Computer Exercise 1

This is a basic hands-on exercise in ROS to practice the principles shown in class.

We assume very basic know-how in Linux and basic experience in Python. A recommended Python crash course is Google's Python Class. Specifically, make sure you know how to work with classes. If you need, you can use this handy Linux Tutorial.

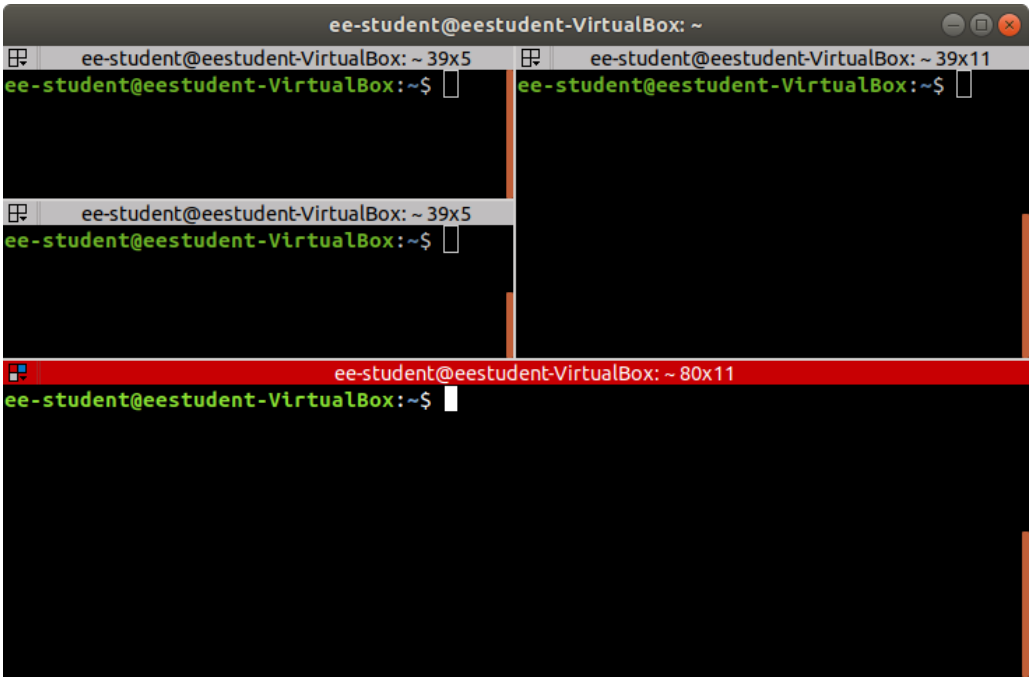
General instructions:

- Submission is in pairs only.
- Submission deadline: 12/4/2021, 23:59
- Questions related to the exercise should be posted in the "Discussion on Computer Exercises" forum on Moodle.
- Submit your report electronically as a PDF file (typed only, handwritten is not accepted). Code submission is not required in this exercise.

Part I: Get to Know ROS

Open your Ubuntu18 machine.

Launch a new terminal. You are warmly encouraged to use Terminator which should already be installed on your machines. Terminator allows to open multiple shells in one window (you'll soon see how useful it is) by right-clicking on the window and choosing *Split Horizontally* or *Split Vertically*:



The first command to execute in the terminal is

```
roscore
```

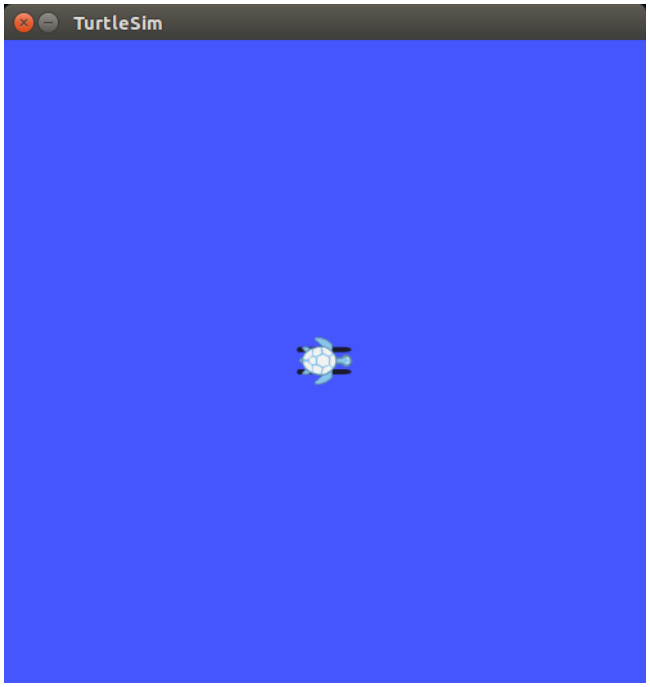
This command launches ROS master together with rosout (the ROS logger) and the ROS parameter server.

Topics and Services

In a new terminal run

```
rosrun turtlesim turtlesim_node
```

A new window should pop up:



This is TurtleSim, a very simple robot simulator.

Open a third terminal and type

```
rosnode
```

Hit *Enter* to view all the options of this command.

If you want more elaborated description of every option, you can use the *-h* flag, for instance:

```
rosnode ping -h
```

Task 1 (5 points)

Use the *rosnode* commands to find the name of the TurtleSim node that you have just lunched and then kill this node. **Add the two commands to your report.**

Note that all ROS command line tools support Tab-completion for package names, node names, etc. – very useful and convenient, try practicing it.

Now that we feel more confident with command line tools, let's explore *rostopic* and *rosservice*. First, resume the TurtleSim node that you previously killed. Then, in a new terminal, type

```
rostopic
```

and hit *Enter*. Read about the different options. Repeat this also for

```
rosservice
```

Task 2 (5 points)

Now, let's make the turtle move. The TurtleSim node listens (subscribes) to a topic where velocity commands are published and moves the turtle on the screen accordingly. We first need to find this topic, and we have two ways for doing that:

1. The TutrleSim wiki page.
2. Find the list of topics related to the TurtleSim node and among them find the topic whose message type is geometry_msgs/Twist (the standard message for velocity commands). This can be done by using the command line tools *rosnode* and *rostopic*.

Publish a *Twist* message with linear velocity of $1.0\left[\frac{m}{sec}\right]$ in the x direction using the *rostopic* command.

Add this command to your report.

Tip: use Tab-completion to create the *Twist* message correctly (simply hit tab twice and it'll generate the template of the message).

See how the turtle moves. After publishing the message, you can hit *CTRL+C* and continue using this terminal.

For a smoother operation of the robot, we can use a tele-operation node which publishes messages to the same velocity command topic based on input from the keyboard. Run

```
rosrun turtlesim turtle_teleop_key
```

and play with the turtle using the arrows. Don't kill this node for now.

Task 3 (5 points)

In the previous case it was fairly easy to find the velocity command topic. In a typical robotic system, there will be many nodes running in parallel, subscribing and publishing on dozens of topics. A useful debugging tool for nodes and topics is rqt_graph.

In a new terminal, launch rqt_graph by executing

```
rosrun rqt_graph rqt_graph
```

The nodes are represented by circles and the topics are represented by arrows. Now let's see what happens when we add another subscriber to the velocity command topic.

In a new (fifth) terminal run

```
rostopic echo ##velocity_command_topic_name##
```

(replace *##velocity_command_topic_name##* with the topic name from the previous task).

This practically creates another node that listens to the velocity command topic.

Now, play with the arrows while you're in focus on the terminal where *teleop_turtle_key* runs.

Refresh the view of rqt_graph by hitting the refresh icon

What has changed? Can you see two arrows with the same topic name? **Paste this graph in your report.**

Task 4 (5 points)

To conclude this part, let's do some stuff using service calls.

Use the *rosservice* commands as well as the TurtleSim wiki page to find out how to:

1. Clear the pen
2. Change the pen color
3. Return the turtle to its starting point

Add the three commands to your report.

Remember to use Tab-completion for the service names and for creating the service payload!

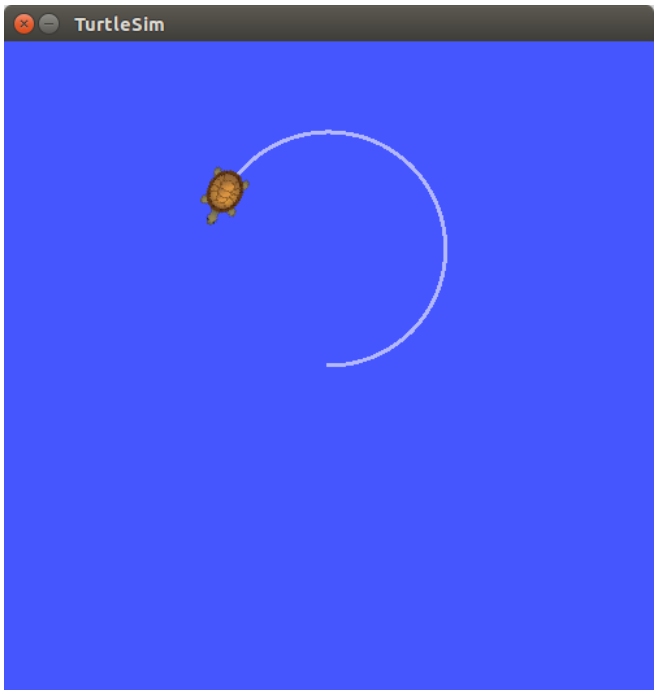
Writing Our First Node

In this part, we will write our first ROS node in a new catkin package and under a new catkin workspace. We will also further play with topics and services, this time from the code API side. For reference you can use the rospy documentation.

You are warmly encouraged to start by going over the two tutorials below. Use them as reference when you write your code.

- Writing a Simple Publisher and Subscriber (Python)
- Writing a Simple Service and Client (Python).

Our task is going to be pretty simple: we want to program TurtleSim to draw an arch at a given angle in the range $[0,2\pi]$:



At first, let's create our new catkin workspace. In a new terminal execute

```
cd ~
mkdir -p my_ws/src
cd my_ws/src
catkin_init_workspace
```

We can now create a catkin package under the workspace we have just created:

```
catkin_create_pkg my_pkg rospy std_srvs std_msgs turtlesim geometry_msgs
```

This command creates a catkin package named *my_pkg* which is dependent on the packages *rospy*, *std_srvs*, *std_msgs, turtlesim* and *geometry_msgs*.

If you open the file *package.xml* under *my_pkg* you will see all the packages mentioned above as build and run dependencies of the new package. Those dependencies will be also expressed in *CMakeLists.txt* of our package.

Change directory to the new package:

```
cd my_pkg
```

The common convention is that .cpp files are located under the *src* folder whereas .py files are located under another folder names *scripts*.

```
mkdir scripts
cd scripts
```

Create a new empty .py file (name it *my_node.py*) and make it executable:

```
touch my_node.py
chmod +x my_node.py
```

Open the file using your preferred editor (*geany*, *atom* and *sublime* are recommended ones but you'll have to install them yourself) and paste the following code (alternatively, use the provided *my_node.py* file):

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


import numpy as np
import rospy
from std_srvs.srv import Empty, Trigger, TriggerResponse
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from std_msgs.msg import Float32


class Draw(object):
    def __init__(self):

        # Initialize the node
        ## <YOUR CODE HERE - 1> ##


        # Subscribe to pose topic
```

```
        ## <YOUR CODE HERE - 2> ##

        # Subscribe to draw_arch topic
        ## <YOUR CODE HERE - 3> ##

        # Create publisher for cmd_vel topic
        ## <YOUR CODE HERE - 4> ##

        # Create service callback to pause_drawing
        ## <YOUR CODE HERE - 6> ##

        # Create service callback to resume_drawing
        ## <YOUR CODE HERE - 6> ##

        # Create handle to the reset service
        ## <YOUR CODE HERE - 7> ##

        # Indicator that node is now in the process of drawing
        self.is_busy = False

    def pose_callback(self, msg):
        ## <YOUR CODE HERE - 2> ##

    def draw_arch_callback(self, msg):

        # Check availability for drawing a new arch
        if self.is_busy:
            rospy.loginfo('Currently drawing, new request is ignored')
            return

        rospy.loginfo('Recieved draw request')
        self.is_busy = True

        # Reset turtlesim
        ## <YOUR CODE HERE - 7> ##
        self.current_angle = None
```

```
        # Initialize member variables
        self.pose = None
        self.allowed_to_draw = True
        if 0 <= msg.data < np.pi:
            desired_angle = msg.data
        elif np.pi < msg.data <= 2 * np.pi:
            desired_angle = msg.data - 2 * np.pi
        else:
            raise Exception('Angle must be in range [0, 2PI]')

        # Wait for first published pose before drawing
        while self.current_angle == None:
            rospy.sleep(0.01)
        rospy.loginfo('Start drawing')

        while not rospy.is_shutdown():

            if ## <YOUR CODE HERE - 5> ##:
                ## <YOUR CODE HERE - 5> ##
                break

            if not self.allowed_to_draw:
                continue

            twist_msg = Twist()
            twist_msg.linear.x = 1
            twist_msg.linear.y = 0
            twist_msg.linear.z = 0
            twist_msg.angular.x = 0
            twist_msg.angular.y = 0
            twist_msg.angular.z = 0.5

            rospy.sleep(0.01)
            ## <YOUR CODE HERE - 4> ##

        rospy.loginfo('Finished drawing')
        self.is_busy = False
```

```
    def pause_callback(self, reqt):
        ## <YOUR CODE HERE - 6> ##
        return TriggerResponse(success=True, message='paused drawing')

    def resume_callback(self, req):
        ## <YOUR CODE HERE - 6> ##
        return TriggerResponse(success=True, message='resumed drawing')

if __name__ == '__main__':
    Draw()
    rospy.spin()
```

We will now gradually fill the missing parts in the code above. For debugging, either use rospy's logger or just simple prints.

Task 1 – initializing the node (5 points)

Every ROS node must be initialized before any interaction with the master and with other nodes. Replace *## <**YOUR CODE HERE – 1**> ##* with the suitable command. Name the node *drawer*.

Task 2 – subscribing to the pose topic (5 points)

When we draw the arch, we need to know when to stop. For that purpose, we'll need to subscribe to the pose topic and read its value.

- First, we need to declare the subscriber to the pose topic (find its exact name in the topics list) whose type is Pose (already imported in the code). In the subscription command we also define the callback (*self.pose_callback*) to which the pose messages are to be routed.
- Then, in the callback itself, we need to save the current orientation to the class member variable named *self.current_angle*. Note that the orientation is one field of the Pose message.

Replace *## <**YOUR CODE HERE – 2**> ##* in both places.

Task 3 – creating the *draw_arch* topic (5 points)

We would like our node to start drawing once invoked by a message sent on a dedicated topic – let's name it *draw_arch*. The message will contain the desired arch angle (as a Float32 number). In order to create a new topic, we do the same thing we did in Task #2 – create a subscriber. Replace *## <**YOUR CODE HERE – 3**> ##* with your code. Make sure you route this topic to the correct callback (which already exists in the code).

Task 4 – creating a *cmd_vel* publisher (5 points)

As you have already seen, we are controlling the robot by sending *Twist* messages on the *turtle1/cmd_vel* topic. We want to initialize a publisher to this topic. Replace ## <***YOUR CODE HERE – 4***> ## with your code – at first initialize the publisher and store it as a member variable, then publish the *Twist* message which is already instantiated in the code.

Task 5 – stop condition (5 points)

The turtle needs to move as long as it doesn't reach the desired angle. Complete the stop condition of the loop in ## <***YOUR CODE HERE – 5***> ##. You should use a threshold for the proximity, don't expect the turtle to reach exactly the specified angle. Please use the already set variable named *desired_angle* which converts the angle to the range $[-\pi, \pi]$ used by TurtleSim. In addition, once the stop condition is met, send a zero *Twist* message to make the turtle immediately stop.

Task 6 – creating services (5 points)

Now that we practiced sending and receiving messages on topics, it's time to play with services. Let's add two services: one that allows to pause the turtle (name it *pause_drawing*) and another that resumes (name it *resume_drawing*). Complete the missing parts marked ## <***YOUR CODE HERE – 6***> ##. In the service declarations, specify std_srvs/Trigger as the service type and route the messages to the relevant callbacks declared below. In the callbacks, all you need to do is change the value of the member *self.allowed_to_draw*.

Task 7 – calling an existing service (5 points)

Before every new arch is drawn, we'd like to clear the pen and return the turtle to its original position. The */reset* service (implemented by TurtleSim) does exactly that. In ## <***YOUR CODE HERE – 7***> ## declare the use of this service, save the function handle as a member variable and then call this handle in the appropriate place in the code. Note that the *reset* service works with the *Empty* message and therefore no arguments need to be passed to the handle.

Task 8 – building and debugging (5 points)

The next thing to do is build the project.

```
cd ~/my_ws
catkin_make
```

If you previously closed it, launch again TurtleSim.

In another terminal, first source the *setup.bash* script of our workspace:

```
source ~/my_ws/devel/setup.bash
```

and then run *my_node*:

```
rosrun my_pkg my_node.py
```

For sanity check, run

```
rosnode list
```

and identify your node in the list.

Now run

```
rosnode info drawer
```

and make sure you see the correct interfaces: topics and services.

In a new terminal send a *draw_arch* message and see the turtle moving. Remember that the angles are given in radians. Repeat this several times with several different angles. Validate also the functionality of the pause/resume services – again, send the requests from the terminal.

Choose an arbitrary angle and paste a screenshot of the drawn arch in your report.

Debug tip: apart from the rqt_graph tool that we have already seen, you can use rqt_console to view info/debug/error/… messages. rqt_console allows you to easily filter the messages that are relevant to you. You run it by

```
rqt_console
```

Add your complete code to the report.

Launch Files

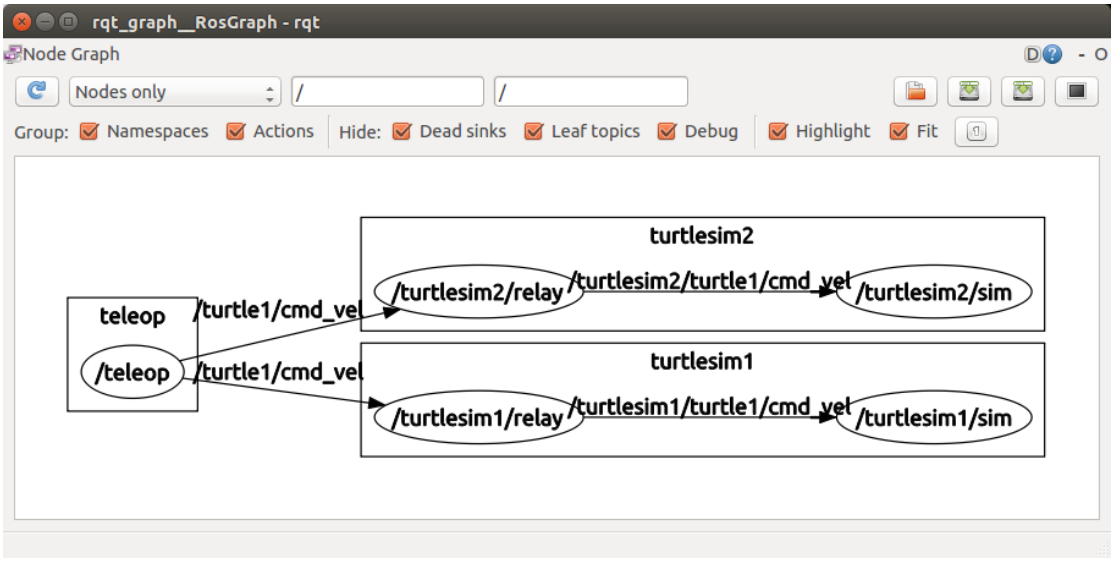
Launch files are XMLs which are used to define a system configuration. These files allow us to run multiple nodes with a single command. For more details, refer to roslaunch/XML.

Task (10 points)

Launch simultaneously two turtles, in two different windows of TurtleSim, but make them both response to the same velocity command topic.

The components that we need in our system are the following:

- Two TurtleSim nodes
- One teleoperation node that outputs *Twist* messages to the topic */turtle1/cmd_vel*
- Two relay nodes that remap the messages on */turtle1/cmd_vel* to the input velocity command topics of the two TurtleSim nodes



Under *my_ws* create a new package and a launch folder under it

```
cd ~/my_ws/src
catkin_create_pkg two_turtles
cd two_turtles
mkdir launch
```

In this directory create a new file named *two_turtles.launch* and paste the following skeleton in it:

```
<?xml version="1.0"?>
<launch>
  <node name='teleop' pkg='turtlesim' type='turtle_teleop_key' />
```

```
  <group ns="turtlesim1">
    <!-- turtlesim node - your code here -->
    <!-- cmd_vel relay node - your code here -->
  </group>

  <group ns="turtlesim2">
    <!-- turtlesim node - your code here -->
    <!-- cmd_vel relay node - your code here -->
  </group>

</launch>
```

To understand how we add a node to the launch file, let's look at the teleoperation node which is already there:

```
  <node name='teleop' pkg='turtlesim' type='turtle_teleop_key' />
```

Each *<node>* entry in a launch file has three mandatory attributes:

1. *name*: a symbolic name that we choose and should be unique (*teleop* in this case)
2. *pkg*: the catkin package under which the requested node resides (*turtlesim* in this case)
3. *type*: the executable name of the desired node (*tutle_teleop_key* in this case)

Recall that we previously ran the command

```
rosrun turtlesim turtle_teleop_key
```

in order to bring up the teleoperation node. From this command you could infer the package and type attributes.

By default, nodes that are launched from launch files don't output their stdout to the screen. We can change that by adding the attribute *output="screen"* in the *<node>* entry.

Now we want to launch two TurtleSims and for that we'll need two different namespaces (that already exists in the skeleton). Under each group (namespace) add the TurtleSim node – should be very similar to what we did with the teleoperation node. Note that you may give the same name to the two TurtleSim nodes; ROS will automatically add the namespace as a prefix to prevent name conflicts.

Now, run the *setup.bash* script of *my_ws* and launch the file you have just created:

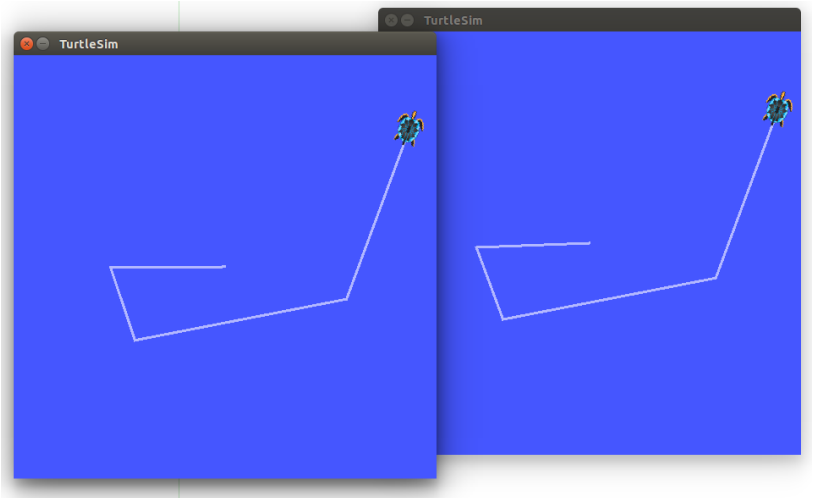
```
source ~/my_ws/devel/setup.bash

roslaunch two_turtles two_turtles.launch
```

Use the command line tools to list the topic names, the service names and the node names. Can you see the prefixes automatically added?

The one thing which is still missing is the relay nodes. For that purpose, we'll use an existing utility in ROS; add the following line under each group (replace ***##output topic##*** with the correct topic name):

```
<node pkg="topic_tools" type="relay" name="relay" args="/turtle1/cmd_vel ##output topi
c##" />
```

Launch your launch file again. Play with the arrows and make sure that the two turtles are moving together:



Add the launch file to the report.

ROS Parameters

The parameter server in ROS is automatically launched with the ROS master. The parameters are accessible to all nodes and are configurable in command line, API and launch files. For more information, refer to Parameter Server.

The command line tool which allows observability and control of parameters is *rosparam*.

Launch again your launch file from the previous section and execute

```
rosparam list
```

As you can see, there are global parameters but also node-specific parameters (you can identify them by the namespace prefix before them). Let's change the background color of *turtlesim1*:

1. Using *rosparam set*, change at least one of the three parameters *background_r*, *background_g* and *background_b*
2. In order for the change to take effect, call the */clear* service under the namespace *turtlesim1*.

Task (10 points)

Let's see now how to do the same thing but from a launch file. Under the same package create another launch file (name it *change_background_color.launch*) that changes the background of *turtlesim1* to your preferred color and the background of *turtlesim2* to another color. You may use the following skeleton:

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

<launch>

  <group ns="turtlesim1">
    <!-- setting the relevant parameter - your code here -->
    <node name="call_clear" pkg="rosservice" type="rosservice" args="call clear" />
  </group>

  <group ns="turtlesim2">
    <!-- setting the relevant parameter - your code here -->
    <node name="call_clear" pkg="rosservice" type="rosservice" args="call clear" />
  </group>

</launch>
```

Use the roslaunch/XML documentation.

Make sure that *two_turtles*.launch is still running and in a new terminal execute

```
source ~/my_ws/devel/setup.bash

roslaunch two_turtles change_background_color.launch
```

Make sure you got the expected result:

Add the launch file to the report.

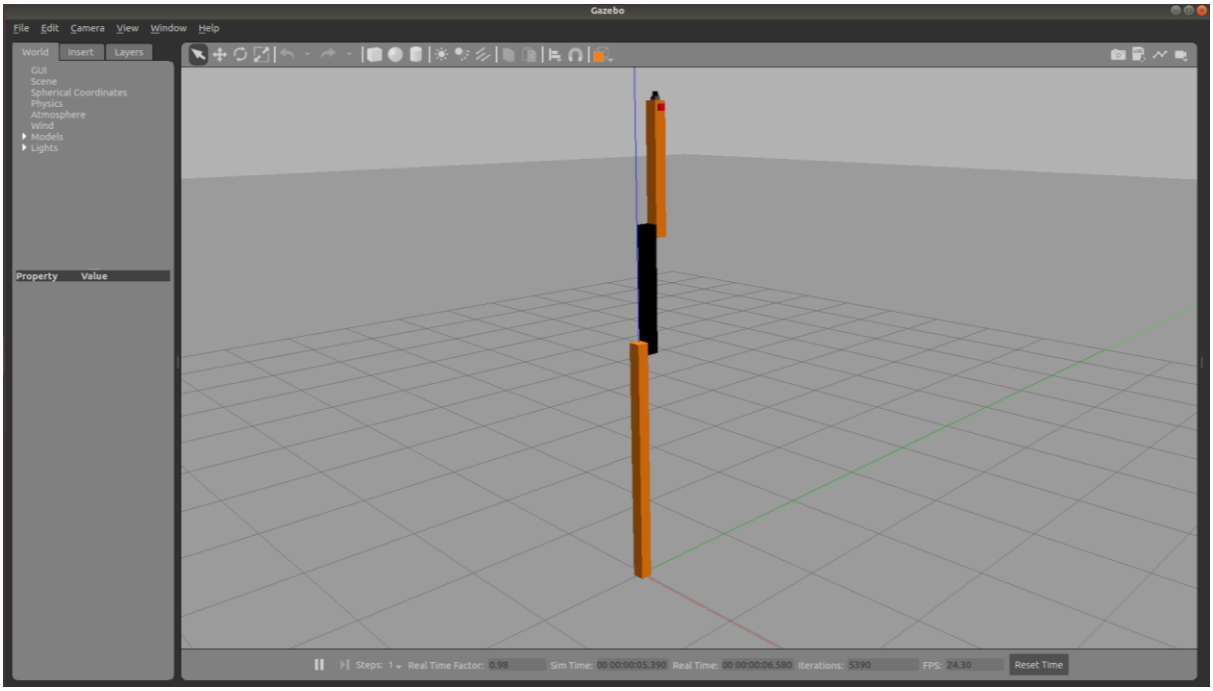
Part II: Gazebo

In this part you will get to know Gazebo – the common robot simulator which integrates with ROS.

First, close all your previous terminals and open a new one. Launch Gazebo by executing

```
source ~/rrbot_ws/devel/setup.bash
roslaunch rrbot_gazebo rrbot_world.launch
```

Once Gazebo launches, you should see RRBOT located exactly at the origin of the global frame (whose axes are marked in red, blue and green lines):



It shouldn't take longer than a few seconds before the robot links "collapse". That is because of the effect of gravity which is modeled by Gazebo's physics engine.

Task 1 (5 points)

Change the viewpoint using the mouse and CTRL and SHIFT buttons.

Paste in your report a screenshot of RRBOT from an arbitrary viewpoint.

Task 2 (5 points)

We want to start playing with RRBOT, but first let's add a coke can model to this Gazebo world.

Run the following command

```
rosrun gazebo_ros spawn_model -database coke_can -sdf -model coke_can3 -x 1 -z 1
```

This might take a while since Gazebo is downloading the model from an online repository (and then saves it locally for future use).

Could you see the coke can dropped from the air to the ground?

Gazebo exposes a set of topics and services. Those allow us to know the exact state ("ground truth") of objects in the world.

Use the command line tools to query the topic */gazebo/model_states*. What is the exact pose of the coke can once it reaches the ground?

Add the command you used and the pose you found to your report. The pose should be relative to the global origin and include the position $[x \quad y \quad z]$ and orientation, expressed as a quaternion.

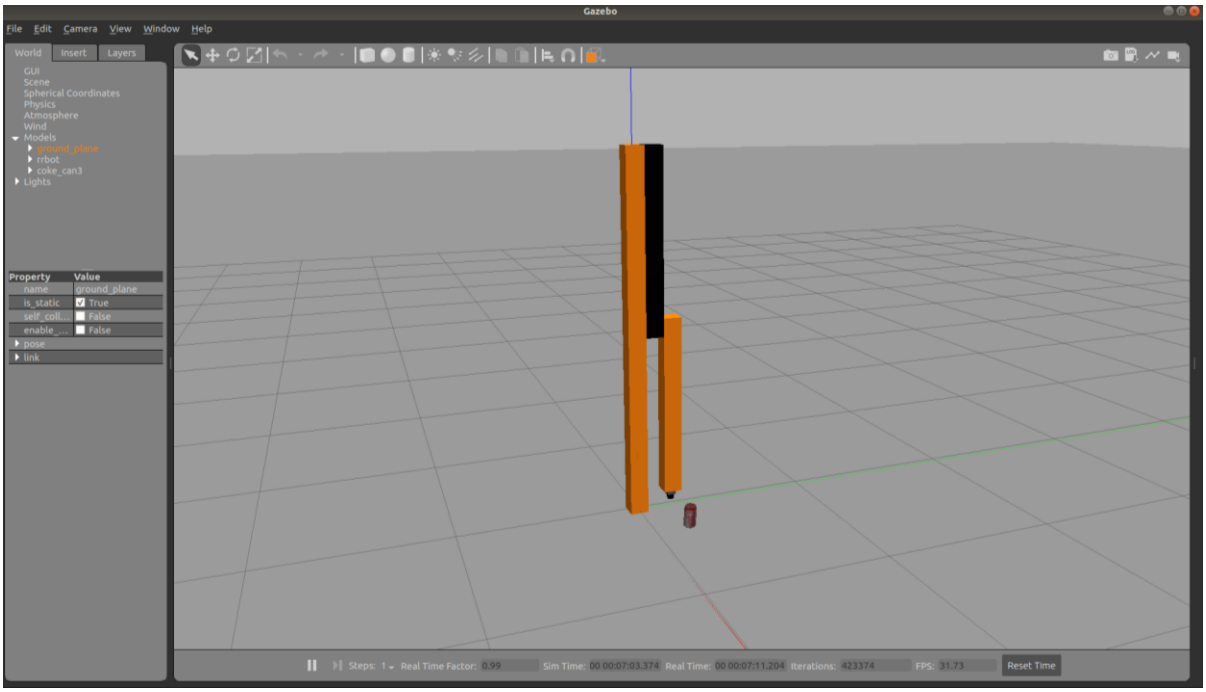
Task 3 (10 points)

Let's hit the coke can with the robot as far as we can.

First, we need to bring the can closer to the robot. Gazebo expose the service *gazebo/set_model_state* which allows us to easily do that:

```
rosservice call /gazebo/set_model_state '{model_state: { model_name: coke_can3, pose:
{ position: { x: 0.3, y: 0.2 ,z: 0 }, orientation: {x: 0, y: 0, z: 0, w: 1.0 } }, twis
t: { linear: {x: 0.0 , y: 0 ,z: 0 } , angular: { x: 0.0 , y: 0 , z: 0.0 } } , referenc
e_frame: world } }'
```

Make sure the can is indeed placed now near the robot's end effector:



Use again the service *gazebo/set_model_state* to change the pose of RRBOT's end effector such that it will eventually hit the can.

Add the command you used to your report. Explain your considerations.
Add also the can's new pose.