ROBOT PROGRAMMING IE416

LAB 1 TO 17

- **KENI PATEL : 20210194**
- **UMANG TRIVEDI: 202101471**
- **KANISHK PAREKH: 202101134**

09.05.2024

VIDEO AND CODE LINK :

- https://drive.google.com/drive/folders/1uBp1Kfd9fpoQbp7zacNjY9Lr2O20rM13?us p=sharing

ROBOT PROGRAMING IE416

LAB 1 - 17

Here are the contents which must be included in the lab report:

IE416 Robot Programming

Group 15

Group Members:

202101194 Keni Patel

202101471 Umang Trivedi

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images
- IPython: Creating notebooks, Typical workflows

A Brief Note on Python Versions

As of Janurary 1, 2020, Python has officially dropped support for `python2` . We'll be using Python 3.7 for this iteration of the course. You can check your Python version at the command line by running `python --version` . In Colab, we can enforce the Python version by clicking `Runtime -> Change Runtime Type` and selecting `python3` . Note that as of April 2020, Colab uses Python 3.6.9 which should run everything without any errors.

```
!python --version
```

```
    Python 3.10.12
```

Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

```
def quicksort(arr):
     if len(arr) <= 1:
         return arr
pivot = arr[len(arr) // 2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
     right = [x for x in arr if x > pivot]
return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

```
     [1, 1, 2, 3, 6, 8, 10]
```

䅘 Basic data types

䅘 Numbers

Integers and floats work as you would expect from other languages:

```
x = 3
print(x, type(x))
```

```
     3 <class 'int'>
```

```
print(x + 1) # Addition
print(x - 1) # Subtraction
print(x * 2) # Multiplication
 print(x ** 2) # Exponentiation
```

```
     4
     2
     6
     9
```

```
x += 1
print(x)
 x *= 2
print(x)
```

```
     4
     8
```

```
y = 2.5
print(type(y))
print(y, y + 1, y * 2, y ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

䅇 Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&` , `||` , etc.):

```
t, f = True, False
 print(type(t))
```

```
<class 'bool'>
```

Now we let's look at the operations:

```
print(t and f) # Logical AND;
 print(t or f) # Logical OR;
 print(not t)      # Logical
 NOT; print(t != f) # Logical
 XOR;
```

```
False
True
False
True
```

䅇 Strings

```
hello = 'hello'    # String literals can use single quotes
 world = "world"   # or double quotes; it does not matter
 print(hello, len(hello))
```

```
hello 5
```

```
hw = hello + ' ' + world # String concatenation
 print(hw)
```

```
hello world
```

```
hw12 = '{} {} {}'.format(hello, world, 12) # string formatting
 print(hw12)
```

```
hello world 12
```

String objects have a bunch of useful methods; for example:

```
s = "hello"
print(s.capitalize()) # Capitalize a string
print(s.upper())   # Convert a string to uppercase; prints "HELLO"
 print(s.rjust(7)) # Right-justify a string, padding with spaces
 print(s.center(7))      # Center a string, padding with spaces
print(s.replace('l', '(ell)')) # Replace all instances of one substring with another
 print('     world '.strip()) # Strip leading and trailing whitespace
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

You can find a list of all string methods in the documentation.

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists

A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

```
xs = [3, 1, 2]     # Create a
list
print(xs[-  [2])  # Negative indices count from the end of the list; prints
1])                "2"
```

```
[3, 1, 2] 2
2
```

```
xs[2] =          # Lists can contain elements of different
'foo'            types
print(xs)
```

```
    [3, 1, 'foo']
```

```
xs.append('bar') # Add a new element to the end of the list
 print(xs)
```

```
    [3, 1, 'foo', 'bar']
```

```
 x =                   # Remove and return the last element of the
 xs.pop()              list
 print(x,
 xs)
```

```
    bar [3, 1, 'foo']
```

As usual, you can find all the gory details about lists in the documentation.

程 Slicing

In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

```
 nums =                       # range is a built-in function that creates a list of
 print(nums) # Prints "[0, 1, 2, 3, 4]"
 print(nums[2:4])   # Get a slice from index 2 to 4 (exclusive); prints "[2,
 3]" print(nums[2:])      # Get a slice from index 2 to the end; prints "[2,
 3, 4]"
 print(nums[:2])    # Get a slice from the start to index 2 (exclusive); prints "[0,
 1]" print(nums[:]) # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
 print(nums[:-1])   # Slice indices can be negative; prints ["0, 1, 2,
 3]" nums[2:4] = [8, 9] # Assign a new sublist to a slice
 print(nums) # Prints "[0, 1, 8, 9, 4]"
```

```
    [0, 1, 2, 3, 4]
    [2, 3]
    [2, 3, 4]
    [0, 1]
    [0, 1, 2, 3, 4]
    [0, 1, 2, 3]
    [0, 1, 8, 9, 4]
```

程 Loops

You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']
 for animal in animals:
print(animal)
```

```
    cat
    dog
```

```
monkey
```

If you want access to the index of each element within the body of a loop, use the built-in enumerate function:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
print('#{}: {}'.format(idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

稓 List comprehensions:

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
 print(squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
 print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
 print(even_squares)
```

```
[0, 4, 16]
```

稓 Dictionaries

A dictionary stores (key, value) pairs, similar to Java or an object in Javascript. You can a use it like this:

```
d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some
 data print(d['cat'])    # Get an entry from a dictionary; prints "cute"
print('cat' in d)  # Check if a dictionary has a given key; prints "True"
```

```
    cute
    True
```

```
 d['fish'] =          # Set an entry in a
 'wet'                dictionary
 print(d['fish'        # Prints "wet"
 ])
```

```
    wet
```

```
#print(d['monkey'])      # KeyError: 'monkey' not a key of d
```

```
print(d.get('monkey', 'N/A')) # Get an element with a default; prints "N/A"
 print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
    N/A
    wet
```

```
 del                  # Remove an element from a
 print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints
 "N/A"
```

```
    N/A
```

You can find all you need to know about dictionaries in the documentation.

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
 for animal, legs in d.items():
print('A {} has {} legs'.format(animal, legs))
```

```
    A person has 2 legs
    A cat has 4 legs
    A spider has 8 legs
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
 print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

程 Sets

A set is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}
print('cat' in animals)  # Check if an element is in a set; prints "True"
 print('fish' in animals) # prints "False"
```

```
True
False
```

```
animals.add('fish')     # Add an element to a
print('fish' in         set
animals)                 # Number of elements in a
print(len(animals))      set;
```

```
True
3
```

```
animals.add('cat')      # Adding an element that is already in the set does
print(len(animals)      nothing
)                       # Remove an element from a
animals.remove('ca      set
t')
print(len(animals)
)
```

```
3
2
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
print('#{}: {}'.format(idx + 1, animal))
```

```
#1: fish
#2: cat
#3: dog
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

稙 Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
d = {(x, x + 1): x for x in range(10)} # Create a dictionary with tuple keys
 t = (5, 6)  # Create a tuple
print(type(t))
 print(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

```
#t[0] = 1
```

稙 Functions

Python functions are defined using the keyword. For example:

```
def sign(x):
if x > 0:
         return 'positive'
     elif x < 0:
         return 'negative'
     else:
return 'zero'

 for x in [-1, 0, 1]:
     print(sign(x))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments, like this:

```
 def hello(name,
     loud=False): if loud:
         print('HELLO,
     {}'.format(name.upper())) else:
print('Hello, {}!'.format(name))

hello('Bob')
hello('Fred', loud=True)
```

```
     Hello, Bob!
     HELLO, FRED
```

程 Classes

The syntax for defining classes in Python is straightforward:

```
class Greeter:

# Constructor
def_init_(self, name):
self.name = name # Create an instance variable

# Instance method
     def greet(self, loud=False):
         if loud:
           print('HELLO, {}'.format(self.name.upper()))
         else:
print('Hello, {}!'.format(self.name))

g = Greeter('Fred') # Construct an instance of the Greeter class
g.greet()    # Call an instance method; prints "Hello, Fred"
 g.greet(loud=True)      # Call an instance method; prints "HELLO,
 FRED!"
```

```
     Hello, Fred!
     HELLO, FRED
```

程 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

To use Numpy, we first need to import the package:

```
import numpy as np
```

䅇 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
a = np.array([1, 2, 3]) # Create a rank 1 array
 print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5    # Change an element of the array print(a)
```

```
    <class 'numpy.ndarray'> (3,) 1 2 3
    [5 2 3]
```

```
b = np.array([[1,2,3],[4,5,6]])# Create a rank 2 array
 print(b)
```

```
    [[1 2 3]
     [4 5 6]]
```

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
    (2, 3)
    1 2 4
```

Numpy also provides many functions to create arrays:

```
a = np.zeros((2,2)) # Create an array of all zeros
 print(a)
```

```
    [[0. 0.]
     [0. 0.]]
```

```
b = np.ones((1,2)) # Create an array of all ones
 print(b)
```

```
    [[1. 1.]]
```

```
c = np.full((2,2), 7) # Create a constant array
 print(c)
```

```
    [[7 7]
     [7 7]]
```

```
d =                         # Create a 2x2 identity
np.eye(2)                   matrix
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.62557958 0.66335353]
 [0.0144399  0.18211925]]
```

蕕 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create the following rank 2 array with shape (3,
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2
rows
# and columns 1 and 2; b is the following array of shape (2,
2): # [[2 3]
#     [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

```
print(a[0, 1])
b[0, 0] = 77 # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
# Create the following rank 2 array with shape (3, 4)
 a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

```
row_r1 = a[1, :] # Rank 1 view of the second row of a
 row_r2 = a[1:2, :] # Rank 2 view of the second row of a
 row_r3 = a[[1], :] # Rank 2 view of the second row of a
 print(row_r1, row_r1.shape)
print(row_r2,  row_r2.shape)
 print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

```
# We can make the same distinction when accessing columns of an array:
 col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
 print()
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,)
 and print(a[[0, 1, 2], [0, 1, 0]])

# The above example of integer array indexing is equivalent to
 this: print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
[1 4 5]
```

```
# When using integer array indexing, you can reuse the same
 # element from the source array:
print(a[[0, 0], [1, 1]])

# Equivalent to the previous integer array indexing example
 print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
# Create a new array from which we will select elements
 a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
# Create an array of indices
 b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
 print(a[np.arange(4), b]) # Prints "[ 1    6     7 11]"
```

```
[ 1  6  7 11]
```

```
# Mutate one element from each row of a using the indices in b
 a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2) # Find the elements of a that are bigger than 2;
# this returns a numpy array of Booleans of the same # shape as a, where
                     each slot of bool_idx tells
# whether that element of a is > 2.

print(bool_idx)
```

```
[[False  False]
 [ True True] [
 True True]]
```

```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

䅇 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
x = np.array([1, 2]) # Let numpy choose the datatype
y = np.array([1.0, 2.0]) # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64) # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

程 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
 print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
# Elementwise difference; both produce the array
 print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
# Elementwise product; both produce the array
 print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division; both produce the array
 # [[ 0.2    0.33333333]
#     [ 0.42857143 0.5    ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
# Elementwise square root; produces the
 array # [[ 1.     1.41421356]
#     [ 1.73205081 2.     ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
 print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

You can also use the `@` operator which is equivalent to numpy's operator.

```
print(v @ w)
```

```
219
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
 print(x.dot(v))
print(np.dot(x, v))
 print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
```

```
  print(x.dot(y))
  print(np.dot(x,
  y)) print(x @ y)
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum` :

```
x = np.array([[1,2],[3,4]])

print(np.sum(x)) # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0)) # Compute sum of each column; prints "[4 6]"
 print(np.sum(x, axis=1)) # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
print(x)
print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

```
v = np.array([[1,2,3]])
 print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

䅇 Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
# We will add the vector v to each row of the matrix x,
 # storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)     # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
 for i in range(4):
y[i, :] = x[i, :] + v

print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix x is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv. We could implement this approach like this:

```
  vv = np.tile(v, (4, 1)) # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

```
y = x + vv # Add x and vv elementwise
 print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix
 x, # storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # Add v to each row of x using broadcasting
 print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)` , where each row was a copy of `v` , and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

```
# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
 w = np.array([4,5])      # w has shape
 (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
 # an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
# Add a vector to each row of a
 matrix x = np.array([[1,2,3],
 [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2,
 3), # giving the following matrix:
print(x + v)
```

```
    [[2 4 6]
     [5 7 9]]
```

```
# Add a vector to each column of a matrix
 # x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
 # yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
    [[ 5  6  7]
     [ 9 10 11]]
```

```
# Another solution is to reshape w to be a row vector of shape (2, 1);
 # we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
    [[ 5  6  7]
     [ 9 10 11]]
```

```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
 # these can be broadcast together to shape (2, 3), producing the
 # following array:
print(x * 2)
```

```
    [[ 2  4  6]
     [ 8 10 12]]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

萢 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the matplotlib.pyplot module, which provides a plotting system similar to that of MATLAB.

```
import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline:

```
%matplotlib inline
```

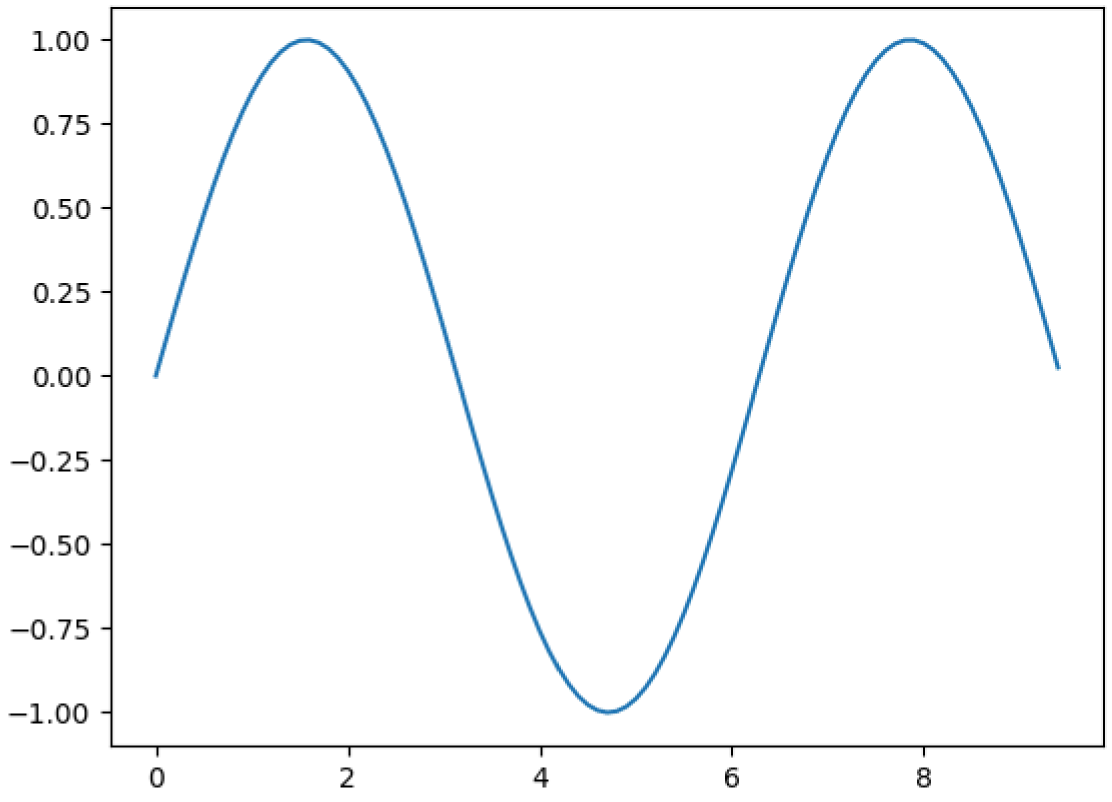
Plotting

The most important function in simple matplotlib is plot, which allows you to plot 2D data. Here is a example:

```
# Compute the x and y coordinates for points on a sine
 curve x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using
 matplotlib plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7c7044fe5120>]
```

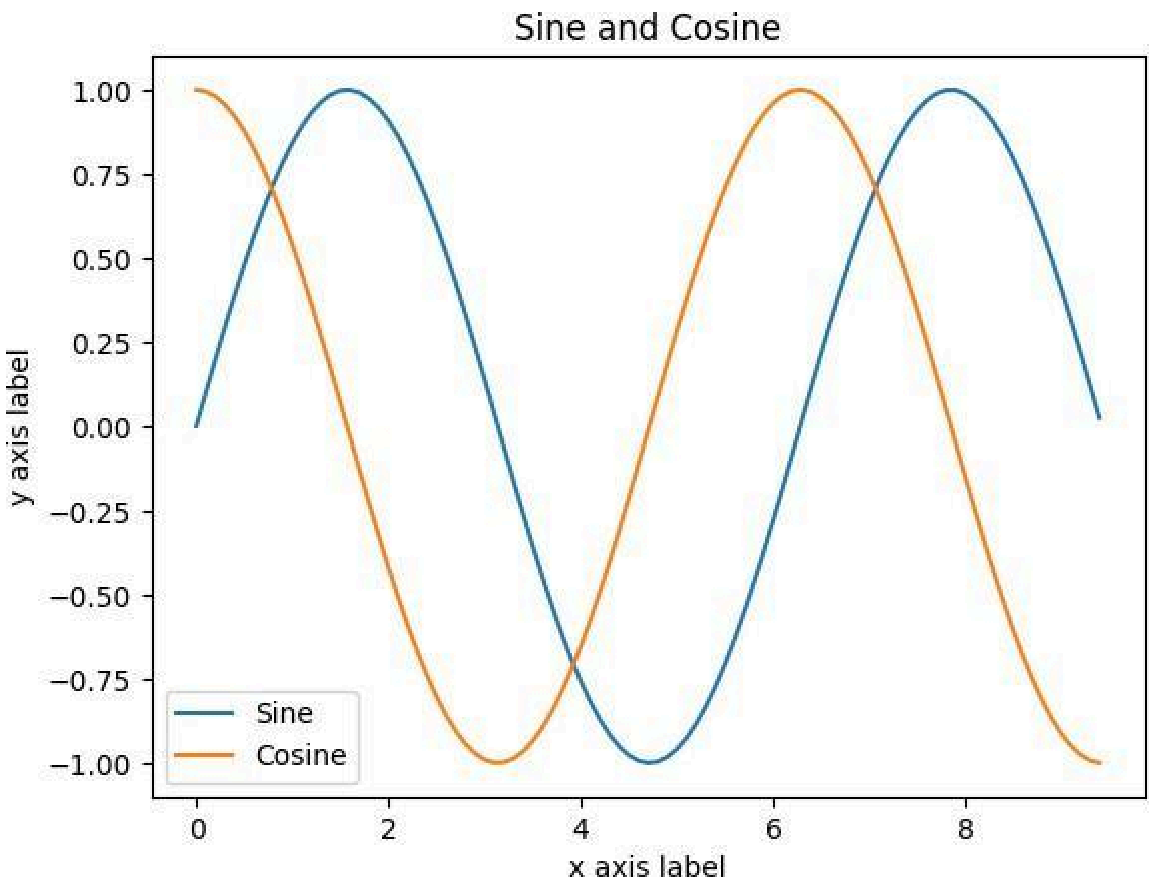


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
y_sin = np.sin(x)
 y_cos = np.cos(x)

# Plot the points using matplotlib
 plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
 plt.ylabel('y axis label')
 plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

```
<matplotlib.legend.Legend at 0x7c704437b970>
```



穃 Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

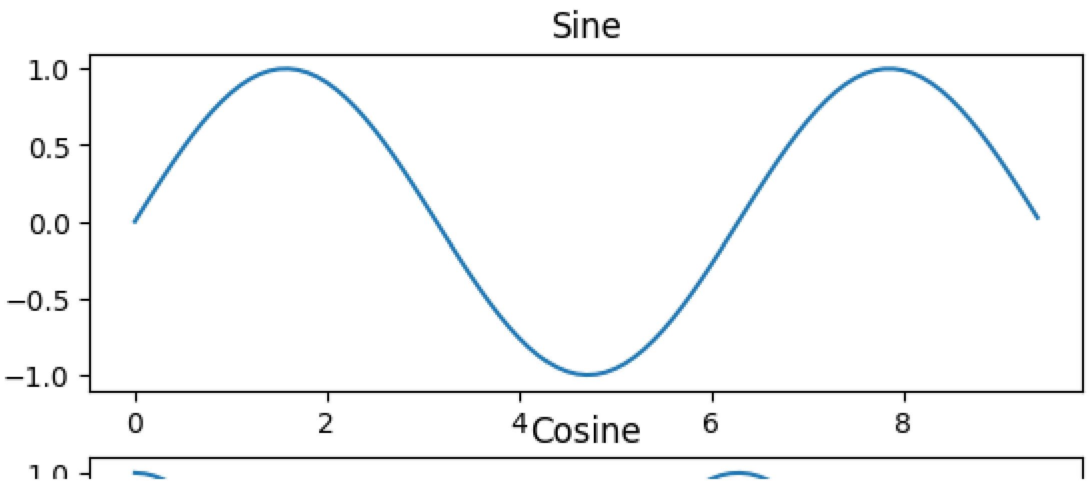
```
# Compute the x and y coordinates for points on sine and cosine
curves x = np.arange(0, 3 * np.pi, 0.1)
y_sin =
np.sin(x)
y_cos =
np.cos(x)

# Set up a subplot grid that has height 2 and width
1, # and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first
plot plt.plot(x,
y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second
plot. plt.subplot(2, 1, 2)
plt.plot(x,
y_cos)
plt.title('Cosin
e')

# Show the
figure.
plt.show()
```



Lab Assignment 2

Turtlesim simulation in ROS:

Steps:

1.Start the roscore:

When you run "roscore," it starts the ROS Master, which acts as a centralized coordination point for ROS nodes. The ROS Master provides services such as parameter server, naming, and registration, allowing ROS nodes to find and communicate with each other.
Command: roscore

2.To install and start the turtlesim:

This command is used to install the ROS package named "turtlesim" for the specific ROS distribution that you are currently using. Let me break down the command:
Command: sudo apt-get install ros-$(rosversion -d) -turtlesim

3.Run turtlesim:

The command rosrun turtlesim turtlesim_node is used to run the "turtlesim_node" executable within the ROS environment.

Here's a breakdown of the command:
rosrun: This command is used to run a specific ROS package and its executables.
turtlesim: This is the name of the ROS package.
turtlesim_node: This is the name of the executable within the "turtlesim" package. The "turtlesim_node" is responsible for launching the turtlesim simulator, a simple turtle graphics simulator in ROS.
Command: rosrun turtlesim turtlesim_node

4.move turtle using the keyboard:

This command used to move turtle using keyboard keys.
Command: rosrun turtlesim turtle_teleop_key

Click here for the video.

Lab Assignment 3

Python in ROS Simulation:

Steps:

1. Create Publisher and Subscriber Files:

Python Publisher Code(publisher_node.py):

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

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

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

Python Subscriber Code(subscriber_node.py):

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

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    def listener():
```

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

2. Make the scripts executable:

chmod +x publisher_node.py
chmod +x subscriber_node.py

Command explanation:
The command chmod +x publisher_node.py is saying: "Change the file permissions of 'publisher_node.py' to make it executable.

3. Build Your ROS Package(if not done already):

cd ~/catkin_ws catkin_make

Command explanation:
The catkin_make command is used to build (compile) the source code of a ROS workspace.

4. Run Publisher Node In One Terminal:

rosrun my_package publisher_node.py

Command explanation:
The command rosrun my_package publisher_node.py is used to run a ROS node from a specific ROS package.

5. Run Subscriber Node In Another Terminal:

rosrun my_package subscriber_node.py

Click here to get a video.

Micro Lab-4: Robotics Math in ROS Simulation

Position Publisher File:

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import Int32

def position_publisher():

    rospy.init_node('pos_pub_node')
    pub = rospy.Publisher('position_out', Int32, queue_size=10)
    rate = rospy.Rate(0.5)

    while not rospy.is_shutdown():
        pos_msg = 15      # distance in meter
        pub.publish(pos_msg)
        print(pos_msg)
        rate.sleep()

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

Velocity Subscriber File:

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import Int32

def pos_callback(data):
    pos = data.data
    omega = 20
    velocity = omega * pos
    print(velocity)


def velocity_subscriber():

    rospy.init_node('velocity_sub_node', anonymous=True)
    rospy.Subscriber("position_out", Int32, pos_callback)
    rospy.spin()

if __name__ == '__main__':
    velocity_subscriber()
```

Here, Publisher Python File Takes position as input and Subscriber Python File calculates velocity for a given position. When we run publisher file it will print position of object and Subscriber File will print velocity of the same.

Micro Lab-5: Publisher and Subscriber Implementation in Python with Launch File

Publisher Node:

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def my_subscribe():

    # run simultaneously.
    rospy.init_node('hello_world_sub_node: ')
    rospy.Subscriber("hello_world", String, callback)

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

if __name__ == '__main__':
    my_subscribe()
```

Subscriber Node:

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def hello_world_pub():
    rospy.init_node('hello_world_pub_node')
    pub = rospy.Publisher('hello_world', String, queue_size=10)
    rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

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

Launch File:

```
<launch>
        <node pkg="robot_math_2" type="publisher.py" name="pub_node"/>
        <node pkg="robot_math_2" type="subscriber.py" name="sub_node"/>
</launch>
```

Here, When we run launch File, it will start two nodes, Publisher Node and Subscriber Node.
The Publisher will start publishing to the Topic at the rate of 10 messages per second. The Subscriber will start listening to Topic.

Micro Lab-6: Creating URDF and Xacro Files for a Rover

Rviz File:

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

<launch>


    <param name="robot_description" textfile="$(find
robot_ie416)/urdf/rover.urdf"/>

    <arg name="rviz_file" default="$(find robot_ie416)/config/robot.rviz"/>


    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher"/>

    <node pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"
name="joint_state_publisher_gui"/>

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(arg rviz_file)"/>


</launch>
```

Rviz_xacro launch:

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

<launch>


    <param name="robot_description" command="xacro '$(find
robot_ie416)/urdf/rover.urdf.xacro'"/>
```

```
    <arg name="rviz_file" default="$(find robot_ie416)/config/robot.rviz"/>



    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher"/>

    <node pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"
name="joint_state_publisher_gui"/>

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(arg rviz_file)"/>



</launch>
```

Here after successfully running the ROS environment and executing the roslaunch command, we were able to see the rover model in rviz file.

Micro Lab-7: Analyzing Rover Forward Kinematics (FK) with RViz

Rviz launch:

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

<launch>



    <param name="robot_description" textfile="$(find
robot_ie416)/urdf/rover.urdf"/>

    <arg name="rviz_file" default="$(find robot_ie416)/config/robot.rviz"/>



    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher"/>

    <node pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"
name="joint_state_publisher_gui"/>

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(arg rviz_file)"/>



</launch>
```

Rviz_xacro.launch

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

<launch>



    <param name="robot_description" command="xacro '$(find
robot_ie416)/urdf/rover.urdf.xacro'"/>

    <arg name="rviz_file" default="$(find robot_ie416)/config/robot.rviz"/>
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher"/>

    <node pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"
name="joint_state_publisher_gui"/>

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(arg rviz_file)"/>



</launch>
```

Here after successfully running the ROS environment and executing the roslaunch command, we were able to see the rover model in rviz file and were also be able to observe the forward kinematics i.e. the position, velocity and forces/torques etc. using the command rostopic echo /joint_states.

Micro Lab-8: URDF File Creation for a Robot Arm

Launch File for Robotic arm (Gazebo):

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

    <param name="robot_description" textfile="$(find
robotic_arm)/urdf/gazebo_robotic_arm.urdf"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model robotic_arm"    />

    <node name="RSP" pkg="robot_state_publisher"
type="robot_state_publisher" output="screen"/>

    <node name="JSP" pkg="joint_state_publisher"
type="joint_state_publisher" output="screen"

</launch>
```

URDF File for Robotic Arm (Gazebo):

```
<?xml version="1.0"?>
<robot name="robotic_arm">

  <link name="world"/>

    <link name="base_link">
        <visual>
            <geometry>
                <cylinder length="0.05" radius="0.1"/>
            </geometry>
            <material name="silver">
                <color rgba="0.75 0.75 0.75 1"/>
            </material>
            <origin rpy="0 0 0" xyz="0 0 0.025"/>
        </visual>

    <collision>
        <geometry>
            <cylinder length="0.05" radius="0.1"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.025"/>
    </collision>

    <inertial>
        <origin rpy="0 0 0" xyz="0 0 0.025"/>
        <mass value="5.0"/>
        <inertia ixx="0.0135" ixy="0.0" ixz="0.0" iyy="0.0135" iyz="0.0"
izz="0.05"/>
    </inertial>


    </link>

  <joint name="fixed" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
  </joint>

  <link name="link_1">
    <visual>
        <geometry>
            <cylinder length="0.5" radius="0.05"/>
        </geometry>
        <material name="silver">
            <color rgba="0.75 0.75 0.75 1"/>
        </material>
        <origin rpy="0 0 0" xyz="0 0 0.25"/>
    </visual>

    <collision>
        <geometry>
            <cylinder length="0.5" radius="0.05"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.25"/>
    </collision>

    <inertial>
        <origin rpy="0 0 0" xyz="0 0 0.25"/>
        <mass value="5.0"/>
        <inertia ixx="0.107" ixy="0.0" ixz="0.0" iyy="0.107" iyz="0.0"
izz="0.0125"/>
    </inertial>

  </link>

  <joint name="joint_1" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="link_1"/>
    <origin rpy="0 0 0" xyz="0.0 0.0 0.05"/>
  </joint>

<link name="link_2">
    <inertial>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
        <mass value="2.0"/>
        <inertia ixx="0.027" ixy="0.0" ixz="0.0" iyy="0.027" iyz="0.0"
izz="0.0025"/>
    </inertial>

    <visual>
        <geometry>
        <cylinder length="0.4" radius="0.05"/>
        </geometry>
        <material name="silver"/>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
    </visual>

    <collision>
        <geometry>
                <cylinder length="0.4" radius="0.05"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
    </collision>

</link>

<joint name="joint_2" type="continuous">
<axis xyz="0 1 0"/>
<parent link="link_1"/>
<child link="link_2"/>
<origin rpy="0 1.5708 0" xyz="0.0 -0.1 0.45"/>
</joint>

<link name="link_3">
    <inertial>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
        <mass value="2.0"/>
        <inertia ixx="0.027" ixy="0.0" ixz="0.0" iyy="0.027" iyz="0.0"
izz="0.0025"/>
    </inertial>

    <visual>
        <geometry>
        <cylinder length="0.4" radius="0.05"/>
        </geometry>
        <material name="silver"/>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
        </visual>

    <collision>
        <geometry>
            <cylinder length="0.4" radius="0.05"/>
        </geometry>
        <origin rpy="0 0 0" xyz="0 0 0.2"/>
    </collision>
</link>

<joint name="joint_3" type="continuous">
<axis xyz="0 0 1"/>
```

```
<parent link="link_2"/>
<child link="link_3"/>
<origin rpy="0 0 0" xyz="0.0 0.1 0.35"/>
</joint>


<link name="link_4">
    <inertial>
        <mass value="1.0"/>
        <inertia ixx="0.00042" iyy="0.00042" izz="0.00042" ixy="0"
ixz="0" iyz="0"/>
    </inertial>

    <visual>
        <geometry>
            <box size="0.05 0.05 0.05"/>
        </geometry>
        <material name="silver"/>
    </visual>

    <collision>
        <geometry>
            <box size="0.05 0.05 0.05"/>
        </geometry>
    </collision>
</link>

<joint name="joint_4" type="continuous">
<axis xyz="0 1 0"/>
<parent link="link_3"/>
<child link="link_4"/>
<origin rpy="0 0 0" xyz="0.0 0.0 0.425"/>
</joint>



<gazebo reference="base_link">
    <material>Gazebo/Black</material>
</gazebo>
```

```
<gazebo reference="link_1">
    <material>Gazebo/Grey</material>
</gazebo>
<gazebo reference="link_2">
    <material>Gazebo/Orange</material>
</gazebo>
<gazebo reference="link_3">
    <material>Gazebo/Black</material>
</gazebo>
<gazebo reference="link_4">
    <material>Gazebo/Yellow</material>
</gazebo>

</robot>
```

This URDF File contains physical shapes and parameters of Robotic Arm. We will run the Gazebo_Robotic_arm.launch file in the Robotic_arm package to see Gazebo simulation.

Micro Lab-9: Adding Controller Plugin for Smooth Control of Robot Arm

Now we will add 3 type of controller in Robotic Arm:

1)Robot Effort Controller
2)Robot Position Controller
3)Robot Trajectory Controller

So, the purpose of using a controller launch file is to separate the setup and launch of controllers from the general setup of the robotic system. Here are controller Node Launch File Codes.

Robot Effort Controller Launch File:

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

    <param name="robot_description" textfile="$(find
robotic_arm)/urdf/robot_effort_controller.urdf"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model robotic_arm"/>

    <node name="RSP" pkg="robot_state_publisher"
type="robot_state_publisher" output="screen"/>

    <rosparam  file="$(find robotic_arm)/config/effort_controller.yaml"/>

    <node name="Controller_Spawner" pkg="controller_manager"
type="spawner" args="joint_1_controller joint_2_controller
joint_3_controller joint_4_controller" />

</launch>
```

Robot Position Controller launch File:
```
<?xml version="1.0"?>
<launch>

    <param name="robot_description" textfile="$(find
robotic_arm)/urdf/robot_pos_controller.urdf"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model robotic_arm"/>

    <node name="RSP" pkg="robot_state_publisher"
type="robot_state_publisher" output="screen"/>

    <rosparam  file="$(find
robotic_arm)/config/position_controller.yaml"/>

    <node name="Controller_Spawner" pkg="controller_manager"
type="spawner" args="joint_1_controller joint_2_controller
joint_3_controller joint_4_controller" />

</launch>
```

Robot Trajectory Controller Launch File:
```
<?xml version="1.0"?>
<launch>

    <param name="robot_description" textfile="$(find
robotic_arm)/urdf/robot_traj_controller.urdf"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch" />

    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model robotic_arm"    />

    <node name="RSP" pkg="robot_state_publisher"
type="robot_state_publisher" output="screen"/>
```

```
    <rosparam  file="$(find
robotic_arm)/config/joint_trajectory_controller.yaml"/>

    <node name="Controller_Spawner" pkg="controller_manager"
type="spawner" args="arm_controller" />

</launch>
```

Micro Lab-10: Modeling the Kuka-Kr5 Robot

manipulator_rviz_JSP_gui.launch

```
<!-- -*- mode: XML -*- -->

<launch>


  <param name="robot_description" command="cat $(find
alexa_ros)/urdf/manipulator.urdf" />

  <node name="sim" pkg="rviz" type="rviz" output="screen" args="-d $(find
alexa_ros)/config/config.rviz" />


  <node name="awaien" pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui" output="screen"/>
```

```
  <node name="robot_states" pkg="robot_state_publisher"
type="robot_state_publisher" output="screen"/>


</launch>
```

alexa_ros.launch

```
<!-- -*- mode: XML -*- -->

<launch>


  <!-- these are the arguments you can pass this launch file, for example
paused:=true -->

  <arg name="paused" default="false"/>

  <arg name="use_sim_time" default="true"/>

  <arg name="gui" default="true"/>

  <arg name="headless" default="false"/>

  <arg name="debug" default="false"/>


  <!-- We resume the logic in empty_world.launch -->

  <include file="$(find gazebo_ros)/launch/empty_world.launch">

    <arg name="debug" value="$(arg debug)" />

    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>

    <arg name="use_sim_time" value="$(arg use_sim_time)"/>

    <arg name="headless" value="$(arg headless)"/>

  </include>



  <!-- Load the URDF into the ROS Parameter Server -->



  <param name="robot_description" command="cat $(find
alexa_ros)/urdf/manipulator.urdf" />



  <!-- Run a python script to the send a service call to gazebo_ros to spawn a
URDF robot -->

  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen"   args="-urdf -model manipulator -param robot_description -z
0.0"/>



  <!-- ros_control KUKA launch file -->

  <include file="$(find alexa_ros)/launch/manipulator_control.launch" />



    <!-- alexa ROS interface launch file -->

  <include file="$(find alexa_ros)/launch/alexa_interface.launch" />



</launch>



</launch>
```

Micro Lab-11: Integrating Alexa Developer Console with Flask Micro-web Framework in Python

Launch server and remote interface using “roslaunch alexa_ros alexa_interface.launch” command. Then run “ngrok http –domain=closely-helping-cricket.ngrok-free.app 5000” command.

ngrok will create a secure tunnel to your local web server running on port 5000 and assign it the custom domain closely-helping-cricket.ngrok-free.app, allowing external users to access your local web server via this domain.

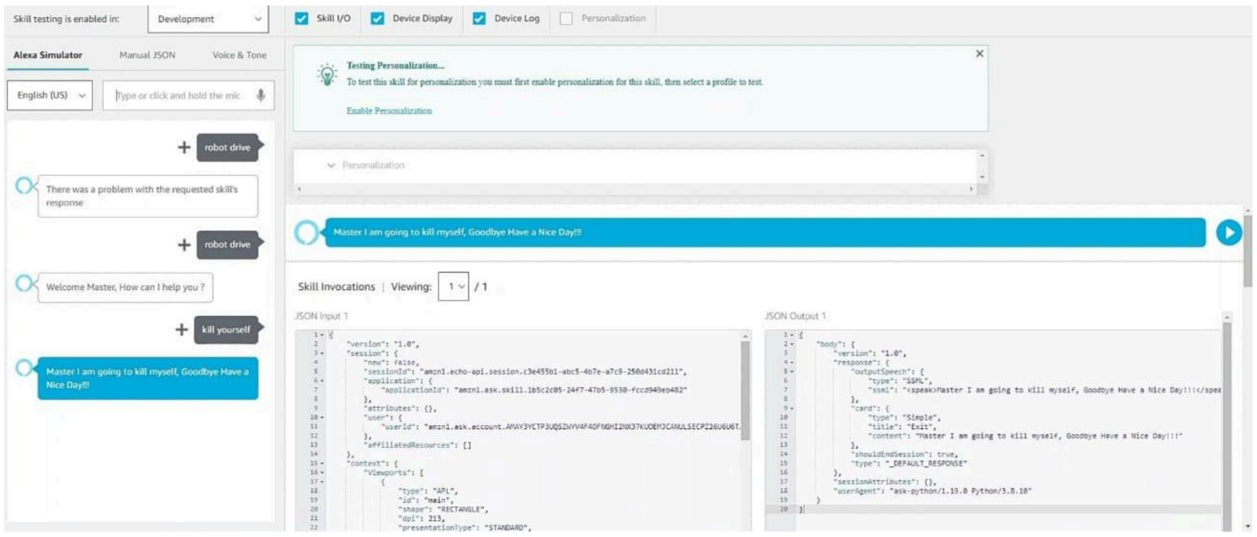
Alexa_interface.launch File:

```
<launch>

    <!-- Launch the task action server -->
    <node pkg="alexa_ros" type="alexa_task.py" name="alexa_task"
respawn="true" output="screen"/>

    <!-- Launch the remote interface node that handles remote requests
-->
    <node pkg="alexa_ros" type="alexa_ros_interface.py"
name="alexa_ros_interface" respawn="true" output="screen"/>

</launch>
```



Testing Personalization...
To test this skill for personalization you must first enable personalization for this skill, then select a profile to test.
Enable Personalization

robot drive

There was a problem with the requested skill's response

robot drive

Welcome Master, How can I help you ?

kill yourself

Master I am going to kill myself, Goodbye Have a Nice Day!!!

Master I am going to kill myself, Goodbye Have a Nice Day!!!

Skill Invocations | Viewing: 1 / 1

JSON Input 1

```
{
    "version": "1.0",
    "session": {
        "new": false,
        "sessionId": "amzn1.echo-api.session.c3e455b1-abc5-4b7e-a7c9-250d431cd211",
        "application": {
            "applicationId": "amzn1.ask.skill.1b5c2c05-24f7-47b5-9530-fccd949eb482"
        },
        "attributes": {},
        "user": {
            "userId": "amzn1.ask.account.AMAY3YCTP3UQSZWYV4F4OFNGHI2NX37KUOEM3CANULSECPZ26U6U6T.
        },
        "affiliatedResources": []
    },
    "context": {
        "Viewports": [
            {
                "type": "APL",
                "id": "main",
                "shape": "RECTANGLE",
                "dpi": 213,
                "presentationType": "STANDARD",
```

JSON Output 1

```
{
    "body": {
        "version": "1.0",
        "response": {
            "outputSpeech": {
                "type": "SSML",
                "ssml": "<speak>Master I am going to kill myself, Goodbye Have a Nice Day!!!</spea
            },
            "card": {
                "type": "Simple",
                "title": "Exit",
                "content": "Master I am going to kill myself, Goodbye Have a Nice Day!!!"
            },
            "shouldEndSession": true,
            "type": "_DEFAULT_RESPONSE"
        },
        "sessionAttributes": {},
        "userAgent": "ask-python/1.19.0 Python/3.8.10"
    }
}
```

We can give commands to Alexa using this developer console.

Micro Lab-12: Developing an Alexa Voice Assistant to Control the Kuka-Kr5 Robot via Amazon Web Services

Launch alexa_ros.launch file in Gazebo and then connect to developer console using “ngrok http –domain=closely-helping-cricket.ngrok-free.app 5000” command.

Give commands to Alexa like Sleep, Wakeup, Pick the object etc. And Alexa will control the robotic arm.

Alexa_ros.launch FIle:

```
<!-- -*- mode: XML -*- -->
<launch>

  <!-- these are the arguments you can pass this launch file, for example
paused:=true -->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->

  <param name="robot_description" command="cat $(find
alexa_ros)/urdf/manipulator.urdf" />

  <!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
```

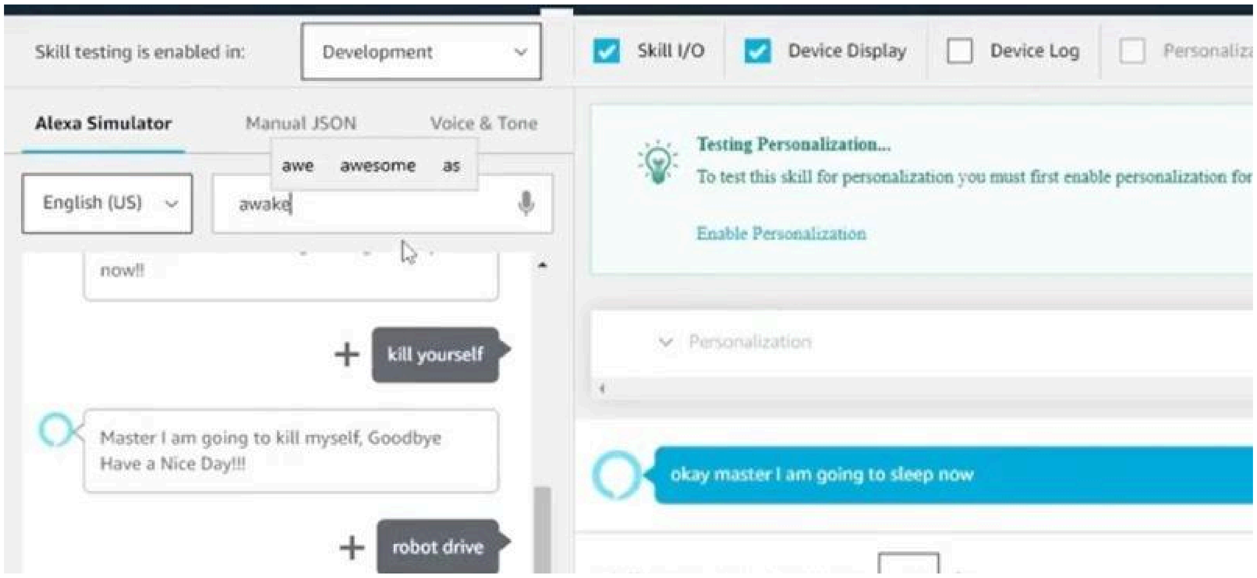
```
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"   args="-urdf -model manipulator -param
robot_description -z 0.0"/>

  <!-- ros_control KUKA launch file -->
  <include file="$(find alexa_ros)/launch/manipulator_control.launch" />

    <!-- alexa ROS interface launch file -->
  <include file="$(find alexa_ros)/launch/alexa_interface.launch" />

</launch>
```

Say to Awaken the robot to Alexa:



Skill testing is enabled in: Development

Skill I/O ✓ Device Display ✓ Device Log Personaliz

awake

now!!

kill yourself

Master I am going to kill myself, Goodbye Have a Nice Day!!!

robot drive

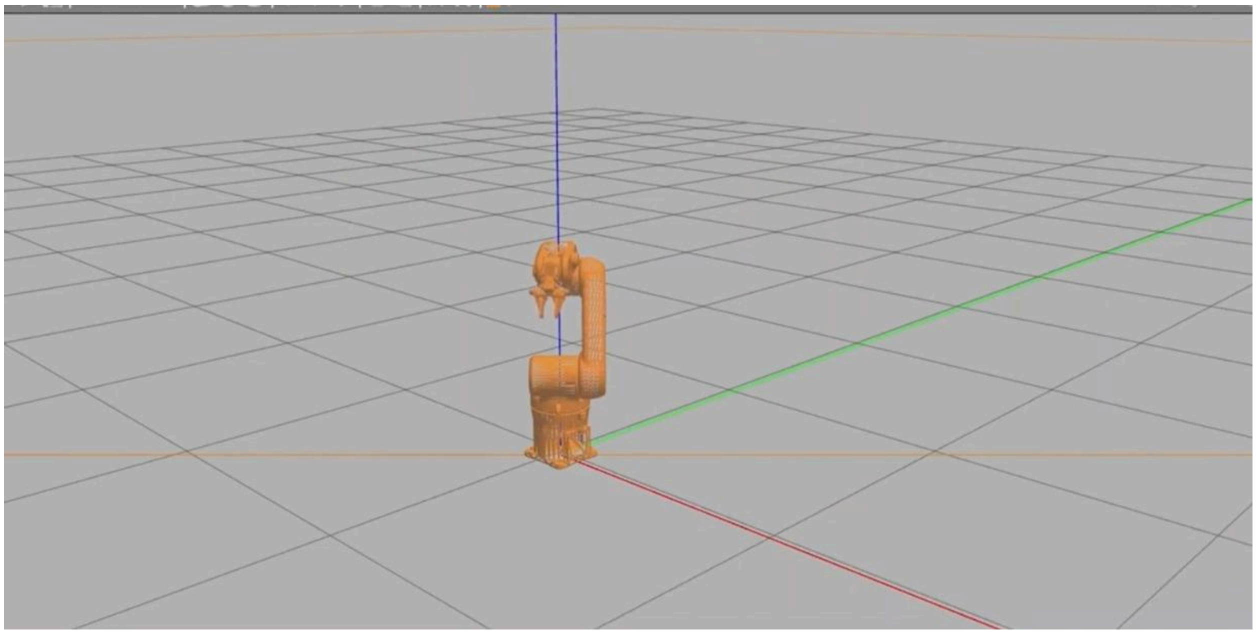
Testing Personalization...

To test this skill for personalization you must first enable personalization for

Enable Personalization

okay master I am going to sleep now

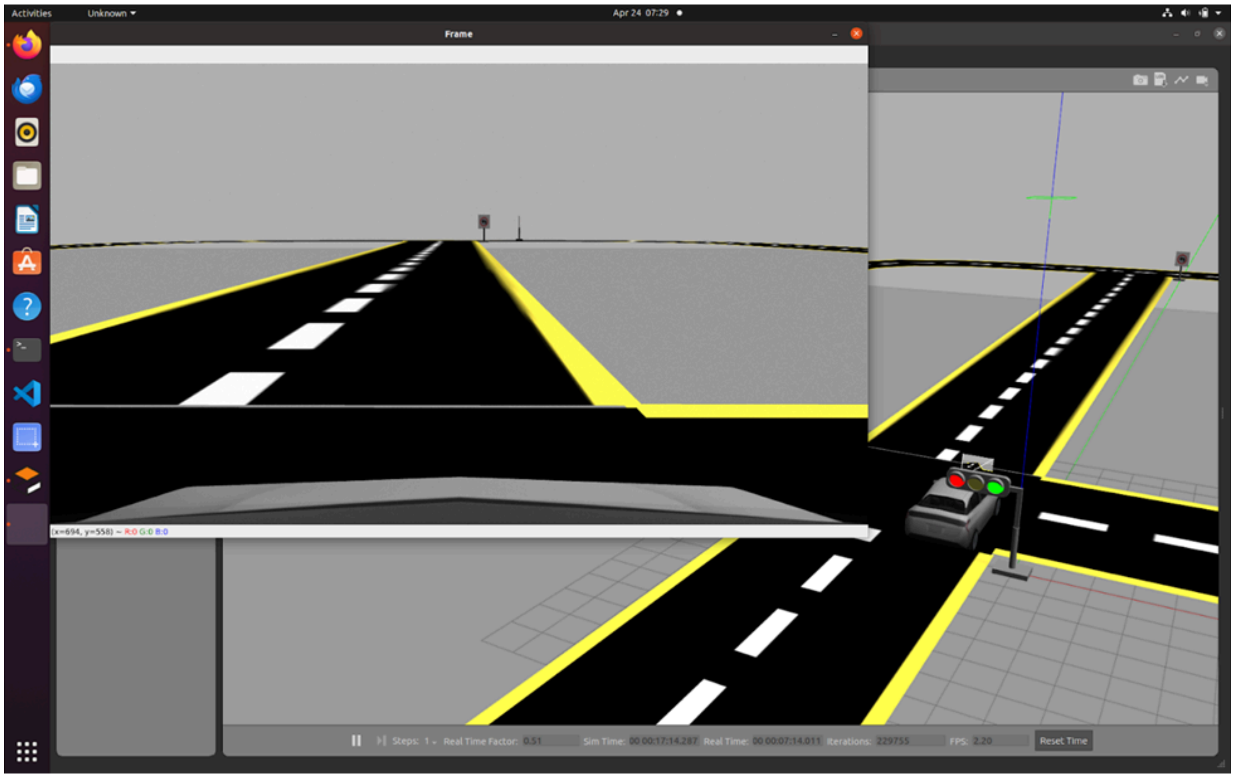
Result:



Micro Lab-13: Building a Simulation Environment for Self-Driving Cars (SDC) using URDF, Xacro, and Computer Vision (CV)

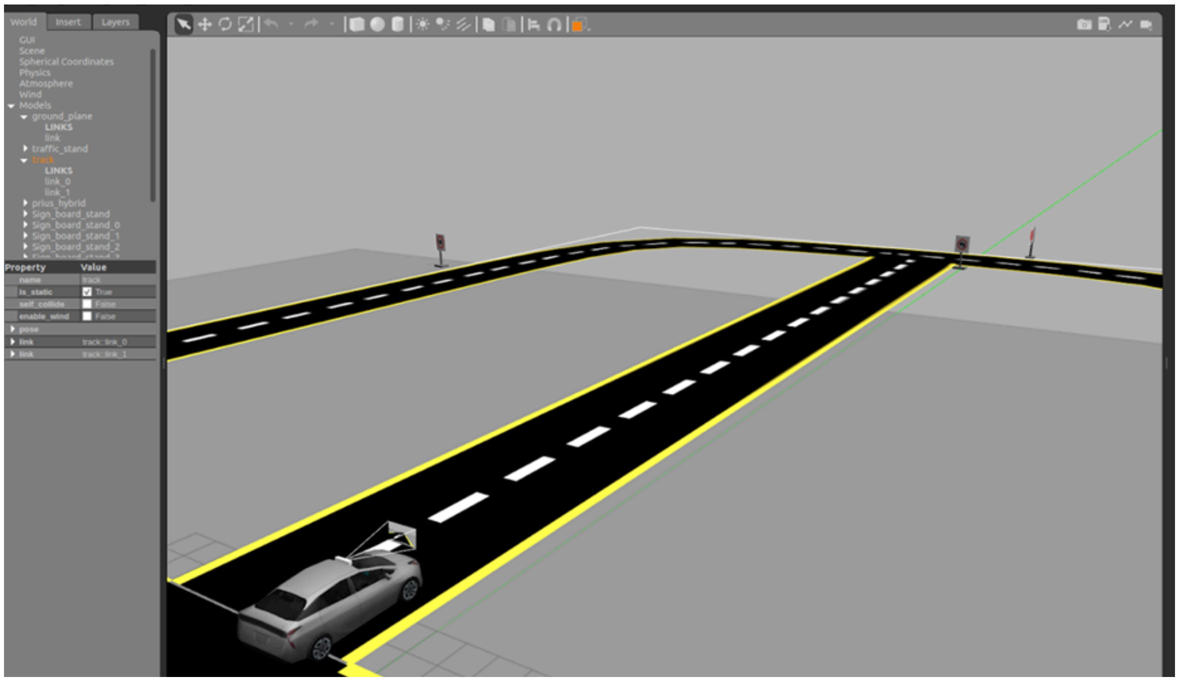
Steps:

1. Creating and activating virtual environment
2. Building the repo
3. Sourcing the setup.bash files
4. Running the simulation
5. Running the sdc controller



Micro Lab-14: Implementing Lane Detection for Simulated SDC using CV Techniques

1. Creating and activating virtual environment
2. Building the repo
3. Sourcing the setup.bash files
4. Running the simulation
5. Running the lane detection controller

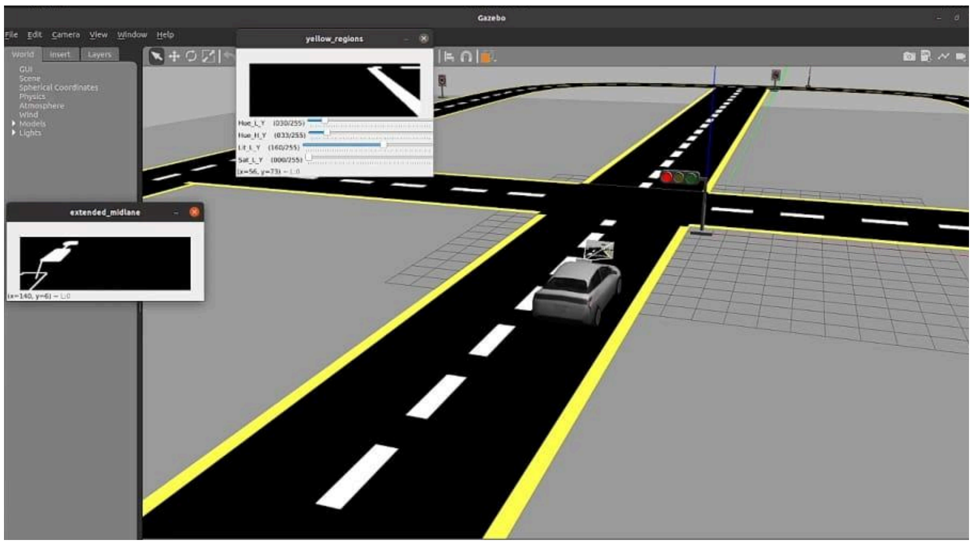


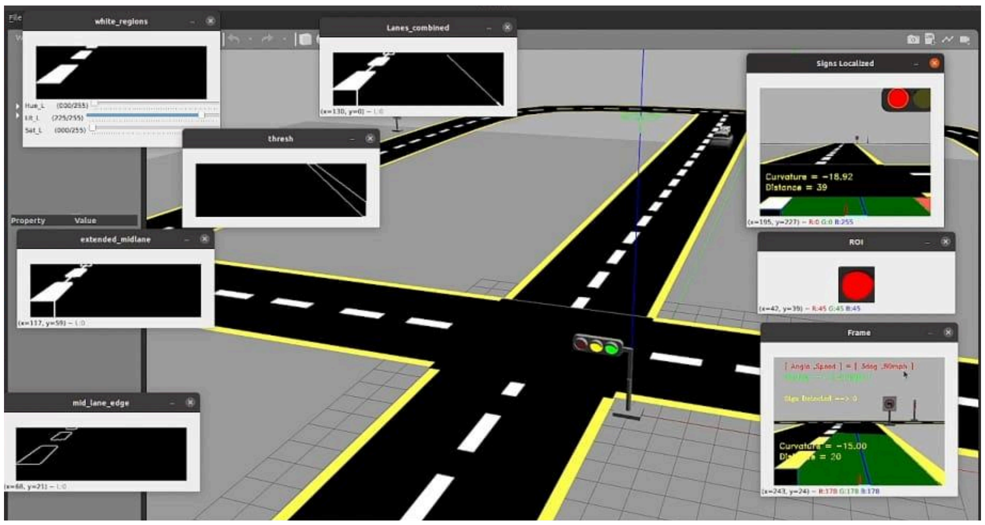
Property	Value
name	track
is_static	✓ True
self_collide	False
enable_wind	False
pose	
link	track::link_0
link	track::link_1

Micro Lab-15: Creating a CNN-based Sign Classification System for SDC

Steps:

1. Activate virtual environment
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature2_Cruise_Control/
    c. Pip3 install -r note/requirenments.txt
2. Build SDC_Car
    a. cp -r self_driving_car_pkg/models/* ~/.gazebo/models/
    b. colcon build
    c. nano sdc_car/COLCONE_IGNORE
3. Run the project
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature2_Cruise_Control/
    c. source install/setup.bash
    d. source /opt/ros/foxy/setup.bash
    e. Ros2 launch self_driving_car_pkg world_gazebo.launch.py
4. To move the car
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature2_Cruise_Control/
    c. source install/setup.bash
    d. Source /opt/ros/foxy/setup.bash
    e. Ros2 run self_driving_car_pkg computer_vision_node



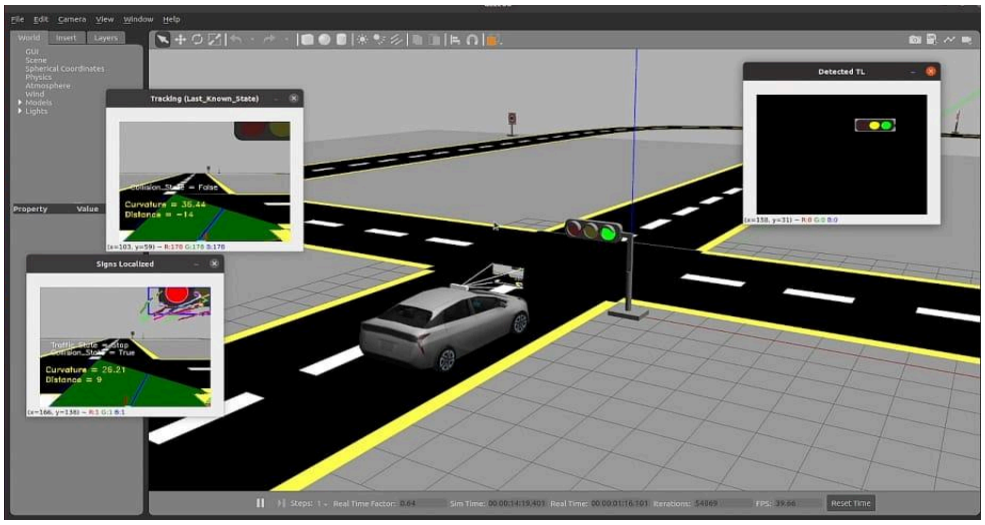


We’ve run the World_Gazebo.launch.py file. To move the car we will launch Computer Vision Node.

Micro Lab-16: Detecting Traffic Lights Using Haar Cascades for Road Intersection Crossings

Steps:

1. Activate virtual environment
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature2_Cruise_Control/
    c. Pip3 install -r note/requirenments.txt
2. Build SDC_Car
    a. cp -r self_driving_car_pkg/models/* ~/.gazebo/models/
    b. colcon build
    c. nano sdc_car/COLCONE_IGNORE
3. Run the project
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature4_Crossing_Intersection/
    c. source install/setup.bash
    d. source /opt/ros/foxy/setup.bash
    e. Ros2 launch self_driving_car_pkg world_gazebo.launch.py
4. To move the car
    a. source sdc_car/bin/activate
    b. cd SDC_Car/Feature2_Cruise_Control/
    c. source install/setup.bash
    d. Source /opt/ros/foxy/setup.bash
    e. Ros2 run self_driving_car_pkg computer_vision_node



CONCLUSION

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

REFERENCES

1. Lorem ipsum dolor sit amet
2. Consectetuer adipiscing elit
3. Sed diam nonummy nibh euismod