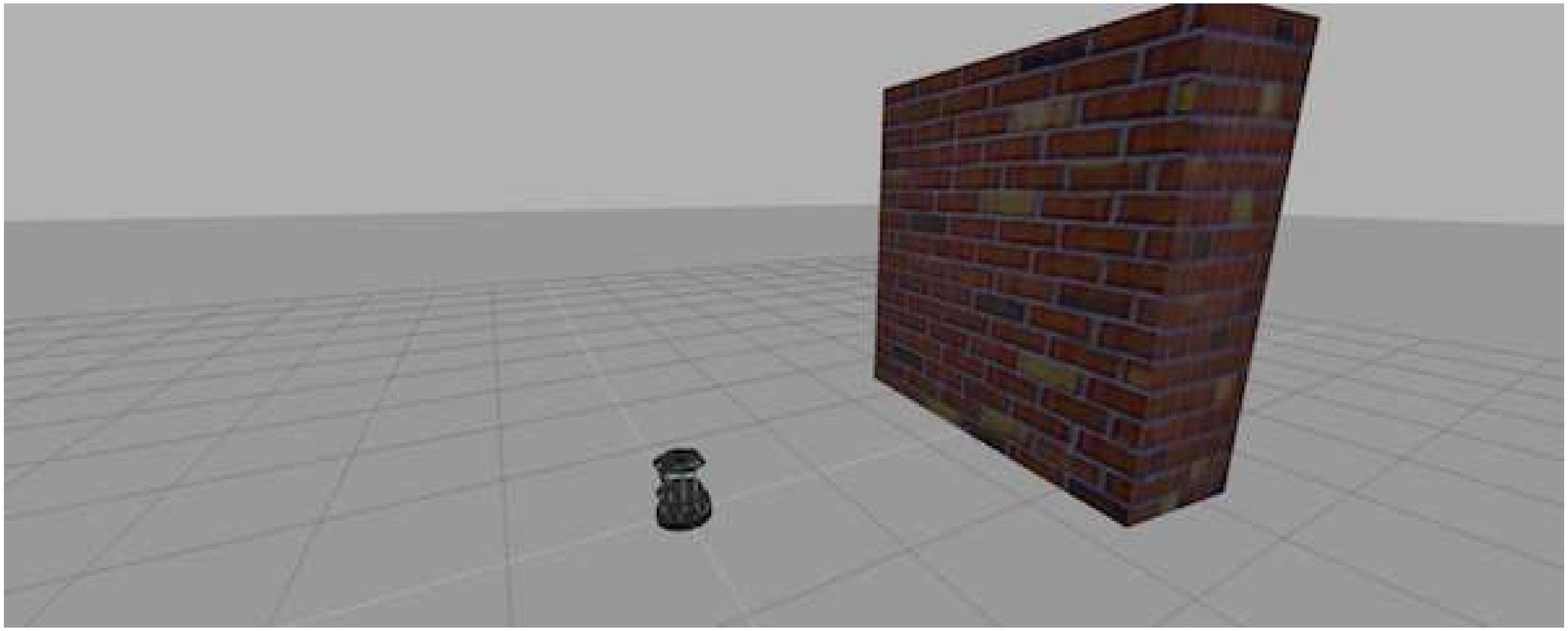
ROS Basics in 5 days

Unit 4 Understanding ROS Topics: Subscribers & Messages

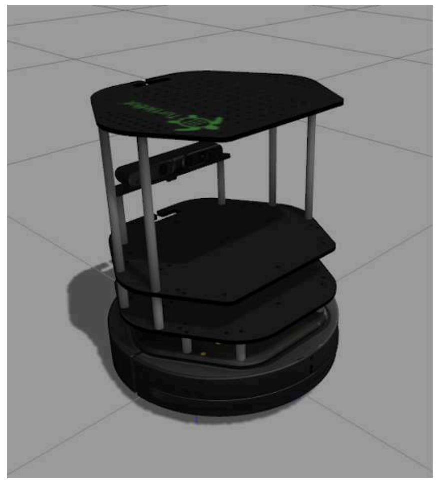


- Summary -

Estimated time to completion: 2.5 hours

What will you learn with this unit?

- What is a Subscriber and how to create one
- How to create your own message



- Real Robot Project -

By the end of this unit, you'll be able to start the course project on a real robot. Please read the instructions at the end of this unit to get started once you finish this course unit.

- Real Robot Project -

- End of Summary -

4.1 Topic Subscriber

You've learned that a topic is a channel where nodes can either write or read information. You've also seen that you can write into a topic using a publisher, so you may be thinking that there should also be some kind of similar tool to read information from a topic. And you're right! That's called a subscriber. **A subscriber is a node that reads information from a topic**. Let's execute the next code:

- Example 4.1 -

For this and following examples, you should create a ROS package named **my_subscriber_example_pkg**. Within this package you will place all the files that you will be creating during this unit.

- First, create the package **my_subscriber_example_pkg**, executing the following comands in the Web Terminal:

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src
```

In [ ]:
```
catkin_create_pkg my_subscriber_example_pkg rospy std_msgs
```

- Next, create a new folder inside your ROS package named **scripts**:

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src/my_subscriber_example_pkg
```

In [ ]:
```
mkdir scripts
```

- Inside the **scripts** folder, create the example Python script named **simple_topic_subscriber.py**. Remember that you can also create it using the IDE.

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws/src/my_subscriber_example_pkg/scripts
```

In [ ]:
```
touch simple_topic_subscriber.py
```

Remember that to run a python script using `rosrun` , the script has to be executable:

In [ ]:
```
chmod +x simple_topic_subscriber.py
```

Python Program: simple_topic_subscriber.py

In [ ]:
```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

def callback(msg):
  print (msg.data)

rospy.init_node('topic_subscriber')
sub = rospy.Subscriber('/counter', Int32, callback)
rospy.spin()
```

END Python Program: simple_topic_subscriber.py

And don't forget to compile and source your workspace!

▶ Execute in Terminal #1

In [ ]:
```
cd ~/catkin_ws
```

In [ ]:
```
catkin_make
```

In [ ]:
```
source devel/setup.bash
```

Now you can execute the script with the command `rosrun` :

▶ Execute in Terminal #1

In [ ]:
```
rosrun my_subscriber_example_pkg simple_topic_subscriber.py
```

- End Example 4.1 -

What's up? Nothing happened again? Well, that's not actually true... Let's do some checks.

Go to your Terminal and type the following:

▶ Execute in Terminal #2

In [ ]:
```
rostopic echo /counter
```

You should see an output like this:

Terminal #2 Output

```
WARNING: no messages received and simulated time is active.
Is /clock being published?
```

And what does this mean? This means that **nobody is publishing into the `/counter` topic**, so there's no information to be read. Let's then publish something into the topic and see what happens. For that, let's introduce a new command:

In [ ]:
```
rostopic pub <topic_name> <message_type> <value>
```

This command will publish the message you specify with the value you specify, in the topic you specify.

Open another Terminal (leave the one with the ***rostopic echo*** opened) and type the next command:

▶ Execute in Terminal #3

In [ ]:
```
rostopic pub /counter std_msgs/Int32 5
```

Now check the output of the console where you did the ***rostopic echo*** again. You should see something like this: .

Terminal #2 Output

```
WARNING: no messages received and simulated time is active.
Is /clock being published?
data:
5
---
```

This means that the value you published has been received by your subscriber program (which prints the value on the screen).

Before explaining everything with more detail, let's explain the code inside **simple_topic_subscriber.py**

In [ ]:

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

import rospy
from std_msgs.msg import Int32

def callback(msg):                                  # Define a function called 'callback' that receives a p
                                                    # named 'msg'

    print (msg.data)                                # Print the value 'data' inside the 'msg' parameter

rospy.init_node('topic_subscriber')                 # Initiate a Node called 'topic_subscriber'

sub = rospy.Subscriber('/counter', Int32, callback)  # Create a Subscriber object that will listen to the /c
                                                    # topic and will cal the 'callback' function each time
                                                    # something from the topic
rospy.spin()                                        # Create a loop that will keep the program in execution
```

So now, let's explain what has just happened. You've basically created a subscriber node that listens to the **/counter** topic, and each time it reads something, it calls a function that does a print of the msg. Initially, nothing happened since nobody was publishing into the **/counter** topic, but when you executed the `rostopic pub` command, you published a message into the **/counter** topic, so the function has printed the number to the console and you could also see that message in the `rostopic echo` output. Now everything makes sense, right?

Now let's do some exercises to put into practice what you've just learned!

- Exercise 4.2 -

Modify the code in order to print the odometry of the robot.

- End of Exercise 4.2 -

- Notes for Exercise 4.2 -

1. **The odometry of the robot is published by the robot into the /odom topic.**
2. **You will need to figure out what message uses the /odom topic, and how the structure of this message is.**

- End of Notes -

- Exercise 4.3 -

1. Add to {Exercice 4.2}, a Python file that creates a publisher that indicates the age of the robot.
2. For that, you'll need to create a new message called **Age.msg**. To see how you can do that, have a look at the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

- End of Exercise 4.3 -

4.2 Prepare CMakeLists.txt and package.xml for custom Message compilation

Now you may be wondering... in case I need to publish some data that is not an Int32, which type of message should I use? You can use all ROS defined (***rosmsg list***) messages. But, in case none fit your needs, you can create a new one.

In order to create a new message, you will need to do the following steps:

1. Create a directory named 'msg' inside your package
2. Inside this directory, create a file named Name_of_your_message.msg (more information down)
3. Modify CMakeLists.txt file (more information down)
4. Modify package.xml file (more information down)
5. Compile
6. Use in code

For example, let's create a message that indicates age, with years, months, and days.

1) Create a directory msg in your package.

In [ ]:
```
roscd <package_name>
```

In [ ]:
```
mkdir msg
```

2) The **Age.msg** file must contain this:

In [ ]:
```
float32 years
float32 months
float32 days
```

3) **In CMakeLists.txt**

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_message_files()
- generate_messages()
- catkin_package()

I. find_package()

This is where all the packages required to COMPILE the messages of the topics, services, and actions go. In package.xml, you have to state them as **build_depend**.

HINT 1: If you open the CMakeLists.txt file in your IDE, you'll see that almost all of the file is commented. This includes some of the lines you will have to modify. Instead of copying and pasting the lines below, find the equivalents in the file and uncomment them, and then add the parts that are missing.

In [ ]:

```
find_package(catkin REQUIRED COMPONENTS
       rospy
       std_msgs
       message_generation   # Add message_generation here, after the other packages
)
```

II. add_message_files()

This function includes all of the messages of this package (in the msg folder) to be compiled. The file should look like this.

In [ ]:

```
add_message_files(
      FILES
      Age.msg
    ) # Dont Forget to UNCOMENT the parenthesis and add_message_files TOO
```

III. generate_messages()

Here is where the packages needed for the messages compilation are imported.

In [ ]:

```
generate_messages(
      DEPENDENCIES
      std_msgs
) # Dont Forget to uncoment here TOO
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the package.xml as **exec_depend**.

In [ ]:

```
catkin_package(
      CATKIN_DEPENDS rospy message_runtime   # This will NOT be the only thing here
)
```

Summarizing, this is the minimum expression of what is needed for the **CMakelists.txt** file to work:

Note: Keep in mind that the name of the package in the following example is **topic_ex**, so in your case, the name of the package may be different.

In [ ]:
```
cmake_minimum_required(VERSION 2.8.3)
project(topic_ex)


find_package(catkin REQUIRED COMPONENTS
  std_msgs
  message_generation
)

add_message_files(
  FILES
  Age.msg
)

generate_messages(
  DEPENDENCIES
  std_msgs
)


catkin_package(
  CATKIN_DEPENDS rospy message_runtime
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

4) **Modify package.xml**

Just add these 3 lines to the package.xml file.

In [ ]:

```
<build_depend>message_generation</build_depend>

<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

This is the minimum expression of the package.xml

Note: Keep in mind that the name of the package in the following example is **topic_ex**, so in your case, the name of the package may be different.

package.xml

In [ ]:

```
<?xml version="1.0"?>
<package format="2">
  <name>topic_ex</name>
  <version>0.0.0</version>
  <description>The topic_ex package</description>


  <maintainer email="user@todo.todo">user</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <build_export_depend>rospy</build_export_depend>
  <exec_depend>rospy</exec_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>std_msgs</exec_depend>
  <build_export_depend>message_runtime</build_export_depend>
  <exec_depend>message_runtime</exec_depend>
  <export>

  </export>
</package>
```

5) Now you have to compile the msgs. To do this, you have to type in a Terminal:

▶ Execute in Terminal #1

In [ ]:

```
roscd; cd ..
```

In [ ]:
```
catkin_make
```

In [ ]:
```
source devel/setup.bash
```

VERY IMPORTANT: When you compile new messages, there is still an extra step before you can use the messages. You have to type in the Terminal, in the **catkin_ws** directory, the following command: **source devel/setup.bash**.

This executes this bash file that sets, among other things, the newly generated messages created through the catkin_make.

If you don't do this, it might give you a python import error, saying it doesn't find the message generated.

HINT 2: To verify that your message has been created successfully, type in your Terminal `rosmsg show Age`. If the structure of the Age message appears, it will mean that your message has been created successfully and it's ready to be used in your ROS programs.

▶ Execute in Terminal #1

In [ ]:
```
rosmsg show Age
```

Terminal #1 Output

```
[topic_ex/Age]:
float32 years
float32 months
float32 days
```

WARNING

There is an issue in ROS that could give you problems when importing msgs from the msg directory. If your package has the same name as the Python file that does the import of the msg, this will give an error saying that it doesn't find the msg element. This is due to the way Python works. Therefore, you have to be careful to not name the Python file exactly the same as its parent package.

Example:
Package name = "my_package"
Python file name = "my_package.py"

This will give an import error because it will try to import the message from the **my_package.py** file, from a directory **.msg** that doesn't exists.

4.3 Topics Quiz

With all you've learned during this course, you're now able to do a small Quiz to put everything together. Subscribers, Publisher, Messages... you will need to use all of this concepts in order to succeed!

For evaluating this Quiz, we will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly test your Quiz, and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

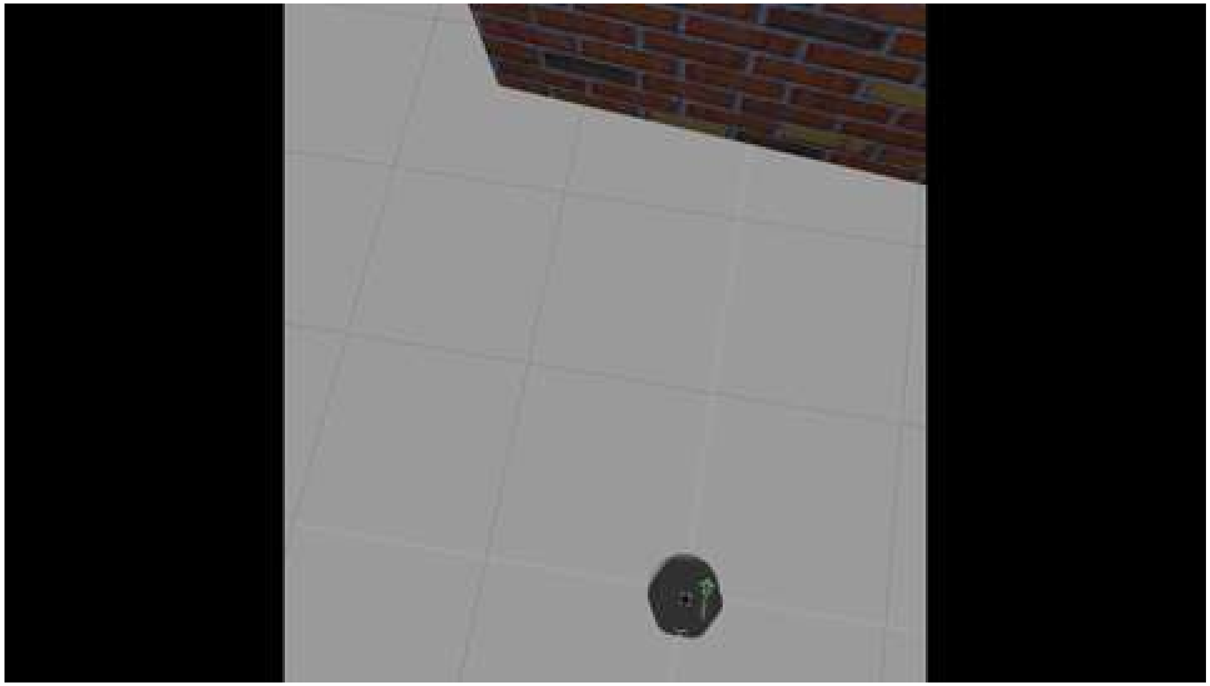
1. Create a Publisher that writes into the `/cmd_vel` topic in order to move the robot.
2. Create a Subscriber that reads from the `/kobuki/laser/scan topic` . This is the topic where the laser publishes its data.
3. Depending on the readings you receive from the laser's topic, you'll have to change the data you're sending to the `/cmd_vel` topic in order to avoid the wall. This means, use the values of the laser to decide.

Your program should follow the following logic:

1. If the laser reading in front of the robot is higher than 1 meter (there is no obstacle closer than 1 meter in front of the robot), the robot will move forward.
2. If the laser reading in front of the robot is lower than 1 meter (there is an obstacle closer than 1 meter in front of the robot), the robot will turn left.
3. If the laser reading at the right side of the robot is lower than 1 meter (there is an obstacle closer than 1 meter at the right side of the robot), the robot will turn left.
4. If the laser reading at the left side of the robot is lower than 1 meter (there is an obstacle closer than 1 meter at the left side of the robot), the robot will turn right.

The logic explained above has to result in a behavior like the following:

The robot starts moving forward until it detects an obstacle in front of it which is closer than 1 meter. Then it begins to turn left in order to avoid it.



The robot keeps turning left and moving forward until it detects that it has an obstacle at the right side which is closer than 1 meter. Then it turns left in order to avoid it.

Finally, the robot will continue moving forward since it won't detect any obstacle (closer than 1 meter) neither in front of it nor in its sides.

HINT 1: The data that is published into the `/kobuki/laser/scan` topic has a large structure. For this project, you just have to pay attention to the 'ranges' array.

▶ Execute in Terminal #1

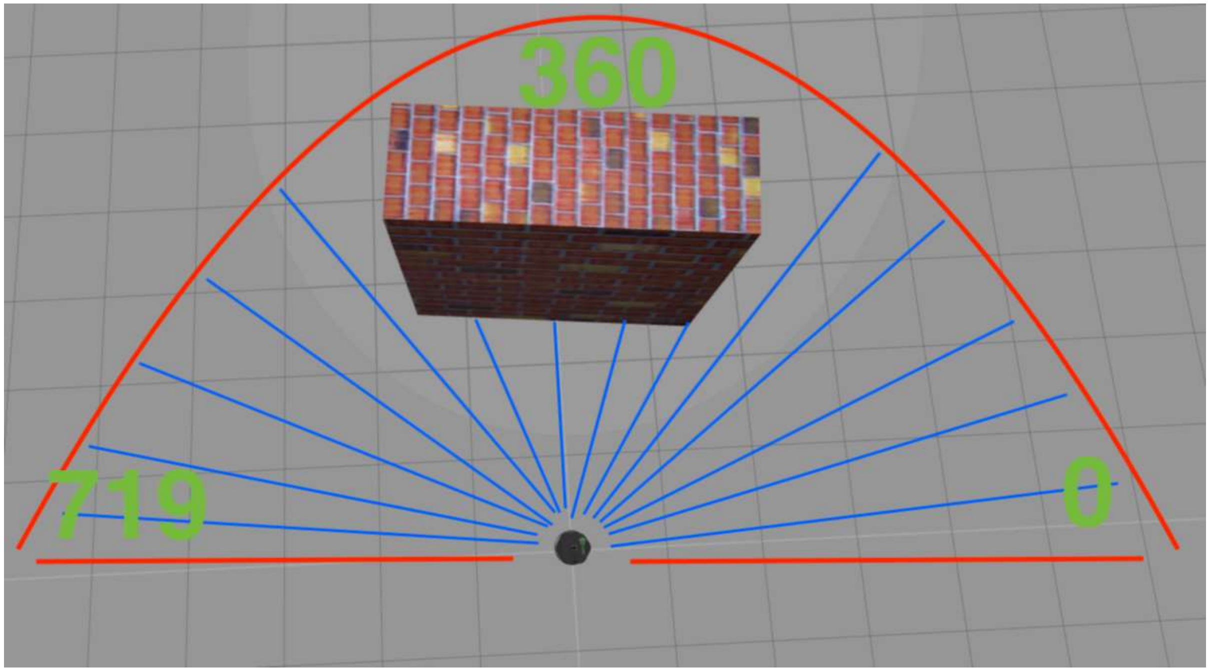
In [ ]:
```
rosmsg show sensor_msgs/LaserScan
```

Terminal #1 Output

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges <-- Use only this one
float32[] intensities
```

HINT 2: The 'ranges' array has a lot of values. The ones that are in the middle of the array represent the distances that the laser is detecting right in front of it. This means that the values in the middle of the array will be the ones that detect the wall. So in order to avoid the wall, you just have to read these values.

HINT 3: The scope of the laser is about 180 degrees from right to left. This means that the values at the beginning and at the end of the 'ranges' array will be the ones related to the readings on the sides of the laser (left and right), while the values in the middle of the array will be the ones related to the front of the laser. Have a look at the image below in order to better understand this.



HINT 4: The laser has a range of 30m. So, if you get a value that is under 30, this will mean that the laser is detecting some kind of obstacle in that direction (the wall). When the laser doesn't detect any obstacle on its way, it returns an `inf` (infinite) value.

Specifications

- The name of the package where you'll place all the code related to the Quiz will be **topics_quiz**.
- The name of the launch file that will start your program will be **topics_quiz.launch**.
- The name of the ROS node that will be launched by your program will be **topics_quiz_node**. *This is the node name you **must** specify in the launch file, regardless of what you specify in the script*.
- Before correcting your Quiz, make sure that all your Python scripts are executalbe. They need to have full execution permissions in order to be executed by our autocorrection system. You can give them full execution permissions with the following command:

In [ ]:
```
chmod +x my_script.py
```

- Before correcting your Quiz, make sure you have terminated all the programs in your Web Terminals.

Grading Guide

The following will be checked, in order. *If a step fails, the steps following are skipped.*

1. Does the package exist?
2. Did the package compile successfully?
3. Can the package be launched with the launch file?
4. Was the subscriber created as specified?
5. Was the robot publishing to `/cmd_vel` ?
6. Did the robot avoid the obstacle?

The grader will provide as much feedback on any failed step so you can make corrections where necessary.

Quiz Correction

When you have finished the Quiz, you can correct it in order to get a Mark. For that, just click on the following button at the top of this Notebook.

Final Mark

In case you fail the Quiz, or you don't get the desired mark, do not get frustrated! You will have the chance to resend the Quiz in order to improve your score.

It is now time that you start the project for this course!

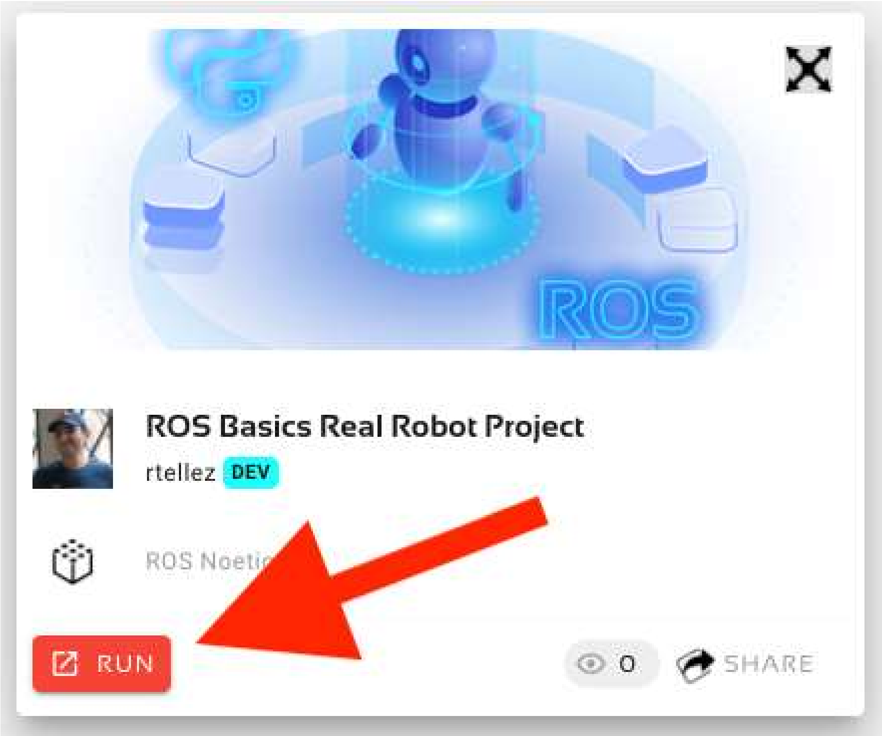
The project is going to be done in a different environment, which we call the **ROS Development Studio** (ROSDS). The ROSDS is an environment closer to what you will find when programming robots for companies. It is not as guided as this academy.

The ROSDS is included with your subscription, and is integrated inside The Construct. So no extra effort needs to be made by you. Well... you will need to expend some extra learning effort! But that's why you are here!

To start the project, you first need to get a copy of the ROS project (rosject), which contains the project instructions. Do the following:

1. **Copy the project rosject** to your ROSDS area (see instructions below).
2. Once you have it, go to the *My Rosjects* area in The Construct

3. **Open the rosject** by clicking *Run* on this course rosject



1. Then follow the instructions of the rosject to **finish the PART I of the project**.

You can now copy the project rosject by [clicking here (https://app.theconstructsim.com/#/l/3de1ad91/)](https://app.theconstructsim.com/#/l/3de1ad91/). This will automatically make a copy of it.

You should finish PART I of the rosject before attempting next unit of this course!

English proofread