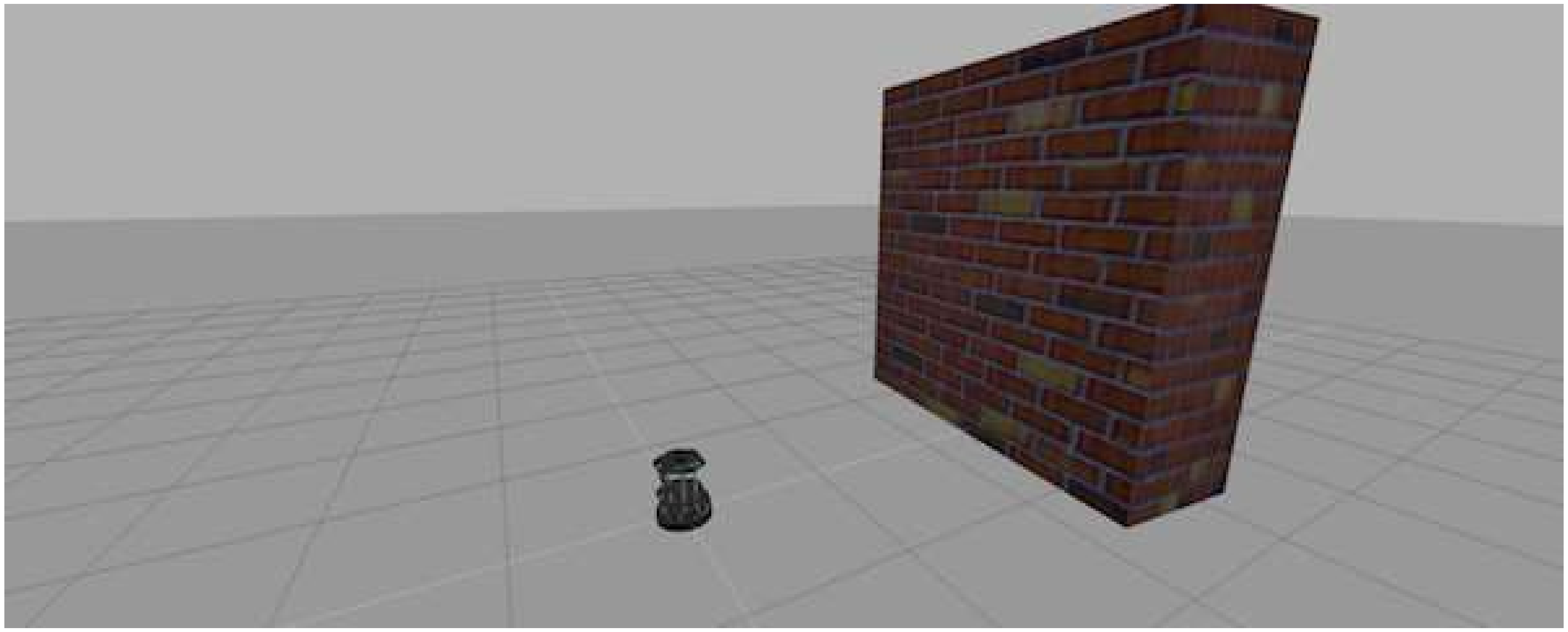
ROS Basics in 5 days (C++)

Unit 4 Understanding ROS Topics: Subscribers & Messages

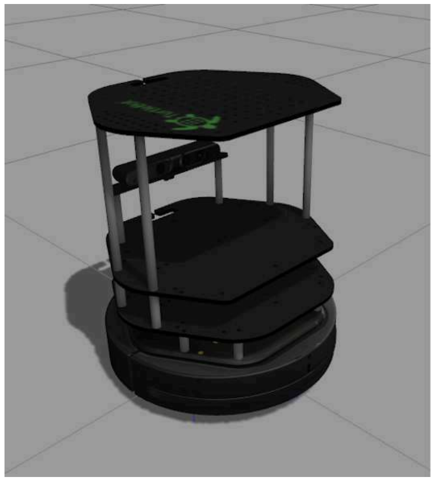


- Summary -

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



- 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 4.3 -

- Create a new package named **topic_subscriber_pkg**. When creating the package, add as dependencies **roscpp** and **std_msgs**.
- Inside the src folder of the package, create a new file named **simple_topic_subscriber.cpp**. Inside this file, copy the contents of simple_topic_subscriber.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

- End of Example 4.3 -

C++ Program {4.2}: simple_topic_subscriber.cpp

In [ ]:

```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

void counterCallback(const std_msgs::Int32::ConstPtr& msg)
{
  ROS_INFO("%d", msg->data);
}

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

    ros::init(argc, argv, "topic_subscriber");
    ros::NodeHandle nh;

    ros::Subscriber sub = nh.subscribe("counter", 1000, counterCallback);

    ros::spin();

    return 0;
}
```

END C++ Program {4.2}: simple_topic_subscriber.cpp

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

Go to your webshell and type the following:

▶ Execute in Shell #1

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

You should see an output like this:

Shell #1 Output

In [ ]:
```
user ~ $ rostopic echo /counter
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 webshell (leave the one with the ***rostopic echo*** opened) and type the next command:

▶ Execute in Shell #2

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

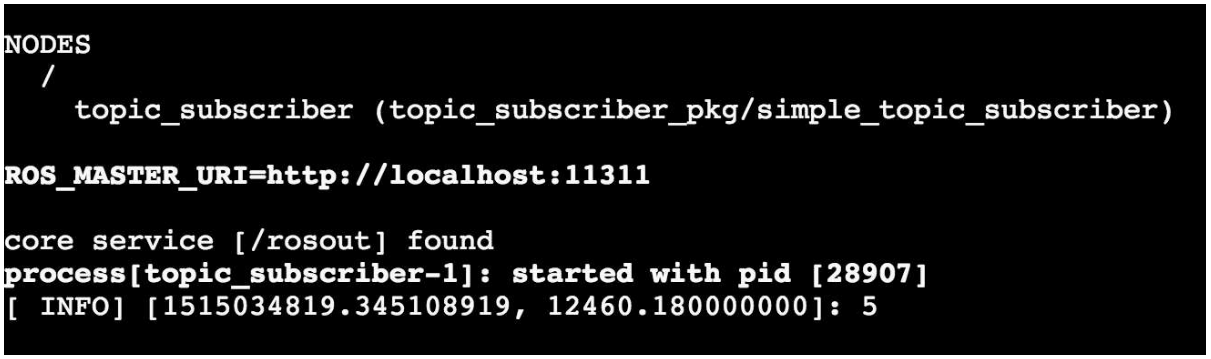
Now check the output of the console where you did the ***rostopic echo*** again. You should see something like this: .

Shell #1 Output

In [ ]:
```
user ~ $ rostopic echo /counter
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).

Now check the output of the shell where you executed your subscriber code. You should now see something like this:



```
NODES
  /
    topic_subscriber (topic_subscriber_pkg/simple_topic_subscriber)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[topic_subscriber-1]: started with pid [28907]
[ INFO] [1515034819.345108919, 12460.180000000]: 5
```

Before explaining everything with more detail, let's explain the code you executed.

In [ ]:
```
#include <ros/ros.h>
#include <std_msgs/Int32.h>

void counterCallback(const std_msgs::Int32::ConstPtr& msg) // Define a function called 'callback' that receiv
{
  ROS_INFO("%d", msg->data); // Print the value 'data' inside the 'msg' parameter
}

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

    ros::init(argc, argv, "topic_subscriber"); // Initiate a Node called 'topic_subscriber'
    ros::NodeHandle nh;

    ros::Subscriber sub = nh.subscribe("counter", 1000, counterCallback); // Create a Subscriber object that
                                                                          // call the 'callback' function eac

    ros::spin(); // Create a loop that will keep the program in execution

    return 0;
}
```

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 your subscriber has printed that number 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 learned!

- Exercise 4.4 -

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

- End of Exercise 4.4 -

- Notes for Exercise 4.4 -

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.**
3. **Remember to compile again your package in order to update your executable.**

- End of Notes -

- Exercise 4.5 -

1. Add to the {Exercice 4.4}, a C++ file that creates a publisher that indicates the age of the robot, to the previous package.
2. For that, you'll need to create a new message called Age.msg. See the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

- End of Exercise 4.5 -

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>
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
       roscpp
       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 roscpp std_msgs message_runtime   # This will NOT be the only thing here
)
```

Summarizing, this is the minimum expression of what is needed for the CMakaelist.txt 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
  roscpp
  std_msgs
  message_generation
)

add_message_files(
  FILES
  Age.msg
)

generate_messages(
  DEPENDENCIES
  std_msgs
)

catkin_package(
  CATKIN_DEPENDS roscpp std_msgs 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.

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>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <exec_depend>roscpp</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 WebShell:

▶ Execute in Shell #1

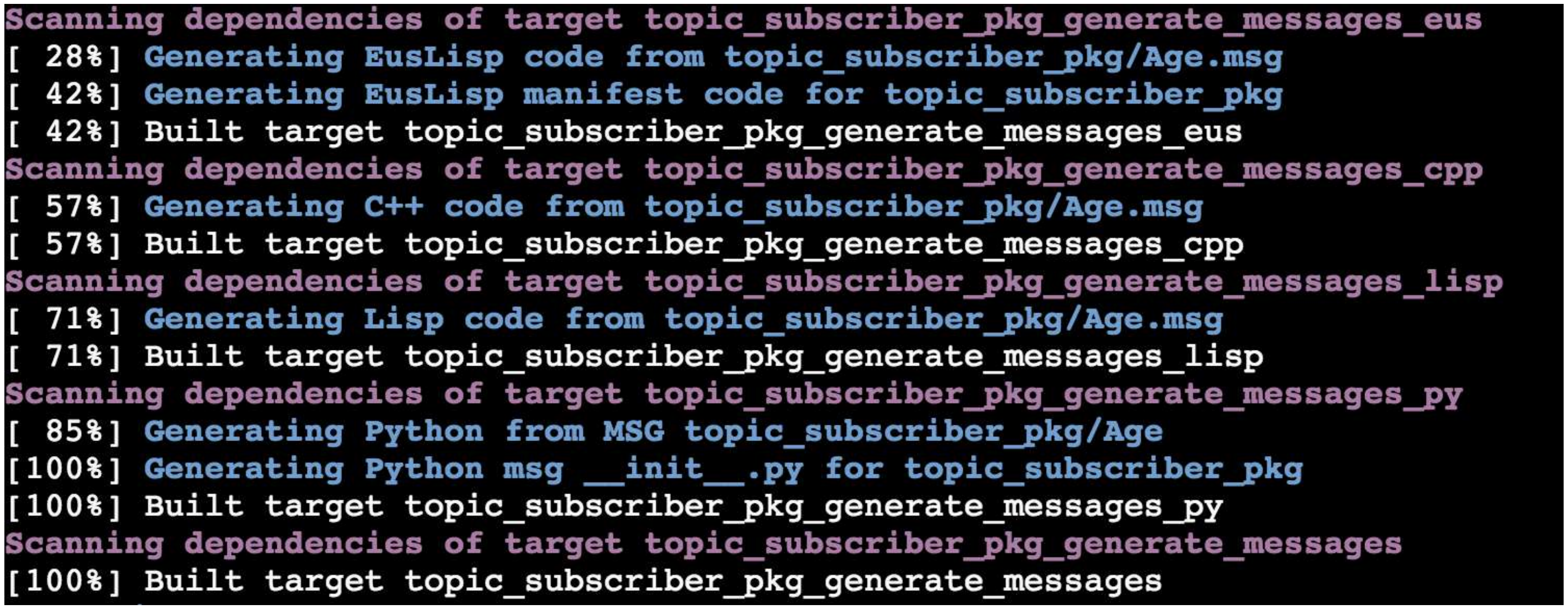
In [ ]:
```
roscd; cd ..
catkin_make
source devel/setup.bash
```

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 Webshell, in the **catkin_ws**, 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 an import error, saying it doesn't find the message generated.

If your compilation goes fine, you should see something similar to this:


```
Scanning dependencies of target topic_subscriber_pkg_generate_messages_eus
[ 28%] Generating EusLisp code from topic_subscriber_pkg/Age.msg
[ 42%] Generating EusLisp manifest code for topic_subscriber_pkg
[ 42%] Built target topic_subscriber_pkg_generate_messages_eus
Scanning dependencies of target topic_subscriber_pkg_generate_messages_cpp
[ 57%] Generating C++ code from topic_subscriber_pkg/Age.msg
[ 57%] Built target topic_subscriber_pkg_generate_messages_cpp
Scanning dependencies of target topic_subscriber_pkg_generate_messages_lisp
[ 71%] Generating Lisp code from topic_subscriber_pkg/Age.msg
[ 71%] Built target topic_subscriber_pkg_generate_messages_lisp
Scanning dependencies of target topic_subscriber_pkg_generate_messages_py
[ 85%] Generating Python from MSG topic_subscriber_pkg/Age
[100%] Generating Python msg __init__.py for topic_subscriber_pkg
[100%] Built target topic_subscriber_pkg_generate_messages_py
Scanning dependencies of target topic_subscriber_pkg_generate_messages
[100%] Built target topic_subscriber_pkg_generate_messages
```

HINT 2: To verify that your message has been created successfully, type in your webshell *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 Shell #1

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

Shell #1 Output

In [ ]:
```
user ~ $ rosmsg show Age
[topic_ex/Age]:
float32 years
float32 months
float32 days
```

4.3 Use custom messages in your programs

You will have to add to your **CMakeLists.txt** the following extra lines to compile and link your executable (in this example its called **publish_age.cpp**)
:

In [ ]:
```
add_executable(publish_age src/publish_age.cpp)
add_dependencies(publish_age ${publish_age_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
target_link_libraries(publish_age
   ${catkin_LIBRARIES}
 )
add_dependencies(publish_age topic_ex_generate_messages_cpp)
```

As you can see, there is nothing new here except for the last line:

In [ ]:
```
add_dependencies(publish_age topic_ex_generate_messages_cpp)
```

And why do you need to add this extra **add_dependencies()**? Well, this is to make sure that all the messages contained in the package (**topic_ex**) are compiled before we create the **publish_age** executable. In this case, the **publish_age** executable uses the custom message we have just created: **Age.msg**. So... what would happen if we try to build the executable before those messages are built? Well, it would fail, of course. Then, with this line, you will make sure that the messages are built before trying to build your executable.

4.4 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 Cpp 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.

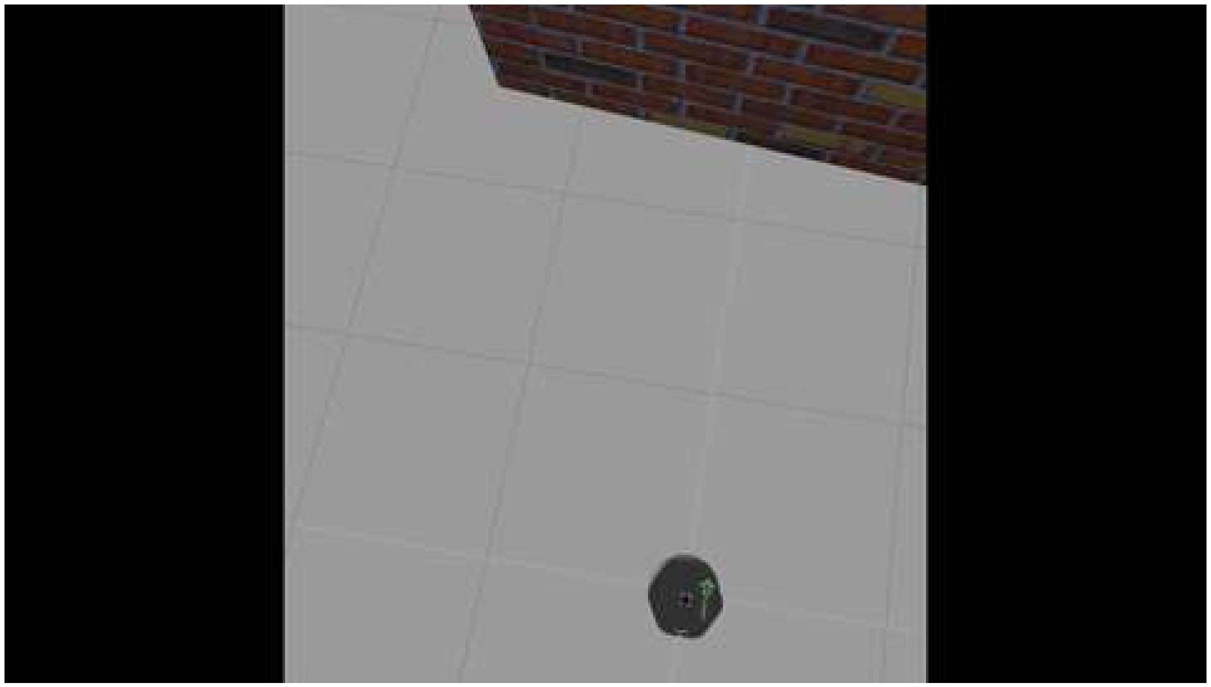
1. Create a Publisher that writes into the `/cmd_vel` topic in order to move the robot.
2. Create a Subscriber that reads from the `/kobuki/laser/scan topic` . This is the topic where the laser publishes its data.
3. Depending on the readings you receive from the laser's topic, you'll have to change 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 Shell #1

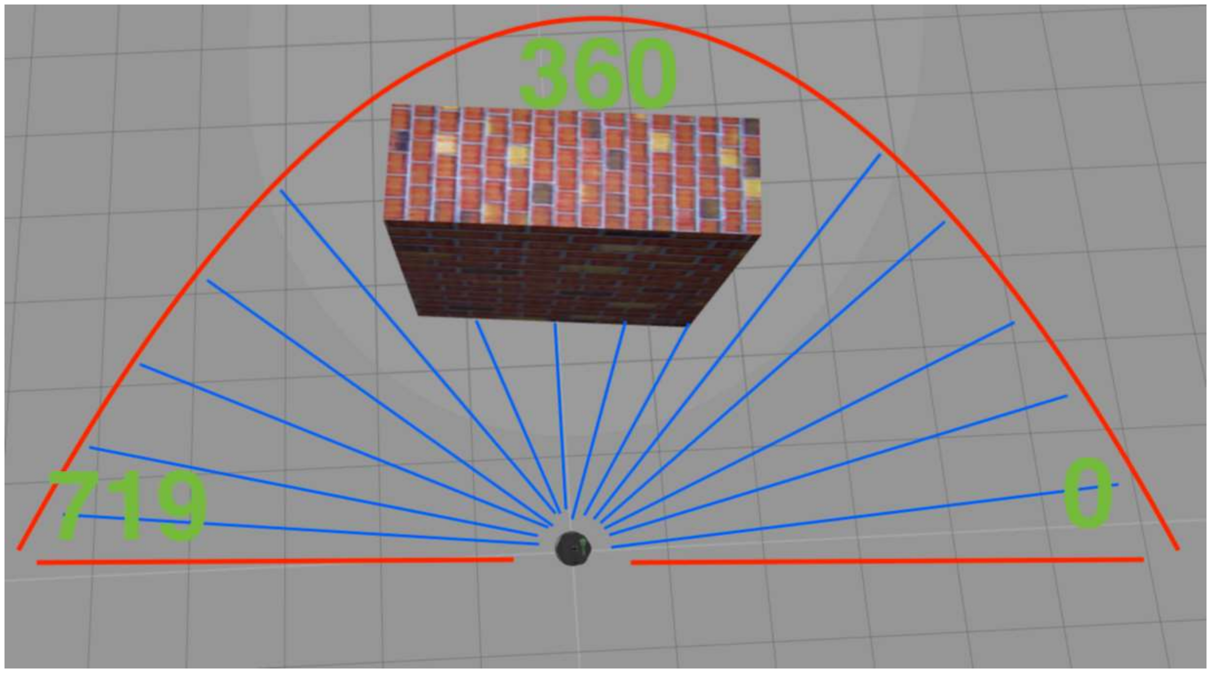
In [ ]:
```
rosmsg show sensor_msgs/LaserScan
```

Shell #1 Output

In [ ]:
```
user ~ $ rosmsg show sensor_msgs/LaserScan
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 him. 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 you have terminated all the programs in your Web Shells.

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.

NEW IMPORTANT UPDATE FOR THIS COURSE !!!

(done on 18th Feb 2021)

Starting from the 18th of February, **the Unit 11 Project of this course is deprecated**.

From that day on, the Unit 11 Project of this course must be done in a completely different way (see instructions below).

The reason for this change is to include a more realistic project and to **provide you with practice with real robots**. Remember that everything is included in your subscription at no extra cost.

If you have already started the Unit 11 Project for this course, you can keep doing it and finish it. You don't need to change to the new project. However, the Unit 11 Project will definitely be cancelled on March 31st, so you have to finish it before that date.

If you haven't started the Unit 11 Project yet, then don't do it. Instead, go to the new project, the procedure for which is explained below.

We are always thinking of ways to provide you with the best learning experience. We hope this change will help you better understand how to program robots with ROS.

Each course in this academy includes a project that must be solved applying the knowledge gained from the whole course.

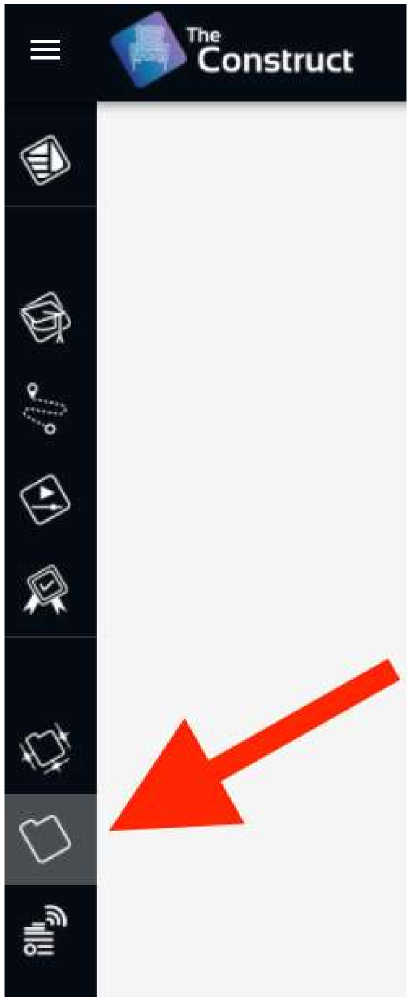
It is now time that you start the project for this course!

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.

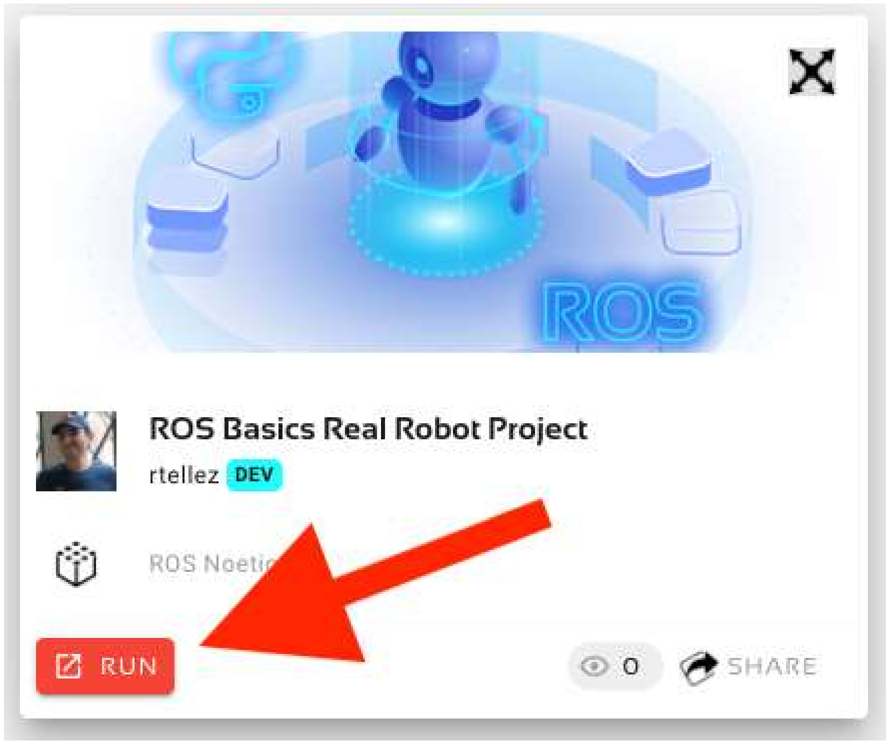
The ROSDS is included with your subscription, and is integrated inside The Construct. So no extra effort needs to be made by you. Well... you will need to expend some extra learning effort! But that's why you are here!

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