello world 164]
[ INFO] [1653086957.882232211]: I heard: [hello world 165]
[ INFO] [1653086957.982245414]: I heard: [hello world 166]
```

ROS Publisher/Subscriber Nodes in Python.

WRITING A PYTHON ROS PUBLISHER

```python
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Python Publisher Node

- Start writing your ROS Python publisher node inside **beginner_tutorial/scripts/talker.py** file.

PYTHON PUBLISHER EXAPLAINED.

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

Every Python ROS Node will have this declaration to make sure the script is executed as a Python script.

Python Publisher Exaplained.

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

Every Python ROS Node will have this declaration to make sure the script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

rospy is used for writing a ROS Node in Python and the std_msgs.msg imports String message type.

Python Publisher Exaplained.

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

Every Python ROS Node will have this declaration to make sure the script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

rospy is used for writing a ROS Node in Python and the std_msgs.msg imports String message type.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node ('talker', anonymous=True)
```

declares a publishing node with **chatter** topic using the message type String. The queue size is 10 messages, if not recieved they will be removed. The node name is done by the init_node method.

PYTHON PUBLISHER EXAPLAINED.

```
rate = rospy.Rate(10) # 10hz
```

for looping at the desired rate of 10 Hz.

PYTHON PUBLISHER EXAPLAINED.

```
rate = rospy.Rate(10) # 10hz
```

for looping at the desired rate of 10 Hz.

```
while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo( hello_str )
        pub.publish( hello_str )
        rate.sleep()
```

This loop is standard in rospy. It checks the rospy.is_shutdown() flag and then publish the message.

PYTHON PUBLISHER EXAPLAINED.

```
rate = rospy.Rate(10) # 10hz
```

for looping at the desired rate of 10 Hz.

```
while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo( hello_str )
        pub.publish( hello_str )
        rate.sleep()
```

This loop is standard in rospy. It checks the rospy.is_shutdown() flag and then publish the message.

```
try :
        talker ()
        except rospy.ROSInterruptException:
pass
```

catches if CTRL-C is pressed

WRITING A PYTHON ROS SUBSCRIBER.

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Python Subscriber Node

- Start writing your ROS Python subscriber node inside **beginner_tutorial/scripts/listener.py** file.

PYTHON SUBSCRIBER EXAPLAINED.

```
rospy.init_node ('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
rospy.spin()
```

This declares that your node subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, *callback* is invoked with the message as the first argument.

PYTHON SUBSCRIBER EXAPLAINED.

```
rospy.init_node ('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
rospy.spin()
```

This declares that your node subscribes to the chatter topic which is of type std_msgs.msgs.String. When new messages are received, *callback* is invoked with the message as the first argument.

```
def callback(data):
        rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

The callback for handling the recieved message.

BUILDING THE PYTHON NODES.

1. Change nodes mode to executable [only in Python]

```
$ cd
~/catkin_ws/src/beginner_tutorials/scripts
$ chmod a+x talker.py
$ chmod a+x listener.py
```

2. Compile the ROS workspace:

```
$ cd ~/catkin_ws/
$ catkin_make
```

3. Run the **talker** and **listener**

```
$ rosrun beginner_tutorials talker.py
$ rosrun beginner_tutorials listener.py
```

talker node

```
user:~$ rosrun beginner_tutorials talker.py
[INFO] [1653091387.113462]: hello world 1653091387.11
[INFO] [1653091387.213632]: hello world 1653091387.21
[INFO] [1653091387.313957]: hello world 1653091387.31
[INFO] [1653091387.413691]: hello world 1653091387.41
[INFO] [1653091387.513732]: hello world 1653091387.51
[INFO] [1653091387.613700]: hello world 1653091387.61
[INFO] [1653091387.713722]: hello world 1653091387.71
```

listener node

```
user:~$ rosrun beginner_tutorials listener.py
[INFO] [1653091761.276077]: I heard hello world 1653091761.28
[INFO] [1653091761.376341]: I heard hello world 1653091761.38
[INFO] [1653091761.475969]: I heard hello world 1653091761.48
[INFO] [1653091761.575958]: I heard hello world 1653091761.58
[INFO] [1653091761.676152]: I heard hello world 1653091761.68
[INFO] [1653091761.776017]: I heard hello world 1653091761.78
[INFO] [1653091761.876076]: I heard hello world 1653091761.88
[INFO] [1653091761.975920]: I heard hello world 1653091761.98
```

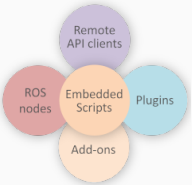
ROS nodes in Python don't require modification for CMakeLists.txt and package.xml files.

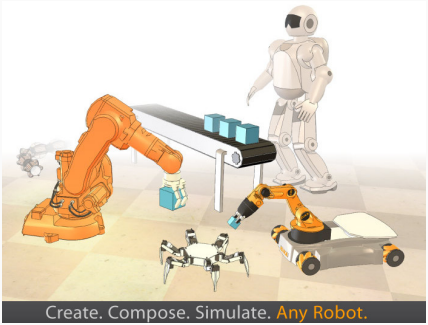
CoppeliaSim Simulation with ROS.

COPPELIASIM SIMULATION WITH ROS.

CoppeliaSim:

- A Physics-based simulator to simulate and control robots.
- It can work with existing models or import CAD models.





▸ CoppeliaSim Demo ▸ CoppeliaSim complete tutorial

COPPELIASIM SIMULATION WITH ROS.

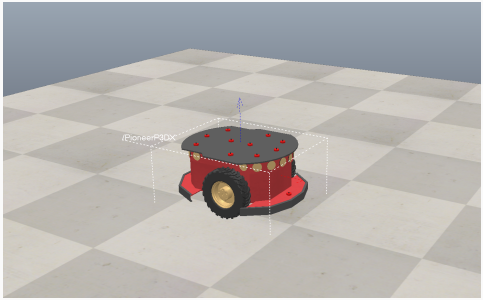
Download CoppeliaSim.

1. Download the Educational version suitable to your Ubuntu version.
2. Run the following commands into the terminal

```
$ sudo apt install
libboost-all-dev lua5.1
liblua5.1-0 liblua5.1-0-dev
qtcreator qt5-default
libqt5serialport5-dev
```

3. Extract the downloaded folder and run:

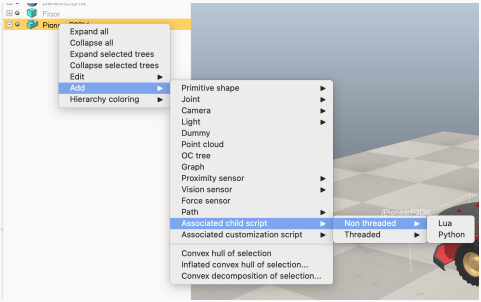
```
$ cd <directory-of-vrep>
$ ./coppeliasim.sh
```



COPPELIASIM SIMULATION WITH ROS.

Task one:
The task is to republish the command velocity message published from the turtle_telop_key node to the pioneer robot using an intermediate node called pioneer_interface.

- Drag a Pioneer3XD robot into the environment.
- Delete the attached script and add a non-threaded script to the robot (Lua language).



COPPELIASIM SIMULATION WITH ROS.

The script contains four essential functions:

```
function sysCall_init()
    -- do some initialization here
end

function sysCall_actuation()
    -- put your actuation code here
end

function sysCall_sensing()
    -- put your sensing code here
end

function sysCall_cleanup()
    -- do some clean-up here
end
```

COPPELIASIM SIMULATION WITH ROS.

sysCall_init(): Called once the simulation starts

```
function sysCall_init ()

    pioneer_robot = sim.getObjectHandle("/PioneerP3DX")
    rightWheel = sim.getObjectHandle("/PioneerP3DX/rightMotor")
    leftWheel = sim.getObjectHandle("/PioneerP3DX/leftMotor")

    if simROS then
        sim.addLog(sim.verbosity_scriptinfos ," ROS interface was found.")
        sub=simROS.subscribe('/pioneer/cmd_vel', 'geometry_msgs/Twist', 'subscriber_callback')
    else
        sim.addLog(sim.verbosity_scripterrors ," ROS interface was not found. Cannot run.")
    end

end
```

COPPELIASIM SIMULATION WITH ROS.

subscriber_callback(): Called once a Twist message received

```
function subscriber_callback(msg)
    -- This is the subscriber callback function

    lin_vel = msg.linear.x
    rot_vel = msg.angular.z

    L = 0.33 --m
    vLeft= lin_vel - ((L/2)* rot_vel )
    vRight= lin_vel + ((L/2)* rot_vel )

    sim.setJointTargetVelocity(leftWheel,vLeft)
    sim.setJointTargetVelocity(rightWheel,vRight)

end
```

COPPELIASIM SIMULATION WITH ROS.

sysCall_cleanup(): Called when simulation stopped

```
function sysCall_cleanup()
    if simROS then
        simROS.shutdownSubscriber(subscriber)

    end
end
```

COPPELIASIM SIMULATION WITH ROS.

ROS Python Interface:

- Create a new ROS package.

  `catkin_create_pkg pioneer_mover`

  `std_msgs roscpp rospy`

- Add a Python script pioneer_interface.py.

```python
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

def velovity_recieved_callback(robot_velocity):
    rospy.loginfo("The robot velocity is ({}, {})".format(robot_velocity.linear.x,
    robot_velocity.angular.z))
    robot_velocity.linear.x = robot_velocity.linear.x/4
    robot_velocity.angular.z = robot_velocity.angular.z/4

    velocity_pub.publish(robot_velocity)

def pioneer_init():

    rospy.init_node('pioneer_interface', anonymous=True)

    rospy.Subscriber("/turtle1/cmd_vel", Twist, velovity_recieved_callback)
    global velocity_pub
    velocity_pub = rospy.Publisher('/pioneer/cmd_vel', Twist, queue_size=10)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    pioneer_init()
```

COPPELIASIM SIMULATION WITH ROS.

Don't forget to add the goemtry_msgs in CMakeLists.txt and Package.xml files

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  geometry_msgs
)
```

CMakeList.txt

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
  <build_depend>geometry_msgs</build_depend>

<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
  <build_export_depend>geometry_msgs</build_export_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
  <exec_depend>geometry_msgs</exec_depend>
```

Package.xml

Finally make the pioneer_interface.py file executable

```
$ chmod a+x pioneer_interface.py
```

COPPELIASIM SIMULATION WITH ROS.

ROS Python Listener:

- Run the ROS master.

```
$ roscore
```

- Run the simulation.

```
$ cd <VREP folder>
$ ./coppeliasim.sh
```

- rosrun the listener node.

```
$ rosrun pioneer_mover pioneer_interface.py
```

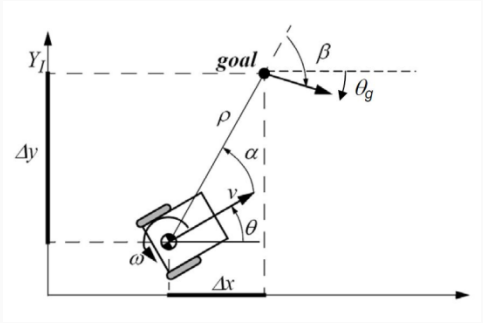
COPPELIASIM SIMULATION WITH ROS.

Task two:
The task is to do closed loop motion control to move the robot to a certain pose (x_g, y_g, θ_g)

- Drag a Pioneer3XD robot into the environment.
- Delete the attached script and add a non-threaded script to the robot (Lua language).
- A linear control low:

$$\boxed{v = k_\rho \rho, \qquad \omega = k_\alpha \alpha + k_\beta \beta}$$

$$k_\rho > 0,\ k_\beta < 0,\ k_\alpha - k_\rho > 0$$



$$\rho = \sqrt{\Delta x^2 + \Delta y^2}$$

$$\alpha = \text{atan2}(\Delta y, \Delta x) - \theta, \quad \alpha = [-\pi, \pi)$$

$$\beta = -\alpha - \theta + \theta_g$$

COPPELIASIM SIDE.

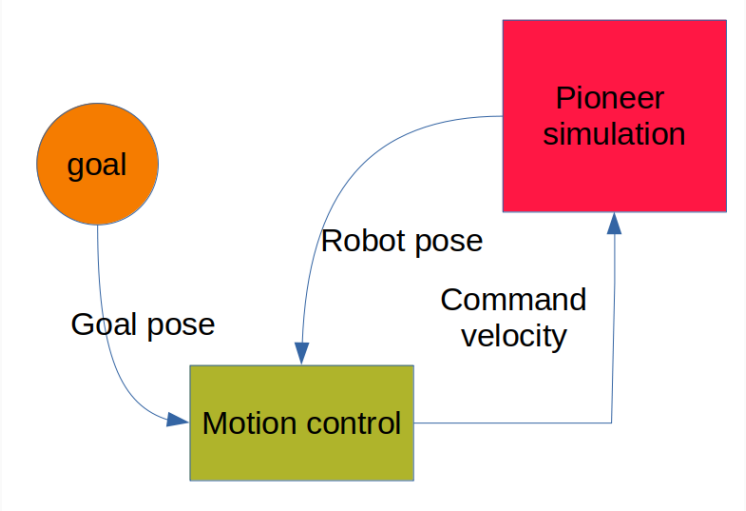
- In **sysCall_init()** add a publisher to publish the robot pose:

```
function sysCall_init()
    if simROS then
        pub=simROS.advertise('/pioneer/pose', 'geometry_msgs/Pose2D')
    else
        sim.addLog(sim.verbosity_scripterrors,"ROS interface was not found. Cannot run.")
    end
end
```

- In **sysCall_actuation()**: Continuously send robot pose

```
function sysCall_actuation()
    pos = sim.getObjectPosition(robot,-1)
    eulerAngles=sim.getObjectOrientation(robot, -1)
    alpha, beta, gamma= sim.yawPitchRollToAlphaBetaGamma(eulerAngles[1], eulerAngles[2], eulerAngles[3])
    local msg = {}
    msg['x'] = pos[1]
    msg['y'] = pos[2]
    msg['theta'] = alpha -- rotation around z
    simROS.publish(pub,msg)
end
```

PYTHON SIDE.


Pioneer simulation
goal
Robot pose
Command velocity
Goal pose
Motion control

PYTHON SIDE: INITIALIZATION

```python
def controller_init():
    rospy.init_node('turtle_closed_loop', anonymous=True)
    global k_rho, k_alpha, k_beta
    k_rho = 3.14 #rospy.get_param("k_rho")
    k_alpha = 5 #rospy.get_param("k_alpha")
    k_beta = -3 #rospy.get_param("k_beta")
    rospy.Subscriber("/pioneer/pose", Pose2D, pose_recieved_callback)
    rospy.Subscriber("/pioneer/goal", Pose2D, goal_recieved_callback)
    global velocity_pub

    velocity_pub = rospy.Publisher('/pioneer/cmd_vel', Twist, queue_size=10)
    rospy.spin()

if __name__ == '__main__':
    controller_init()
```

PYTHON SIDE: RECEIVING ROBOT'S POSE

```python
def pose_recieved_callback(turtle_pose):
    global x_turtle, y_turtle, theta_turtle

    x_turtle = turtle_pose.x
    y_turtle = turtle_pose.y
    theta_turtle = turtle_pose.theta
    rospy.loginfo("I heard ({:2f}, {:2f}, {:2f})".format(x_turtle, y_turtle, theta_turtle))
```

PYTHON SIDE: FEEDBACK CONTROL.

```
def goal_recieved_callback(turtle_goal):
    x_g = turtle_goal.x
    y_g = turtle_goal.y
    theta_g = turtle_goal.theta
    rho = np.sqrt(np.power(x_g - x_turtle,2) + np.power(y_g - y_turtle,2))
    turtle_speed = Twist()
    while not rho<0.1:
        rho = np.sqrt(np.power(x_g - x_turtle,2) + np.power(y_g - y_turtle,2))
        alpha = np.arctan2((y_g - y_turtle),(x_g - x_turtle)) -theta_turtle
        alpha = np.mod( alpha+np.pi, 2*np.pi) - np.pi
        turtle_speed.linear.x = k_rho*rho
        turtle_speed.angular.z = (k_alpha * alpha)
        velocity_pub.publish(turtle_speed)
        rospy.sleep(0.01)
    print("Reach position")
    beta = theta_turtle - theta_g
    while not np.abs(beta)<0.05:
        beta = theta_turtle - theta_g
        turtle_speed.linear.x = 0
        turtle_speed.angular.z = (k_beta * beta)
        velocity_pub.publish(turtle_speed)
        rospy.sleep(0.01)
    print("Reach orientation")
    turtle_speed.linear.x = 0
    turtle_speed.angular.z = 0
    velocity_pub.publish(turtle_speed)
```

COPPELIASIM SIMULATION WITH ROS.

ROS Python Listener:

- Run the ROS master.

```
$ roscore
```

- Run the simulation.

```
$ cd <VREP folder>
$ ./coppeliasim.sh
```

- rosrun the listener node.

```
$ rosrun closed_loop_turtle move_turtle_to_goal.py
```

End of Lecture

haitham.elhussieny@ejust.edu.eg