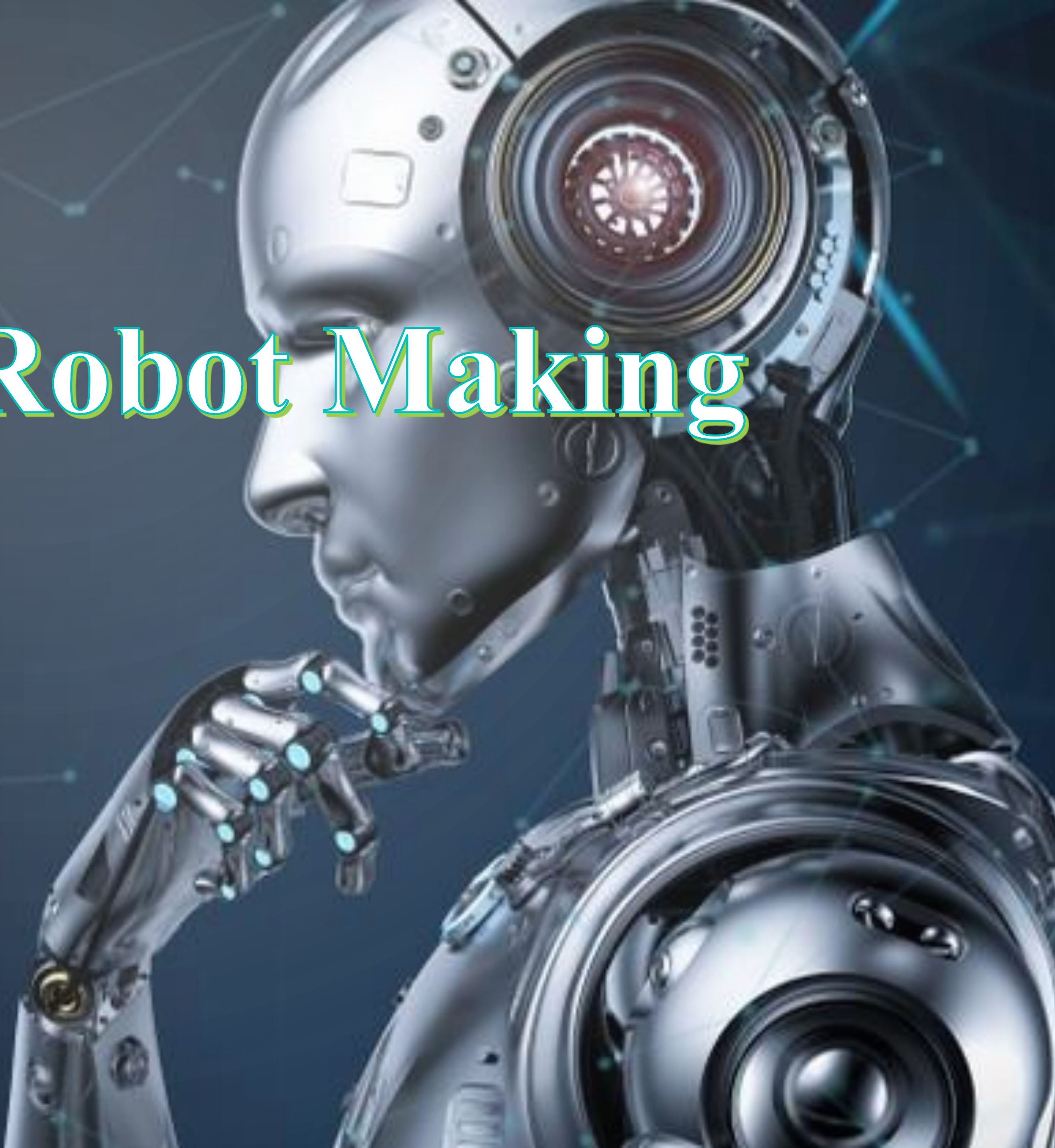
Humanoid Robot Making (Level-2)



HUMANOID ROBOT MAKING (LEVEL-2)

Course Code:	--	Credits:	--
		CIE Marks:	30
Exam Hours:	03	SEE Marks:	20

Course Learning Outcome (CLOs): After Completing this course successfully, the student will be able to…

CLO	Learning Outcome
CLO 1	Understand the structure and functionality of humanoid robots.
CLO 2	Program and control humanoid robot movements effectively.
CLO 3	Integrate advanced sensors for stability and feedback in humanoids.
CLO 4	Assemble and test humanoid robots, applying engineering principles.
CLO 5	Implement IoT-based solutions for humanoid control and monitoring.
CLO 6	Design and refine humanoid robots for real-world applications.

SUMMARY OF COURSE CONTENT

Serial No.	SUMMARY OF COURSE CONTENT	Hours	CLOs
1	Structure and functionality of humanoid robots	4	CLO 1
2	Programming and controlling humanoid robot movements effectively	6	CLO 2
3	Integrating advanced sensors for stability and feedback in humanoids	5	CLO 3
4	Assembling and testing humanoid robots using engineering principles	7	CLO 4
5	Implementing IoT-based solutions for humanoid control and monitoring	6	CLO 5
6	Designing and refining humanoid robots for real-world applications	8	CLO 6

Textbooks:

- **"Humanoid Robotics: A Reference" by Prahlad Vadakkepat**
- **"Introduction to Robotics: Mechanics and Control" by John J. Craig**

Additional References:

- **"Arduino Robotics" by John-David Warren**
- **"Robot Dynamics and Control" by Mark W. Spong**

ASSESSMENT PATTERN

CIE- Continuous Internal Evaluation (30 Marks)

Bloom's Category Marks (out of 90)	Lab Participation (10)	Assignments (10)	Quizzes (10)
Remember			05
Understand	05		
Apply		05	
Analyze	05		
Evaluate		05	05
Create			

SEE- Semester End Examination (20 Marks)

Bloom's Category	Test
Remember	
Understand	
Apply	10
Analyze	
Evaluate	
Create	10

COURSE PLAN

Week	Topics	Teaching-Learning Strategy(s)	Class Hour	Practice Hour	Assessment Strategy(s)	Mapping with CLO
01	Anatomy of Humanoid Robots: Understanding humanoid robot structure and functions.	Lecture, Demonstration, Interactive Q&A	5h	3h	Participation, Short Quiz	CLO 1
02-03	Motion Control in Humanoids: Skills in programming basic humanoid movements.	Lecture, Hands-on Exercises, Programming Practice	10h	6h	Practical Exercises, Code Review	CLO 2
04-05	Advanced Sensors for Humanoids: Proficiency in using advanced sensors for humanoids.	Demonstration, Practical Sessions, Troubleshooting	10h	6h	Hands-on Assignments, Sensor Integration Evaluation	CLO 3
06-07	Humanoid Programming Basics: Ability to program humanoid robots for interactive tasks.	Coding Tutorials, Debugging Sessions, Hands-on Practice	10h	6h	Coding Tests, Lab Performance	CLO 2
08-10	Assembly of a Humanoid Prototype: Hands-on experience in humanoid robot construction and testing.	Guided Assembly, Team Collaboration	15h	10h	Prototype Evaluation, Practical Assessment	CLO 4
11-12	IoT Control for Humanoids: Skills in connecting humanoids to IoT platforms.	IoT Integration Workshop, Cloud Platform Tutorials	10h	6h	Cloud Control Assignment, Test on IoT Integration	CLO 5
13-14	Large Humanoid Project Phase 1: Proficiency in project planning and initial design.	Brainstorming, Planning Sessions, Prototype Development	10h	6h	Progress Reports, Peer Review	CLO 6
15-16	Large Humanoid Project Phase 2: Practical experience in full-scale humanoid development.	Hands-on Development, Testing, Refinement	10h	10h	Final Testing, Functionality Demonstration	CLO 6
17	Final Assessment: Comprehensive evaluation of humanoid robot-making skills.	Presentation, Practical Demonstration	5h	1h	Final Exam, Project Presentation	CLO 1-6

Lab Experiment # 01

Objective:
Demonstration of “Peter Corke” MATLAB Robotics Toolbox. Apply the translational changes during the simulation of stationary robot in the toolbox.

Equipment:
PCs with installed MATLAB and Peter Corke Robotics Toolbox

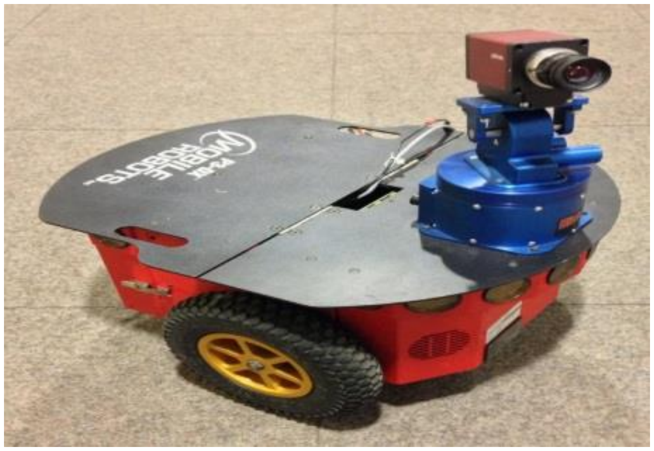
Theory:
Introduction to Robotics:

- *Robotics* is defined as an *Art* with some knowledge base to design and build useful autonomous or semiautonomous systems which can provide assistance to Humans [1].
- *Robot* is normally referred to system which is providing assistance with some autonomy.

Types of Robots:
We will discuss two types of Robots in our course:

1. Industrial Robotic Arm:
   A robotic arm, sometimes referred to as an industrial robot, is often described as a 'mechanical' arm. It is a device that operates in a similar way to a human arm, with a number of joints that either move along an axis or can rotate in certain directions [1].

2. Moving Robot (Rover):
   A rover (or sometimes planetary rover) is a planetary surface exploration device designed to move across the solid surface on a planet or other planetary mass celestial bodies [2].



Robot Components:

A Robot consists of several components, following are:

- Manipulator (or rover): main body of robot
- Links: Joining parts of robot which are associated with main body or other parts
- Joints: Provide connectivity between different parts
- Actuators: Agents which are responsible to provide force for motion on different parts
- Sensors: Agents which are responsible to sense rate of motion of robot itself or objects nearer to robot with further details
- End effector: The part at the last joint (hand/gripper)
- Controller: A numerically supported agent responsible to control all motions of robot
- Processor (Brain): A numerically supported agent responsible to determine and estimate behavior of robot and its surroundings and to generate commands to different parts
- Software: A series of instructions which are provided to Processor; for designing of Robot we will be using two software's in our lab sessions; MATLAB & ROS

MATLAB:

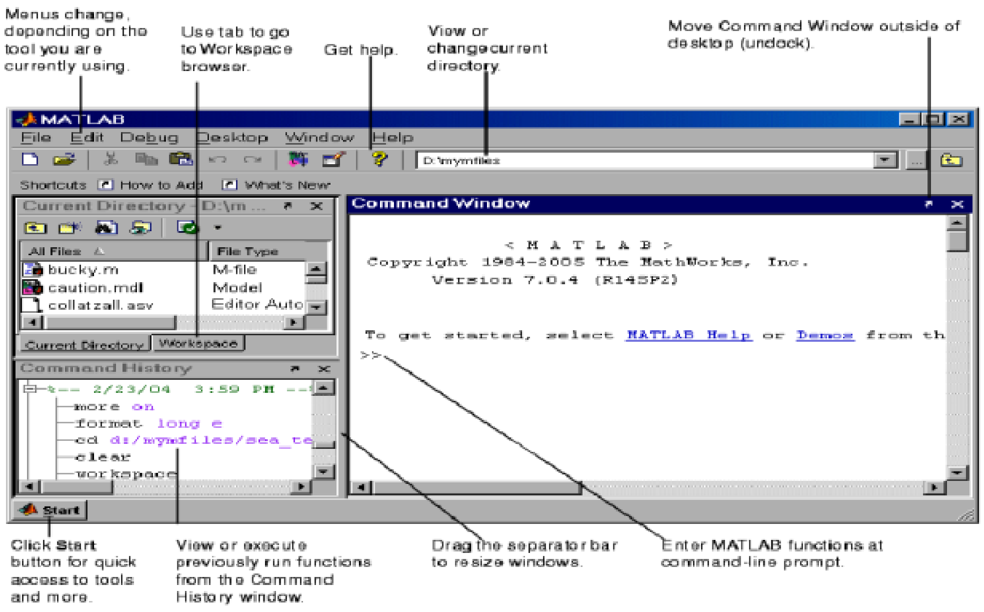
The name MATLAB stands for MATrix LABoratory. M ATLAB was written originally to provide easy access to matrix software developed by the LINPACK (linear system package) and EISPACK (Eigen system package) projects.

MATLAB is a high-performance language for technical computing. It integrates Computation, visualization, and programming environment. Furthermore, MATLAB is a modern programming language environment: it has sophisticated data structures, contains built-in editing and debugging tools, and supports object-oriented programming. These factors make MATLAB an excellent tool for teaching and research.

It has powerful built-in routines that enable a very wide variety of computations. It also has easy to use graphics commands that make the visualization of results immediately available. Specific applications are collected in packages referred to as toolbox. There are toolboxes for signal processing, symbolic computation, control theory, simulation, optimization, and several other fields of applied science and engineering.

Starting MATLAB:

After logging into account, double-click on the MATLAB shortcut icon (MATLAB) on Windows desktop. A special window called the MATLAB desktop appears. The desktop is a window that contains other windows. The major tools within or accessible from the desktop are: The Command Window the Command History the Workspace the Current Directory the Help Browser the Start button


Menus change, depending on the tool you are currently using.
Use tab to go to Workspace browser.
Get help.
View or change current directory.
Move Command Window outside of desktop (undock).
MATLAB
File Edit Debug Desktop Window Help
D:\mymfiles
Shortcuts How to Add What's New
Current Directory - D:\m ...
All Files
File Type
bucky.m M-file
caution.mdl Model
collatzall.asv Editor Auto
Current Directory Workspace
Command History
Command Window
Start
Click Start button for quick access to tools and more.
View or execute previously run functions from the Command History window.
Drag the separator bar to resize windows.
Enter MATLAB functions at command-line prompt.

```
< M A T L A B >
Copyright 1984-2005 The MathWorks, Inc.
      Version 7.0.4 (R14SP2)


To get started, select MATLAB Help or Demos from th
>>
```

```
%-- 2/23/04  3:59 PM --
  more on
  format long e
  cd d:/mymfiles/sea_te
  clear
  workspace
```

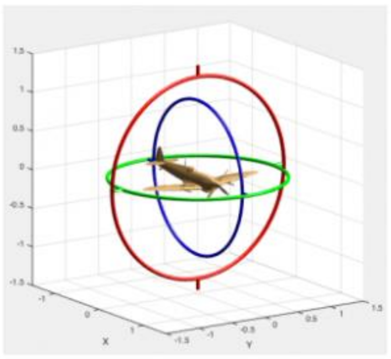
Peter Corke Robotics Toolbox:

The toolbox has developed by Prof. Peter Corke (QUT, Australia) and demonstrated in his robotic book [3]. The toolbox provided many functions that are useful for the study and simulation of classical arm-type robotics, for example such things as kinematics, dynamics, and trajectory generation. The Toolbox also provides functions for manipulating and converting between data types such as vectors, homogeneous transformations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

The Toolbox uses a very general method of representing the kinematics and dynamics of serial-link manipulators as MATLAB objects. Robot objects can be created by the user for any serial-link manipulator and a number of examples are provided for well-known robots from Kinova, Universal Robotics, and Rethink as well as classical robots such as the Puma 560 and the Stanford arm.

The toolbox also supports mobile robots with functions for robot motion models (unicycle, bicycle), path planning algorithms (bug, distance transform, D*, PRM), dynamic planning (lattice, RRT), localization (EKF, particle filter), map building (EKF) and simultaneous localization and mapping (EKF), and a Simulink model a of non-holonomic vehicle. The Toolbox also includes a detailed Simulink model for a quadrotor flying robot.

The programming commands of this toolbox have provided in the book and students can find softcopies on provided web links.



Advantages of the Toolbox:

- The code is mature and provides a point of comparison for other implementations of the same algorithms;
- The routines are generally written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. If you feel strongly about computational efficiency then you can always rewrite the function to be more efficient, compile the M-file using the MATLAB compiler, or create a MEX version;
- Since source code is available there is a benefit for understanding and teaching.

Installing the Toolbox:

There are two versions of the Robotics Toolbox:

- RTB9.10, the last in the 9th release is what is used in Robotics, Vision & Control (1st edition) and the Robot Academy.
- RTB10.x is the current release and is used in Robotics, Vision & Control (2nd edition)
  Both are available for installation using one of three installation methods:

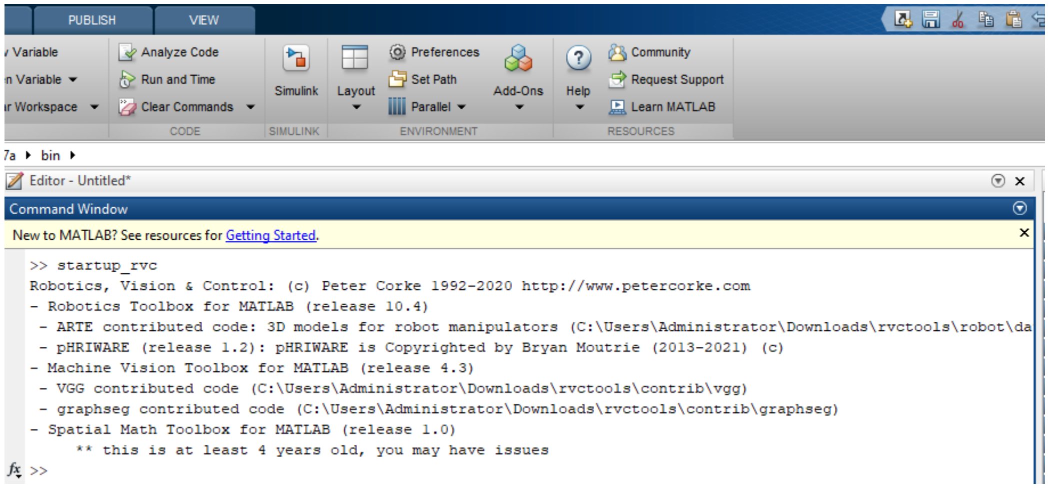
1. Direct access to a shared MATLAB Drive folder (for MATLAB19a onward)
2. Download a MATLAB Toolbox install file (.mltbx type), this is the latest version from GitHub
3. Clone the source files from GitHub

<u>Install from shared MATLAB Drive folder</u>

This will work for MATLAB Online or MATLAB Desktop provided you have MATLAB drive setup.
Note that this includes the Machine Vision Toolbox (MVTB) as well.

RVC 2nd edition: RTB10+MVTB4 (2017)

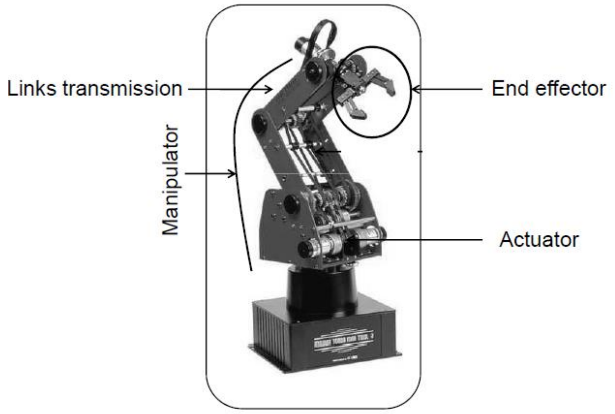
1. Download the required toolbox
2. Save it at any location
3. Open your MATLAB
4. Go to Home tab then select the Set path
5. Add with Subfolder, browse the location of the RVC toolbox then Save and Close
6. For checking the proper installation of RVC tool's installed files; write command **startup_rvc**
7. This must show the following results as shown below:



```
>> startup_rvc
Robotics, Vision & Control: (c) Peter Corke 1992-2020 http://www.petercorke.com
- Robotics Toolbox for MATLAB (release 10.4)
 - ARTE contributed code: 3D models for robot manipulators (C:\Users\Administrator\Downloads\rvctools\robot\da
 - pHRIWARE (release 1.2): pHRIWARE is Copyrighted by Bryan Moutrie (2013-2021) (c)
- Machine Vision Toolbox for MATLAB (release 4.3)
 - VGG contributed code (C:\Users\Administrator\Downloads\rvctools\contrib\vgg)
 - graphseg contributed code (C:\Users\Administrator\Downloads\rvctools\contrib\graphseg)
- Spatial Math Toolbox for MATLAB (release 1.0)
     ** this is at least 4 years old, you may have issues
>>
```

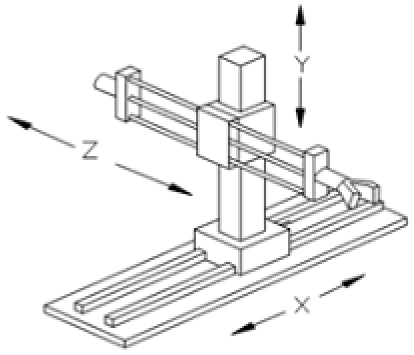
In order to see the usage of the tool box, lets apply the translational changes occurring in stationary robot and simulating the translational changes in the robotics toolbox

Industrial Robots or Manipulators:
"A robot is a re-programmable, multifunction manipulator designed to move material, parts, tools, or special devices through variable programmed motions for the performance of a variety of tasks" [1].



Cartesian Co-Ordinate Robotic Manipulators:
The Cartesian co-ordinate robot is one that consists of a column and an arm [1]. It is sometimes called an x-y-z robot, indicating the axes of motion. The x-axis is lateral motion, the y-axis is longitudinal motion, and the z-axis is vertical motion. Thus, the arm can move up and down on the z-axis; the arm can slide along its base on the x-axis; and then it can telescope to move to and from the work area on the y-axis. The Cartesian co-ordinate robot was developed mainly for arc welding, but it is also suited for many other assembly operations.
Robots with Cartesian configurations consists of links connected by linear joints (L). Gantry robots are Cartesian robots (LLL). A robot with 3 prismatic joints – the axes consistent with a Cartesian coordinate system.
Commonly used for:

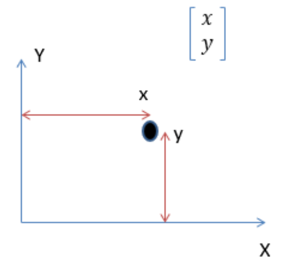
- Pick and place work
- Assembly operations
- Handling machine tools
- Arc welding



In Cartesian coordinates translation may be represented by a position vector, A_p, If A is not given the world coordinate frame is assumed. Many representations of 3D orientation have been proposed but the most used in robotics are orthonormal rotation matrices. The homogeneous transformation is a 4 ×4 matrix which represents translation and orientation and can be compounded simply by matrix multiplication. Such a matrix representation is well matched to MATLAB's powerful capability for matrix manipulation. Homogeneous transformations describe the relationships between Cartesian coordinate frames in terms of a Cartesian translation, p, and orientation expressed as an orthonormal rotation matrix, R is a 3 × 3.

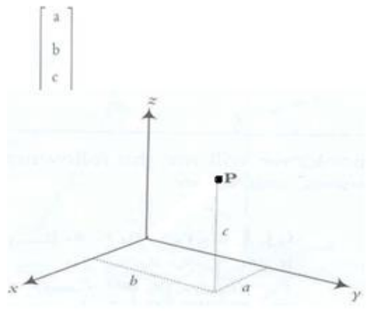
Representation of a Point in Space:
A point P in space can be represented by three coordinates relative to a reference frame [1]:

$$P = a_x i + b_y j + c_z k$$



A point in 2D space

Two pieces of information is required



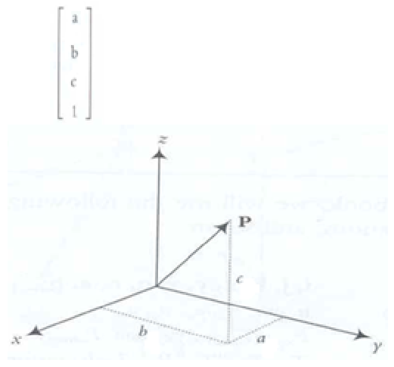
A point in 3D space

Representation of a vector in Space:
A vector can be presented by three coordinates of its tail and its head. If the vector starts at point A and end at point B, then it can be represented by $P_{AB} = (B_x - A_x)i + (B_y - A_y)\, j + (B_z - A_z)\, k$.

Where a_x, b_y and c_z are three components of the vector in the reference frame. The three components of the vector can also be written in matrix form as:

$$P = \begin{bmatrix} a_x \\ b_y \\ c_z \end{bmatrix}$$

this representation can be slightly modified to also include a scale factor w such that if P_x, P_y and P_z are divide by w, they will yield a_x, b_y, and c_z.



A simple vector in space

Three pieces of information is required

$$a = \frac{A}{w}$$

$$b = \frac{B}{w}$$

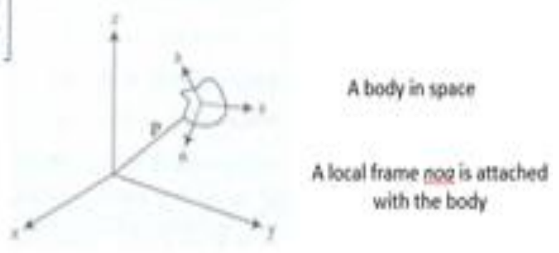
$$c = \frac{C}{w}$$

Scale Factor representation

Representation of a Frame at the origin of a fixed reference frame:
A frame is generally represented by three mutually orthogonal axes (x, y and z). since we may have more than 1

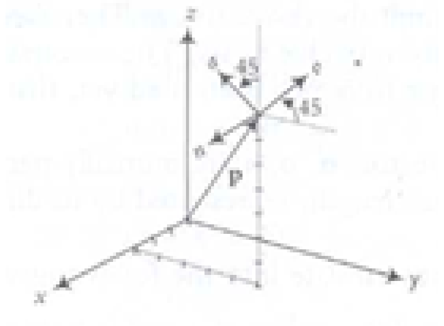
frame at any given time, we will use axes x, y and z to represent the fixed Universe reference frame $F_{x,v,z}$ and a set of axes n, o and a to represent another moving frame $F_{n,o,a}$ relative to the reference frame.

$$\begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Twelve pieces of information is represented. Three for locations and nine for orientations

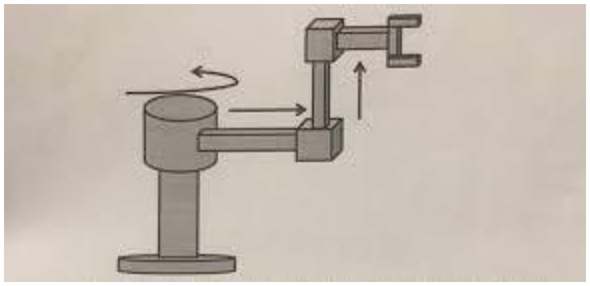
$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 0.707 & -0.707 & 5 \\ 0 & 0.707 & 0.707 & 7 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Transformation:
Movement of a local frame in space with respect to a fixed frame is known as Transformation
There are two different types of transformations:

- Translations: Having displacement without orientation change
- Rotations: Having only orientation change
- Combined transformations: Having both changes (Translations & Rotations)

Translational Transformation:
If a frame moves in a space without any change in its orientation then the transformation is pure translation. In this case, the vector remains in the same direction and thereforer don’t change . the only thong that change is the location of the origin of the frame relative to the refrance frame.



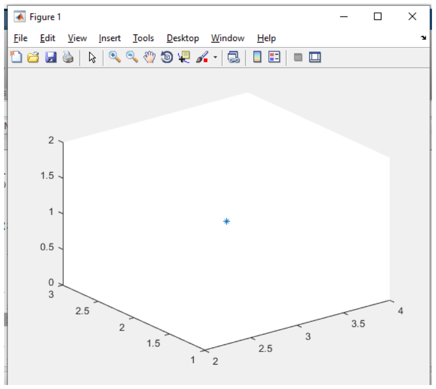
Procedure:
We are using MATLAB 2019(a) for performing these operations, by using MATLAB Toolbax ‘Petercorke’

First we define a point (3, 2,1) relative to the world frame which is a column vector and add it to the plot

```
P = [3 ; 2 ; 1]

P =

    3
    2
    1
```

Here we are describing the position of a point in 3D space by using following command

```
>> plot3(P(1), P(2),  P(3), '*');
```



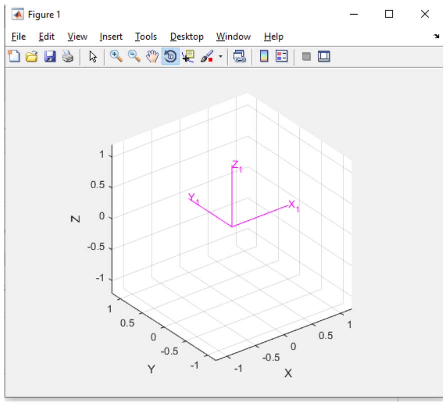
We create a homogeneous transformation (4*4) using the function TRANSL, which is a translational matrix at origin of a world frame having no change in its direction

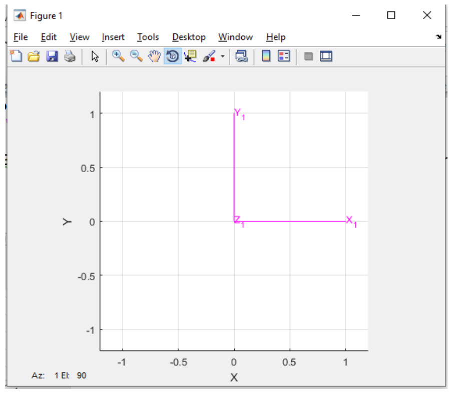
```
>> T1 = transl(0, 0, 0)

T1 =

    1    0    0    0
    0    1    0    0
    0    0    1    0
    0    0    0    1
```

which represents a translation of (0, 0, 0) with a rotation of 0°, where TRANSL(X, Y, Z) is an homogeneous transform (4x4) representing a pure translation of X, Y and Z.
We can plot this, relative to the world coordinate frame, by

```
 >> trplot(T1, 'frame', '1', 'color', 'm')
```

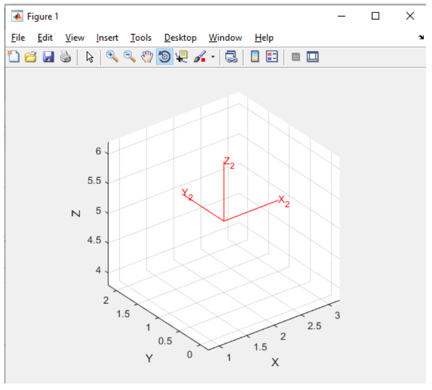
The options TRPLOT plot a 3D coordinate frame specifies that the label for the frame is {1} and it is colour magenta and this is shown in Fig.

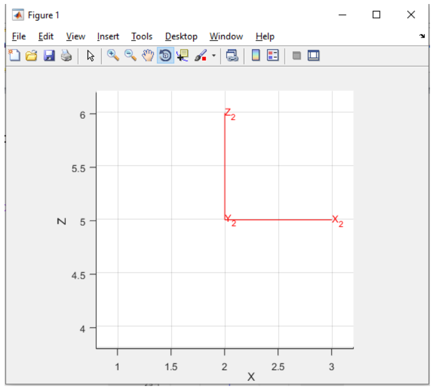




We create another relative pose which has a displacement of (2, 1,5) and again with zero rotation according to the world frame
And then plot a 3D coordinate frame specifies that the label for the frame is {2} and it is colour red

```
>> T2 = transl(2, 1, 5)
T2 =

   1   0   0   2
   0   1   0   1
   0   0   1   5
   0   0   0   1

>> axis([0 5 0 5]);
>> trplot(T2, 'frame', '2', 'color', 'r');
```

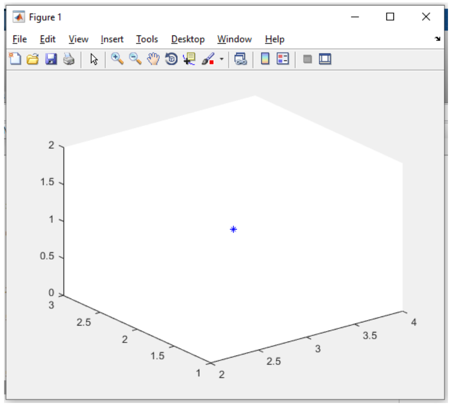




Here determine the coordinate of the point with respect to {1} & {2} respectively

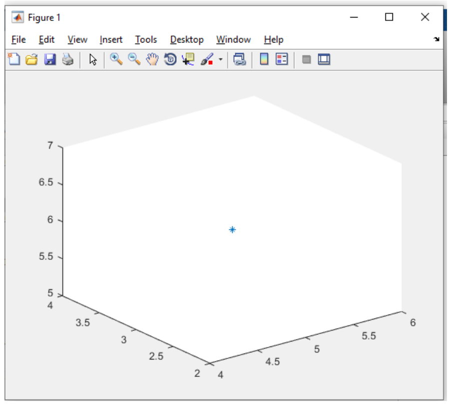
```
>> P1 = T1 * [P ; 1]
P1 =
   3
   2
   1
   1
```

```
>> plot3(P1(1), P1(2), P1(3), '*', 'color', 'b');
```



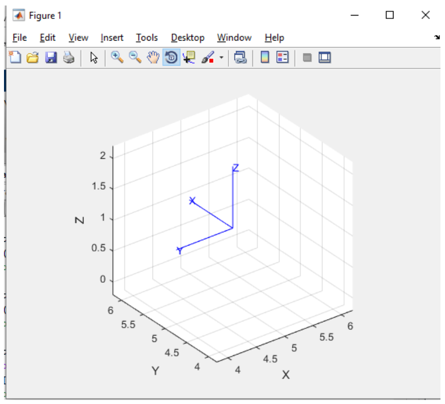
The same point P with respect to {2} is

```
>> P2 = T2 * [P ; 1]
P2 =

    5
    3
    6
    1
```



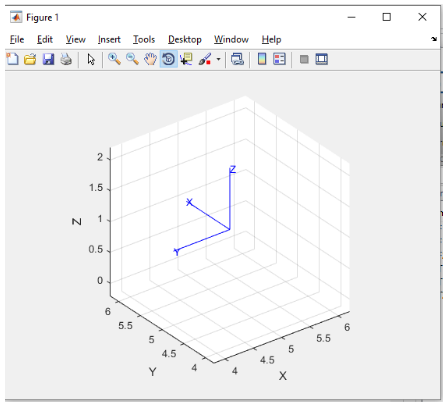
Next we have a plot of a body which is parallel to z-axis and 90^o rotation in its x and y axes and has a position of 5,3,2 units in x, y and z axes respectively

```
>> F1 = [0, -1, 0, 5; 1, 0, 0, 5; 0, 0, 1, 1 ; 0, 0, 0, 1]

F1 =
    0   -1    0    5
    1    0    0    5
    0    0    1    1
    0    0    0    1
>> trplot(F1)
```



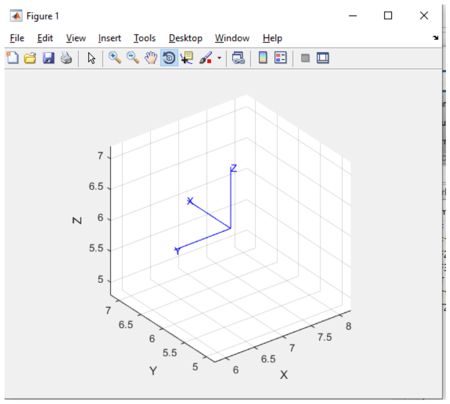
Now we expect the change in its position only according to the given translational matrix {1} & {2} respectively

```
>> F2 = T1 * F1
F2 =

    0   -1    0    5
    1    0    0    5
    0    0    1    1
    0    0    0    1
>> trplot(F2)
```



The same frame F1 with respect to {2} is

```
>> F3 = T2 * F1
F3 =

    0   -1    0    7
    1    0    0    6
    0    0    1    6
    0    0    0    1
>> trplot(F3)
```



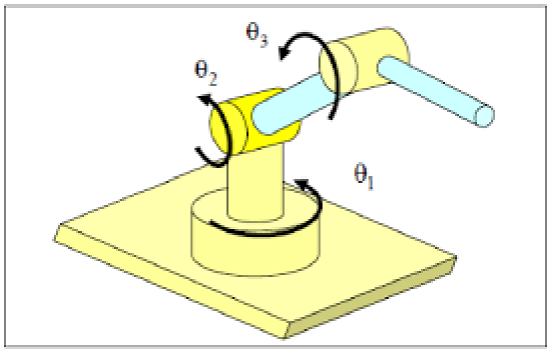
Tasks:

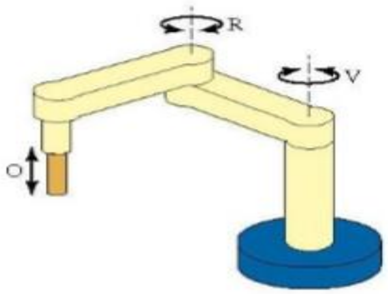
1. Run the startup_rvc command and provide the output message generating on MATLAB terminal.
2. Initially a robotic base is at the coordinate (1,2,0) now your task is to move the base to a new location when it translated to dx=5, dy= 2 and dz=0.
3. Now move the initial located base representing by a frame F1 in which its local x-axis is parallel to z-axis and move it to a new location where dx=2, dy=3 and dz=4

Lab Experiment #02

Objective:
Compute the rotational changes during the simulation of stationary robot in the robotics toolbox.

Theory:
Representing Orientation in 3-Dimensions:
Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis. The following figures are showing the two different industrial manipulators which are equipped with rotating joints and can perform different industrial operations [1].





Orthonormal Rotation Matrix
Just as for the 2-dimensional case we can represent the orientation of a coordinate frame by its unit vectors expressed in terms of the reference coordinate frame. Each unit vector has three elements and they form the columns of a 3 × 3 orthonormal matrix ${}^{A}R_{B}$

$$\begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix} = {}^{A}R_{B} \begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix}$$

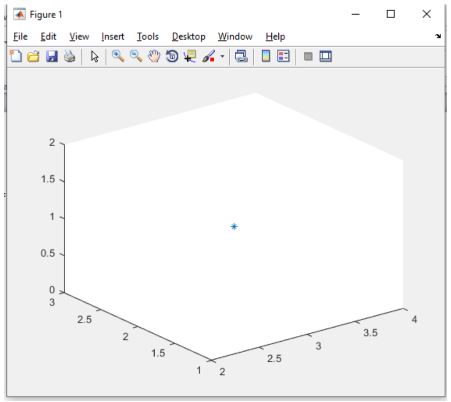
which rotates a vector defined with respect to frame {B} to a vector with respect to {A}.

Procedure:
We are using MATLAB 2019(a) for performing these operations, by using MATLAB Toolbax 'Peter Corke'
First we are defining a point (3, 2, 1) relative to the world frame which is a column vector and add it to the plot

```
P = [3 ; 2 ; 1]

P =

     3
     2
     1
```

Now we can describe a vector P, describing position of a point in 3D space by using following command

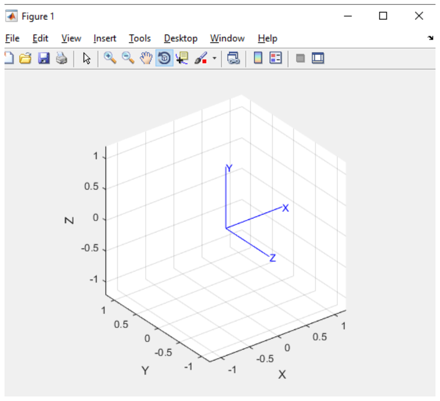
```
>> plot3(P(1), P(2),  P(3), '*');
```

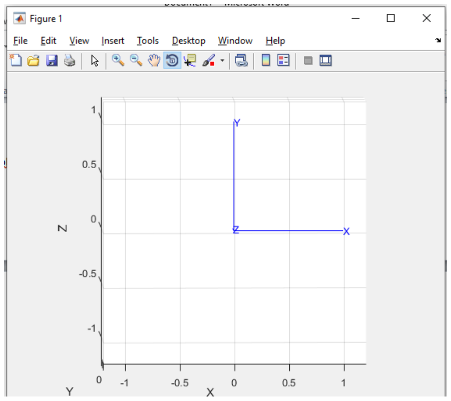


Now we are creating an orthonormal rotation matrix (3x3) which is showing the rotation about the x-axis of the reference frame

```
>> Rx = rotx(pi/2)
Rx =

     1     0     0
     0     0    -1
     0     1     0
>> trplot(Rx)
>> tranimate(R)
```

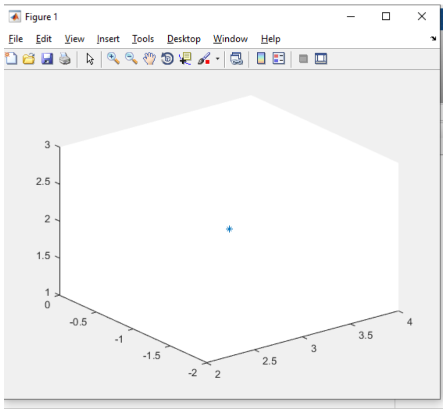
TRANIMATE animate a 3D coordinate frame





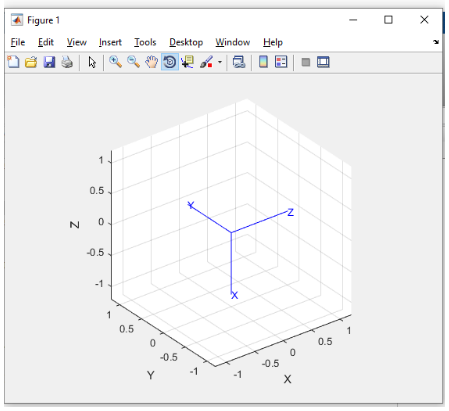
Now Px is the rotated frame multiplied by the rotation matrix in the reference frame

```
>> Px = Rx * P
Px =
     3
    -1
     2
```



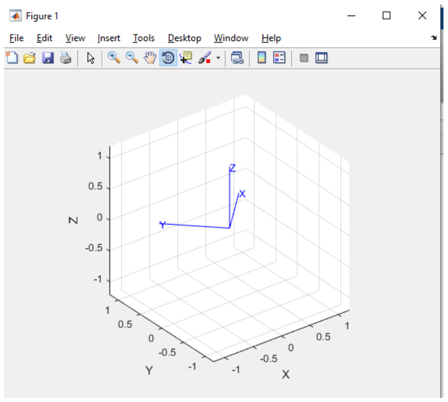
Similarly we have a rotation about the y-axis

```
>> Ry = roty(pi/2)
Ry =

     0     0     1
     0     1     0
    -1     0     0
>> trplot(Ry)
```



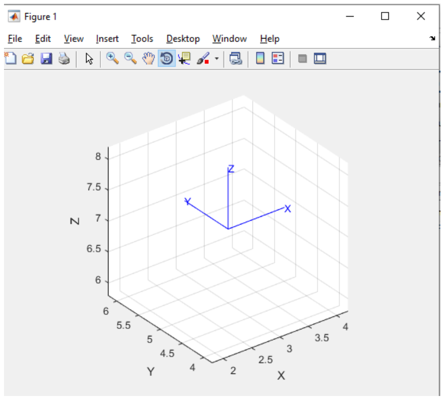
And in last we have a rotation about the z-axis

```
>> Rz = rotz(pi/4)

Rz =

    0.7071   -0.7071         0
    0.7071    0.7071         0
         0         0    1.0000
>> trplot(Rz)
```



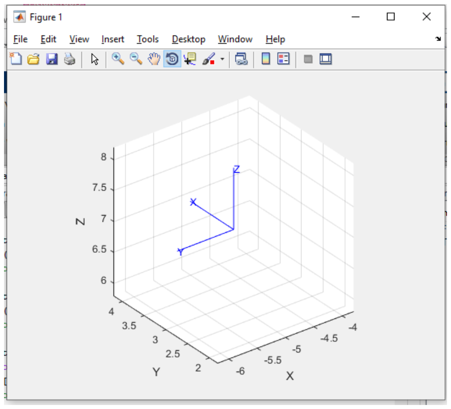
Next we have a plot of a body which is 0^o rotated about x- axis and it has a position of 3,5,7 units in x, y and z axes respectively

```
>> F1 = [1, 0, 0, 3; 0, 1, 0, 5; 0, 0, 1, 7; 0, 0, 0, 1]
F1 =

    1    0    0    3
    0    1    0    5
    0    0    1    7
    0    0    0    1
>> trplot(F1)
```

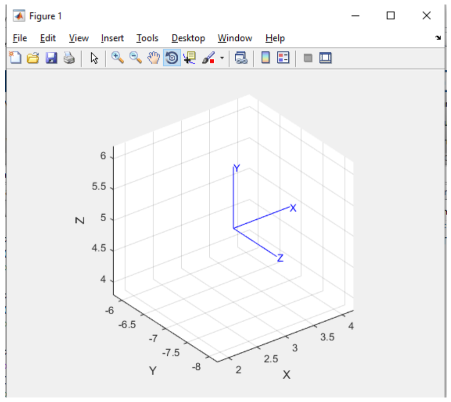


Here we have a frame which has some rotation in y-axis and trotx is a homogeneous transformation (4x4) representing a rotation % of theta radians about the x-axis.

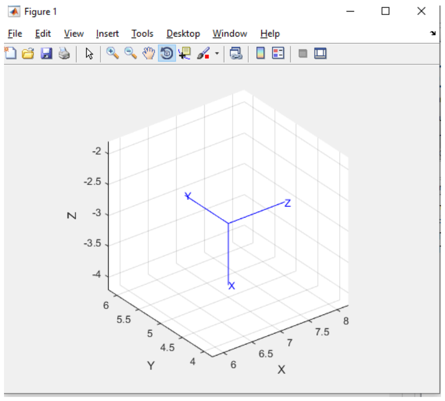
```
>> T = trotz(pi/2) * F1

T =

    0   -1    0   -5
    1    0    0    3
    0    0    1    7
    0    0    0    1
>> trplot(T)
```



```
>> T1 = trotx(pi/2) * F1
T =

    1    0    0    3
    0    0   -1   -7
    0    1    0    5
    0    0    0    1
>> trplot(T1)
```

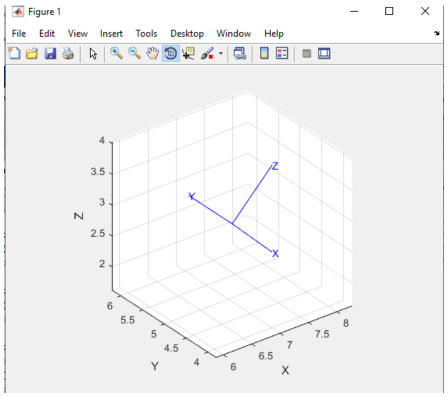


```
>> T2 = troty(pi/2) * F1
T2 =

    0    0    1    7
    0    1    0    5
   -1    0    0   -3
    0    0    0    1
>> trplot(T2)
```



At this time we already have a new frame with 90 degree rotation about x-axis, now we are multiplying it by the rotation about y-axis to achieve a new location

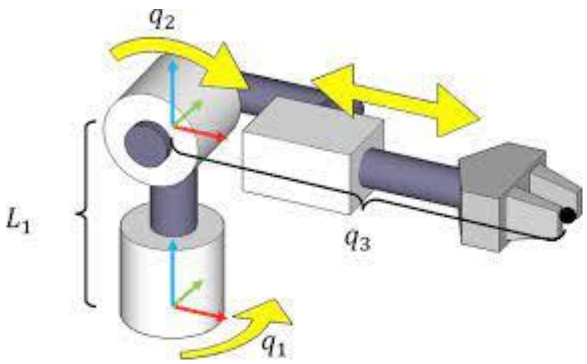
```
>> F2 = troty(pi/4) * F1

F2 =

    0.7071    0.4999    0.4999    7.0711
         0    0.7070   -0.7070    5.0000
   -0.7071    0.4999    0.4999    2.8284
         0         0         0    1.0000
>> trplot(F2)
```



Combining Translation and Orientation

Combine transformation consist of number of successive translations and rotations about a fixed reference frame axes. Any transformation can be resolved into a set of translations and rotations in a particular order.

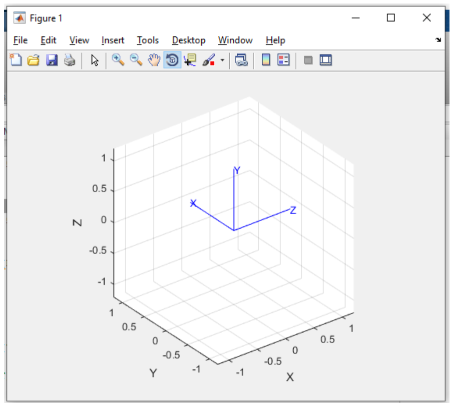
We return now to representing relative pose in three dimensions, the position and orientation change, between two coordinate frames. We have discussed several different representations of orientation, and we need to combine this with translation, to create a tangible representation of relative pose. The two most practical representations are: the quaternion vector pair and the 4 × 4 homogeneous transformation matrix.



The 4 × 4 homogeneous transformation is very commonly used in robotics and computer vision, is supported by the Toolbox and will be used throughout this book as a concrete representation of 3-dimensional pose. The Toolbox has many functions to create homogeneous transformations.

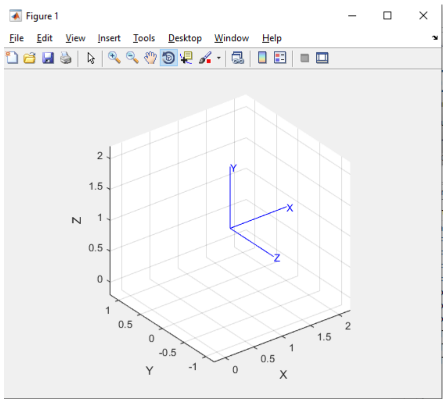
Here we are compounding 2 rotations

```
rotx(pi/2) * roty(pi/2)

ans =

     0     0     1
     1     0     0
     0     1     0
>> trplot(ans)
```



we can demonstrate composition of transforms by

```
>> T = transl(1, 0, 0) * trotx(pi/2) * transl(0, 1, 0)
T =

     1     0     0     1
     0     0    -1     0
     0     1     0     1
     0     0     0     1
>> trplot(T)
```

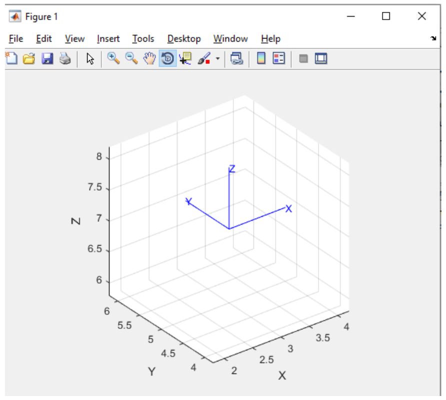


Here we are finding a coordinates of the point relative to the reference frame which is currently at the point P (7,3,1), which is subjected to a reference frame by the rotation of 90^o about z-axis , translation of [4,-3,7] and a rotation of 90^o about y-axis

```
>> T = troty(pi/2) * transl (4, -3, 7) * trotz(pi/2) * [7;3;1;1]
T =

    8
    4
   -1
    1
```

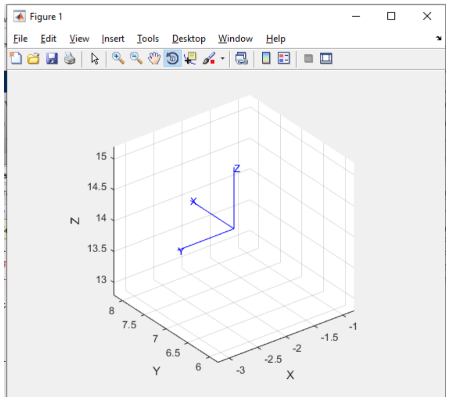
Next we have a plot of a body which is 0^o rotated about x- axis and it has a position of 3,5,7 units in x, y and z axes respectively

```
>> F1 = [1, 0, 0, 3; 0, 1, 0, 5; 0, 0, 1, 7; 0, 0, 0, 1]
F1 =

    1     0     0     3
    0     1     0     5
    0     0     1     7
    0     0     0     1
>> trplot(F1)
```



Here we are generating a new frame by providing some rotation and translational change to the previous frame

```
F2 = trotz(pi/2) * transl (4, -3, 7) * F1
T =
0   -1    0   -2
1    0    0    7
0    0    1   14
0    0    0    1

>> trplot(F2)
```



Tasks:

1. Initially a robot's component is at point P (7,3,1). Now move it to a new location by giving it a rotation of 90^o about the z-axis.
2. Initially a robot's component is at point P (7,3,1). Now move it to a new location by giving it a rotation of 90^o about the z-axis, followed by a rotation of 90^o about x-axis,
3. Initially a robot's component is at point P (7,3,1). Now move it to a new location by giving it a rotation of 90^o about the z-axis, followed by a rotation of 90^o about x-axis, followed by a translation of [4, -3, 7].
4. In this case assume the same point P (7,3,1). But the transformation is performed in different order. First give it a rotation of 90^o about the z-axis, followed by a translation of [4, -3, 7], and in last followed by a rotation of 90^o about y-axis
5. Initially a frame is at point P (2,3,8) and having rotation of 90^O in its x-axis, now repeat the task#4 for it.
6. Use tranimate to show translation and rotational motion about various axes.
7. Animate rotation about all axes.

Lab Experiment # 03

Objective:
Use the kinematic principle for a 2 DOF industrial manipulator and to simulate in the toolbox.

Theory:

- Kinematics is the branch of classical mechanics that describes the motion of points, bodies (objects) and systems of bodies (groups of objects) without consideration of the causes of motion. It is the study of the position, velocity and acceleration, and all higher order derivatives of the position variables
- Forward kinematics refers to the use of the kinematic equations of a robot to compute the Cartesian location and orientation of robotic arm by using known joints data parameters. It is the solution gives the coordinate frame, or pose, of the last link
- Forward kinematics (for a robot arm) takes as input joint angles and calculates the Cartesian position and orientation of the end effector. It is the problem of solving the Cartesian position and orientation of the end-effector given knowledge of the kinematic structure and the joint coordinates. The kinematic structure of a serial-link manipulator can be succinctly described in terms of DenavitHartenberg parameters. Within the Toolbox the manipulator's kinematics are represented in a general way by a dh matrix which is given as the first argument to Toolbox kinematic functions. The dh matrix describes the kinematics of a manipulator using the standard Denavit-Hartenberg conventions, where the link and joint parameters may be summarized as:

  theta: link angle
  d: link offset
  alpha: link twist
  a: link length
  offset: joint coordinate offset
  name: joint coordinate name
  flip: joint moves in opposite direction

- Typical robots are serial-link manipulators comprising a set of bodies, called links, in a chain, connected by joints. For a manipulator with n joints, there are n+1 links
- A seriallink manipulator comprises a chain of mechanical links and joints. Each joint can move its outward neighbouring link with respect to its inward neighbour. One end of the chain, the base, is generally fixed and the other end is free to move in space and holds the tool or end-effector.
- A serial-link manipulator comprises a set of bodies, called links, in a chain and connected by joints. Each joint has one degree of freedom, either translational (a sliding or prismatic joint) or rotational (a revolute joint). Motion of the joint changes the relative angle or position of its neighbouring links.
- Denavit and Hartenberg proposed a matrix method of assigning coordinate systems to each link

Procedure:

We are creating a 2-link robot shown in given figure:

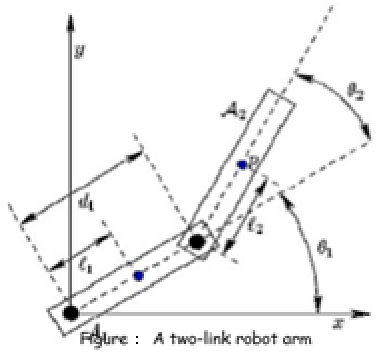


Figure : A two-link robot arm

A Link object holds all information related to a robot joint and link such as kinematics parameters, rigid-body inertial parameters, and motor and transmission parameters. First we are creating link 1 by the name of L(1)

```
>> L(1) = Link([0 0 1 0])

L =
Revolute(std): theta=q, d=0, a=1, alpha=0, offset=0
```

Now we are generating a link transform matrix by a 4 × 4 homogeneous transformation with a Denavit-Hartenberg parameter theta (revolute)

```
>> L.A(0)


ans =
     1     0     0     1
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Here it indicates the number of links in kinematic parameter

```
>> L.a

ans =

   1
```

It is a link transform matrix by a 4 × 4 homogeneous transformation with a Denavit-Hartenberg parameter theta (revolute)

```
>> L.A(pi/2)

ans =
     0    -1     0     0
     1     0     0     1
     0     0     1     0
     0     0     0     1
```

This same can be done by defining a link which contain an offset

```
>>  L.offset = pi/2

L =
Revolute(std): theta=q, d=0, a=1, alpha=0, offset=1.5708
```

Now again generate its transform matrix

```
>> L.A(0)

ans =
        0      -1       0       0
        1       0       0       1
        0       0       1       0
        0       0       0       1
```

Here we are creating a link 2

```
>> L(2) = Link([0 0 1 0])

L =
Revolute(std):  theta=q1   d=0         a=1         alpha=0         offset=1.571
Revolute(std):  theta=q2   d=0         a=1         alpha=0         offset=0
```

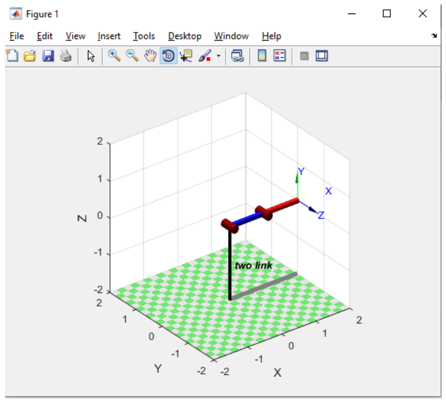
A concrete class that represents a serial-link arm type robot. Each link and joint in the chain is described by a Link-class object using Denavit-Hartenberg parameters, where “two_link” is a variable name and constructing a serial link for L; L (1) & L (2)

```
>> two_link = SerialLink(L, 'name', 'two link')

two_link =

two link:: 2 axis, RR, stdDH, slowRNE
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|          0|          1|          0|     1.5708|
|  2|         q2|          0|          1|          0|          0|
```

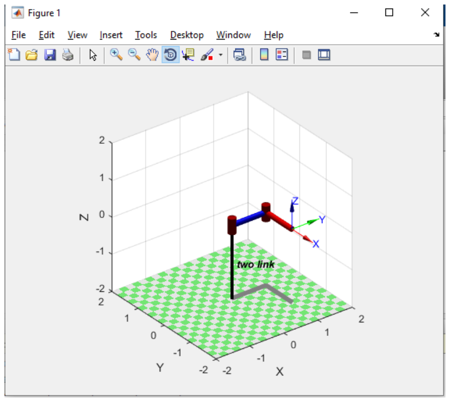
And here we are plotting it at 0-axes by using

```
>> two_link.plot([0 0])
```

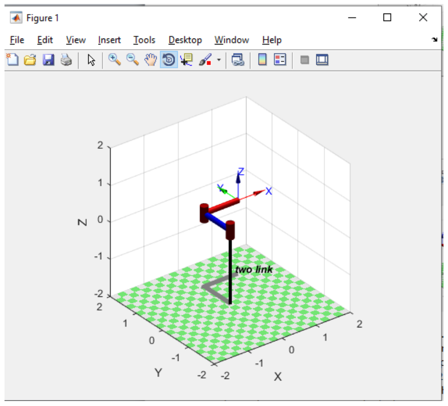
Here the first element of matrix defining the link 1 and the second element is defing the link 2



Here its pose changes,
>> two_link.plot([0 -pi/2])



Here its pose changes,
>> two_link.plot([pi/2 -pi/2])



This provides a concise description of the robot. We see that it has 2 revolute joints as indicated by the structure string 'RR', it is defined in terms of standard DenavitHartenberg parameters, that gravity is acting in the default z-direction. The kinematic parameters of the link objects are also listed and the joint variables are shown as variables q1 and q2. We have also assigned a name to the robot which will be shown whenever the robot is displayed graphically. The script **MDL_TWOLINK** is a script that creates the workspace creating a SerialLink object named twolink which describes the kinematic and dynamic characteristics of a simple planar 2-link mechanism moving in the xz-plane, it experiences gravity loading.

```
>> mdl_twolink
```

Obtain the number of joints,

```
>> twolink.n

ans =

     2
```

Now this returns a vector of Link objects comprising the robot,

```
>> links = twolink.links

links =
Revolute(std):  theta=q1   d=0          a=1          alpha=0          offset=0
Revolute(std):  theta=q2   d=0          a=1          alpha=0          offset=0
```

Now we can put the robot arm to work. The forward kinematics is computed using the fkine method where fkine is forward kinematics

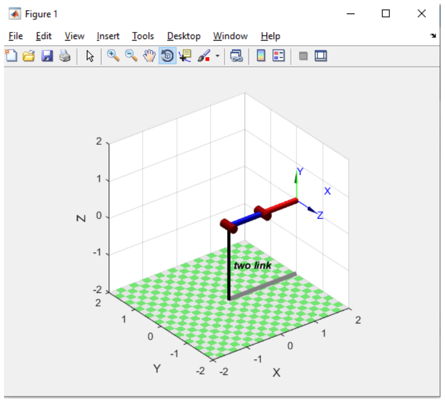
```
>> twolink.fkine([0 0])


ans =
         1         0         0         2
         0         1         0         0
         0         0         1         0
         0         0         0         1
```

The method returns the homogenous transform that represents the pose of the second link coordinate frame of the robot

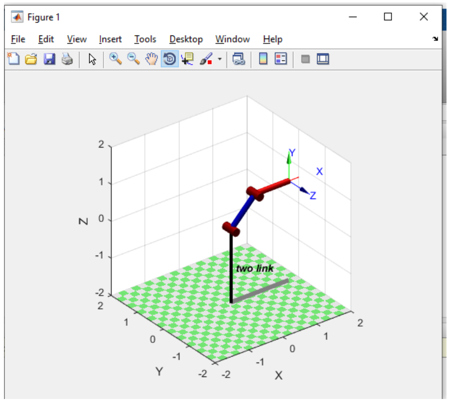
```
>> twolink.fkine([pi/4 -pi/4])


ans =
         1         0         0     1.707
         0         1         0     0.707
         0         0         1         0
         0         0         0         1
```

The robot can be visualized graphically, the pose (0, 0);

```
>> twolink.plot([0 0])
```



Here the pose (pi/4 -pi/4)
>> twolink.plot([pi/4 -pi/4])



Tasks:

1. For the simple two link manipulator shown in above figure, consider a1=a2=1, and the joint angle variations for both joints θ1=pi and θ2=pi/2.
2. For a serial link manipulator generate a 3 link robotic arm at angle (0, pi/2, –pi/2)

Lab Experiment # 04

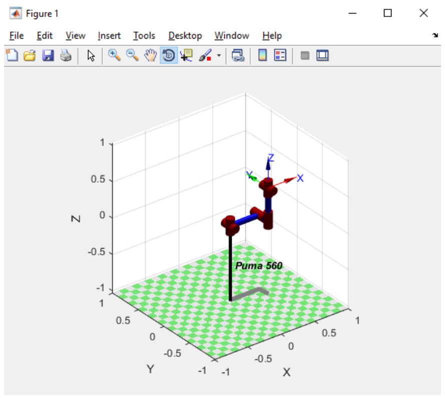
Objective:
Illustrate the inverse kinematic principle of a PUMA industrial manipulator and to test in toolbox

Theory:

- Kinematics is the branch of classical mechanics that describes the motion of points, bodies (objects) and systems of bodies (groups of objects) without consideration of the causes of motion. It is the study of the position, velocity and acceleration, and all higher order derivatives of the position variables.
- Inverse kinematics is the use of kinematic equations to determine the motion of a robot to reach a desired position or calculating joints data by using known location and orientation of robotic arm.
- In computer animation and robotics, inverse kinematics is the mathematical process of calculating the variable joint parameters needed to place the end of a kinematic chain, such as a robot manipulator or animation character's skeleton, in a given position and orientation relative to the start of the chain.
- We have shown how to determine the pose of the end-effector given the joint coordinates. A problem of real practical interest is the inverse problem: given the desired pose of the end-effector what are the required joint coordinates? For example, if we know the Cartesian pose of an object, what joint coordinates does the robot need in order to reach it? This is the inverse kinematics problem. In general, this solution is non-unique, and for some classes of manipulator no closed-form solution exists.
- Inverse kinematics is the problem of finding the robot joint coordinates, given a homogeneous transform representing the last link of the manipulator. It is very useful when the path is planned in Cartesian space, for instance a straight-line path as shown in the trajectory demonstration.
- With the help of inverse kinematics, we will be able to determine the value of each joint in order to place the robot at a desired position and orientation.

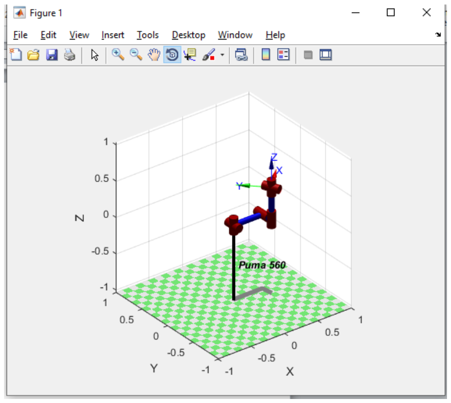
Procedure:
We will explore inverse kinematics using the Puma robot model, First generate the transform corresponding to a particular joint coordinate, q, using forward kinematics
Now

```
>> mdl_puma560
>> q = [0 0 0 0 0 0]

q =

    0    0    0    0    0    0

>> T = p560.fkine(q)

T =
        1        0        0   0.4521
        0        1        0    -0.15
        0        0        1   0.4318
        0        0        0        1
>> qi = p560.ikine(T)
qi =
     0     0     0     0     0     0

>> p560.plot (qi)
```



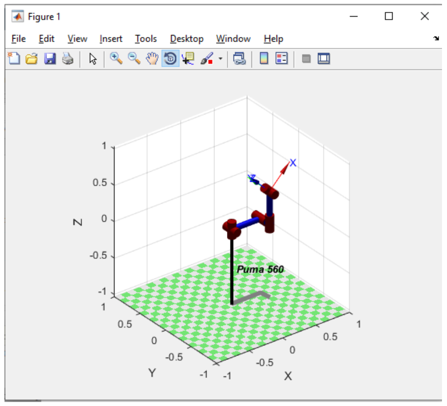
Now

```
q = [0 0 0 0 0 pi/4]

q =

         0         0         0         0         0    0.7854

>> T = p560.fkine(q)



T =
    0.7071   -0.7071         0    0.4521
    0.7071    0.7071         0     -0.15
         0         0         1    0.4318
         0         0         0         1
>>  qi = p560.ikine(T)

qi =

    0.0000   -0.0000    0.0000    0.3918   -0.0000    0.3936

>> p560.plot (qi)
```



Now

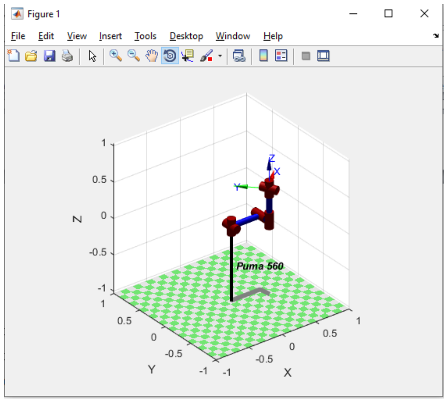
```
>> q = [0 0 0 0 pi/4 0]

q =

         0         0         0         0    0.7854         0

>> T = p560.fkine(q)


T =
    0.7071         0   -0.7071    0.4521
         0         1         0     -0.15
    0.7071         0    0.7071    0.4318
         0         0         0         1
>> qi = p560.ikine(T)

qi =

    0.0000   -0.0000    0.0000   -0.0000    0.7854    0.0000

>>  p560.plot (qi)
```



Now

```
>> q = [0 0 0 pi/4 0 0]
```

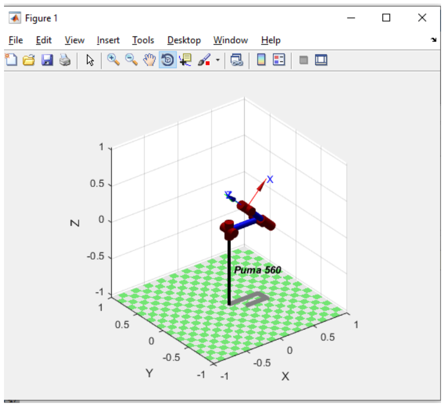
```
q =

         0         0         0    0.7854         0         0

>> T = p560.fkine(q)


T =
    0.7071   -0.7071         0    0.4521
    0.7071    0.7071         0     -0.15
         0         0         1    0.4318
         0         0         0         1
>> qi = p560.ikine(T)

qi =

    0.0000   -0.0000    0.0000    0.3918   -0.0000    0.3936

>>  p560.plot (qi)
```



Now

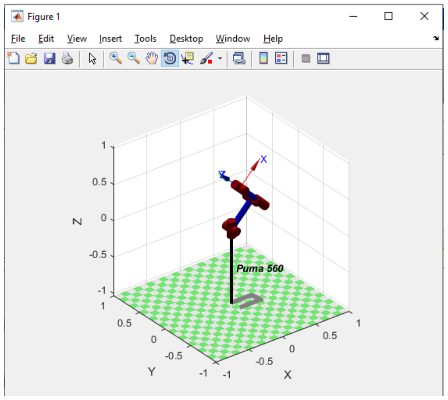
```
>> q = [0 0 pi/4 0 0 0]

q =

         0         0    0.7854         0         0         0

>> T = p560.fkine(q)


T =
    0.7071         0   -0.7071    0.1408
         0         1         0     -0.15
    0.7071         0    0.7071    0.3197
         0         0         0         1
>> qi = p560.ikine(T)
```

```
qi =

   -0.0000   -0.0000    0.7854    0.0207    0.0000   -0.0207

>> p560.plot (qi)
```



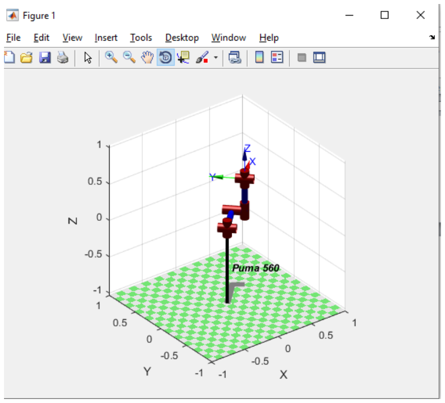
Now

```
>> q = [0 pi/4 0 0 0 0]

q =
         0    0.7854         0         0         0         0

>> T = p560.fkine(q)

T =
    0.7071         0   -0.7071   0.01435
         0         1         0     -0.15
    0.7071         0    0.7071     0.625
         0         0         0         1
>> qi = p560.ikine(T)

qi =

   -0.0000    0.7854    0.0000   -0.0055   -0.0000    0.0055

>>  p560.plot (qi)
```



Now

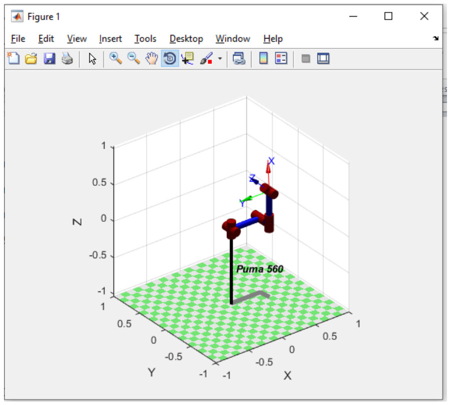
```
>> q = [pi/4 0 0 0 0 0]

q =

    0.7854         0         0         0         0         0

>> T = p560.fkine(q)


T =
    0.7071   -0.7071         0    0.4258
    0.7071    0.7071         0    0.2136
         0         0         1    0.4318
         0         0         0         1
>> qi = p560.ikine(T)

qi =

    0.7854   -0.0000    0.0000    0.0061   -0.0000   -0.0061

>>  p560.plot (qi)
```



Now

```
>> mdl_puma560
>> q = [0 0 0 0 pi pi]
```

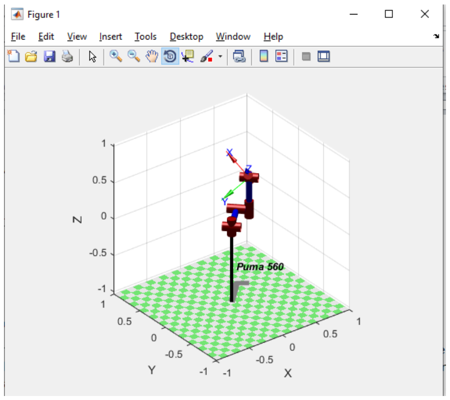
```
q =

         0         0         0         0    3.1416    3.1416

>> T = p560.fkine(q)


T =
         1         0         0    0.4521
         0        -1         0     -0.15
         0         0        -1    0.4318
         0         0         0         1
>> qi = p560.ikine(T)

qi =

     0     0     0     0     0     0

>> p560.plot (qi)
```



Now

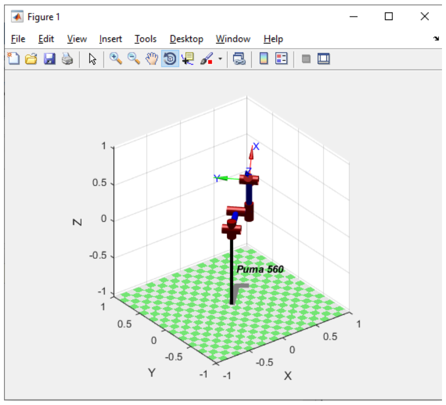
```
>> q = [pi/4 0 0 0 pi/4 pi/4]

q =

    0.7854         0         0         0    0.7854    0.7854

>> T = p560.fkine(q)


T =
   -0.1464   -0.8536   -0.5000    0.4258
    0.8536    0.1464   -0.5000    0.2136
    0.5000   -0.5000    0.7071    0.4318
         0         0         0         1
```

```
>> qi = p560.ikine(T)

qi =

    0.7854   -0.0000    0.0000   -0.0000    0.7854    0.7854

>> p560.plot (qi)
```



Now

```
>> q = [pi/4 0 0 0 pi/4 0]

q =

    0.7854         0         0         0    0.7854         0

>> T = p560.fkine(q)


T =
    0.5000   -0.7071   -0.5000    0.4258
    0.5000    0.7071   -0.5000    0.2136
    0.7071         0    0.7071    0.4318
         0         0         0         1
>> qi = p560.ikine(T)

qi =

    0.7854   -0.0000    0.0000   -0.0000    0.7854    0.0000

>> p560.plot (qi)
```



Now

```
>>  mdl_puma560
```

At the nominal joint coordinates (The wrist angles are defined relative to a nominal posture of the forearm and hand: the nominal posture is the forearm hanging straight down while the palm is parallel to the sagittal plane and facing towards the body)

```
>>  qn
qn =

        0    0.7854    3.1416         0    0.7854         0
```

the end-effector pose is

```
>>  T = p560.fkine(qn)
T =
        0         0         1    0.5963
        0         1         0   -0.1501
       -1         0         0   -0.01435
        0         0         0         1
```

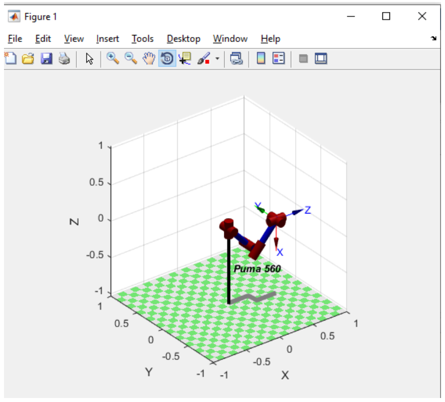
Now the inverse kinematic procedure for any specific robot can be derived symbolically and in general an efficient closed-form solution can be obtained. However we are given only a generalized description of the manipulator in terms of kinematic parameters so an iterative solution will be used. The procedure is slow, and the choice of starting value affects search time and the solution found, since in general a manipulator may have several poses which result in the same transform for the last link. The starting point for the first point may be specified, or else it defaults to zero (which is not a particularly good choice in this case)

We use the method ikine (inverse kinematics using iterative numerical method) to compute the general inverse kinematic solution, which is different to the original value (qn) but does result in the correct tool pose

```
>> qi = p560.ikine(T)

qi =
   -0.0000   -0.8335    0.0940    0.0000   -0.8312   -0.0000


>>  p560.fkine(qi)


ans =

    0.0000   -0.0000         1    0.5963
   -0.0000         1    0.0000   -0.1501
        -1   -0.0000    0.0000   -0.01435
         0         0         0         1
```

Plotting the pose

```
>> p560.plot (qi)
```



OR

```
>>mdl_puma560

>>q = [0 -pi/4 -pi/4 0 pi/8 0]
q =
         0   -0.7854   -0.7854         0    0.3927         0

>>T = p560.fkine(q)

T =
    0.3827         0    0.9239    0.7371
         0         1         0   -0.1501
   -0.9239         0    0.3827   -0.3256
         0         0         0         1


>>qi = p560.ikine(T)
```

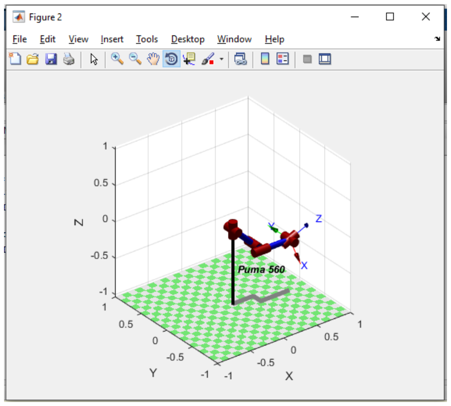
```
>>qi
qi =
   0.0000   -0.7854   -0.7854    0.0000    0.3927   -0.0000
```

and the result is the same as the original set of joint angles

```
>>q
q =
        0   -0.7854   -0.7854         0    0.3927         0
```

```
>> p560.plot (qi)
```

shows clearly that ikine has found the elbow-down configuration



Tasks:

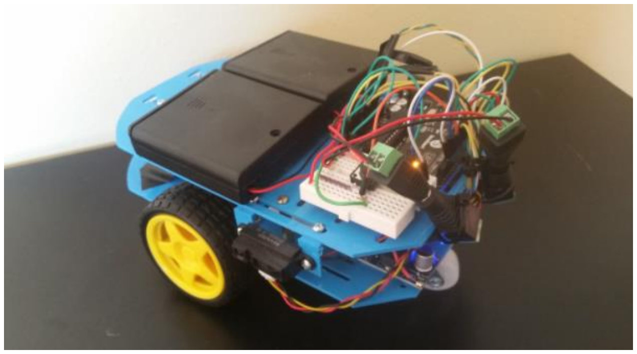
1. Do the example 2.23 of the book, “Introduction to Robotics by Saeed B. Niku” [1] with 3 DOF and then plot the result.
2. As per the example 2.28 mentioned in the book (page number 88), confirm results of the inverse kinematics by using the given equation.

Lab Experiment # 05

Objective:
Determine the translational changes during the simulation of moving robot in the robotics toolbox

Theory:
A mobile robot is a machine controlled by software that uses sensors and other technology to identify its surroundings and move around its environment. In other words, robot having capability to change its location is known as moving robot [2].

.



Classifications of mobile robots:

A simple moving robot is comprised of following elements:

1. Manipulator:
Just like the human arm, the robot consists of what is called a manipulator having several joints and links.

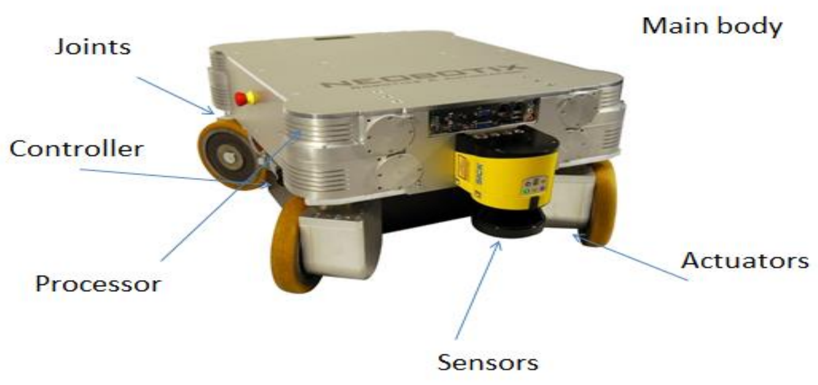
2. End effector:
The base of the manipulator is fixed to base support and at its other free end, the End effector is attached.

3. The Locomotion Device:
For the robot the power for the movement (locomotion) is provided by the motors. The motors used for providing locomotion in robots are of three types depending on the source of energy: Electric, Hydraulic or Pneumatic.

4.The Controller:
The digital computer (both the hardware and the software) acts as a controller to the robot. The controller functions in a manner analogous to the human brain. With the help of this controller, the robot is able to carry out the assigned tasks. The controller directs and controls the movement of the Manipulator and the End effector. In other words, the controller controls the robot.

5.The Sensors:
Without the data supplied by the sense organs, the controller (the computer) of the robot cannot do any meaningful task, if the robot is not with a component analogous to the sense organs of the human body. Thus,

the fifth and the most important component of the robot is the set of sensors. Sensors are nothing but measuring instruments which measures quantities such as position, velocity, force, torque, proximity, temperature, etc.



Examples of different environment mobile robots include:

- Polar robots that are designed to traverse icy, uneven environments.
- Aerial robots, also known as unmanned aerial vehicles (UAVs) or drones, which fly through the air.
- Land or home robots, or unmanned ground vehicles (UGVs), that navigate on dry land or within houses.
- Underwater robots or autonomous underwater vehicles (AUVs) that can direct themselves and travel through water.
- Delivery and transportation mobile robots that are designed to move materials and supplies around a work environment.

Types of mobile robots according to their drives:

1. **Differential Drive/Differential wheel:**

It is a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels and hence does not require an additional steering motion.

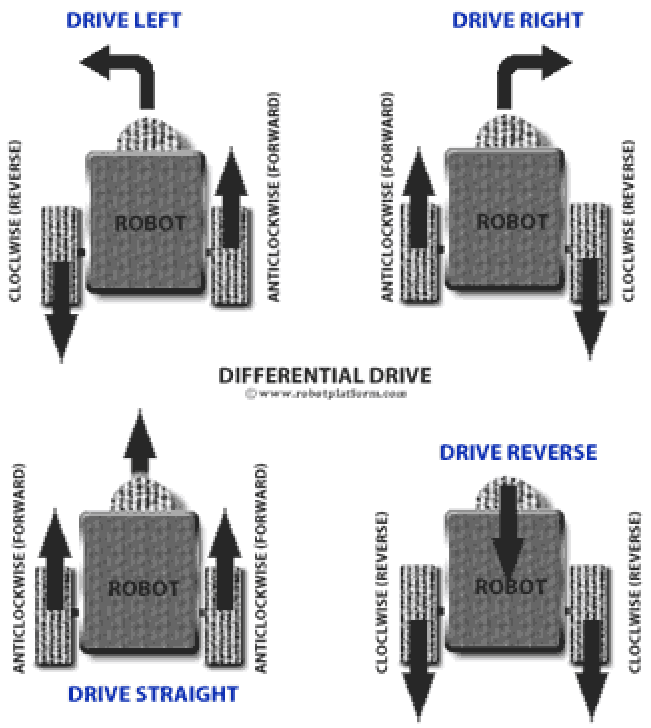
To balance the robot, additional wheels or casters may be added. The concept is simple; Velocity difference between two motors drive the robot in any required path and direction. Hence the name "**Differential**" drive. Differential wheeled robot can have two independently driven wheels fixed on a common horizontal axis or three wheels where two independently driven wheels and a roller ball or a castor attached to maintain equilibrium. To balance the robot, additional wheels or casters may be added.

There are three fundamental cases which can happen in a differential wheeled robot:

I. If the angular velocities are identical in terms of both values and direction, i.e. if both the wheels are driven at the same speed and same direction (either clockwise or anticlockwise) then the robot tends to spin around its vertical axis. This complete turn capability is one of the greatest advantages of a differentially driven robot (a.k.a zero radius turn).

II. If the angular velocities are identical in terms of values and opposite in direction, i.e. if both the wheels are driven in the same speed but in the opposite direction (One clockwise and other anticlockwise) then the robot is more likely to follow a linear path, either forward or backward based on the motors spin.

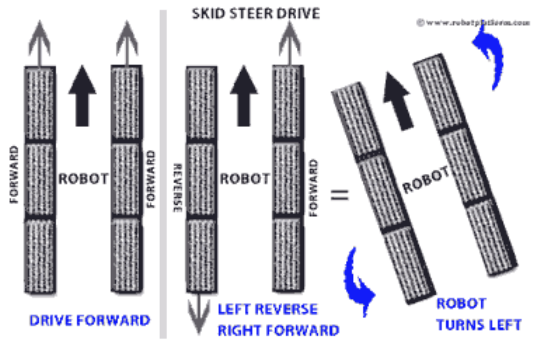
III. If the angular velocities are different in terms of values (same or different direction), i.e. if the wheels are driven at different speeds in the same direction or opposite direction, then the robot makes a curve motion. Lastly, if one of the wheels rotate and the other stays still then the robot almost makes a 90° turn. Manipulating the drive speed and direction can give some interesting drive paths.



One of the major disadvantages of this control is that the robot does not drive as expected. It neither drives along a straight line nor turn exactly at expected angles, especially when we use DC motors. This is due to difference in the number of rotations of each wheel in a given amount of time.

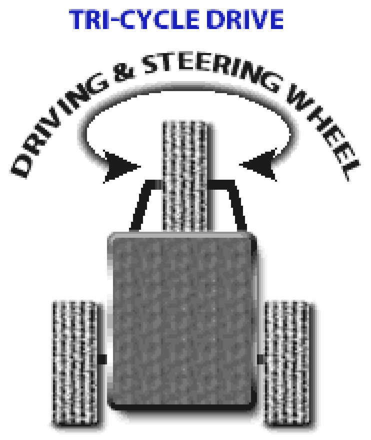
2. **Skid steering**

It is another driving mechanism implemented on vehicles with either tracks or wheels which uses differential drive concept. Most common Skid steered vehicles are tracked tanks and bulldozers. This method engages one side of the tracks or wheels and turning is done by generating differential velocity at opposite side of a vehicle as the wheels or tracks in the vehicle are non-steerable.



3. **Tricycle Drive**:

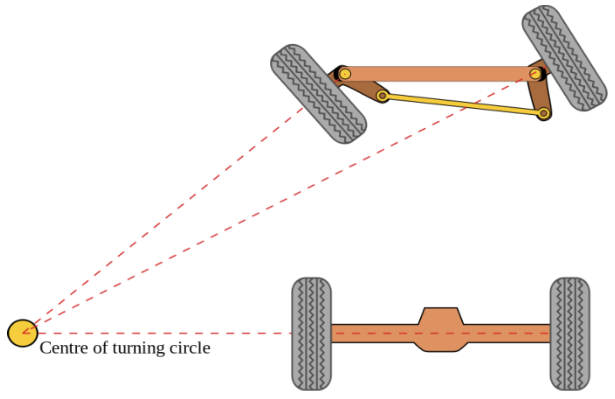
Tricycle robot is designed with a front steering wheel controlled by a motor. The two rear wheels are attached to a common axle driven by a single motor with two degrees of freedom (either forwards or reverse). The disadvantage is that they cannot spin like a differential drive robot and does not have a 90° turn due to their limited radius of curvature.



4. **Ackermann Steering:**

One of the most common configurations found in cars is Ackerman steering which mechanically coordinates the angle of two front wheels which are fixed on a common axle used for steering and two rear wheels fixed on another axle for driving.

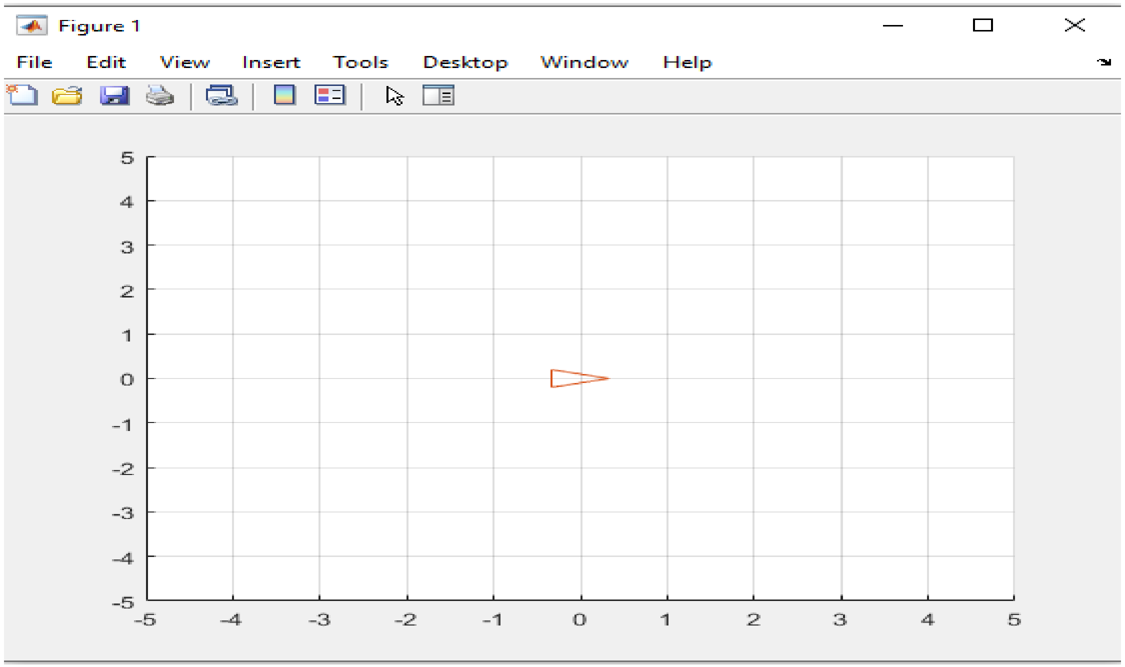
The advantage in this design is increased control, better stability and maneuverability on road, less slippage and less power consumption.



Procedure:
We are using MATLAB 2021(b) alongwith ‘Peter Corke Robotics Toolbox’ to perform the simulations of moving robot (triangle). To perform this task we will provide translation and rotation to the robot and will observe the result.

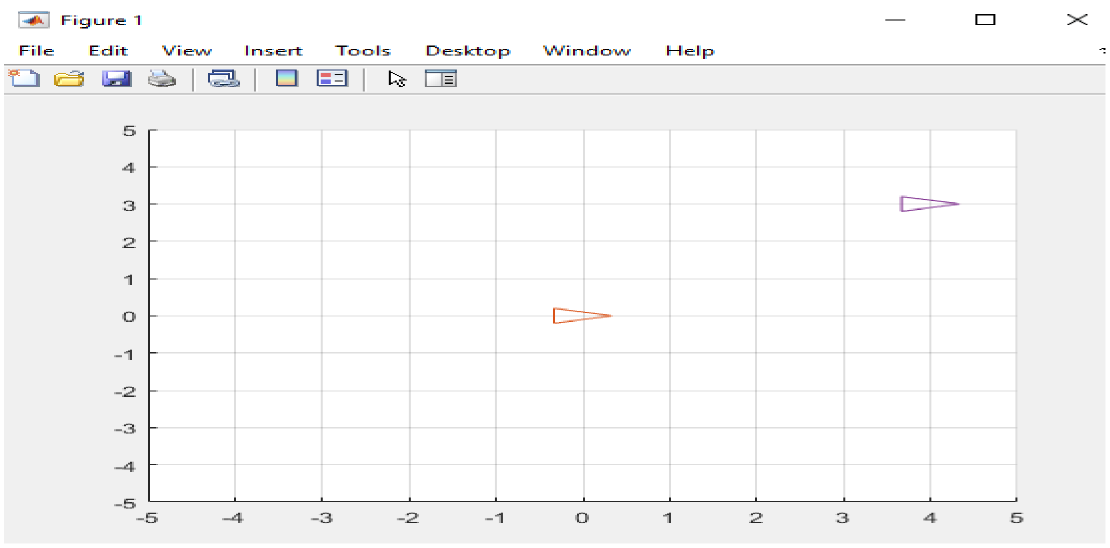
Initially the robot is resting at origin

```
axis([-5 5 -5 5]);
grid on
x0=[0;0;0];
plot_vehicle(x0, 'r', 'retain');
```



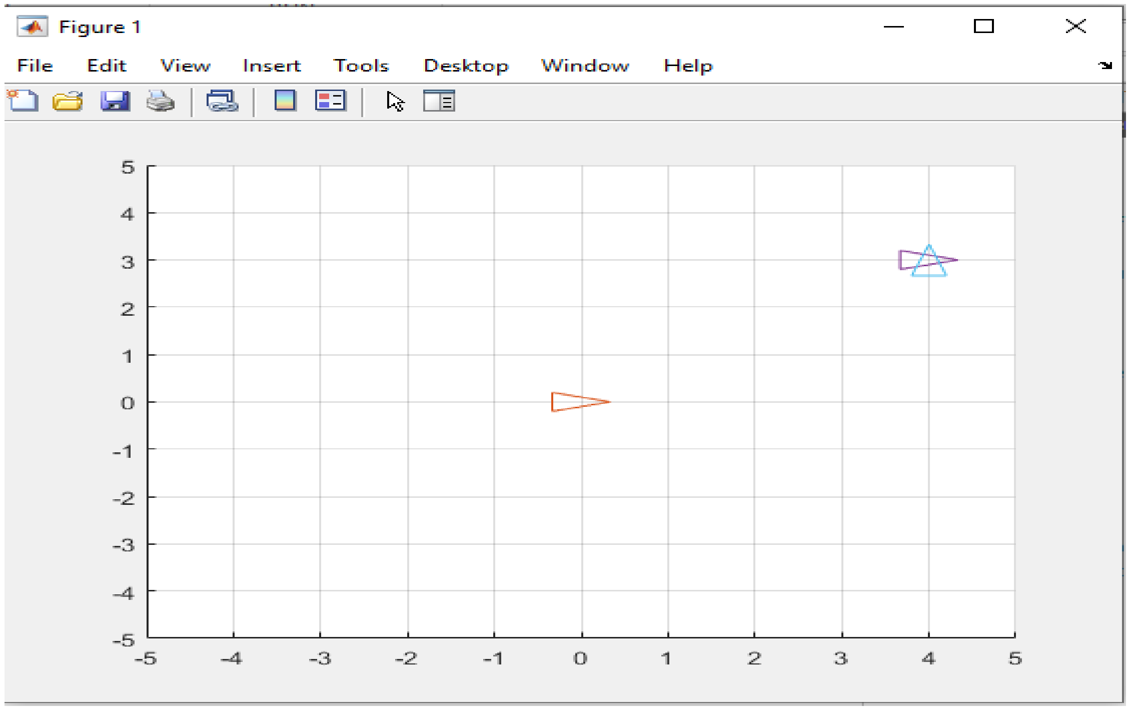
Performing translation on x-axis and y-axis respectively:

```
F = transl(x0(1)+4,  x0(2)+3, 0);
x = F(1,1);
A=F(1,4);
B=F(2,4);
Y=acos(x);
x1p=[A;B;Y];
plot_vehicle(x1p, 'r', 'retain');
```



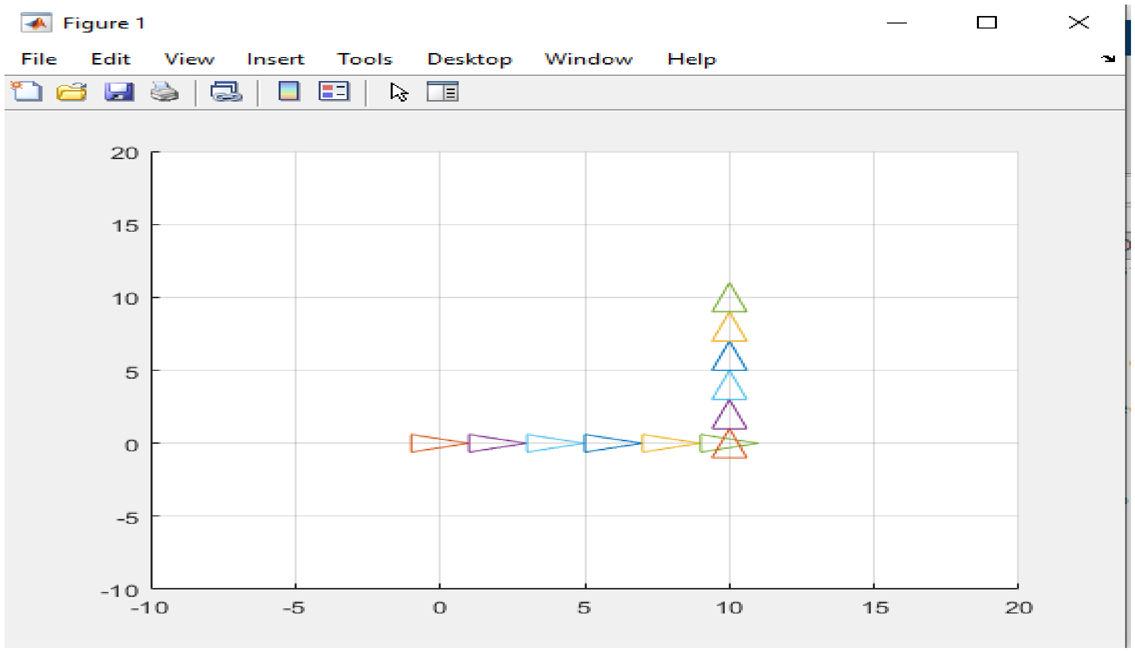
Upon rotation of 90degrees:

```
FNEW=F*trotz(pi/2);
x = FNEW(1,1);
A=FNEW(1,4);
B=FNEW(2,4);
C=FNEW(3,4);
Y=acos(x);
x2p=[A;B;Y];
plot_vehicle(x2p, 'r', 'retain');
```



Code:

```
clear all; %Clear removes all variables from the current workspace, releasing them from system memory.
close all; %Close one or more figures.
clc       %Clears all the text from the Command Window

axis([-10 20 -10 20]);  %Axis([xmin xmax ymin ymax]) sets the limits for the x- and y-axis of the current axes
grid on                %Grid on displays the major grid lines for the current axes
hold on                %Retains the current plot and certain axes properties so that subsequent graphing commands
add to the existing graph

for i=2:2:10           %For loop to repeat specified number of times.
x0=[i;0;0];            % The robot will travel along x-axis
plot_vehicle(x0, 'r', 'retain'); %PLOT_VEHICLE(X,OPTIONS) draws an oriented triangle to represent the pose
end

P = [x0(1); x0(2) ; 0;1]; %extract single value out of a matrix using vectors
F = transl(x0(1),  x0(2), 0); %T = transl(x, y, z) is an SE(3) homogeneous transform (4x4) representing a pure
translation of x, y and z.


FNEW=F*trotz(pi/2)        %T = trotz(theta) is a homogeneous transformation (4x4) representing a rotation of
theta radians about the z-axis
x = FNEW(1,1)             %Indexing into a matrix is a means of selecting a subset of elements from the matrix, x
will conatin an angle
A=FNEW(1,4)               % A will contain the position of x-axis
B=FNEW(2,4)               %B will contain the position of y-axis
C=FNEW(3,4)               %C will contain the position of z-axis
Y=acos(x)               %Converting angle into radians

x1p=[A;B;Y]               % Forming the new vector point that has position in x and y axis and angle at z-axis
plot_vehicle(x1p, 'r', 'retain'); %PLOT_VEHICLE(X,OPTIONS) draws an oriented triangle to a new position
with respect to its angle

for j=B:2:10              %For loop to repeat specified number of times.
   x2=[A;B+j;Y]           % The robot will travel along y-axis

plot_vehicle(x2, 'r', 'retain'); %PLOT_VEHICLE(X,OPTIONS) draws an oriented triangle to a new position
without changing the angle
end
```



Tasks:

1. Write a simple code to bring the moving robot back to its origion from (5,5,0)

2. Consider result of exercise 1 as the initial position of the robot, write a simple code to perform translation of (x_0+dx)= 5 and (y_0+dy)=3 , after reaching the final destination robot should rotate at 180 Degrees.

3. Perform a motion forming a triangular shape having same origin and destination and displacement of 20 steps atleast.

Lab Experiment # 06

Objective:
Examine the concept of localization of a mobile robot and testing the movement of a robot using MATLAB based programming**.**

Theory:
A process used to estimate the location of a robot in the world is known as localization and to calculate localization we will use Kalman Filter (KF) [3].

A Kalman Filter is an algorithm that takes data inputs from multiple sources and estimates unknown variables, despite a potentially high level of signal noise. Often used in navigation and control technology, the Kalman Filter has the advantage of being able to predict unknown values more accurately than if individual predictions are made using singular methods of measurement.
It has two steps:

1. **Prediction step:** the next step state of the system is predicted given the previous measurements
2. **Update step:** the current state of the system is estimated given the measurement at that time step

The terms “prediction” and “update” are often called “propagation” and “correction,” respectively.
Basics
Here are the most important concepts you need to know:

- Kalman Filters are discrete. That is, they rely on measurement samples taken between repeated but constant periods of time.
- Kalman Filters are recursive. This means its prediction of the future relies on the state of the present (position, velocity, acceleration, etc) as well as a guess about what any controllable parts tried to do to affect the situation (such as a rudder or steering differential).
- Kalman Filters work by making a prediction of the future, getting a measurement from reality, comparing the two, moderating this difference, and adjusting its estimate with this moderated value.
- The more you understand the mathematical model of your situation, the more accurate the Kalman filter's results will be.
- If your model is completely consistent with what's actually happening, the Kalman filter's estimate will eventually converge with what's actually happening.

When you start up a Kalman Filter, these are the things it expects:

- The mathematical model of the system, represented by matrices A, B, and H.
- An initial estimate about the complete state of the system, given as a vector x.
- An initial estimate about the error of the system, given as a matrix P.
- Estimates about the general process and measurement error of the system, represented by matrices Q and R.

During each time step, you are expected to give it the following information:

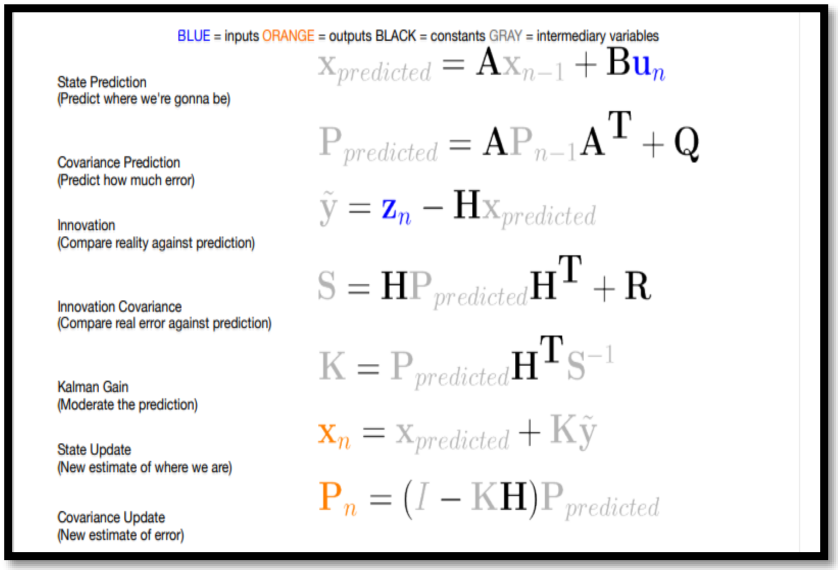
- A vector containing the most current control state (vector "u"). This is the system's guess as to what it did to affect the situation (such as steering commands).
- A vector containing the most current measurements that can be used to calculate the state (vector "z").

After the calculations, you get the following information:

- The most current estimate of the true state of the system.

- The most current estimate of the overall error of the system

The KF algorithm:
The equations are here for exposition and reference [3]. You aren't expected to understand the equations on the first read. The Kalman Filter is like a function in a programming language: it's a process of sequential equations with inputs, constants, and outputs. Here we have color-coded the filter equations to illustrate which parts are which. If you are using the Kalman Filter like a black box, you can ignore the gray intermediary variables. BLUE = inputs, ORANGE = outputs, BLACK = constants, GRAY = intermediary variables


BLUE = inputs ORANGE = outputs BLACK = constants GRAY = intermediary variables

State Prediction
(Predict where we're gonna be)

$$x_{predicted} = A x_{n-1} + B u_n$$

Covariance Prediction
(Predict how much error)

$$P_{predicted} = A P_{n-1} A^T + Q$$

Innovation
(Compare reality against prediction)

$$\tilde{y} = z_n - H x_{predicted}$$

Innovation Covariance
(Compare real error against prediction)

$$S = H P_{predicted} H^T + R$$

Kalman Gain
(Moderate the prediction)

$$K = P_{predicted} H^T S^{-1}$$

State Update
(New estimate of where we are)

$$x_n = x_{predicted} + K\tilde{y}$$

Covariance Update
(New estimate of error)

$$P_n = (I - KH) P_{predicted}$$

Inputs:
Un = Control vector. This indicates the magnitude of any control system's or user's control on the situation.
Zn = Measurement vector. This contains the real-world measurement we received in this time step.
Outputs:
Xn = Newest estimate of the current "true" state.
Pn = Newest estimate of the average error for each part of the state.
Constants:
A = State transition matrix. Basically, multiply state by this and add control factors, and you get a prediction of the state for the next time step.
B = Control matrix. This is used to define linear equations for any control factors.
H = Observation matrix. Multiply a state vector by H to translate it to a measurement vector.
Q = Estimated process error covariance.
R = Estimated measurement error covariance.

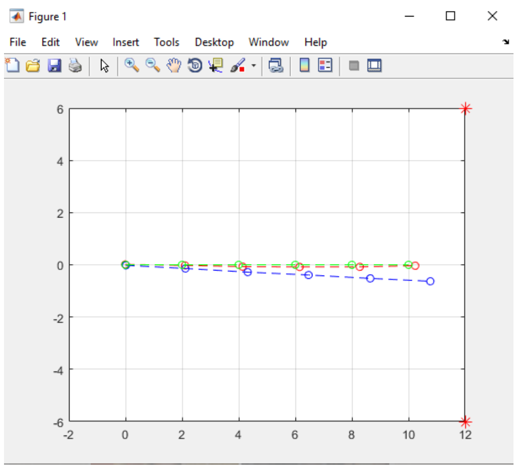
Procedure:
Following is the Matlab coding for performing the localization:

```
clc;
clear;
close all;
```

```
% Basic KF-Localisation implementation
% true value =[ x y theta]
true_val=[0 0 0;
         2 0 0;
         4 0 0;
         6 0 0;
         8 0 0;
        10 0 0];

odom=[  0.02    -0.02         0;
       2.1     -0.12     -0.024;
       2.2     -0.14     -0.026;
       2.15    -0.11     -0.023;
       2.18    -0.13     -0.026;
       2.12    -0.11     -0.022];  % Odometric data (dx, dy, dtheta)

% range sensor data (range 1, bearing 1,  range 2, bearing 2)
ranges=[    13.42       0.463       13.42       -0.463;
            11.55       0.528       11.57       -0.526;
            9.98        0.643       9.99        -0.6334;
            8.46        0.779       8.45        -0.778;
            7.05        0.977       7.04        -0.978;
            6.33        1.251       6.41        -1.253 ]; %13.2;0.46;13.2;-0.46


sig_t1=0.2;sig_th1=6*pi/180;sig_fx=0.6;sig_fy=0.5;   % variance for initialiasating P
sig_t=0.08;sig_th=5*pi/180;        % variance for Q
sig_row=0.1;sig_alpha=2*pi/180;    % variance for R
dx=0;dy=0;dth=0;
HA=0; HB=0; HC=0; HD=0; HE=0; HF=0;

X=[0;0;0];   %initialising X
P=[(sig_t1)^2  0.1         0.1;
    0.1       (sig_t1)^2   0.1;
    0.1        0.1       (sig_th1)^2];


A=[1 0 -dy;
   0 1 dx ;
   0 0 1  ]; % initialising A

Q=[(sig_t)^2  0.001     0.001       ;
    0.001    (sig_t)^2  0.001       ;
    0.001     0.001     (sig_th)^2  ]; %initialising Q


R=[(sig_row)^2  0;
```

```
      0        (sig_alpha)^2]; %initialising R
I=eye(3,3);

odomtx=0;
odomty=0;
odomt=[odomtx odomty];
XT=[X(1) X(2)];
% Plot
figure(1);
plot(XT(:,1),XT(:,2),'r--o'); %KF pose
hold on;
grid on;
hold on;
plot(odomt(:,1),odomt(:,2),'b--o'); %odometry pose
hold on;
plot(true_val(1,1),true_val(1,2),'g--o'); %actual pose
hold on;
plot(12,6,'g*','MarkerSize',10);%actual feature1 plot
hold on;
plot(12,-6,'g*','MarkerSize',10);%actual feature2 plot
hold on;

% KF localisation loop
for i=1:size(odom,1)
    dx = odom(i,1);
    dy = odom(i,2);
    dth = odom(i,3);
    odomtx=odomtx+dx;
    odomty=odomty+dy;
    odomt=[odomt;[odomtx odomty]];

    % Prediction
    A(1,3) = -dy;
    A(2,3) = dx;

    XP = X + [dx;dy;dth];
    PP = A * P * A' + Q;

    % Correction
    range1=ranges(i,1);
    bearing1=ranges(i,2);
    range2=ranges(i,3);
    bearing2=ranges(i,4);

    for j=1:2

        if j==1
```

```
            fx=12;
            fy=6;
            zn=[range1;bearing1];
        else
            fx=12;
            fy=-6;
            zn=[range2;bearing2];
        end

        dist=sqrt((XP(1)-fx)^2+(XP(2)-fy)^2);
        angp=atan2((fy-XP(2)),(fx-XP(1)))-XP(3);
        zp=[dist;angp];
        HA= (XP(1)-fx)/dist;
        HB= (XP(2)-fy)/dist;
        HD= (fy-XP(2))/dist^2;
        HE= (XP(1)-fx)/dist^2;

        H=[HA HB  0 ;
           HD HE -1 ];



        y= zn - zp;
        s= H * PP * H' + R;
        k= PP * H' * inv(s);
        XP= XP + k *y;
        PP= (I - k * H) * PP;
    end

    X = XP;
    P = PP;

    XT=[XT;[X(1) X(2)]];
    % plots
    plot(XT(:,1),XT(:,2),'r--o');   %KF pose
    hold on;
    plot(odomt(:,1),odomt(:,2),'b--o');   %odometry pose
    hold on;
    plot(true_val(1:i,1),true_val(1:i,2),'g--o'); %true pose
    hold on;
    plot(12,6,'r*','MarkerSize',10); %KF feature 1 location
    hold on;
    plot(12,-6,'r*','MarkerSize',10); %KF feature 2 location
    hold on;
end
asd=0;
```



Tasks:

1. Run the code once again by using following odometric values and provide your result.

```
odom=[  0.02     -0.02       0;
        1.9      -0        -0.024;
        2.1      -0.17     -0.026;
        2.3      -0.15     -0.023;
        2.25     -0.09     -0.026;
        2.12     -0.10     -0.022 ];
```

2. Run the code by making your new range values for given feature locations (13, 5 and 13,-5) and provide your result.

```
ranges=[    ];
```

Lab Experiment # 07

Objective:
Derive the concept of simultaneous localization and mapping of a mobile robot and testing the working of a robot using MATLAB based programming**.**

Theory:

The mobile robots performing any task, must acquire data from the environment. This data may be used to measure the motion and location of rover. The level or nature of data varies with application. For example, a simple toy type robot just detects the obstacle at front of it from suitable range, a line following robot detects line on his path and try to follow that line. However, a mobile robot being used in a warehouse continue scan the environment surrounded it and saves that data for further processing like map building and path planning.

The process of collecting the information of environment for development of map and self-localize in measured map is called Simultaneous Localization and Mapping or SLAM. The type of map varies with respect to the environment. The environment would be indoor, outdoor. It could be under water, surface of water, floor of the ground or aerial.

For environments with locally distinguishable features, landmark-based maps have been extensively used. If we assume that the position of the robot is always known, it simply remains to maintain an estimate about the positions of the individual landmarks over time [2].

A mobile robot scanning the envirnment and generate landmark based map

Mathematical Representation of SLAM

Let a mobile robot is roaming in an unknown environment. It starts from an unknown location$\boldsymbol{x_0}$. The uncertainty in the motion making it more complicate to determine the pose of mobile robot. The robot senses the environment with some induced noise. The process of developing the map of the environment and determining the poses is called Simultaneous Localization and Mapping (SLAM).

At time t the location of the robot is $\boldsymbol{x_t}$. The path of the robot can be expresses as

$$\boldsymbol{X_T} = \{\boldsymbol{x_0} + \boldsymbol{x_1} + \boldsymbol{x_2} + \boldsymbol{x_3} + \cdots \boldsymbol{x_T}\} \qquad ...... 1$$

T is terminal time and may be infinite. Relative information between two positions can be determined by odometry. Let $\boldsymbol{u_t}$ is the odometry caricaturists between time t-1 and t, so the motion of the robot can be express as

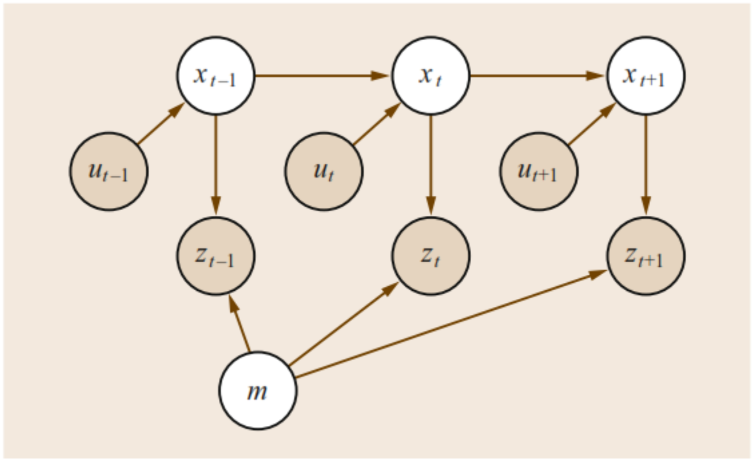
$$\boldsymbol{U_T} = \{\boldsymbol{u_0} + \boldsymbol{u_1} + \boldsymbol{u_2} + \boldsymbol{u_3} + \cdots \boldsymbol{u_T}\} \qquad ...... 2$$

If the true map of the environment is *m*, presenting the locations of landmarks, objects and surfaces. The robot establishes the information between self-location and features in *m*. The sequence of measurements is

$$\boldsymbol{Z_T} = \{\boldsymbol{z_0} + \boldsymbol{z_1} + \boldsymbol{z_2} + \boldsymbol{z_3} + \cdots \boldsymbol{z_T}\} \qquad \text{...... 3}$$

Now it is established that the recovering the model of *m* using measurement and sensor data is called SLAM.

The graphical model of the SLAM is presented in the below figure. In this figure, the measurement z_{t-1} is measured by the information of motion u_{t-1} and location x_{t-1} and scanned information from *m*. The information x_{t-1} is added to x_t with new motion information and generates zt, and the process continues till x_n



Graphical model of SLAM

Literary there are two major forms of SLAM. One is full SLAM and other is online SLAM.

First one is estimating the posterior over the entire path together

$$\boldsymbol{p(X_T, m|Z_T U_T)} \qquad \text{...... 4}$$

The second type seeks to recover the present location of robot.

$$\boldsymbol{p(x_t, m|Z_t U_t)} \qquad \text{...... 5}$$

The problem of SLAM categorizes along with several dimensions.

Kalman Filters base SLAM

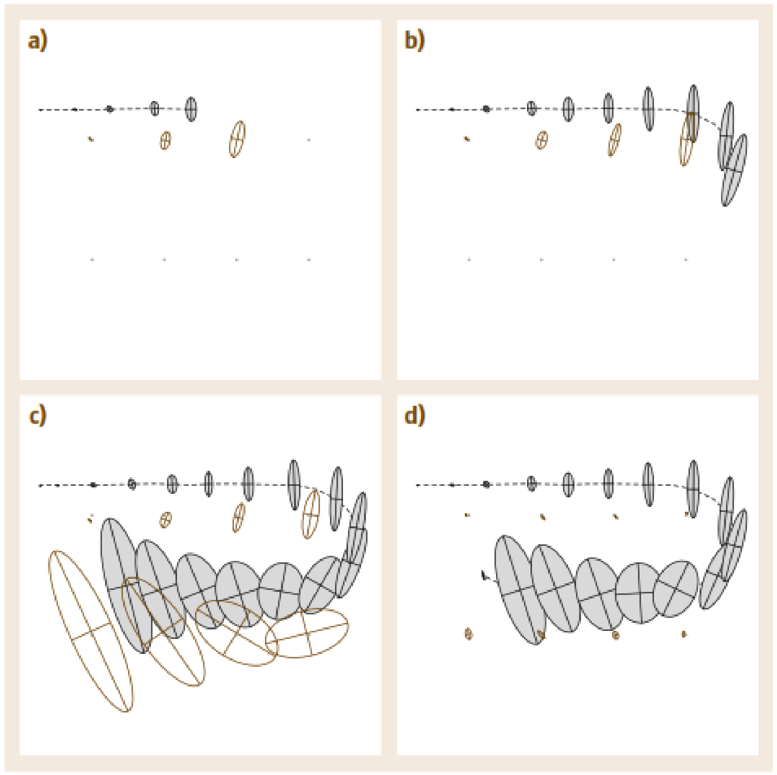
Earliest and most influential algorithm of SLAM is Extended Kalman Filter EKF SLAM. It was introduced by Randall Smith in 1987. A group of researchers in 1990 proposed an experimental setup using this algorithm. They propose the use of a single state vector to estimate the locations of the robot and set of the features. A covariance matrix represents the uncertainty, including the correlations. As the rover moves the state vector and the covariance matrix are updated using the EKF. New states are added on the observation of new feature, so size of system covariance matrix grows. A multivariate Gaussian represents the EKF algorithm [3].

$$\boldsymbol{p(x_t, m|Z_t U_t)} = \boldsymbol{\mathcal{N}(\mu_t, \textstyle\sum_t)} \qquad \text{...... 6}$$

The vector $\boldsymbol{\mu_t}$ holds the best estimate of rover's current location $\boldsymbol{x_t}$ and feature's location. $\sum_t$ Is the covariance of the rover and its expected error in guess $\boldsymbol{\mu_t}$.

The practical application of EKF SLAM is presented in the below figure. Robot starts to navigate from a point that serves as an origin. As the rover moves, pose and uncertainty increases. The size of the ellipse presents this increment in data. The rover also senses landmarks in its range and map them. The fixed measurement uncertainty with the growing POS uncertainty is also added in estimated map.

The dotted line in figure represents the path of the robot. The shaded ellipse presents the estimated own position. Small dots represent landmarks and their estimated location is white ellipse. In figure a, b and c uncertainty of the robot position is increased. In figure d robot senses first land mark second time, so the uncertainty reduced



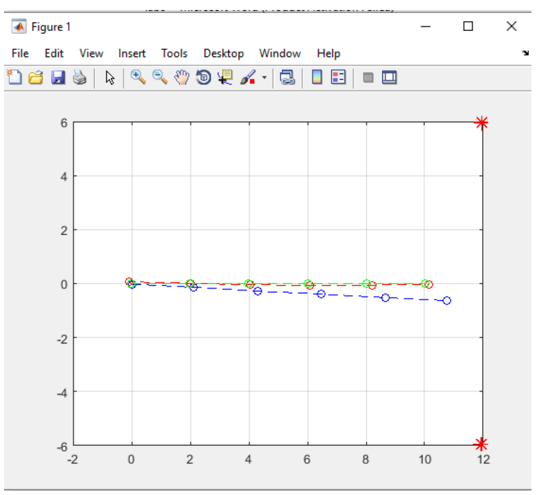
Online SLAM with EKF(a, b and c) uncertainty of the robot position is increased. (d) Robot senses first land mark second time, so the uncertainty reduced.

The quadratic nature of the covariance matrix is a key concern of this algorithm. There are several extensions have been proposed. They considered a special form of matrices for their algorithm and a compressed filter that reduce the computation requirements. A researcher's claim that Decoupled Stochastic Mapping (DSM) is an efficient approach for large scale mapping and localization. They divided the environment or map with small overlapped maps. It reduced the burden of computing.

Procedure:

Following is the KF based Matlab implementation of the SLAM.

```
clc;
clear;
close all;


% Basic EKF-SLAM implementation
true_val=[0 0 0;
          2 0 0;
          4 0 0;
          6 0 0;
          8 0 0;
         10 0 0];

odom=[  0.02    -0.02         0;
        2.1     -0.12     -0.024;
        2.2     -0.14     -0.026;
        2.15    -0.11     -0.023;
        2.18    -0.13     -0.026;
        2.12    -0.11     -0.022];  % Odometric data (dx, dy, dtheta)

% range sensor data (range 1, bearing 1,  range 2, bearing 2)
ranges=[    13.42       0.463       13.42       -0.463;
            11.55       0.528       11.57       -0.526;
            9.98        0.643       9.99        -0.6334;
            8.46        0.779       8.45        -0.778;
            7.05        0.977       7.04        -0.978;
            6.33        1.251       6.41        -1.253 ]; %13.2;0.46;13.2;-0.46


sig_t1=0.2;sig_th1=6*pi/180;sig_fx=0.6;sig_fy=0.5;   % variance for initialiasating P
sig_t=0.08;sig_th=5*pi/180;         % variance for Q
sig_row=0.1;sig_alpha=2*pi/180;     % variance for R
dx=0;dy=0;dth=0;
HA=0; HB=0; HC=0; HD=0; HE=0; HF=0;

X=[0;0;0;11.95;5.96;11.93;-5.96];   %initialising X
P=[(sig_t1)^2  0.1         0.1          0.1         0.1           0.1        0.1;
    0.1       (sig_t1)^2   0.1          0.1         0.1           0.1        0.1;
    0.1        0.1        (sig_th1)^2   0.1         0.1           0.1        0.1;
    0.1        0.1         0.1         (sig_fx)^2   0.1           0.1        0.1;
    0.1        0.1         0.1          0.1        (sig_fy)^2     0.1        0.1;
    0.1        0.1         0.1          0.1         0.1          (sig_fx)^2  0.1;
    0.1        0.1         0.1          0.1         0.1           0.1       (sig_fy)^2];

A=[1 0 -dy 0 0 0 0;
   0 1 dx  0 0 0 0;
   0 0 1   0 0 0 0;
   0 0 0   0 0 0 0;
   0 0 0   0 0 0 0;
   0 0 0   0 0 0 0;
   0 0 0   0 0 0 0]; % initialising A

Q=[(sig_t)^2  0.001     0.001       0.001        0.001          0.001        0.001;
    0.001    (sig_t)^2  0.001       0.001        0.001          0.001        0.001;
    0.001     0.001     (sig_th)^2  0.001        0.001          0.001        0.001;
    0.001     0.001      0.001      0.001        0.001           0.001        0.001;
    0.001     0.001      0.001      0.001        0.001           0.001        0.001;
    0.001     0.001      0.001      0.001        0.001           0.001        0.001;
    0.001     0.001      0.001      0.001        0.001           0.001        0.001];
%initialising Q


R=[(sig_row)^2  0;
      0        (sig_alpha)^2]; %initialising R
I=eye(7,7);

odomtx=0;
odomty=0;
odomt=[odomtx odomty];
XT=[X(1) X(2)];
% Plot
figure(1);
plot(XT(:,1),XT(:,2),'r--o'); %KF pose
hold on;
grid on;
hold on;
plot(odomt(:,1),odomt(:,2),'b--o'); %odometry pose
hold on;
plot(true_val(1,1),true_val(1,2),'g--o'); %actual pose
hold on;
plot(X(4),X(5),'g*','MarkerSize',10);%actual feature1 plot
hold on;
plot(X(6),X(7),'g*','MarkerSize',10);%actual feature2 plot
hold on;

% KF SLAM loop
for i=1:size(odom,1)
    dx = odom(i,1);
    dy = odom(i,2);
    dth = odom(i,3);
    odomtx=odomtx+dx;
    odomty=odomty+dy;
    odomt=[odomt;[odomtx odomty]];

    % Prediction
    A(1,3) = -dy;
    A(2,3) = dx;

    XP = X + [dx;dy;dth;0;0;0;0];
    PP = A * P * A' + Q;

    % Correction
    range1=ranges(i,1);
    bearing1=ranges(i,2);
    range2=ranges(i,3);
    bearing2=ranges(i,4);

    for j=1:2

        if j==1
            fx=XP(4);
            fy=XP(5);
            zn=[range1;bearing1];
        else
            fx=XP(6);
            fy=XP(7);
            zn=[range2;bearing2];
        end

        dist=sqrt((XP(1)-fx)^2+(XP(2)-fy)^2);
        angp=atan2((fy-XP(2)),(fx-XP(1)))-XP(3);
        zp=[dist;angp];
        HA= (XP(1)-fx)/dist;
        HB= (XP(2)-fy)/dist;
        HD= (fy-XP(2))/dist^2;
        HE= (XP(1)-fx)/dist^2;

        if j==1
            H=[HA HB  0 -HA -HB 0 0;
               HD HE -1 -HD -HE 0 0];
        else
            H=[HA HB  0 0 0 -HA -HB;
               HD HE -1 0 0 -HD -HE];
        end

        y= zn - zp;
        s= H * PP * H' + R;
        k= PP * H' * inv(s);
        XP= XP + k *y;
        PP= (I - k * H) * PP;
    end

    X = XP;
    P = PP;

    XT=[XT;[X(1) X(2)]];
    % plots
    plot(XT(:,1),XT(:,2),'r--o');   %KF pose
    hold on;
    plot(odomt(:,1),odomt(:,2),'b--o');   %odometry pose
    hold on;
    plot(true_val(1:i,1),true_val(1:i,2),'g--o'); %true pose
    hold on;
    plot(X(4),X(5),'r*','MarkerSize',10); %KF feature 1 location
    hold on;
    plot(X(6),X(7),'r*','MarkerSize',10); %KF feature 2 location
    hold on;
end
asd=0;
% real scanner's plot
%
% lidar spec
% min_ang1=-2.35619449615;max_ang1 =2.35619449615;ang_step1 =0.00436332309619; % Hokuyo
%
% reading horizontal scan
% Scan = readtable('E:\lidar_short.xlsx','ReadVariableNames',1);%horizontal
% ranges = table2array(Scan(:,12:359));%12:1080
% ranges = table2array(Scan(:,12:1080));
% time = table2array(Scan(:,1));
% time= (time(:,1)- time(1,1))/1000000;          %s1 timestamp
% s1_ranges = [time,ranges];
%
s1_cartcheck=scan_polar_to_cartesian_index(s1_ranges(1,2:end),min_ang1,ang_step1,1,1,1);
% figure(100);
% plot(s1_cartcheck(:,1),s1_cartcheck(:,2),'g.');
% hold on;
% grid on;
% hold on;
% %
```



Tasks:

1. Run the code once again by using following odometric values and provide your result in the workbook.

```
odom=[  0.02    -0.02      0;
        1.9     -0        -0.024;
        2.1     -0.17     -0.026;
        2.3     -0.15     -0.023;
        2.25    -0.09     -0.026;
        2.12    -0.10     -0.022 ];
```

2. Run the code by making your new range values for given feature locations (13, 5 and 13,-5) and provide your result in the workbook.

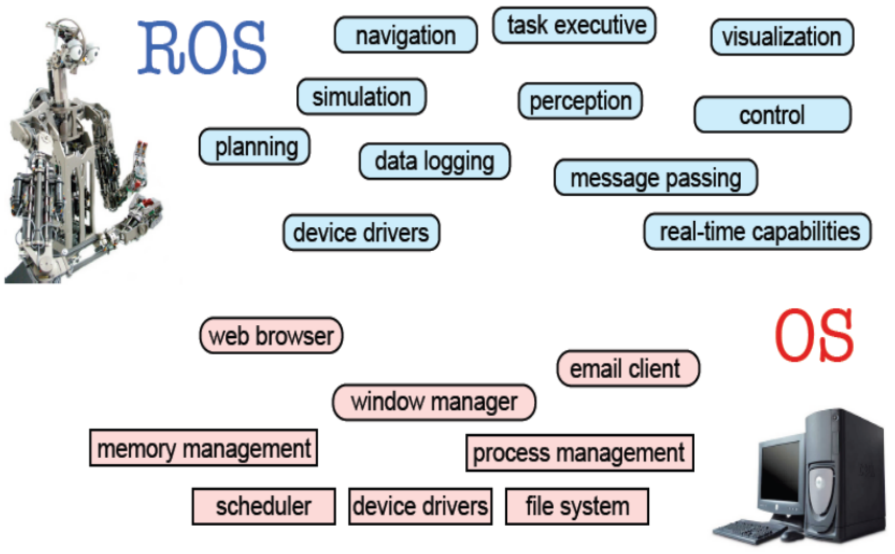
```
ranges=[    ];
```

Lab Experiment # 08

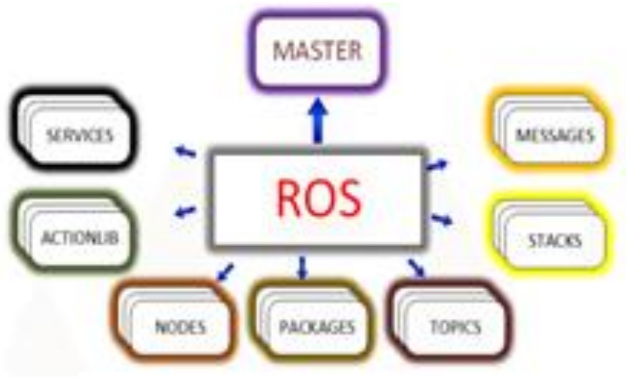
Objective:
Analyze the functioning of Robot Operating System (ROS)

Equipment:
Linux operated PCs having ROS installed

Theory:
Introduction to ROS
ROS is an open-source robot operating system, it is a set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms [4].



ROS Core Concept:



Main Features:
The Robot Operating System ROS is not an actual operating system, but a framework and set of tools that provide functionality of an operating system on a heterogeneous computer cluster. Its usefulness is not limited to robots, but the majority of tools provided are focused on working with peripheral hardware.

Installation and Running ROS

1. Make sure linux is running
2. Install ROS. Make sure the version you install is Kinetic, as that is the most recent LTS release.

3. Set up the environment:

```
$ source /opt/ros/kinetic/setup.bash
```

4. You will have to do this every time you start.
5. Create a catkin workspace. Catkin is a program like 'make' that is used by ROS.
6. ROS workspace is created to manage packages (software libraries) to execute required task.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

7. Initially, the workspace will contain only the toplevel CMakeLists.txt
8. catkin_make command builds the workspace and all the packages within it

```
cd ~/catkin_ws
catkin_make
```

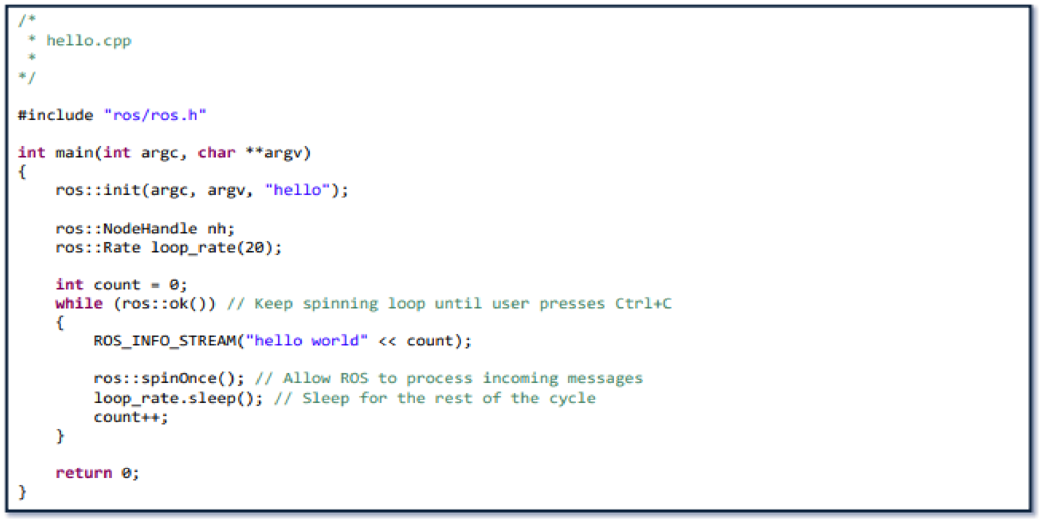
9. catkin_create_pkg creates a new package with the specified dependencies

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

10. For example, create a first_pkg package:

```
$ catkin_create_pkg first_pkg std_msgs rospy roscpp
```

ROS Catkin Workspace for Packages



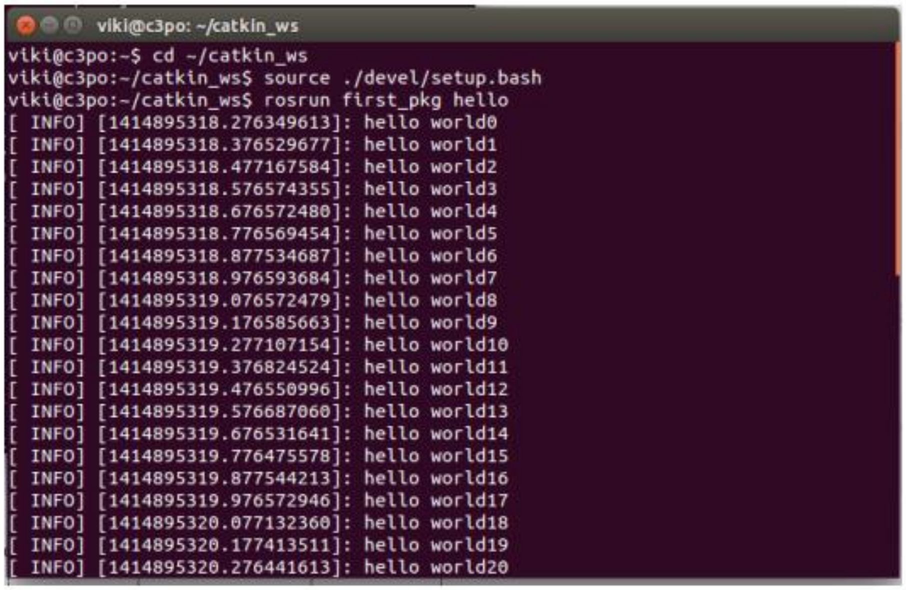
```
/*
 * hello.cpp
 *
*/

#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");

    ros::NodeHandle nh;
    ros::Rate loop_rate(20);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);

        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
```

11. To compile the cpp file, need some C++ project editor like Eclipse, install it and press Ctrl-B
12. Source the workspace's setup.sh file after calling catkin_make:
13. To build the package in the terminal call catkin_make.

```
$ cd ~/catkin_ws
$ source ./devel/setup.bash
```

14. Can also add this line to .bashrc startup file
15. Now write command rosrun to execute your node:

```
$ rosrun first_pkg hello
```



```
viki@c3po: ~/catkin_ws
viki@c3po:~$ cd ~/catkin_ws
viki@c3po:~/catkin_ws$ source ./devel/setup.bash
viki@c3po:~/catkin_ws$ rosrun first_pkg hello
[ INFO] [1414895318.276349613]: hello world0
[ INFO] [1414895318.376529677]: hello world1
[ INFO] [1414895318.477167584]: hello world2
[ INFO] [1414895318.576574355]: hello world3
[ INFO] [1414895318.676572480]: hello world4
[ INFO] [1414895318.776569454]: hello world5
[ INFO] [1414895318.877534687]: hello world6
[ INFO] [1414895318.976593684]: hello world7
[ INFO] [1414895319.076572479]: hello world8
[ INFO] [1414895319.176585663]: hello world9
[ INFO] [1414895319.277107154]: hello world10
[ INFO] [1414895319.376824524]: hello world11
[ INFO] [1414895319.476550996]: hello world12
[ INFO] [1414895319.576687060]: hello world13
[ INFO] [1414895319.676531641]: hello world14
[ INFO] [1414895319.776475578]: hello world15
[ INFO] [1414895319.877544213]: hello world16
[ INFO] [1414895319.976572946]: hello world17
[ INFO] [1414895320.077132360]: hello world18
[ INFO] [1414895320.177413511]: hello world19
[ INFO] [1414895320.276441613]: hello world20
```

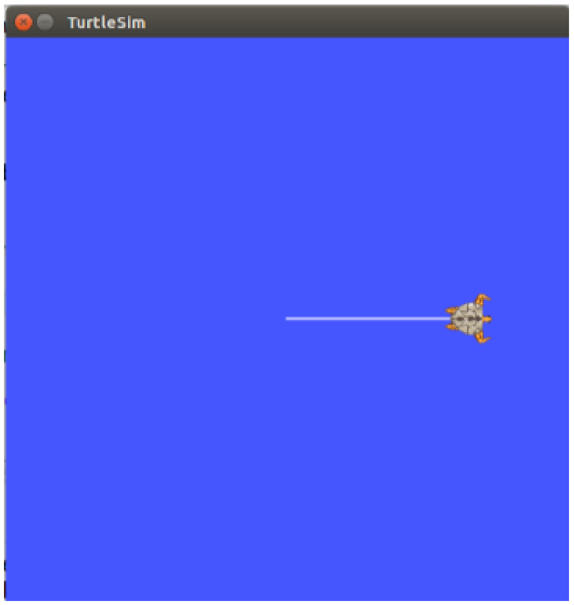
16. Try launching: roslaunch is a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server
17. It takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch
18. If you use roslaunch, you do not have to run roscore manually.
19. Add move_turtle.launch to your package:

```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
output="screen" />
</launch>
```

20. Run the launch file:

```
$ roslaunch my_turtle move_turtle.launch
```

You should see the turtle in the simulator constantly moving forward until it bumps into the wall



Task:

1. Create and build your own ROS package and run the given example in it.

Lab Experiment # 09

Objective:
Imitate the urdf exemplary model for a 2 DOF industrial robot and to simulate in ROS

Equipment:
ubuntu and ROS operated PCs

Theory:
Understanding robot modeling using URDF:
We have to create a file and write the relationship between each link and joint in the robot and save the file with the .urdf extension.
The following tags are the commonly used URDF tags to compose a URDF robot model:
link: The link tag represents a single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes size, shape, color, and can even import a 3D mesh to represent the robot link. We can also provide dynamic properties of the link such as inertial matrix and collision properties.
The syntax is as follows:

```
<link name="<name of the link>">
<inertial>...........</inertial>
  <visual> ............</visual>
  <collision>..........</collision>
 </link>
```

The Visual section represents the real link of the robot, and the area surrounding the real link is the Collision section.

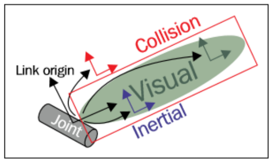


Figure 1 : Visualization of a URDF link

joint: The joint tag represents a robot joint. The joint tag supports the different types of joints such as revolute, continuous, prismatic, fixed, floating, and planar. The syntax is as follows:

```
 <joint name="<name of the joint>">
  <parent link="link1"/>
<child link="link2"/>
<calibration .... />
<dynamics damping ..../>
  <limit effort .... />
</joint>
```

A URDF joint is formed between two links; the first is called the Parent link and the second is the Child link. The following is an illustration of a joint and its link:

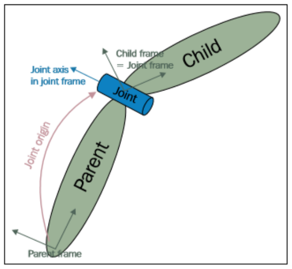


Figure 2 : Visualization of a URDF joint

robot: This tag encapsulates the entire robot model that can be represented using URDF. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot.
The syntax is as follows:

<robot name="<name of the robot>"
<link> ..... </link>
<link> ...... </link>
  <joint> .... </joint>
  <joint> ....</joint>
 </robot>

A robot model consists of connected links and joints. Here is a visualization of the robot model:

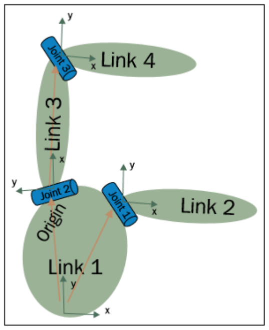


Figure 3 : Visualization of a robot model having joints and links

gazebo: This tag is used when we include the simulation parameters of the Gazebo simulator inside URDF. We can use this tag to include gazebo plugins, gazebo material properties, and so on. The following shows an example using gazebo tags:

  <gazebo reference="link_1">
<material>Gazebo/Black</material>
 </gazebo>

Creating the ROS package for the Robot description:
let's create a ROS package in the catkin workspace so that the robot model keeps using the following command:
$ catkin_create_pkg mastering_ros_robot_description_pkg roscpp tf geometry_msgs urdf rviz xacro
Before creating the urdf file for this robot, let's create three folders called urdf, meshes, and launch inside the package folder. The urdf folder can be used to keep the urdf/xacro files that we are going to create. The meshes folder keeps the meshes that we need to include in the urdf file and the launch folder keeps the ROS launch files.
Creating our first URDF model:
 After learning about URDF and its important tags, we can start some basic modeling using URDF. The first robot mechanism that we are going to design is a pan and tilt mechanism as shown in the following figure.
There are three links and two joints in this mechanism. The base link is static, in which all other links are mounted. The first joint can pan on its axis and the second link is mounted on the first link and it can tilt on its

axis. The two joints in this system are of a revolute type.

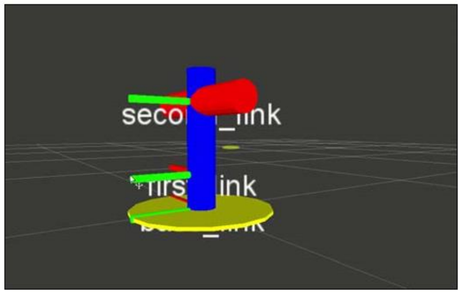


Figure 4 : Visualization of a pan and tilt mechanism in RViz

Procedure:
Let's see the URDF code of this mechanism. mastering_ros_robot_description_pkg/urdf and open pan_tilt.urdf. The code indentation in URDF is not mandatory for URDF but it keeping indentation can improve code readability:

```
<?xml version="1.0"?>
 <robot name="pan_tilt">
  <link name="base_link">
 <visual>      <geometry>
<cylinder length="0.01" radius="0.2"/>
</geometry>      <origin rpy="0 0 0" xyz="0 0 0"/>
 <material name="yellow">
  <color rgba="1 1 0 1"/>
 </material>
</visual>
 </link>
  <joint name="pan_joint" type="revolute">
 <parent link="base_link"/>
 <child link="pan_link"/>
<origin xyz="0 0 0.1"/>
<axis xyz="0 0 1" />
</joint>
  <link name="pan_link">
 <visual>      <geometry>
   <cylinder length="0.4" radius="0.04"/>
 </geometry>
<origin rpy="0 0 0" xyz="0 0 0.09"/>
<material name="red">
        <color rgba="0 0 1 1"/>
</material>
    </visual>
  </link>
  <joint name="tilt_joint" type="revolute">
   <parent link="pan_link"/>
<child link="tilt_link"/>
    <origin xyz="0 0 0.2"/>
    <axis xyz="0 1 0" />
</joint>
```

```
  <link name="tilt_link">
<visual>
<geometry>
 <cylinder length="0.4" radius="0.04"/>
</geometry>
      <origin rpy="0 1.5 0" xyz="0 0 0"/>
<material name="green">
        <color rgba="1 0 0 1"/>
</material>
</visual>
 </link>
</robot>
```

Explaining the URDF file

When we check the code, we can add a <robot> tag at the top of the description:

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

The <robot> tag defines the name of the robot that we are going to create. Here, we named the robot pan_tilt.

If we check the sections after the <robot> tag definition, we can see link and joint definitions of the pan and tilt mechanism:

```
  <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.01" radius="0.2"/>
   </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="yellow">
        <color rgba="1 1 0 1"/>
      </material>
    </visual>
  </link>
```

The preceding code snippet is the base_link definition of the pan and tilt mechanism. The <visual> tag can describe the visual appearance of the link, which is shown on the robot simulation. We can define the link geometry (cylinder, box, sphere, or mesh) and the material (color and texture) of the link using this tag:

```
  <joint name="pan_joint" type="revolute">
<parent link="base_link"/>
    <child link="pan_link"/>
   <origin xyz="0 0 0.1"/>
<axis xyz="0 0 1" />
</joint>
```

In the preceding code snippet, we define a joint with a unique name and its joint type. The joint type we used here is revolute and the parent link and child link are base_link and the pan_link respectively. The joint origin is also specified inside this tag.

Save the preceding URDF code as pan_tilt.urdf and check whether the urdf contains errors using the following command:

```
$ check_urdf pan_tilt.urdf
```

The check_urdf command will parse urdf and show an error, if any. If everything is OK, it will show an output as follows:

```
robot name is: pan_tilt
---------- Successfully Parsed XML --------------
root Link: base_link has 1 child(ren)
child(1):  pan_link
child(1):  tilt_link
```

If we want to view the structure of the robot links and joints graphically, we can use a command tool called urdf_to_graphiz:

```
$ urdf_to_graphiz pan_tilt.urdf
```

This command will generate two files: pan_tilt.gv and pan_tilt.pdf. We can view the structure of this robot using following command:

```
$ evince pan_tilt.pdf
```

We will get the following output:

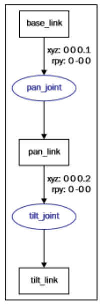


Figure 5 : Graph of joint and links in pan and tilt mechanism

Visualizing the robot 3D model in RViz

After designing URDF, we can view it on RViz. We can create a view_demo.launch launch file and put the following code into the launch folder, code/mastering_ros_robot_description_pkg/launch for the same code:

```
<launch>
<arg name="model" />
 <param name="robot_description" textfile="$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.urdf"
/>
 <param name="use_gui" value="true"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />   <node
name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find mastering_ ros_robot_description_pkg)/urdf.rviz"
required="true" />
</launch>
```

We can launch the model using the following command:

```
$ roslaunch mastering_ros_robot_description_pkg view_demo.launch
```

If everything works fine, we will get a pan and tilt mechanism in RViz

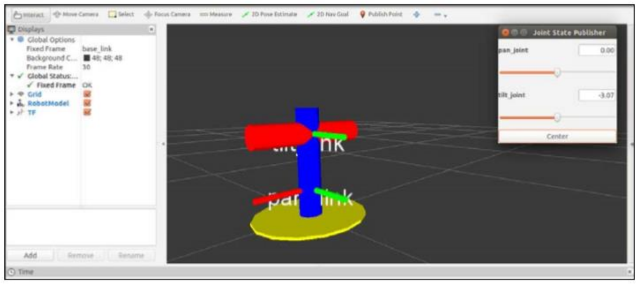


Figure 6 : Joint level of pan and tilt mechanism

Interacting with pan and tilt joints

We can see an extra GUI came along with RViz, which contains sliders to control pan joints and tilt joints. This GUI is called the Joint State Publisher node from the joint_state_publisher package:

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

We can include this node in the launch file using this statement. The limits of pan and tilt should be mentioned inside the joint tag:

<joint name="pan_joint" type="revolute">
<parent link="base_link"/>
<child link="pan_link"/>
<origin xyz="0 0 0.1"/>
<axis xyz="0 0 1" />
<limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
<dynamics damping="50" friction="1"/>
</joint>

The <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/> defines the limits of effort, velocity, and angle limits. The effort is the maximum force supported by this joint, lower and upper indicate the lower and upper limit of the joint in the radian for the revolute type joint, and meters for prismatic joints. The velocity is the maximum joint velocity.

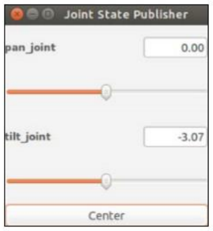


Figure 6 : Joint level of pan and tilt mechanism

The preceding screenshot shows the GUI of Joint State Publisher with sliders and current joint values shown in the box.

Adding physical and collision properties to a URDF model:

Before simulating a robot in a robot simulator, such as Gazebo, V-REP, and so on, we need to define the robot link's physical properties such as geometry, color, mass, and inertia, and the collision properties of the link. We will only get good simulation results if we define all these properties inside the robot model. URDF provides tags to include all these parameters and code snippets of base_link contained in theses properties as given here:

<link>
......
<collision>

```
<geometry>
 <cylinder length="0.03" radius="0.2"/>
  </geometry>
<origin rpy="0 0 0" xyz="0 0 0"/>
</collision>
    <inertial>
<mass value="1"/>
 <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
   </inertial>
 ...........
</link>
```

Here, we define the collision geometry as cylinder and the mass as 1 Kg, and we also set the inertial matrix of the link. The collision and inertia parameters are required in each link; otherwise, Gazebo will not load the robot model properly.

Tasks:

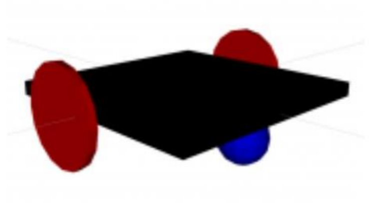
1. Add another joint in the given example and simulate in Gazebo
2. Prepare a 7 DOF arm using XACRO file and simulate in Gazebo

Lab Experiment # 10

Objective:
Imitate the urdf model for a differential drive moving robot and to simulate in ROS

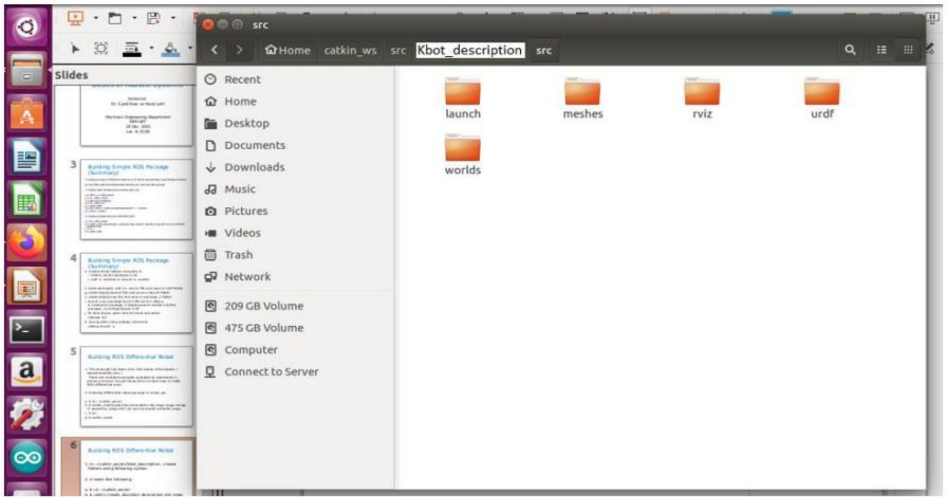
Equipment:
PCs having ubuntu and ROS

Theory:
Differential Robot
Differential drive robot is the most common type of moving robot. A differential drive system works by controlling the velocities of main two co-axial wheels to reach to the desired navigation goal.



To mechanically design and simulate the robot, a Universal Robot Description Format (URDF) needs to generate in URDF folder as per standard ROS procedure. URDF supports XML and XACRO (XML macro) languages. Xacro code is easier to implement, maintain and has better readability. So, lets create a new robot description file in urdf folder by right clicking the mouse and opening gedit toolbox. The path of the file is: urdf/kbot.xacro.

Building ROS Differential Robot:
1. This example has taken from free ebook of Kiranpalla. There are multiple examples available as mentioned in previous lecture. So just follow them to learn how to make ROS differential rover.
2. Creating differential robot package in catkin_ws:
 a. $ cd ~/catkin_ws/src
b. $ catkin_create_pkg kbot_description std_msgs rospy roscpp tf geometry_msgs urdf rviz xacro actionlib actionlib_msgs
c. $ cd ..
 d. $ catkin_make
Placement of Package File:
1. Create manually (using right hand mouse button) or using following command, some folders required to place your package files orderly.
$ cd ~/catkin_ws/src/kbot_description/src
$ mkdir launch urdf rviz world meshes
2. Each created folder has specific purpose:
launch: it will hold launch file (just like exe file) of your package.
urdf: It will hold your robot description file.
rviz: it will hold visualization file for your robot in rviz tool.
world: It will hold environment file where your robot will work.
There are many other folders which we can create when needed such as meshes, config etc. when you will need specific files for them
Now your package folder as highlighted below will appear like this:



Universal Robot Description Format (URDF):
To mechanically design and simulate the robot, a Universal Robot Description Format (URDF) needs to generate in URDF folder as per standard ROS procedure.
URDF supports XML and XACRO (XML macro) languages. Xacro code is easier to implement, maintain and has better readability.
So, let's create a new robot description file in urdf folder by right clicking the mouse and opening gedit toolbox. The path of the file is: urdf/kbot.xacro.
Procedure:
Format of XACRO File:
Writing following lines in kbot.xacro file:

```
•  <?xml version="1.0"?>
• <robot>
• <xmlns:xacro="http://www.ros.org/wiki/xacro" name="kbot">
•
• <xacro:property name="base_width" value="0.16"/>
• <xacro:property name="base_len" value="0.16"/>
• <xacro:property name="wheel_radius" value="0.035"/>
• <xacro:property name="base_wheel_gap" value="0.007"/>
• <xacro:property name="wheel_separation" value="0.15"/>
• <xacro:property name="wheel_joint_offset" value="0.02"/>
 • </robot>
```

In previous file, before </robot> writing following lines:

Box Inertia:
Box inertia is a macro that defines inertia for any box-shaped link. Inertia formulae (ixx, iyy, izz so on) change with the shape of the object and can be obtained by Googling

```
• <xacro:macro name="box_inertia" params="m w h d">
• <inertial>
• <mass value="${m}"/>
• <inertia ixx="${m / 12.0 * (d*d + h*h)}" ixy="0.0" ixz="0.0" iyy="${m / 12.0 * (w*w +
h*h)}" iyz="0.0" izz="${m / 12.0 * (w*w + d*d)}"/>
• </inertial>
• </xacro:macro>
```

Base footprint:
Base footprint is modelled as a tiny box with heavy mass. This represents the center of the robot.

```
• <link name="base_footprint">
• <xacro:box_inertia m="20" w="0.001" h="0.001" d="0.001"/>
• <visual>
• <origin xyz="0 0 0" rpy="0 0 0" />
• <geometry>
• <box size="0.001 0.001 0.001" />
• </geometry>
• </visual>
• </link>
```

Base link
Whereas base link is modelled as a sheet (a box with negligible height) to which rest of the KBot components are going to be attached

Adding following lines:

```
• <link name="base_link">
• <xacro:box_inertia m="10" w="${base_len}" h="${base_width}" d="0.01"/>
• <visual>
• <geometry>
• <box size="${base_len} ${base_width} 0.01"/>
• </geometry>
• <material name="black">
• <color rgba="0 0 0 1"/>
• </material>
• </visual>
• <collision>
• <geometry>
• <box size="${base_len} ${base_width} 0.01"/>
• </geometry>
• </collision>
• </link>
```

Base link joint:
This is for connecting base link to base footprint.

```
<joint name="base_link_joint" type="fixed">
• <origin xyz="0 0 ${wheel_radius + 0.005}" rpy="0 0 0" />
• <parent link="base_footprint"/>
• <child link="base_link" />
```

• *</joint>*

Wheel link and a joint:
Adding wheels in following lines:
• *<xacro:macro name="cylinder_inertia" params="m r h">*
• *<inertial>*
• *<mass value="${m}"/>*
• *<inertia ixx="${m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0" iyy="${m*(3*r*r+h*h)/12}" iyz = "0" izz="${m*r*r/2}"/>*
• *</inertial>*
• *</xacro:macro>*
•
• *<xacro:macro name="wheel" params="prefix reflect">*
• *<link name="${prefix}_wheel">*
• *<visual>*
• *<origin xyz="0 0 0" rpy="${pi/2} 0 0"/>*
• *<geometry>*
• *<cylinder radius="${wheel_radius}" length="0.005"/>*
• *</geometry>*
• *</visual>*
• *<collision>*
• *<origin xyz="0 0 0" rpy="${pi/2} 0 0"/>*
• *<geometry>*
• *<cylinder radius="${wheel_radius}" length="0.005"/>*
• *</geometry>*
• *</collision>*
• *<xacro:cylinder_inertia m="10" r="${wheel_radius}" h="0.005"/>*
• *</link>*

Adding wheel’s joints in following lines:

• *<joint name="${prefix}_wheel_joint" type="continuous">*
• *<axis xyz="0 1 0" rpy="0 0 0" />*
• *<parent link="base_link"/>*
• *<child link="${prefix}_wheel"/>*
• *<origin xyz="${wheel_joint_offset} ${((base_width/2)+base_w*
*heel_gap)*reflect} -0.005" rpy="0 0 0"/>*
• *</joint>*
• *</xacro:macro>*
•
• *<xacro:wheel prefix="left" reflect="1"/>*
• *<xacro:wheel prefix="right" reflect="-1"/>*

The previous code creates a wheel link and a joint to connect it the base link.
Then, using this macro to create a link and a joint for each wheel left_wheel, left_wheel_joint, right_wheel & right_wheel_joint.

Note that a parameter reflect is used to place wheels on either side of base link.

Also, the property wheel_joint_offset determines how far from base plate the wheels are (horizontal offset).

Adding a caster wheel
Adding a caster wheel in following lines:

```
• <xacro:macro name="sphere_inertia" params="m r">
• <inertial>
• <mass value="${m}"/>
• <inertia ixx="${2.0*m*(r*r)/5.0}" ixy="0.0" ixz="0.0" iyy="${2.0*m*(r*r)/5.0}" iyz="0.0"
izz="${2.0*m*(r*r)/5.0}"/>
• </inertial>
• </xacro:macro>
•
• <link name="caster_wheel">
• <visual>
• <origin xyz="0 0 0" rpy="0 0 0"/>
• <geometry>
• <sphere radius="${caster_wheel_radius}"/>
• </geometry>
• </visual>
• <collision>
• <origin xyz="0 0 0" rpy="0 0 0"/>
• <geometry>
• <sphere radius="${caster_wheel_radius}"/>
• </geometry>
• </collision>
• <xacro:sphere_inertia m="5" r="${caster_wheel_radius}"/>
• </link>
```

Adding a caster wheel joint in following lines:

```
• <joint name="caster_wheel_joint" type="continuous">
• <axis xyz="0 1 0" rpy="0 0 0" />
• <parent link="base_link"/>
• <child link="caster_wheel"/>
• <origin xyz="${caster_wheel_joint_offset} 0 -${caster_wheel_radius+0
.005}" rpy="0 0 0"/>
• </joint>
```

Now the basic differential robot is ready. Now some steps are required to visualize and to navigate it in ROS environment.

Creating a launch file:

First, creating a launch file “kbot_base_rviz.launch” in the folder “launch” to read the URDF (xacro) file just created in the URDF folder and to launch rviz (a visualization tool).

The launch file uses xacro utility to paste the URDF and capture it in a parameter robot_description of the parameter server.

Three nodes will be launched: robot_state_publisher, joint_state_publisher ensure that proper transformations between various links (link frames) are published while rviz node initiates rviz visualization tool

Writing the launch file:

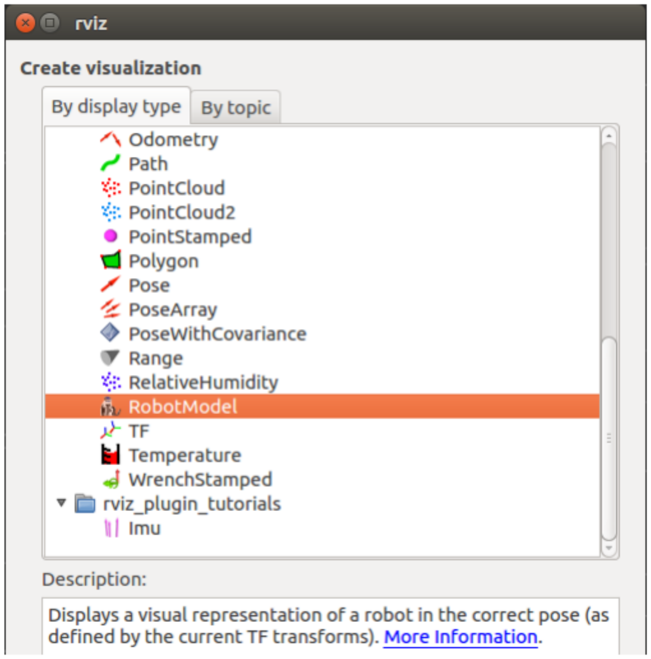
```
• <launch>
• <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find kbot_description)/urdf/kbot.xacro"/>
•
• <node name="robot_state_publisher" pkg="robot_state_publisher" type="s
tate_publisher"/>
•
• <node name="joint_state_publisher" pkg="joint_state_publisher" type="join
t_state_publisher"/>
•
• <node name="rviz" pkg="rviz" type="rviz" required="true"/>
• </launch>
```

Generating a rviz GUI:
The last node opens rviz GUI.

Some settings need to do first in GUI, change value of Global Options -> Fixed Frame from map to base_link.

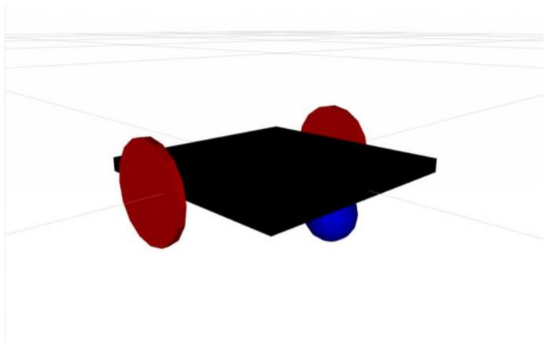
Click on Add in Displays panel and add a Robot Model

Rviz tool provides visualization and navigational help, however it is not a simulator where you can virtually run and observe behavior of robot.
The rviz GUI settings can be seen as follows:



Displays a visual representation of a robot in the correct pose (as defined by the current TF transforms). More Information.

The robot now can be seen in rviz as shown below:



Gazebo simulator
Gazebo is a physics-based real-world robotics simulator. It is the de-facto simulator for most ROS robots.

Gazebo allows robotic engineers to rapidly test their design and algorithms. Gazebo has several pre-built models required to build most kinds of robots and environments (worlds).

Every ROS distribution has some version of Gazebo, however if it is not visible in your ROS then it can be installable using gazebo_ros packages on your host system.

Gazebo supports visualization of URDF-based robot models. Let’s run Gazebo and visualize KBot in the simulator (as well as in rviz).

Creating another launch file in the folder “kbot_description/launch” and named it “kbot_base_rviz_gazebo.launch”. Copy and paste text of earlier launch file and add following text in it.

- *<include file="$(find gazebo_ros)/launch/empty_world.launch">*
- *<arg name="debug" value="false" />*

- *<arg name="gui" value="true" />*
- *<arg name="paused" value="false"/>*
- *<arg name="use_sim_time" value="false"/>*
- *<arg name="headless" value="false"/>*
- *<arg name="verbose" value="true"/>*
- *</include>*
-
- *<!--Launch Gazebo Simulator-->*
- *<node name="spawn_model" pkg="gazebo_ros" type="spawn_model" args="-urdf -param robot_description -model kbot -verbose" output="screen"/>*

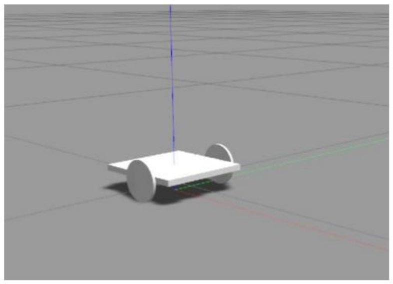
Gazebo requires to load a world file along with a robot model. Currently in this example, an empty world robot environment model is included with some Gazebo parameters.

The KBot model is loaded in Gazebo using spawn_model node as shown in previous slide.

Now running the launch file in another terminal as shown by following command:

$ roslaunch kbot_description kbot_base_rviz_gazebo.launch

The robot model will be appeared like this:



Interfacing Gazebo with URDF:
To navigate the robot, lets add some Gazebo's differential drive plugin in your robot URDF.

It will be able to receive manual steering commands for the robot to run its wheels.

Then a teleop ROS node will be used to send velocity commands to the robot model.

Adding following lines in URDF/XACRO model before the label </robot>:
- *<gazebo>*
- *<plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">*
- *<alwaysOn>false</alwaysOn>*
- *<legacyMode>false</legacyMode>*
- *<updateRate>20</updateRate>*
- *<leftJoint>left_wheel_joint</leftJoint>*

• *<rightJoint>right_wheel_joint</rightJoint>*
• *<wheelSeparation>${wheel_separation}</wheelSeparation>*
• *<wheelDiameter>${wheel_radius * 2}</wheelDiameter>*
• *<torque>20</torque>*
• *<commandTopic>/kbot/base_controller/cmd_vel</commandTopic>*
• *<odometryTopic>/kbot/base_controller/odom</odometryTopic>*
• *<odometryFrame>odom</odometryFrame>*
• *<robotBaseFrame>base_footprint</robotBaseFrame>*
• *</plugin>*
• *</gazebo>*

The newly added code appends a Differential Drive Gazebo plugin to the robot model. Any plugin within tag must be included under tags.

A plugin may have its own parameters. Here left_wheel_joint and right_wheel_joint are the wheel joint names already defined in kbot.xacro. Similarly, values for and come from the main xacro file.

This plugin subscribes to the topic /kbot/base_controller/cmd_vel and publishes topic /kbot/base_controller/odom.

Now modifying the launch file to call tele-op node.

Adding following lines in launch file before </launch> tag:
• *<node name="teleop" pkg="teleop_twist_keyboard" type="teleop_twist_keyboar
d.py" output="screen">*
• *<remap from="/cmd_vel" to="/kbot/base_controller/cmd_vel"/>*
• *</node>*

ROS provides this package to steer a robot base using the keyboard.

A node of package teleop_twist_keyboard publishes a topic called cmd_vel that represents velocity commands for the robot.

Now compiling the complete project (package):

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.sh
```

launch rviz and Gazebo using below command in a terminal.

```
$ roslaunch kbot_description kbot_base_rviz_gazebo.launch
```

In another terminal launch teleop_twist_keyboard node.

```
$roslaunch kbot_simple_control kbot_control_teleop.launch
```

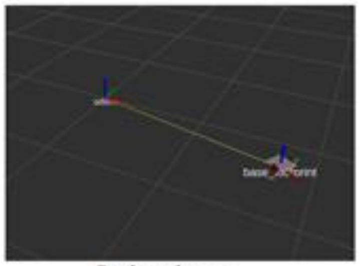
Some configuration parameters in rviz must be tweaked to run the package correctly.

First, set Displays -> Global Options -> Fixed Frame to odom frame instead of map.

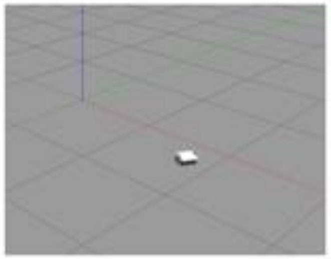
Next, add RobotModel visualization to Displays by clicking ‘add’ and selecting it. Your robot model should be now visible in the map area.
In a similar way, add TF visualization to see co-ordinate frames.

Also, change Current View -> Target Frame to odom. This ensure that odom frame at the initial position of the robot, while base_footprint and robot model navigate the world.

Go to the terminal where teleop_keyboard_twist was launched. Press key ‘i’ and see the robot move forward. Please a relevant key to stop it. Go back to rviz and Gazebo to review the current position of the robot.



Rviz view


gazebo view

Tasks:

1. Add a box on the top surface of dimension 10x10x10 cubic centimeter and again simulate the differential rover
2. Prepare your own differential robot with different dimensions and simulate on Gazebo

Lab Experiment # 11

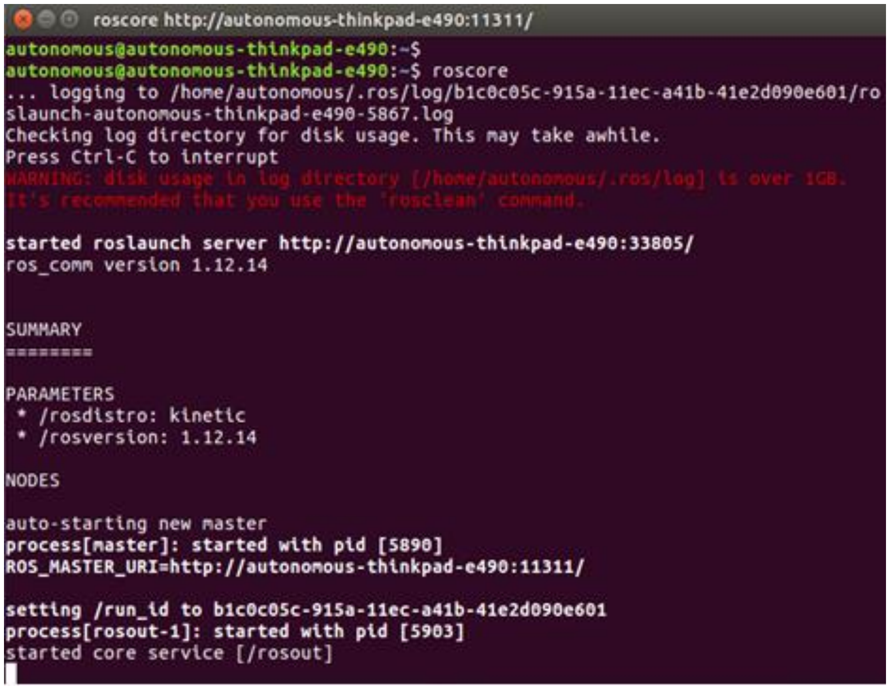
Objective:
Operate under supervision the working of a 4 DOF industrial robot and to interface it with ROS

Equipment:
PCs having ubuntu, ROS and 4 DOF industrial robotic unit

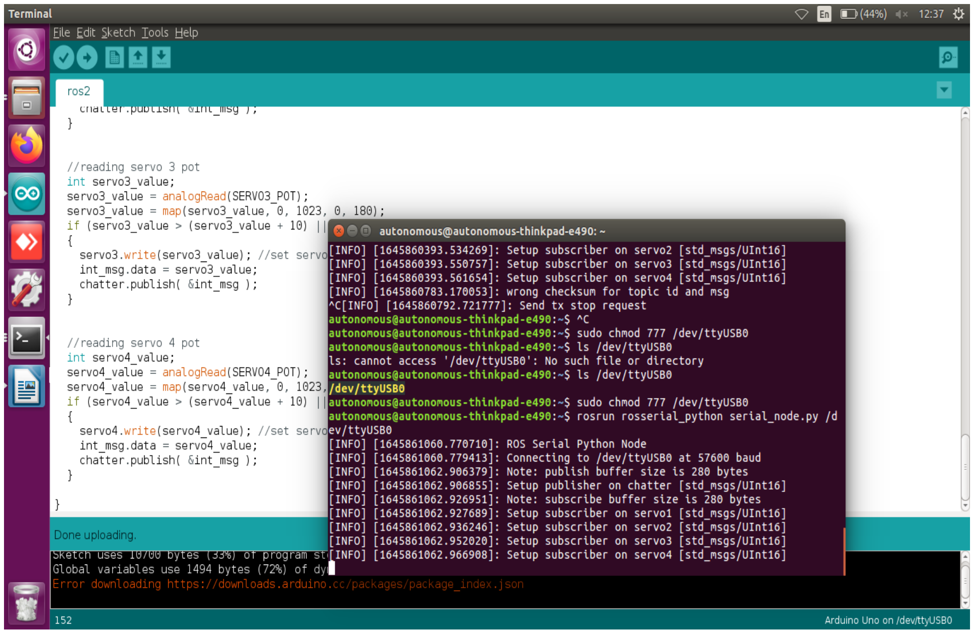
Theory:
4 DOF:
The 4DOF robotic arm has 4 joints to imitate a human upper arm namely joint 1,2,3 and 4 that rotate around x,y and z axis respectively. The joint moves four arm links to get the required posture of the wrist that will be assembled with the hand in future application.

Degrees of Freedom, in a mechanics context, are specific, define modes in which a mechanical device or system can move. The number of degrees of freedom is equal to the total number of independent displacements or aspects of motion. A machine may operate into 3 dimensions but have more than 3 degrees of freedom.
Consider a robotic arm built to work like a human arm. Shoulder motion can take place as pitch (up and down) or yaw (left and right). Elbow motion can occur only as pitch. Wrist motion can occur as pitch or yaw. Rotation (roll) may also be possible for wrist and shoulder.

Procedure:



```
autonomous@autonomous-thinkpad-e490:~$
autonomous@autonomous-thinkpad-e490:~$ roscore
... logging to /home/autonomous/.ros/log/b1c0c05c-915a-11ec-a41b-41e2d090e601/ro
slaunch-autonomous-thinkpad-e490-5867.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
WARNING: disk usage in log directory [/home/autonomous/.ros/log] is over 1GB.
It's recommended that you use the 'rosclean' command.

started roslaunch server http://autonomous-thinkpad-e490:33805/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [5890]
ROS_MASTER_URI=http://autonomous-thinkpad-e490:11311/

setting /run_id to b1c0c05c-915a-11ec-a41b-41e2d090e601
process[rosout-1]: started with pid [5903]
started core service [/rosout]
```



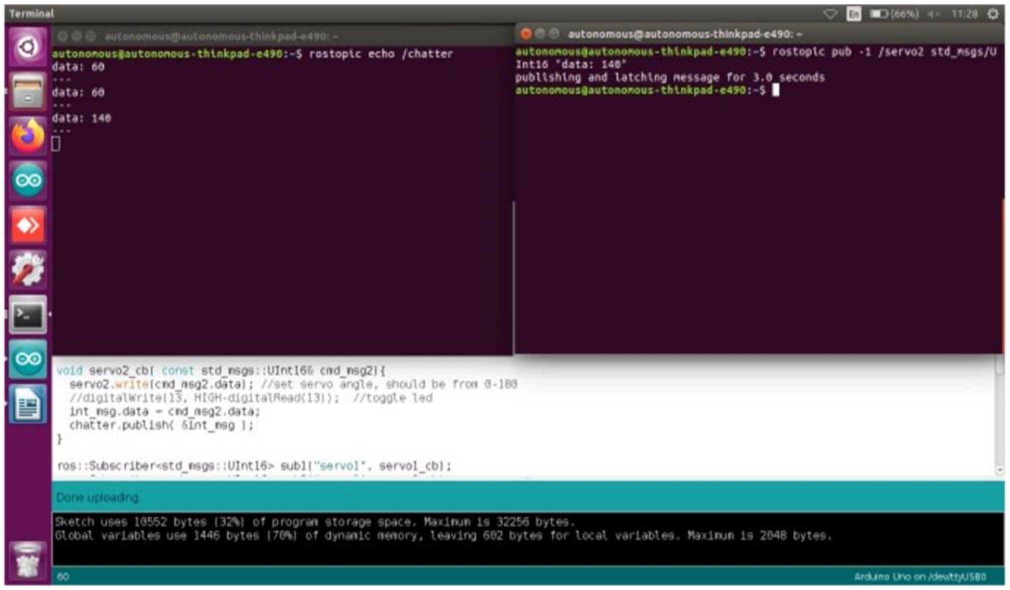
```
  }


  //reading servo 3 pot
  int servo3_value;
  servo3_value = analogRead(SERVO3_POT);
  servo3_value = map(servo3_value, 0, 1023, 0, 180);
  if (servo3_value > (servo3_value + 10) ||
  {
    servo3.write(servo3_value); //set servo
    int_msg.data = servo3_value;
    chatter.publish( &int_msg );
  }


  //reading servo 4 pot
  int servo4_value;
  servo4_value = analogRead(SERVO4_POT);
  servo4_value = map(servo4_value, 0, 1023,
  if (servo4_value > (servo4_value + 10) ||
  {
    servo4.write(servo4_value); //set servo
    int_msg.data = servo4_value;
    chatter.publish( &int_msg );
  }

}
```

Done uploading.

```
Sketch uses 10700 bytes (33%) of program st
Global variables use 1494 bytes (72%) of dy
Error downloading https://downloads.arduino.cc/packages/package_index.json
```

152 Arduino Uno on /dev/ttyUSB0

```
[INFO] [1645860393.534269]: Setup subscriber on servo2 [std_msgs/UInt16]
[INFO] [1645860393.550757]: Setup subscriber on servo3 [std_msgs/UInt16]
[INFO] [1645860393.561654]: Setup subscriber on servo4 [std_msgs/UInt16]
[INFO] [1645860783.170053]: wrong checksum for topic id and msg
^C[INFO] [1645860792.721777]: Send tx stop request
autonomous@autonomous-thinkpad-e490:~$ ^C
autonomous@autonomous-thinkpad-e490:~$ sudo chmod 777 /dev/ttyUSB0
autonomous@autonomous-thinkpad-e490:~$ ls /dev/ttyUSB0
ls: cannot access '/dev/ttyUSB0': No such file or directory
autonomous@autonomous-thinkpad-e490:~$ ls /dev/ttyUSB0
/dev/ttyUSB0
autonomous@autonomous-thinkpad-e490:~$ sudo chmod 777 /dev/ttyUSB0
autonomous@autonomous-thinkpad-e490:~$ rosrun rosserial_python serial_node.py /d
ev/ttyUSB0
[INFO] [1645861060.770710]: ROS Serial Python Node
[INFO] [1645861060.779413]: Connecting to /dev/ttyUSB0 at 57600 baud
[INFO] [1645861062.906379]: Note: publish buffer size is 280 bytes
[INFO] [1645861062.906855]: Setup publisher on chatter [std_msgs/UInt16]
[INFO] [1645861062.926951]: Note: subscribe buffer size is 280 bytes
[INFO] [1645861062.927689]: Setup subscriber on servo1 [std_msgs/UInt16]
[INFO] [1645861062.936246]: Setup subscriber on servo2 [std_msgs/UInt16]
[INFO] [1645861062.952020]: Setup subscriber on servo3 [std_msgs/UInt16]
[INFO] [1645861062.966908]: Setup subscriber on servo4 [std_msgs/UInt16]
```



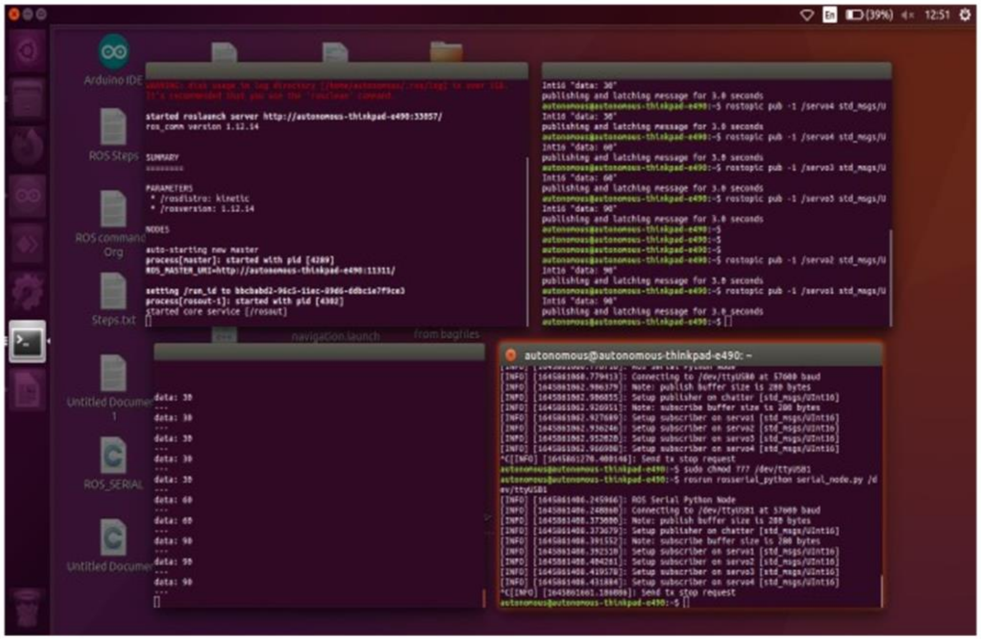
```
autonomous@autonomous-thinkpad-e490:~$ rostopic echo /chatter
data: 60
---
data: 60
---
data: 140
---
```

```
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo2 std_msgs/U
Int16 "data: 140"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$
```

```
void servo2_cb( const std_msgs::UInt16& cmd_msg2){
  servo2.write(cmd_msg2.data); //set servo angle, should be from 0-180
  //digitalWrite(13, HIGH-digitalRead(13));  //toggle led
  int_msg.data = cmd_msg2.data;
  chatter.publish( &int_msg );
}

ros::Subscriber<std_msgs::UInt16> sub1("servo1", servo1_cb);
```

```
Done uploading.
Sketch uses 10552 bytes (32%) of program storage space. Maximum is 32256 bytes.
Global variables use 1446 bytes (70%) of dynamic memory, leaving 602 bytes for local variables. Maximum is 2048 bytes.
```

Terminal
autonomous@autonomous-thinkpad-e490: ~
60
Arduino Uno on /dev/ttyUSB0



```
WARNING: disk usage in log directory [/home/autonomous/.ros/log] is over 1GB.
It's recommended that you use the 'rosclean' command.

started roslaunch server http://autonomous-thinkpad-e490:33857/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [4289]
ROS_MASTER_URI=http://autonomous-thinkpad-e490:11311/

setting /run_id to bbcbabd2-96c5-11ec-8966-ddbc1e7f9ce3
process[rosout-1]: started with pid [4302]
started core service [/rosout]
```

```
Int16 "data: 30"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo4 std_msgs/U
Int16 "data: 30"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo4 std_msgs/U
Int16 "data: 60"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo3 std_msgs/U
Int16 "data: 60"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo3 std_msgs/U
Int16 "data: 90"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$
autonomous@autonomous-thinkpad-e490:~$
autonomous@autonomous-thinkpad-e490:~$
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo2 std_msgs/U
Int16 "data: 90"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$ rostopic pub -1 /servo1 std_msgs/U
Int16 "data: 90"
publishing and latching message for 3.0 seconds
autonomous@autonomous-thinkpad-e490:~$
```

```
data: 30
---
data: 30
---
data: 30
---
data: 30
---
data: 30
---
data: 60
---
data: 60
---
data: 90
---
data: 90
---
data: 90
---
```

```
autonomous@autonomous-thinkpad-e490: ~
[INFO] [1645861060.779413]: Connecting to /dev/ttyUSB0 at 57600 baud
[INFO] [1645861062.906379]: Note: publish buffer size is 280 bytes
[INFO] [1645861062.906855]: Setup publisher on chatter [std_msgs/UInt16]
[INFO] [1645861062.926951]: Note: subscribe buffer size is 280 bytes
[INFO] [1645861062.927689]: Setup subscriber on servo1 [std_msgs/UInt16]
[INFO] [1645861062.936246]: Setup subscriber on servo2 [std_msgs/UInt16]
[INFO] [1645861062.952828]: Setup subscriber on servo3 [std_msgs/UInt16]
[INFO] [1645861062.966980]: Setup subscriber on servo4 [std_msgs/UInt16]
^C[INFO] [1645861270.408146]: Send tx stop request
autonomous@autonomous-thinkpad-e490:~$ sudo chmod 777 /dev/ttyUSB1
autonomous@autonomous-thinkpad-e490:~$ rosrun rosserial_python serial_node.py /d
ev/ttyUSB1
[INFO] [1645861486.245966]: ROS Serial Python Node
[INFO] [1645861486.248960]: Connecting to /dev/ttyUSB1 at 57600 baud
[INFO] [1645861488.373080]: Note: publish buffer size is 280 bytes
[INFO] [1645861488.373679]: Setup publisher on chatter [std_msgs/UInt16]
[INFO] [1645861488.391552]: Note: subscribe buffer size is 280 bytes
[INFO] [1645861488.392510]: Setup subscriber on servo1 [std_msgs/UInt16]
[INFO] [1645861488.404261]: Setup subscriber on servo2 [std_msgs/UInt16]
[INFO] [1645861488.419578]: Setup subscriber on servo3 [std_msgs/UInt16]
[INFO] [1645861488.431884]: Setup subscriber on servo4 [std_msgs/UInt16]
^C[INFO] [1645861601.180080]: Send tx stop request
autonomous@autonomous-thinkpad-e490:~$
```

Tasks:

1. Run the example code by changing joint angles to 45 degree.
2. Apply feedback signal using potentiometer and display on ROS terminal.

Lab Experiment # 12

Objective:
Manipulate with guidance the working and interfacing of a moving rover and to run the rover using ROS

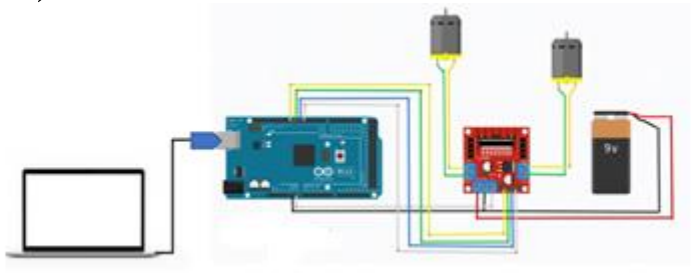
Equipment:
PCs having ubuntu, ROS and a moving rover unit

Theory:
Small Rover Navigation:
The differential drive small rover is made in such a way that it can be navigated through keyboard using ROS (Robot Operating System).

Following are the components that have been used in the small rover:

- Arduino Mega 2560
- Motor Driver (L298N)
- 9V Battery
- DC motors

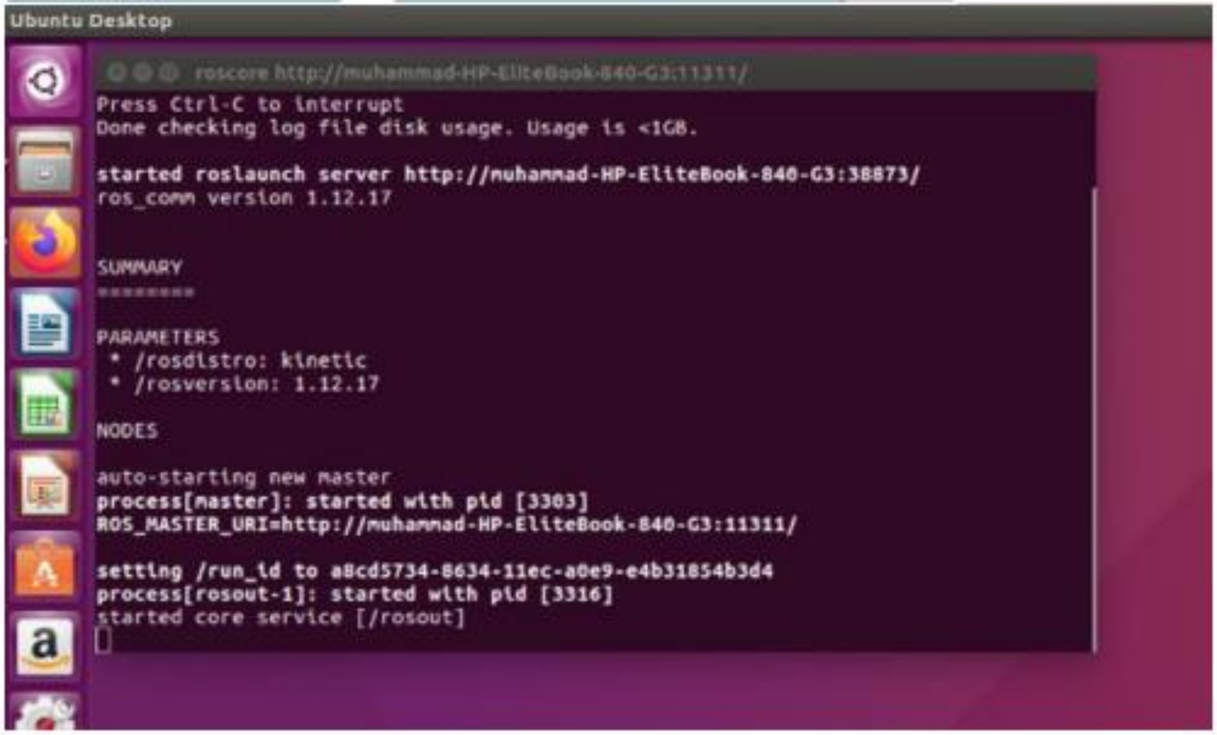
Four motors have been used, each side of the two motors are connected parallelly which is then been controlled by the dual motor driver(L298N).



The PC is installed with ROS (Robot Operating System) which is connected to Arduino Mega which is then connected to the Motor drivers to drive the motors. Laptop is the Master node whereas Arduino Is the slave node which Is receiving instruction from laptop which in turn drive the motors. The above picture includes two motors which represents four motors each is connected parallelly to the other.

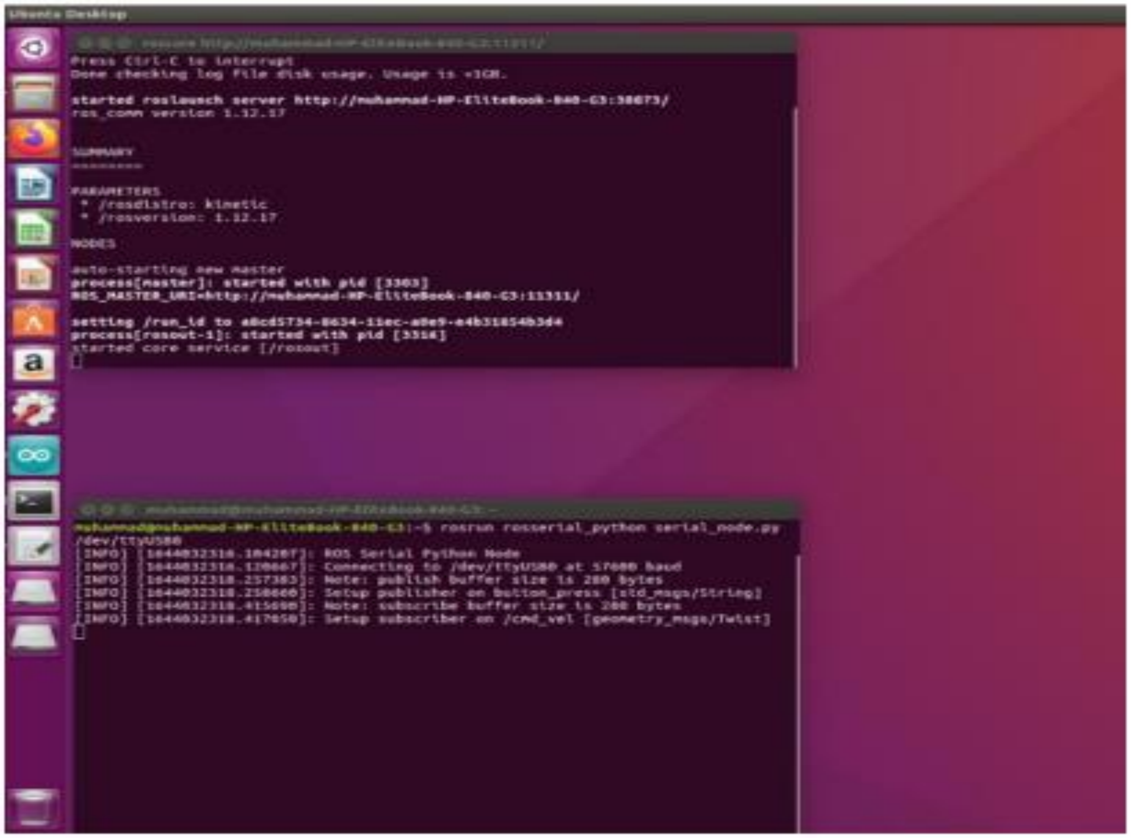
Procedure:
Following where the command run on Linux terminal to run ROS.

1. Opening ROS environment by running the command roscore



```
roscore http://muhammad-HP-EliteBook-840-G3:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://muhammad-HP-EliteBook-840-G3:38873/
ros_comm version 1.12.17


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES

auto-starting new master
process[master]: started with pid [3303]
ROS_MASTER_URI=http://muhammad-HP-EliteBook-840-G3:11311/

setting /run_id to a8cd5734-8634-11ec-a0e9-e4b31854b3d4
process[rosout-1]: started with pid [3316]
started core service [/rosout]
```

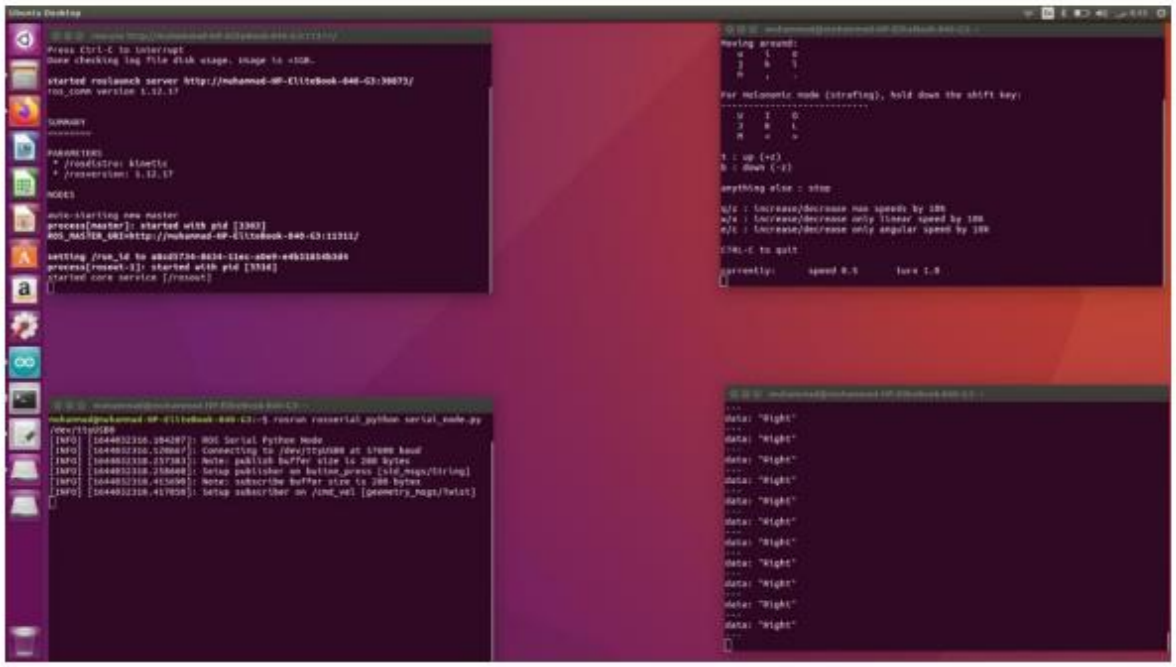
2. After running the above command, rosrun rosserial_python serial_node.py /dev/ttyUSB0, so that the laptop starts communicating with the Arduino board.



```
roscore http://muhammad-HP-EliteBook-840-G3:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://muhammad-HP-EliteBook-840-G3:38873/
ros_comm version 1.12.17


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES

auto-starting new master
process[master]: started with pid [3303]
ROS_MASTER_URI=http://muhammad-HP-EliteBook-840-G3:11311/

setting /run_id to a8cd5734-8634-11ec-a0e9-e4b31854b3d4
process[rosout-1]: started with pid [3316]
started core service [/rosout]
```

```
muhammad@muhammad-HP-EliteBook-840-G3: ~
muhammad@muhammad-HP-EliteBook-840-G3:~$ rosrun rosserial_python serial_node.py
/dev/ttyUSB0
[INFO] [1644032318.104287]: ROS Serial Python Node
[INFO] [1644032318.120867]: Connecting to /dev/ttyUSB0 at 57600 baud
[INFO] [1644032318.257383]: Note: publish buffer size is 280 bytes
[INFO] [1644032318.258660]: Setup publisher on button_press [std_msgs/String]
[INFO] [1644032318.415690]: Note: subscribe buffer size is 280 bytes
[INFO] [1644032318.417050]: Setup subscriber on /cmd_vel [geometry_msgs/Twist]
```

3. Afterwards, we read the current status of the rover movement through rostopic command i.e: rostopic echo button_press and then in the new terminal teleop twist keyboard library has been called by the following command: rosrun teleop_twist_keyboard teleop_twist_keyboard.py



```
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://muhammad-HP-EliteBook-840-G3:39873/
ros_comm version 1.12.17


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES

auto-starting new master
process[master]: started with pid [3303]
ROS_MASTER_URI=http://muhammad-HP-EliteBook-840-G3:11311/

setting /run_id to a8ad5730-8634-11ec-a0e9-e4b318340388
process[rosout-1]: started with pid [3316]
started core service [/rosout]
```

```
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

```
muhammad@muhammad-HP-EliteBook-840-G3:~$ rosrun rosserial_python serial_node.py
/dev/ttyUSB0
[INFO] [1644032316.104207]: ROS Serial Python Node
[INFO] [1644032316.120087]: Connecting to /dev/ttyUSB0 at 57600 baud
[INFO] [1644032318.237383]: Note: publish buffer size is 280 bytes
[INFO] [1644032318.258040]: Setup publisher on button_press [std_msgs/String]
[INFO] [1644032318.415090]: Note: subscribe buffer size is 280 bytes
[INFO] [1644032318.417058]: Setup subscriber on /cmd_vel [geometry_msgs/Twist]
```

```
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
data: "Right"
---
```

ARDUINO CODE:

```
#include <ros.h>
#include <std_msg/String.h>
#include <std_msgs/Ultn16.h>
#include <geometry_msg/Twist.h>

#define BUTTON 10
#define LED1 5
#define LED2 6
#define LED3 7
#define LED4 8

ros::NodeHandle nh;

geometry_msgs::Twist msg;
std_msgs::String button_msg;
std_msgs::UInt16 led_msg;

float move1;
float move2;

void subscriberCallback(const geometry_msgs::Twist& cmd_vel)
{
 move1 = cmd_vel.linear.x;
```

```
move2 = cmd_vel.angular.z;

if (move1>0 && move2 == 0)
{
Forward();
button_msg.data = "Forward";
}
else if (move1<0)
{
Reverse();
button_msg.data = "Reverse";
}
else if (move1 > 0 && move2 < 0)
{
Right();
button_msg.data = "Right";
}
else if (move1 > 0 && move2 > 0)
{
 Left();
button_msg.data = "Left";
}
else
{
die();
button_msg.data = "Stop";
}
}
ros::Publisher button_publisher("button_press", &button_msg);
ros::Subscriber led_subscriber("/cmd_vel", &subscriberCallback);
void setup()
{
// put your setup code here, to run once:
pinMode(LED1, OUTPUT);
pinMode(LED2, OUTPUT);
pinMode(LED3, OUTPUT);
pinMode(LED4, OUTPUT);
pinMode(BUTTON, INPUT);
nh.initNode();
nh.advertise(button_publisher);
nh.subscribe(led_subscriber);
}
void loop()
{
// put your main code here, to run repeatedly:
button_publisher.publish( &button_msg );
nh.spinOnce();
```

```
// Serial.println(led_msg.data);
delay(1);
}
//-----------------Functions-------------------//
void Forward()
{
digitalWrite(LED1, HIGH);
digitalWrite(LED2, LOW);
digitalWrite(LED3, HIGH);
digitalWrite(LED4, LOW);
delay(100);
die();
}
void Reverse()
{
digitalWrite(LED1, LOW);
digitalWrite(LED2, HIGH);
digitalWrite(LED3, LOW);
digitalWrite(LED4, HIGH);
delay(100); die();
}
void Right()
{
digitalWrite(LED1, HIGH);
digitalWrite(LED2, LOW);
digitalWrite(LED3, LOW);
digitalWrite(LED4, HIGH);
delay(100);
die();
}
void Left()
{
digitalWrite(LED1, LOW);
digitalWrite(LED2, HIGH);
digitalWrite(LED3, HIGH);
digitalWrite(LED4, LOW);
delay(100);
die();
}
void die()
{
digitalWrite(LED1, LOW);
digitalWrite(LED2, LOW);
digitalWrite(LED3, LOW);
digitalWrite(LED4, LOW);
}
```

Tasks:

1. Write a program to increase the speed of a rover twice as before.
2. Write a program to run a rover in a straight line by increase the speed twice.
3. Write a program to run the rover in a square shape path, it must return to the starting point

Open Ended Lab

Objective:

Development of student's based small industrial/rover hardware, integration with required electronic components and operation of the robot as per given commands executed using ROS. The complete step by step solution needs to provide by students as instructed during the lab sessions.

REFERENCES

1. Saeed B. Niku, “Introduction to Robotics, Analysis, Control, Applications”, Third edition, Prentice Hall and John Wiley and Sons, 2020.
2. R. Siegwart, I. Nourbakhsh and D. Scaramuzza, “Introduction to Autonomous Mobile Robots”, Second edition, The MIT press, 2011.
3. Peter Corke,”Robotics, Vision and Control: Fundamental Algorithms in Matlab”, Second edition, Springer-Verlag, 2017.
4. M. Quigley, “Programming robots with ROS”, O’Rielly Media Inc., 2015.