Robotics Operating System (ROS)

IMPORTANT Notes

- `roscore` should be run before anything. Read about roscore.

Table of Contents

Workspace

Configure Workspace

1. Add the following command into `.bashrc` file to activate the ROS default workspace

```
source /opt/ros/kinetic/setup.bash
```

Create Workspace

- Reference: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment
- catkin workspace will be used to create and store your own ROS packages (project).
- catkin is the name of the build tool used to compile and execute programs in ROS.

To create catkin workspace (catkin_ws) in HOME directory

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

Packages

> ROS package (project) that you will use to develop programs.

Create Package

1. Go to the `src` folder

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

2. Create your package (specify the dependencies)

```
catkin_create_pkg cs460_package std_msgs rospy roscpp
```

3. Go to catkin_ws and compile to generate executable and configuration files for the project

```
cd ..
catkin_make
```

Make The New Package The Default One

Add the following command in `.bashrc` (in HOME directory)

```
source /home/riotu/catkin_ws/devel/setup.bash
```

replace riotu by your username

Publishers and Subscribers

To understand ROS tobics READ

Get Publisher and Subscribers of a Tobic

```
$ rostopic info /chatter
Type: std_msgs/String
```

```
Publishers:
 * /talker_30225_1538066475268 (http://ubuntu:36549/)

Subscribers:
 * /listener_30271_1538066488910 (http://ubuntu:38827/)
```

You can get the available topics by typing

```
rostopic list
```

Read about rostopic

Publisher

▶ talker.py

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

def talker():
    #create a new publisher. we specify the topic name, then type of message then
the queue size
    pub = rospy.Publisher('chatter', String, queue_size=10)
    #we need to initialize the node
    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'talker' node
    rospy.init_node('talker', anonymous=True)
    #set the loop rate
    rate = rospy.Rate(1) # 1hz
    #keep publishing until a Ctrl-C is pressed
    i = 0
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % i
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()
        i=i+1

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

```
        pass
```

Run the publisher

```
$ rosrun ros_essentials_cpp talker.py
[INFO] [1538046986.881161]: hello world 0
[INFO] [1538046987.895407]: hello world 1
[INFO] [1538046988.882495]: hello world 2
[INFO] [1538046989.883143]: hello world 3
[INFO] [1538046990.882535]: hello world 4
...
```

Read about rosrun

Subscriber

▶ listener.py

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

def callback(message):
    #get_caller_id(): Get fully resolved name of local node
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", message.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node 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()
```

Run the subscriber

```
$ rosrun ros_essentials_cpp listener.py
[INFO] [1538047021.884116]: /listener_4579_1538047020918I heard hello world 35
[INFO] [1538047022.883483]: /listener_4579_1538047020918I heard hello world 36
[INFO] [1538047023.883734]: /listener_4579_1538047020918I heard hello world 37
[INFO] [1538047024.883639]: /listener_4579_1538047020918I heard hello world 38
...
```

Read about rosrun

Messages

- Syntax: package_name/message_type
- Unsigned int only takes positive values.

Set up Message

1. Create `msg` folder in the package_name (e.g. ros_service_assignment).

2. Create `IoTSensor.msg` file in `msg` folder.

3. Add the request and response values in `IoTSensor.msg` file. Example:

```
int32 id
string name
float32 temperature
float32 humidity
```

4. Add two dependencies in `package.xml`

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

5. Modify or add the following in `CMakeLists.txt`

- Add `message_generation`

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

- Uncomment and add `IoTSensor.msg` (`.msg` file)

```
## Generate messages in the 'msg' folder
add_message_files(
   FILES
   IoTSensor.msg
)
```

- Uncomment the following

```
## Generate added messages and services with any dependencies listed here
 generate_messages(
   DEPENDENCIES
   std_msgs
 )
```

- Uncomment and add `message_runtime`

```
######################################
## catkin specific configuration ##
######################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also
need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ros_service_assignment
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

6. Run in terminal

```
cd catkin_ws
catkin_make
```

7. Check if everything is fine

- It must contain `ros_cs460_package/IoTSensor`

```
$ rosmsg list
...
ros_cs460_package/IoTSensor
...
```

Use Message

▶ Details

iot_sensor_publisher.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from ros_cs460_package.msg import IoTSensor
import random

#create a new publisher. we specify the topic name, then type of message then the
queue size
pub = rospy.Publisher('iot_sensor_topic', IoTSensor, queue_size=10)

#we need to initialize the node
rospy.init_node('iot_sensor_publisher_node', anonymous=True)

#set the loop rate
rate = rospy.Rate(1) # 1hz
#keep publishing until a Ctrl-C is pressed
i = 0
while not rospy.is_shutdown():
    iot_sensor = IoTSensor()
    iot_sensor.id = 1
    iot_sensor.name="iot_parking_01"
    iot_sensor.temperature = 24.33 + (random.random()*2)
    iot_sensor.humidity = 33.41+ (random.random()*2)
    rospy.loginfo("I publish:")
    rospy.loginfo(iot_sensor)
    pub.publish(iot_sensor)
    rate.sleep()
    i=i+1
```

▶ iot_sensor_subscriber.py

```
#!/usr/bin/env python
import rospy
from ros_cs460_package.msg import IoTSensor

def iot_sensor_callback(iot_sensor_message):
    rospy.loginfo("new IoT data received: (%d, %s, %.2f ,%.2f)",
```

```
        iot_sensor_message.id,iot_sensor_message.name,
        iot_sensor_message.temperature,iot_sensor_message.humidity)

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

rospy.Subscriber("iot_sensor_topic", IoTSensor, iot_sensor_callback)

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

- Run the publisher

```
$ rosrun ros_cs460_package iot_sensor_publisher.py
[INFO] [1538065355.755153]: I publish:
[INFO] [1538065355.758797]: id: 1
name: "iot_parking_01"
temperature: 24.6894078555
humidity: 34.139884549
[INFO] [1538065356.756542]: I publish:
[INFO] [1538065356.758693]: id: 1
name: "iot_parking_01"
temperature: 26.2780855119
humidity: 33.8374695749
...
```

- Run the subscriber

```
$ rosrun ros_cs460_package iot_sensor_subscriber.py
[INFO] [1538065376.763998]: new IoT data received: (1, iot_parking_01, 25.33
,35.15)
[INFO] [1538065377.765967]: new IoT data received: (1, iot_parking_01, 25.77
,34.45)
...
```

Read about rosrun

Services

Set up Client/Server

1. Create `srv` folder in the package_name (e.g. ros_service_assignment).
2. Create `RectangleAeraService.srv` file in `srv` folder.
3. Add the request and response values in `RectangleAeraService.srv` file. Example:

```
float32 width
float32 height
---
float32 area
```

- First part (before `---`) is the request part.
- Second part (after `---`) is the response part.

4. Add two dependencies in `package.xml`

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

5. Modify or add the following in `CMakeLists.txt`

- Add `message_generation`

```
## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

- Uncomment and add `RectangleAeraService.srv` (`.srv` file)

```
## Generate services in the 'srv' folder
add_service_files(
   FILES
   RectangleAeraService.srv
 )
```

- Uncomment the following

```
## Generate added messages and services with any dependencies listed here
 generate_messages(
   DEPENDENCIES
   std_msgs
 )
```

- Uncomment and add `message_runtime`

```
##################################
## catkin specific configuration ##
##################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also
need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ros_service_assignment
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

6. Run in terminal

```
cd catkin_ws
catkin_make
```

7. Check if everything is fine

- It must contain `ros_service_assignment/RectangleAeraService`

```
$ rossrv list
...
ros_service_assignment/RectangleAeraService
...
```

Read about rossrv

Use Client/Server

▶ Details

area_server.py

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

from ros_service_assignment.srv import RectangleAeraService
from ros_service_assignment.srv import RectangleAeraServiceRequest
from ros_service_assignment.srv import RectangleAeraServiceResponse
```

```
import rospy

def handle_find_area(req):
    print "Returning [%s * %s = %s]"%(req.width, req.height, (req.width *
req.height))
    return RectangleAeraServiceResponse(req.width * req.height)

def find_area_server():
    rospy.init_node('find_area_server')
    s = rospy.Service('find_area', RectangleAeraService, handle_find_area)
    print "Ready to find_area."
    rospy.spin()

if __name__ == "__main__":
    find_area_server()
```

▶ Details

area_client.py

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

import sys
import rospy
from ros_service_assignment.srv import RectangleAeraService
from ros_service_assignment.srv import RectangleAeraServiceRequest
from ros_service_assignment.srv import RectangleAeraServiceResponse

def find_area_client(x, y):
    rospy.wait_for_service('find_area')
    try:
        find_area = rospy.ServiceProxy('find_area', RectangleAeraService)
        resp1 = find_area(x, y)
        return resp1.area
    except rospy.ServiceException, e:
        print "Service call failed: %s" % e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s*%s"%(x, y)
    print "%s * %s = %s"%(x, y, find_area_client(x, y))
```

In two sperate terminals run the following commands (one terminal per command)

```
$ rosrun ros_service_assignment area_server.py
Ready to find_area.
```

```
$ rosrun ros_service_assignment area_client.py 3 4
Requesting 3*4
3 * 4 = 12.0
```

Read about rosrun

Network Configuration

Add the following in `.bashrc`

```
#The IP address for the Master node
export ROS_MASTER_URI=http://192.168.8.111:11311
#The IP address for your device/host IP address
export ROS_HOSTNAME=192.168.8.126
```

- `192.168.8.111` is IP address for the Master node.
- `192.168.8.126` is the IP address for your device/host IP address

To get your IP address

In Windows

```
ipconfig
```

In Ubuntu

```
ifconfig
```

To learn more https://edu.gaitech.hk/turtlebot/network-config-doc.html

---

Robots

Turtlesim

Refernces
[01] http://wiki.ros.org/turtlesim
[02] http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

Start Turtlesim

```
rosrun turtlesim turtlesim_node
```

Read about rosrun

Move Turtle By Using Keyboard

```
rosrun turtlesim turtle_teleop_key
```

Read about rosrun

Create Another Turtle

> This service lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.

```
$ rosservice call /spawn 2 2 0.2 ""
name: turtle2
```

Learn more about rosservice

Motion of Turtlesim

▶ Details
cleaner.py

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

# Q1: What is the limit of rospy.Rate?
# Q2: Is it always good to use large numbers for rospy.Rate?

from turtlesim.msg import Pose
import rospy
from geometry_msgs.msg import Twist
import math
import time
from std_srvs.srv import Empty

x=0
```

```
y=0
z=0
yaw=0

x_min = 0.0
y_min = 0.0
x_max = 11.0
y_max = 11.0


def poseCallback(pose_message):
    # To change the value of the parameters and get the global variables,
    # "global" keyword is used.
    global x
    global y, z, yaw
    # Get the information.
    x= pose_message.x
    y= pose_message.y
    yaw = pose_message.theta

def move(speed, distance, is_forward):
    #declare a Twist message to send velocity commands
    velocity_message = Twist()

    #get current location from the global variable before entering the loop
    x0=x
    y0=y
    #z0=z;
    #yaw0=yaw;


    velocity_message.linear.y = 0
    velocity_message.linear.z = 0
    velocity_message.angular.x = 0
    velocity_message.angular.y = 0
    velocity_message.angular.z = 0

    # assign the x coordinate of linear velocity to the speed.
    if is_forward:
        velocity_message.linear.x = abs(speed)
    else:
        velocity_message.linear.x = -abs(speed)

    distance_moved = 0.0
    loop_rate = rospy.Rate(10) # we publish the velocity at 10 Hz (10 times a
second)

        #task 2. create a publisher for the velocity message on the appropriate
topic.
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)

    while True :
            #task 3. publish the velocity message
            velocity_publisher.publish(velocity_message)
```

```
            loop_rate.sleep()

            #rospy.Duration(1.0)


            #measure the distance moved
            distance_moved = distance_moved+abs(0.5 * math.sqrt(((x-x0) ** 2) +
((y-y0) ** 2)))
            print  distance_moved
            if not (distance_moved < distance):
                rospy.loginfo("reached")
                break

    #task 4. publish a velocity message zero to make the robot stop after the
distance is reached
    velocity_message.linear.x = 0
    velocity_publisher.publish(velocity_message)

def rotate(angular_speed, relative_angle, is_clockwise):
    #declare a Twist message to send velocity commands
    velocity_message = Twist()

    #get current location from the global variable before entering the loop
    x0=x
    y0=y
    #z0=z;
    #yaw0=yaw



    velocity_message.linear.x = 0
    velocity_message.linear.y = 0
    velocity_message.linear.z = 0
    velocity_message.angular.x = 0
    velocity_message.angular.y = 0

    # assign the x coordinate of linear velocity to the speed.
    if is_clockwise:
        velocity_message.angular.z = -abs(angular_speed)
    else:
        velocity_message.linear.x = abs(angular_speed)


    angle_moved = 0.0
    t0 = time.time()
    loop_rate = rospy.Rate(10) # we publish the velocity at 10 Hz (10 times a
second), loop_rate.sleep() will wait for 1/10 seconds


        #task 2. create a publisher for the velocity message on the appropriate
topic.
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
```

```
    while True :
            #task 3. publish the velocity message
            velocity_publisher.publish(velocity_message)

            loop_rate.sleep()

            #rospy.Duration(1.0)


            #measure the distance moved
            t1 = time.time()
            angle_moved = angular_speed * (t1-t0)
            print  angle_moved
            if not (angle_moved < relative_angle):
                rospy.loginfo("reached")
                break

    #task 4. publish a velocity message zero to make the robot stop after the
distance is reached
    velocity_message.angular.z = 0
    velocity_publisher.publish(velocity_message)

def degrees_to_radians(angle_in_degrees):
    return angle_in_degrees *math.pi /180.0

def get_distance(x1, y1, x2, y2):
    return math.sqrt(pow((x1-x2),2)+pow((y1-y2),2))

def set_desired_orientation (desired_angle_radians):
    relative_angle_radians = desired_angle_radians - yaw
    if relative_angle_radians < 0:
        clockwise = True
    else:
        clockwise = False
        rotate(degrees_to_radians(10), abs(relative_angle_radians), clockwise)

def move_to_goal(goal_pose, distance_tolerance):
    #declare a Twist message to send velocity commands
    velocity_message = Twist()

    loop_rate = rospy.Rate(10) # we publish the velocity at 10 Hz (10 times a
second)
    E = 0.0

    #task 2. create a publisher for the velocity message on the appropriate topic.
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)

    while True:

        kp = 1.0
        ki = 0.02

        e = get_distance(x, y, goal_pose.x, goal_pose.y)
        E = E+e
```

```
        velocity_message.linear.x = (kp*e)
        velocity_message.linear.y =0
        velocity_message.linear.z =0
                #angular velocity in the z-axis
        velocity_message.angular.x = 0
        velocity_message.angular.y = 0
        velocity_message.angular.z =4*(math.atan2(goal_pose.y-y, goal_pose.x-x)-
yaw)

        velocity_publisher.publish(velocity_message)

        #rospy.spin()
        loop_rate.sleep()

        if not (get_distance(x, y, goal_pose.x, goal_pose.y)>distance_tolerance):
                rospy.loginfo("reached")
                break
    #task 4. publish a velocity message zero to make the robot stop after the
distance is reached
    velocity_message.angular.z = 0
    velocity_message.linear.x = 0
    velocity_publisher.publish(velocity_message)

def grid_clean():
    loop_rate = rospy.Rate(0.5)
    pose = Pose()
    pose.x=1
    pose.y=1
    pose.theta=0
    move_to_goal(pose, 0.01)
    loop_rate.sleep()
    set_desired_orientation(0)
    loop_rate.sleep()

    move(2.0, 9.0, True)
    loop_rate.sleep()
    rotate(degrees_to_radians(10), degrees_to_radians(90), False)
    loop_rate.sleep()
    move(2.0, 9.0, True)

    rotate(degrees_to_radians(10), degrees_to_radians(90), False)
    loop_rate.sleep()
    move(2.0, 1.0, True)
    rotate(degrees_to_radians(10), degrees_to_radians(90), False)
    loop_rate.sleep()
    move(2.0, 9.0, True)

    rotate(degrees_to_radians(30), degrees_to_radians(90), True)
    loop_rate.sleep()
    move(2.0, 1.0, True)
    rotate(degrees_to_radians(30), degrees_to_radians(90), True)
    loop_rate.sleep()
    move(2.0, 9.0, True)
```

```
    distance = get_distance(x, y, x_max, y_max)

def spiral_clean():
    #declare a Twist message to send velocity commands
    velocity_message = Twist()

    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)


    count = 0

    constant_speed = 4
    vk = 1
    wk = 2
    rk = 0.5
    loop_rate = rospy.Rate(1)

    while True:
        rk = rk+1.0
        velocity_message.linear.x = rk
        velocity_message.linear.y = 0
        velocity_message.linear.z = 0

        velocity_message.angular.x = 0
        velocity_message.angular.y = 0
        velocity_message.angular.z = constant_speed

        velocity_publisher.publish(velocity_message)

        loop_rate.sleep()

        if not (x<10.5) and (y<10.5):
            velocity_message.linear.x = 0
            velocity_publisher.publish(velocity_message)



if __name__=='__main__':
    try:
        rospy.init_node('cleaner', anonymous=True)

        position_topic = "/turtle1/pose"
        pose_subscriber = rospy.Subscriber(position_topic, Pose, poseCallback)

        option = input("For forward move: 0 \nFor angular move: 1\nFor goal move:
2\nFor grid clean: 3\nFor spiral clean: 4\n")

        if option == 0:
            speed = input("Speed:")
            distance = input("Distance:")
            is_forward = input("Is Forward:")
            move(speed, distance, is_forward)
```

```
        elif option == 1:
            angular_speed = input("Angular Speed:")
            relative_angle = input("Relative Angle:")
            is_clockwise = input("Is Clockwise:")
            rotate(degrees_to_radians(angular_speed),
degrees_to_radians(relative_angle), is_clockwise)
        elif option == 2:
            pose = Pose()
            pose.x=1
            pose.y=1
            pose.theta=0
            move_to_goal(pose, 0.01)
        elif option == 3:
            grid_clean()
        elif option == 4:
            spiral_clean()
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo("node terminated.")
```

`cleaner_py.launch` is used to run `turtlesim_node` and `cleaner.py` at the same time

```
<launch>
    <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node"
output="screen"/>
    <node name="cleaner.py" pkg="ros_essentials_cpp" type="cleaner.py"
output="screen"/>
</launch>
```

Run `cleaner_py.launch`

```
roslaunch ros_essentials_cpp cleaner_py.launch
```

Read about roslaunch

Turtlebot

Start Turtlebot

```
roslaunch turtlebot_stage turtlebot_in_stage.launch
```

Move Turtlebot By Using Keyboard

```
roslaunch turtlebot_teleop keyboard_teleop.lanuch
```

Motion of Turtlebot

▶ turtlebot_cleaner.py

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

import rospy
from geometry_msgs.msg import Twist
import math
import time
from std_srvs.srv import Empty

x=0
y=0
z=0
yaw=0


def move(speed, distance, is_forward):
        #declare a Twist message to send velocity commands
        velocity_message = Twist()
        #get current location
        x0=x
        y0=y

        if (is_forward):
            velocity_message.linear.x =abs(speed)
        else:
                velocity_message.linear.x =-abs(speed)

        distance_moved = 0.0
        loop_rate = rospy.Rate(10) # we publish the velocity at 10 Hz (10 times a
second)
        cmd_vel_topic='/cmd_vel_mux/input/teleop'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size=10)

        t0 = rospy.Time.now().to_sec()

        while True :
                rospy.loginfo("Turtlebot moves forwards")
                velocity_publisher.publish(velocity_message)

                loop_rate.sleep()

                #rospy.Duration(1.0)
                t1 = rospy.Time.now().to_sec()
                distance_moved = (t1-t0)*speed
                print  distance_moved
                if  not (distance_moved<distance):
                    rospy.loginfo("reached")
                    break
```

```
        #finally, stop the robot when the distance is moved
        velocity_message.linear.x =0
        velocity_publisher.publish(velocity_message)

def rotate (angular_speed_degree, relative_angle_degree, clockwise):

    global yaw
    velocity_message = Twist()
    velocity_message.linear.x=0
    velocity_message.linear.y=0
    velocity_message.linear.z=0
    velocity_message.angular.x=0
    velocity_message.angular.y=0
    velocity_message.angular.z=0

    #get current location
    theta0=yaw
    angular_speed=math.radians(abs(angular_speed_degree))

    if (clockwise):
        velocity_message.angular.z =-abs(angular_speed)
    else:
        velocity_message.angular.z =abs(angular_speed)

    angle_moved = 0.0
    loop_rate = rospy.Rate(10) # we publish the velocity at 10 Hz (10 times a
second)
    cmd_vel_topic='/cmd_vel_mux/input/teleop'
    velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size=10)

    t0 = rospy.Time.now().to_sec()

    while True :
        rospy.loginfo("Turtlebot rotates")
        velocity_publisher.publish(velocity_message)

        t1 = rospy.Time.now().to_sec()
        current_angle_degree = (t1-t0)*angular_speed_degree
        loop_rate.sleep()



        if  (current_angle_degree>relative_angle_degree):
            rospy.loginfo("reached")
            break

    #finally, stop the robot when the distance is moved
    velocity_message.angular.z =0
    velocity_publisher.publish(velocity_message)


def go_to_goal(x_goal, y_goal):
    global x
```

```
    global y, z, yaw

    velocity_message = Twist()
    cmd_vel_topic='/cmd_vel_mux/input/teleop'

    while (True):
        K_linear = 0.5
        distance = abs(math.sqrt(((x_goal-x) ** 2) + ((y_goal-y) ** 2)))

        linear_speed = distance * K_linear

        K_angular = 4.0
        desired_angle_goal = math.atan2(y_goal-y, x_goal-x)
        angular_speed = (desired_angle_goal-yaw)*K_angular

        velocity_message.linear.x = linear_speed
        velocity_message.angular.z = angular_speed

        velocity_publisher.publish(velocity_message)
        print 'x=', x, 'y=',y


        if (distance <0.01):
            break



if __name__ == '__main__':
    try:

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

        #declare velocity publisher
        cmd_vel_topic='/cmd_vel_mux/input/teleop'
        velocity_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size=10)
        time.sleep(2)
        move(22.0, 90.0, True)
        time.sleep(1)
        move(22.0, 90.0, False)
        time.sleep(1)
        rotate(30, 90, False)
        #go_to_goal(5.0, 9.0)

    except rospy.ROSInterruptException:
        rospy.loginfo("node terminated.")
```

Commands

roscore

> roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the roscore command.

Usage

```
roscore
```

rosnode

> rosnode is a command-line tool for displaying debug information about ROS Nodes, including publications, subscriptions and connections. It also contains an experimental library for retrieving node information.

List active nodes

```
rosnode list
```

Print information about node

```
rosnode info /node_name
```

rostopic

> The rostopic command-line tool displays information about ROS topics. Currently, it can display a list of active topics, the publishers and subscribers of a specific topic, the publishing rate of a topic, the bandwidth of a topic, and messages published to a topic. The display of messages is configurable to output in a plotting-friendly format.

Display messages published to a topic

```
rostopic echo /topic_name
```

Print information about topic

```
rostopic info /topic_name
```

Display a list of current topics

```
rostopic list
```

rosmsg

> The rosmsg command-line tool displays information about ROS Message types.

Display the fields in a ROS message type. You may omit the package name of the type, in which case rosmsg will search for matching types in all packages

```
rosmsg show std_msgs/String
```

or, if you don't know the package name

```
rosmsg show Pose
```

rossrv

> The rossrv command-line tool displays information about ROS services. It has the exact same usage as rosmsg (see what it offers when it runs without sub-command below):

List all the services

```
rossrv list
```

rosrun

> rosrun allows you to run an executable in an arbitrary package from anywhere without having to give its full path or cd/roscd there first.

Usage:

```
rosrun <package> <executable>
```

Example:

```
rosrun roscpp_tutorials talker.py
```

roslaunch

> roslaunch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch

> extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

Usage:

```
roslaunch package_name file.launch
```

roscd

> roscd allows you to change directories using a package name, stack name, or special location.

Usage:

```
roscd <package-or-stack>[/subdir]
```

Example:

```
roscd roscpp
```

Go to the default workspace

```
roscd
```

Others

Create `.launch` file

> `.launch` file is used to run many nodes at the same time.

Example:

`cleaner_py.launch` is used to run `turtlesim_node` and `cleaner.py` at the same time

```
<launch>
    <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node"
output="screen"/>
    <node name="cleaner.py" pkg="ros_essentials_cpp" type="cleaner.py"
output="screen"/>
</launch>
```

Run `cleaner_py.launch`

```
roslaunch ros_essentials_cpp cleaner_py.launch
```

Read about roslaunch

Create shortcuts and aliases in .bashrc (in HOME directory)

Add the following line to make `gb` shortcut to open Gedit to edit `.bashrc`

```
alias gb="gedit /home/riotu/.bashrc"
```
