**Department of Electrical Engineering and   
Computer Science**

**Faculty Member:** Dr. Hafsa Iqbal **Date:** 20/02/2024

**Semester:** 8th **Group:** GP-1

**Lab Engineer:** Munadi Ahmad Sial

**EE-381 Robotics**

Lab 3: Introduction to Robot Operating System

**Group Members**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **PLO5 – CLO4** | | **PLO9 –**  **CLO4** |  |
| **Name** | **Reg. No** | **Analysis of Data in Lab Report** | **Modern Tool Usage** | **Individual and Team Work** | **Total Marks** |
|  |  | **10 Marks** | **5 Marks** | **5 Marks** | **20 Marks** |
| Muhammad Abdullah Sohail | 343642 |  |  |  |  |
| Muhammad Ahmed Mohsin | 333060 |  |  |  |  |
| Muhammad Umer | 345834 |  |  |  |  |
| Hassan Rizwan | 335753 |  |  |  |  |
| Muhammad Saad | 333414 |  |  |  |  |

**Table of Contents**

[2 Introduction to Robot Operating System 3](#_Toc159523230)

[2.1 Introduction 3](#_Toc159523231)

[2.2 Objectives 3](#_Toc159523232)

[2.3 Theory 3](#_Toc159523233)

[3 Lab Tasks 4](#_Toc159523234)

[3.1 Task 1 – Turtlesim 4](#_Toc159523235)

[3.2 Task 2 – RQt 5](#_Toc159523236)

[3.3 Task 3 – Workspaces and Packages 7](#_Toc159523237)

[3.4 Task 4 – Node 1 8](#_Toc159523238)

[3.5 Task 5 – Node 2 – Bubble Sort 9](#_Toc159523239)

[3.6 Task 6 – Node 3 – Find the Angles 11](#_Toc159523240)

[4 Conclusion 17](#_Toc159523241)

# Introduction to Robot Operating System

## Introduction

This laboratory exercise is meant to introduce ROS for programming robots. ROS is a framework for creating and executing nodes that communicate with each other by passing messages. Each node is assigned a modular objective pertaining to robotics such as laser scanning, teleoperation, camera, navigation, localization, mapping and so on.

## Objectives

The following are the main objectives of this lab:

* Learn to use the terminal for basic commands
* Initialize a workspace for ROS 2 environment
* Create packages within the workspace
* Create nodes scripted by python in a package
* Build the workspace with the colcon build tool
* Execute nodes from the terminal

## Theory

Robot Operating System or ROS is a framework for programming robotic platforms in which nodes communicate with other nodes. ROS is widely used within the Linux Ubuntu operating system. To use ROS, a workspace must be created to set up the programming environment for robots. Next, packages can be contained in the workspace. Each package contains the necessary files such as the code, model files and maps etc. that are needed for implementing the robotic simulation. For programming in python, the SublimeText editor is a useful tool. To use ROS effectively, it is necessary to utilize the terminal.

The terminal commands are given as:

**cd <directory>**  change directory

**cd..**  go back to previous directory

**pwd**  print the current directory

**ls** list the contents of the current directory

**python <script.py>** execute python script

# Lab Tasks

## Task 1 – Turtlesim

Turtlesim is a lightweight simulator that shows what ROS 2 can do at the most basic level. We will use turtlesim in this task to get an idea of how ROS 2 works. This is necessary before working on robot simulation.

First, use the terminal to check if you have turtlesim installed or not. You should get a list of Turtlesim’s executables with the following command:

ros2 pkg executables turtlesim

If turtlesim is not installed, use the following commands to install it:

sudo apt update

sudo apt install ros-humble-turtlesim

Launch turtlesim by the following command in the terminal:

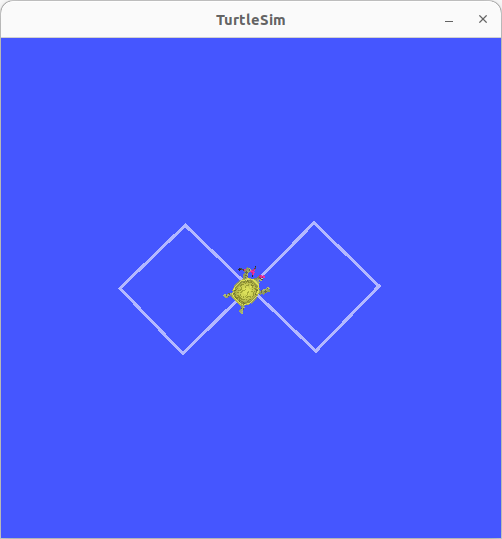
ros2 run turtlesim turtlesim\_node

A new window should appear showing a turtle. Now open a second terminal and type the following command:

ros2 run turtlesim turtle\_teleop\_key

With the second terminal active, press the arrow keys and the turtle should move, leaving a line as it travels.

### TURTLE PATTERN SCREENSHOTS START HERE ###



### TURTLE PATTERN SCREENSHOTS END HERE ###

Now open another terminal and type the following commands one by one. You need to paste a single screenshot of the terminal showing the commands and the output. Keep all commands in the same terminal:

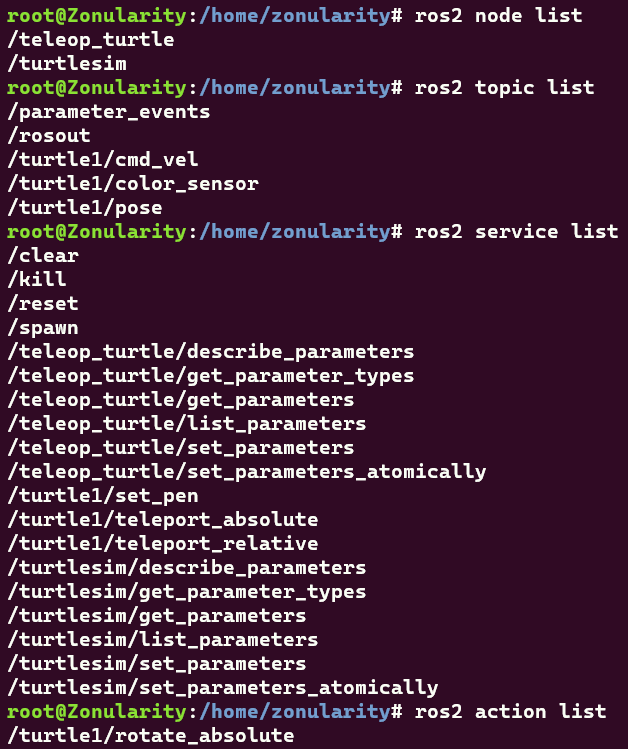
ros2 node list

ros2 topic list

ros2 service list

ros2 action list

### TERMINAL SCREENSHOT STARTS HERE ###



### TERMINAL SCREENSHOT ENDS HERE ###

## Task 2 – RQt

RQt is a GUI tool for ROS 2. Everything done in RQt can also be done on the terminal but it provides a more user-friendly way to handle aspects of ROS. We will get an insight into RQt and for that, we will use the turtlesim simulator again. We will use the service caller (you do not have to concern with the details of the service caller in this lab). Launch RQt:

rqt

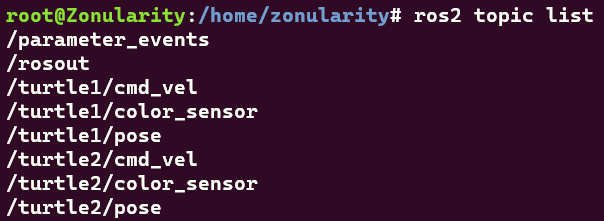
If RQt is not installed, use the following commands to install it:

sudo apt update

sudo apt install ~nros-humble-rqt\*

Reopen the turtlesim simulator and make it move with the arrow keys to draw a few white lines. Next, open up RQt from a new terminal. A blank window will appear. Go to Plugins > Services > Service Caller. The Service Caller will now show in RQt. Use the refresh button, then select /spawn in the service dropdown list. Set the x position at 1 and y position at 2. Change the name to ‘turtle2’. Then, press the call button to spawn a new turtle. Now use the ros2 topic list command and you should see some new additions. Provide the terminal screenshot showing the listed topics:

### TASK 2 TERMINAL SCREENSHOT STARTS HERE ###

******

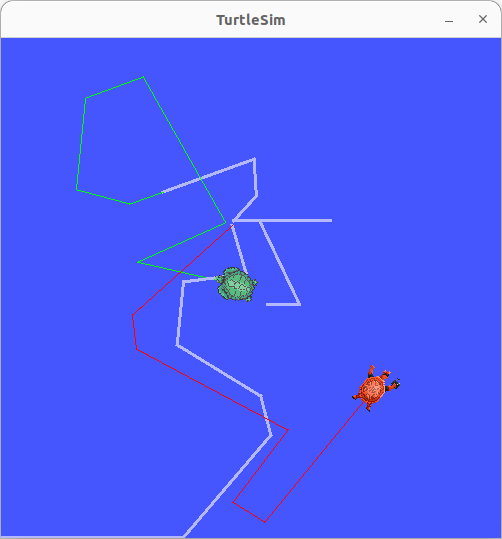
### TASK 2 TERMINAL SCREENSHOT ENDS HERE ###

Next, go to the /turtle1/set\_pen service in the dropdown. Change the g value to 255 and width to 5. Then, press the call button and the turtle should now make a green line as it moves. Move the turtle around and make a few green lines. To move the second turtle, you need to open a new terminal and run the teleop\_key node but this time by remapping for the second turtle:

ros2 run turtlesim turtle\_teleop\_key --ros-args –remap turtle1/cmd\_vel:=turtle2/cmd\_vel

You need to provide a screenshot in which both the turtle’s movements will be shown. You need to first move both the turtles so that their white lines are shown. Next, change the line colors so that turtle1 gives green lines and turtle2 gives red lines. Move both the turtles again and draw a few more lines.

### DOUBLE TURTLE SCREENSHOT STARTS HERE ###



### DOUBLE TURTLE SCREENSHOT ENDS HERE ###

## Task 3 – Workspaces and Packages

Before creating a workspace, you need to install colcon which is used to build the workspace:

sudo apt install python3-colcon-common-extensions

Create your workspace in the home directory:

mkdir -p ~/ros2\_ws/src

From the src folder, create a package:

ros2 pkg create –-build-type ament\_python package1

From the src folder, create another package with following syntax which also creates a minimalist node:

ros2 pkg create –-build-type ament\_python –-node-name node\_test1 package2

Go to the folder where node\_test1.py is present and make two more copies of it. Rename the copies as node\_test2 and node\_test3. You don’t have to use the terminal for this. Open the new python files and edit the display messages. Next, open the setup.py file in the package and add the entry points for node\_test2 and node\_test3. Use commas to separate the entries. Save the setup file.

Next, open a terminal and go to the root of your workspace. Build the workspace:

colcon build

Next, source the setup file:

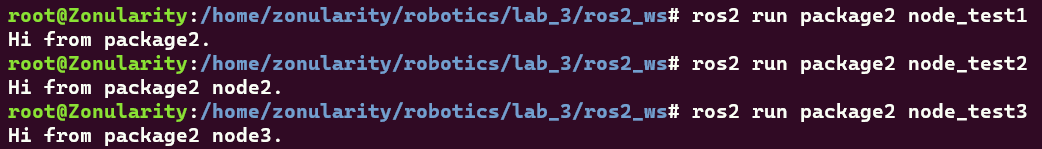
. install/setup.bash

Run the nodes with the following syntax:

ros2 run <package\_name> <node\_name>

Run the 3 nodes from the same terminal and paste the screenshot:

### TASK 3 SCREENSHOT STARTS HERE ###

******

### TASK 3 SCREENSHOT ENDS HERE ###

## Task 4 – Node 1

In the workspace you have created in the previous task, create a new package with rclpy dependency:

ros2 pkg create –-build-type ament\_python package\_group1 –-dependencies rclpy –-node-name node1

You can see the dependencies in the packages.xml file. Next, open up node1.py and replace with the following code:

import rclpy

from rclpy.node import Node

def node1():

# WRITE YOUR TASK CODE IN THIS FUNCTION

class MyNode(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('my\_node\_name')

node1()

def main(args=None):

rclpy.init(args=args)

node = MyNode()

rclpy.spin(node)

rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Provide a name for your node in the super function. In the node1 function, you will write the task code:

Write a program which takes 5 number inputs from the user and then prints their average. You need to give the code and the screenshot of the execution in the terminal.

### NODE 1 CODE STARTS HERE ###

import rclpy

from rclpy.node import Node

*def* node1():

    in\_list = []

    for i in range(5):

        x = *int*(input("Enter a number: "))

        in\_list.append(x)

    avg = sum(in\_list) / len(in\_list)

    print("Average of 5 numbers is: ", avg)

*class* MyNode(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("node\_umer\_1")

        node1()

*def* main(*args*=None):

    rclpy.init(*args*=args)

    node = MyNode()

    rclpy.spin(node)

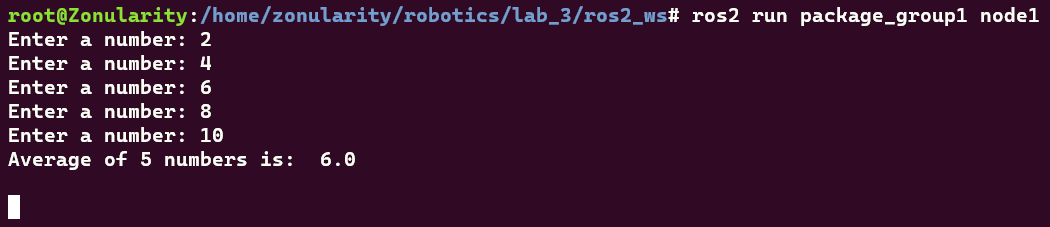
    rclpy.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

### NODE 1 CODE ENDS HERE ###

### NODE 1 SCREENSHOT STARTS HERE ###



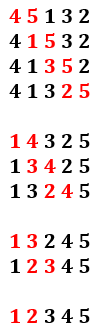
### NODE 1 SCREENSHOT ENDS HERE ###

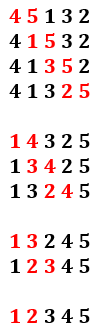
You will use the same procedure as in the previous task to create nodes 2 and 3 in the same package. You must provide both the code and screenshot of execution in terminal for these nodes.

## Task 5 – Node 2 – Bubble Sort

In the bubble sort algorithm, smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom. The bubble sort makes several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array. The comparisons on each pass proceed as follows—the 0th element value is compared to the 1st, the 1st is compared to the 2nd, the 2nd is compared to the third.

The following picture gives an idea of how bubble sort works. The red numbers are the ones being compared for the sorting.

****

****

Write a program that sorts the list [8, 7, 2, 3, 5, 10, 9, 1, 4, 6] using bubble sort. Your program must output the list each time a swap is made.

### NODE 2 CODE STARTS HERE ###

import rclpy

from rclpy.node import Node

*def* node2():

    list\_ts = [8, 7, 2, 3, 5, 10, 9, 1, 4, 6]

    for i in range(len(list\_ts)):

        for j in range(len(list\_ts) - i - 1):

            if list\_ts[j] > list\_ts[j + 1]:

                list\_ts[j], list\_ts[j + 1] = list\_ts[j + 1], list\_ts[j]

                print(list\_ts)

    print("Sorted list is: ", list\_ts)

*class* MyNode(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("node\_umer\_2")

        node2()

*def* main(*args*=None):

    rclpy.init(*args*=args)

    node = MyNode()

    rclpy.spin(node)

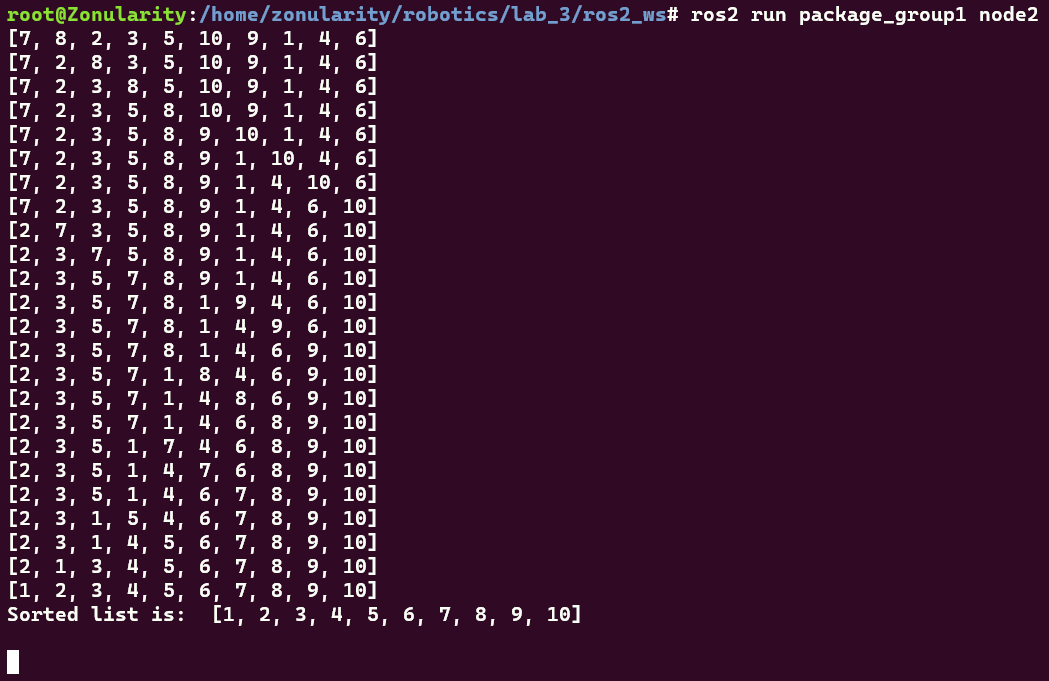
    rclpy.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

### NODE 2 CODE ENDS HERE ###

### NODE 2 SCREENSHOT STARTS HERE ###

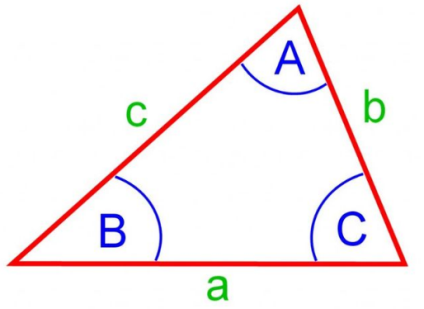


### NODE 2 SCREENSHOT ENDS HERE ###

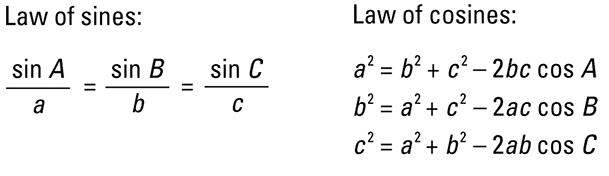
## Task 6 – Node 3 – Find the Angles

In motion planning, collision detection is very important. One way to detect collisions is to mesh the robot and obstacle models as being made up of triangles. Then, the collisions are detected when a triangle in the robot overlaps with a triangle of an obstacle. Because of this, trigonometry is an important requirement. This task serves as a trigonometry exercise as well as a program designing exercise.

A triangle is a polygon which has 3 side lengths (a, b, and c) and 3 angles (A, B and C). Thus, there are a total of 6 parameters in a triangle. If we know 3 of these parameters of which at least 1 is a side length, then we can determine the remaining 3 parameters. This is done by Pythagoras theorem, Law of Sines and Law of Cosines. In this task, you will create a program that determines the unknown angles of a triangle (You don’t need to calculate the unknown side lengths if it’s not necessary).

****

The program will first require some inputs in which the user will input the 3 known parameters (lengths or angles) of which at least 1 must be a side length. Then, using the given inputs, the program must determine the remaining unknown angles. You need to make use of the law of Sines and Cosines for this task:

****

For this task, you will have to use the sine and cosine functions from the math library. Note that the sine and cosine functions take values in radians. You can convert angles to degrees using the math library.

### NODE 3 CODE STARTS HERE ###

import math

import rclpy

from rclpy.node import Node

*def* tri\_angle(*x*, *y*):

    return 180 - (x + y)

*def* node3():

    print("(abc) represents sides, (ABC) represents angles in degrees")

    lengths = {}

    angles = {}

    in\_str = input("Enter the unknown side (i.e., a=3): ")

    side, val = in\_str.split("=")

    val = *float*(val)

    lengths[side] = val

*# input remaining two parameters; can be sides or angles*

    for i in range(2):

        in\_str = input("Enter the unknown side or angle (i.e., a or A=60): ")

        key, val = in\_str.split("=")

        if key.islower():

            lengths[key] = *float*(val)

        else:

            angles[key] = *float*(val)

    assert len(angles) != 3, "At least one side must be known"

*# if all sides are known*

    if len(lengths) == 3:

        a, b, c = lengths.values()

        A = math.degrees(math.acos((b\*\*2 + c\*\*2 - a\*\*2) / (2 \* b \* c)))

        B = math.degrees(math.acos((c\*\*2 + a\*\*2 - b\*\*2) / (2 \* c \* a)))

        C = math.degrees(math.acos((a\*\*2 + b\*\*2 - c\*\*2) / (2 \* a \* b)))

        angles["A"] = A

        angles["B"] = B

        angles["C"] = C

*# if two sides and one angle is known*

    if len(lengths) == 2 and len(angles) == 1:

        if "a" not in lengths:

            if "A" in angles:

                a = math.sqrt(

                    lengths["b"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2

                    \* lengths["b"]

                    \* lengths["c"]

                    \* math.cos(math.radians(angles["A"]))

                )

            elif "B" in angles:

                C = math.asin(

                    lengths["c"] \* math.sin(math.radians(angles["B"]))

/ lengths["b"]

                )

                angles["C"] = math.degrees(C)

                A = tri\_angle(angles["B"], angles["C"])

                angles["A"] = A

                a = math.sqrt(

                    lengths["b"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2 \* lengths["b"] \* lengths["c"] \*

math.cos(math.radians(A))

                )

            else:

                B = math.asin(

                    lengths["b"] \* math.sin(math.radians(angles["C"]))

/ lengths["c"]

                )

                angles["B"] = math.degrees(B)

                A = tri\_angle(angles["B"], angles["C"])

                angles["A"] = A

                a = math.sqrt(

                    lengths["b"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2 \* lengths["b"] \* lengths["c"] \*

math.cos(math.radians(A))

                )

            lengths["a"] = a

        elif "b" not in lengths:

            if "B" in angles:

                b = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2

                    \* lengths["a"]

                    \* lengths["c"]

                    \* math.cos(math.radians(angles["B"]))

                )

            elif "A" in angles:

                C = math.asin(

                    lengths["c"] \* math.sin(math.radians(angles["A"]))

/ lengths["a"]

                )

                angles["C"] = math.degrees(C)

                B = tri\_angle(angles["A"], angles["C"])

                angles["B"] = B

                b = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2 \* lengths["a"] \* lengths["c"] \*

math.cos(math.radians(B))

                )

            else:

                A = math.asin(

                    lengths["a"] \* math.sin(math.radians(angles["C"]))

/ lengths["c"]

                )

                angles["A"] = math.degrees(A)

                B = tri\_angle(angles["A"], angles["C"])

                angles["B"] = B

                b = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["c"] \*\* 2

                    - 2 \* lengths["a"] \* lengths["c"] \*

math.cos(math.radians(B))

                )

            lengths["b"] = b

        else:

            if "C" in angles:

                c = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["b"] \*\* 2

                    - 2

                    \* lengths["a"]

                    \* lengths["b"]

                    \* math.cos(math.radians(angles["C"]))

                )

            elif "A" in angles:

                B = math.asin(

                    lengths["b"] \* math.sin(math.radians(angles["A"]))

/ lengths["a"]

                )

                angles["B"] = math.degrees(B)

                C = tri\_angle(angles["A"], angles["B"])

                angles["C"] = C

                c = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["b"] \*\* 2

                    - 2 \* lengths["a"] \* lengths["b"] \*

math.cos(math.radians(C))

                )

            else:

                A = math.asin(

                    lengths["a"] \* math.sin(math.radians(angles["B"]))

/ lengths["b"]

                )

                angles["A"] = math.degrees(A)

                C = tri\_angle(angles["A"], angles["B"])

                angles["C"] = C

                c = math.sqrt(

                    lengths["a"] \*\* 2

                    + lengths["b"] \*\* 2

                    - 2 \* lengths["a"] \* lengths["b"] \*

math.cos(math.radians(C))

                )

            lengths["c"] = c

*# if two angles and one side is known*

    if len(angles) == 2 and len(lengths) == 1:

        if "A" not in angles:

            A = tri\_angle(angles["B"], angles["C"])

            angles["A"] = A

        elif "B" not in angles:

            B = tri\_angle(angles["A"], angles["C"])

            angles["B"] = B

        else:

            C = tri\_angle(angles["A"], angles["B"])

            angles["C"] = C

        if "a" in lengths:

            b = (

                lengths["a"]

                \* math.sin(math.radians(angles["B"]))

                / math.sin(math.radians(angles["A"]))

            )

            c = (

                lengths["a"]

                \* math.sin(math.radians(angles["C"]))

                / math.sin(math.radians(angles["A"]))

            )

            lengths["b"] = b

            lengths["c"] = c

        elif "b" in lengths:

            a = (

                lengths["b"]

                \* math.sin(math.radians(angles["A"]))

                / math.sin(math.radians(angles["B"]))

            )

            c = (

                lengths["b"]

                \* math.sin(math.radians(angles["C"]))

                / math.sin(math.radians(angles["B"]))

            )

            lengths["a"] = a

            lengths["c"] = c

        else:

            a = (

                lengths["c"]

                \* math.sin(math.radians(angles["A"]))

                / math.sin(math.radians(angles["C"]))

            )

            b = (

                lengths["c"]

                \* math.sin(math.radians(angles["B"]))

                / math.sin(math.radians(angles["C"]))

            )

            lengths["a"] = a

            lengths["b"] = b

    print("\nResults:")

    print(lengths)

    print(angles)

*class* MyNode(*Node*):

*def* \_\_init\_\_(*self*):

*super*().\_\_init\_\_("node\_umer\_3")

        node3()

*def* main(*args*=None):

    rclpy.init(*args*=args)

    node = MyNode()

    rclpy.spin(node)

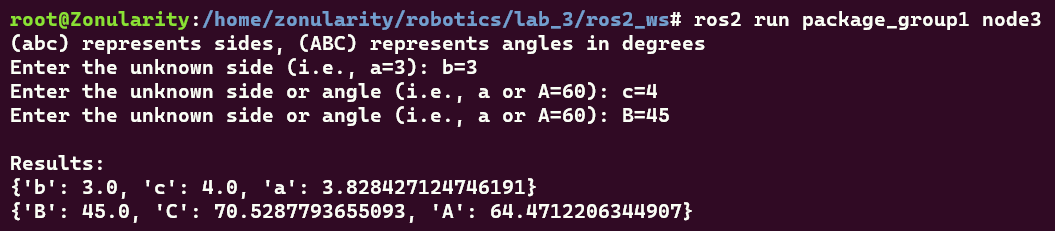
    rclpy.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

### NODE 3 CODE ENDS HERE ###

### NODE 3 SCREENSHOT STARTS HERE ###



### NODE 3 SCREENSHOT ENDS HERE ###

# Conclusion

In conclusion, our exploration into ROS for programming robots has provided us with a foundational understanding of its framework and capabilities. Through the creation and execution of nodes facilitating various functionalities including laser scanning, teleoperation, camera interfacing, navigation, localization, and mapping, we have grasped the modular nature of ROS and its proficiency in enabling seamless communication between nodes via message passing. This laboratory exercise has equipped us with essential skills to navigate and manipulate ROS, laying a solid groundwork for future endeavors in robotics programming and development.