Department of Artificial Intelligence and Data Science

U23AD451 - Artificial Intelligence and Robotics

Laboratory Record

SRI ESHWAR COLLEGE OF ENGINEERING
KINATHUKADAVU, COIMBATORE – 641 202

Sri Eshwar College of Engineering
Coimbatore | Tamilnadu
An Autonomous Institution
Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by

Name: Mr. / Ms. …….

Register No: ……..

of 3rd Year B. Tech. – Artificial Intelligence and Data Science in the **U23AD451 - Artificial Intelligence and Robotics** during the 5th **Semester** of the academic year **2025 – 2026 (Odd Semester).**

Signature of Faculty In-Charge Head of the Department

Submitted for the end semester practical examinations held on …….

Internal Examiner External Examiner

Sri Eshwar College of Engineering
Coimbatore | Tamilnadu
An Autonomous Institution
Affiliated to Anna University, Chennai

Department of Artificial Intelligence and Data Science

Vision, Mission, PEOs, POs and PSOs of the Department

Vision

To build a conducive academic and research environment in the stream of Artificial Intelligence and Data Science for enabling global education, research and entrepreneurship.

Mission

M1: Rigorous update and enrich the curriculum by integrating global trends and technological advancements to ensure the academic excellence.

M2: Promote a research-driven environment by providing access to state-of-the-art infrastructure with emerging technologies.

M3: Empower the faculty through continuous upskilling, international collaborations, and active research engagement to provide globally competent education.

M4: Encourage student-centric, industry-aligned learning that nurtures innovation and entrepreneurship activities for solving real-world and societal challenges.

PEO: Program Educational Objectives

PEO1: To enable graduates to pursue higher education and research or have a successful career in industries associated with Artificial Intelligence and Data Science, or as entrepreneurs.

PEO2: To demonstrate excellence in cutting-edge technologies of Artificial Intelligence and Data Science and solve problems in society.

PEO3: To exhibit professional ethics, involvement in team work in their profession and contributing to the advancement of society.

PO : Program Outcomes:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems

PO2: Problem analysis: Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using the first principles of mathematics, natural sciences, and engineering sciences

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and the cultural, societal, and environmental considerations

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: Environment and sustainability: Understand the impact of professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's work, as a member and leader in a team, to manage projects and in multidisciplinary environments

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PSO: Program Specific Outcomes

PSO1: Apply programming, problem-solving, and hardware integration skills with AI and data science to design intelligent, secure and efficient systems.

PSO2: Develop an AI-driven application by exploring optimized coding competencies using predictive modeling and data analytics technologies.

Sri Eshwar College of Engineering
An Autonomous Institution
Affiliated to Anna University, Chennai

Vision & Mission of the Institution

Vision

To be recognized as a premier institution, grooming students into globally acknowledged engineering professionals

Mission

- Providing outcome and value-based engineering education
- Nurturing research and entrepreneurial culture
- Enabling students to be industry-ready and fulfil their career aspirations
- Grooming students through behavioural and leadership training programs
- Making students socially responsible.

Table of Contents

Ex. No: 1 Configure the ROS environment and organize workspaces for efficient project management

AIM:

To configure the Robot Operating System (ROS) environment and organize workspaces effectively, ensuring efficient project management and streamlined development for robotic applications.

ALGORITHM / STEPS:

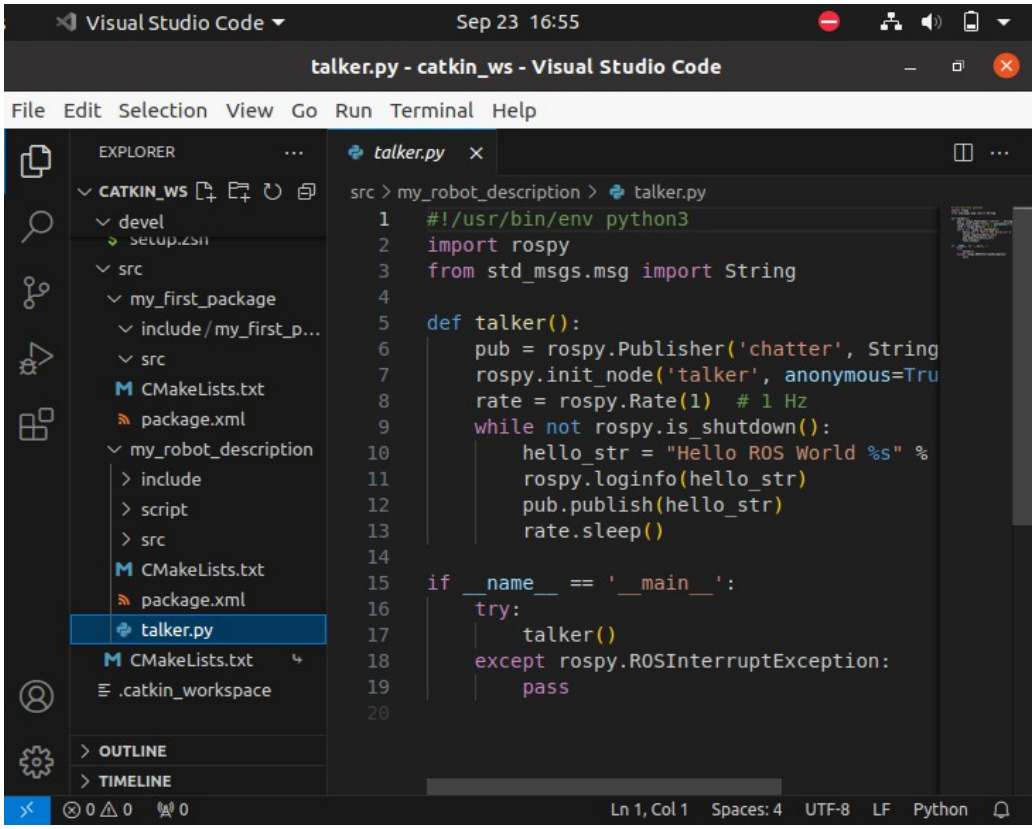
1. Install required ROS distribution and dependencies.
2. Create and initialize a workspace (catkin for ROS1 / colcon for ROS2).
3. Create package skeletons for nodes, messages, services.
4. Use `rosdep` to install external dependencies.
5. Arrange packages into logical folders (e.g., `drivers/`, `algorithms/`, `sim/`, `tools/`).
6. Use version control (git) with `.gitignore` and README templates.
7. Build workspace and source environment. 8. Document workspace layout and build/run procedures

PROGRAM:

```
# install ros (example)
sudo apt update
sudo apt install ros-noetic-desktop-full

# setup ros environment
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc

# create catkin workspace
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make

# create a new package
cd ~/catkin_ws/src
catkin_create_pkg my_robot std_msgs rospy roscpp

# install deps
cd ~/catkin_ws
rosdep install --from-paths src --ignore-src -r -y

# build and source
catkin_make
source devel/setup.bash

# recommended git structure
```

```
cd ~/catkin_ws/src
git init
# add each package as separate repo or submodule
```

OUTPUT:



RESULT

The workspace is now configured for reproducibility. A clear package structure and well-documented build and run steps have been implemented, enhancing team productivity and minimizing integration issues.

Ex. No: 2 Develop and implement basic ROS nodes with publishers and subscribers to facilitate inter-node communication

AIM:

To Implement basic Ros nodes with publishers and subscribers for seamless inter – node communication, enabling message exchange and building a foundation for scalable robotic applications.

ALGORITHM / STEPS:

1. Create a ROS package.
    - Open a terminal and navigate to your workspace:
        - Cd ~/catkin_ws/src
2. Create a New ROS Package:
    - Create catkin package
        - catkin_create_pkg my_robot_description std_msgs rospy roscpp

3. catkin_create_pkg: This is the base command for creating a new ROS package using the Catkin build system. It automatically generates the necessary folder structure, a package.xml file, and a CMakeLists.txt file, pre-filled with the provided information.

4. my_robot_description: This is the desired name for the new package. The name can be anything but is typically descriptive of the package's purpose (e.g., in this case, likely containing Universal Robot Description Format (URDF) files or other description data for a robot).

5. std_msgs rospy roscpp: These are the dependencies for the new package.
   The catkin_create_pkg command automatically adds these packages to the <build_depend> and <exec_depend> lists in the new package's package.xml and CMakeLists.txt files, ensuring they are available at compile and runtime.

    - std_msgs: This package contains standard, primitive ROS message data types (e.g., String, Int64, Float64) that are commonly used for communication between nodes.

    - rospy: This is the client library that provides the Python API for writing ROS nodes. This dependency is included if the package will contain any Python code.

    - roscpp: This is the client library that provides the C++ API for writing ROS nodes. This dependency is included if the package will contain any C++ code.

6. Navigate to your packages directory:
    - Cd my_robot_description
7. Create python publisher and subscriber Nodes

- Create a scripts directory:
    - mkdir scripts

8. Make the script Executable:

    - Chmod +x scripts/talker.py

9. Create a launch file

    - Create a launch Directory
        - mkdir launch

10. Launch your nodes.

    - ros2 launch <package_name> <launch_file_name>

PROGRAM:

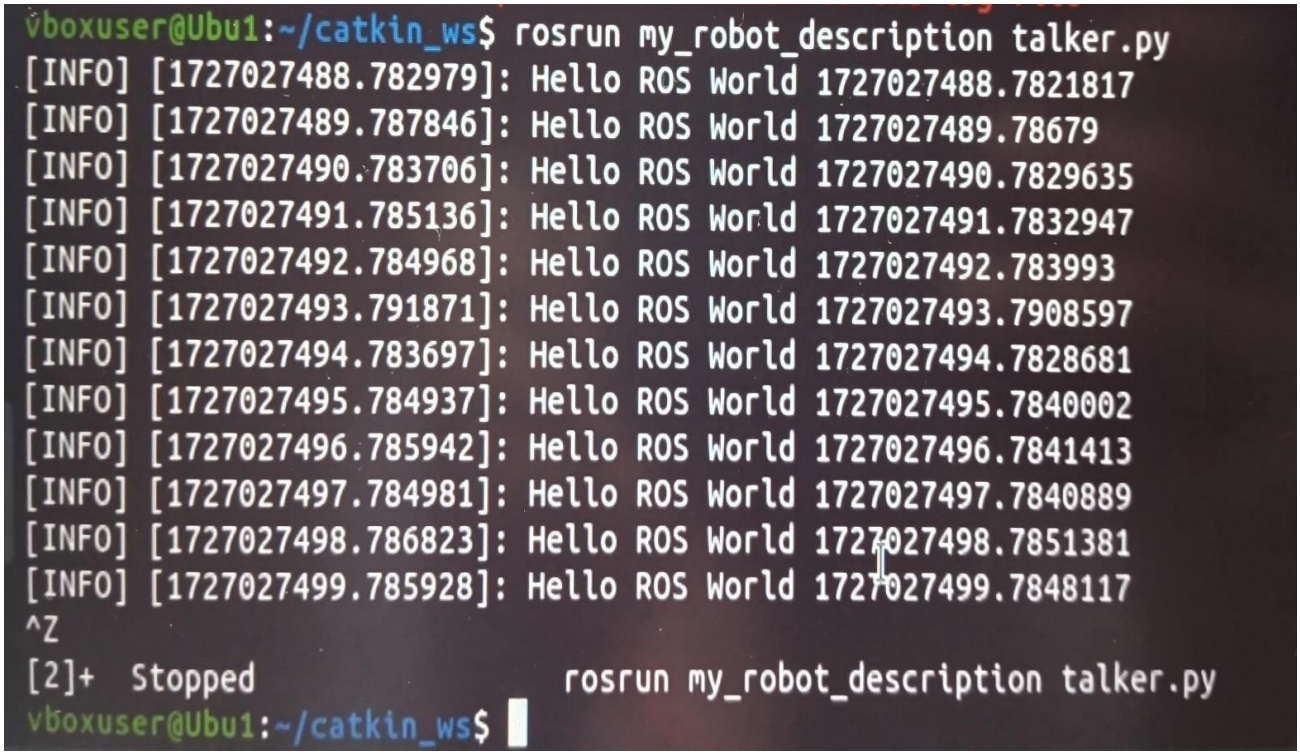
publisher_node.py

```
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(1) # 1 Hz
    count = 0
    while not rospy.is_shutdown():
        msg = f"hello {count}"
        rospy.loginfo(msg)
        pub.publish(msg)
        count += 1
        rate.sleep()

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

subscriber_node.py

```
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(f"I heard: {data.data}")

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber('chatter', String, callback)
    rospy.spin()

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

OUTPUT:



RESULT

Thus the basic ROS nodes with publishers and subscribers, enabling seamless, decoupled, and real-time inter-node communication via a defined set of topics and message types.

Ex. No: 3 Connect and integrate various sensors with a microcontroller to gather and process environmental data

AIM:

To interface various digital and analog sensors with the Arduino Uno microcontroller. A digital sensor converts the measured physical quantity into discrete values, which can be easily processed by computers and other digital devices.

ALGORITHM / STEPS:

1. Wire sensors to microcontroller with appropriate power and pull-ups.
2. Implement sensor drivers in Arduino IDE (or PlatformIO).
3. Calibrate sensors (offsets, scaling).
4. Apply simple filtering (moving average / low-pass).
5. Publish readings over serial or ROS (rosserial/ROS2 micro-ROS).
6. Log and visualize data (rqt_plot / matplotlib).

PROGRAM:

```
#include <DHT.h>
#define DHTPIN 2
#define DHTTYPE DHT22
DHT dht(DHTPIN, DHTTYPE);

// Ultrasonic sensor pins
#define TRIG 3
#define ECHO 4

// Moving average filter variables
float tempSum = 0, humSum = 0, distSum = 0;
const int windowSize = 5;
float tempReadings[windowSize], humReadings[windowSize], distReadings[windowSize];
int index = 0;
```

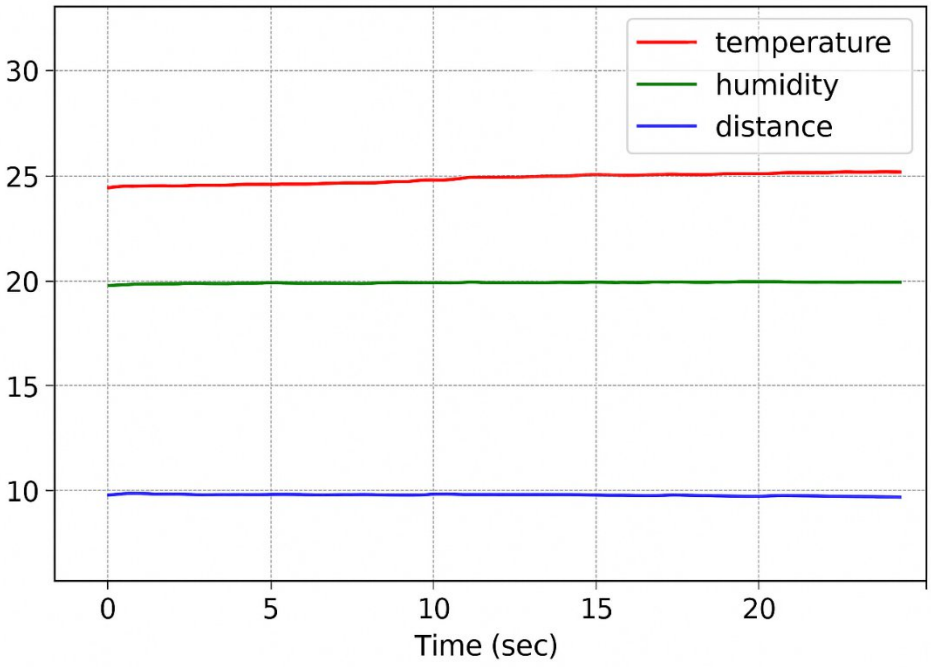
```
void setup() {
  Serial.begin(9600);
  dht.begin();
  pinMode(TRIG, OUTPUT);
  pinMode(ECHO, INPUT);
  for (int i = 0; i < windowSize; i++) {
    tempReadings[i] = humReadings[i] = distReadings[i] = 0;
  }
}

float readDistanceCM() {
  digitalWrite(TRIG, LOW);
  delayMicroseconds(2);
  digitalWrite(TRIG, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG, LOW);
  long duration = pulseIn(ECHO, HIGH);
  float distance = duration * 0.034 / 2;  // cm
  return distance;
}

void loop() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  float d = readDistanceCM();

  if (isnan(h) || isnan(t)) {
    Serial.println("Sensor Error");
    delay(1000);
    return;
  }

  // Calibration offset example
  t = t + 0.5;  // adjust temperature
  d = d - 1.2;  // adjust distance
```

```
  // Moving average filter
  tempSum -= tempReadings[index];
  humSum -= humReadings[index];
  distSum -= distReadings[index];

  tempReadings[index] = t;
  humReadings[index] = h;
  distReadings[index] = d;

  tempSum += t;
  humSum += h;
  distSum += d;

  index = (index + 1) % windowSize;

  float avgTemp = tempSum / windowSize;
  float avgHum = humSum / windowSize;
  float avgDist = distSum / windowSize;

  // Output data
  Serial.print("Temperature(C): ");
  Serial.print(avgTemp);
  Serial.print(" | Humidity(%): ");
  Serial.print(avgHum);
  Serial.print(" | Distance(cm): ");
  Serial.println(avgDist);

  delay(2000);
 }
```

```
rosrun rosserial_python serial_node.py _port:=/dev/ttyUSB0 _baud:=9600
rostopic echo /sensor_data
rqt_plot /sensor_data/temperature /sensor_data/humidity /sensor_data/distance
```

OUTPUT:

Temperature(C): 25.50 | Humidity(%): 57.10 | Distance(cm): 14.80
Temperature(C): 25.60 | Humidity(%): 56.90 | Distance(cm): 14.75
Temperature(C): 25.65 | Humidity(%): 57.00 | Distance(cm): 14.70
Temperature(C): 25.70 | Humidity(%): 57.20 | Distance(cm): 14.72



RESULT

The Sensors are successfully integrated and filtered. Data forwarding to ROS enables higher-level algorithms (navigation, SLAM) to use environment information.

Ex. No: 4 Utilize Gazebo to simulate robotic systems, analyze their behaviour, and test control algorithms

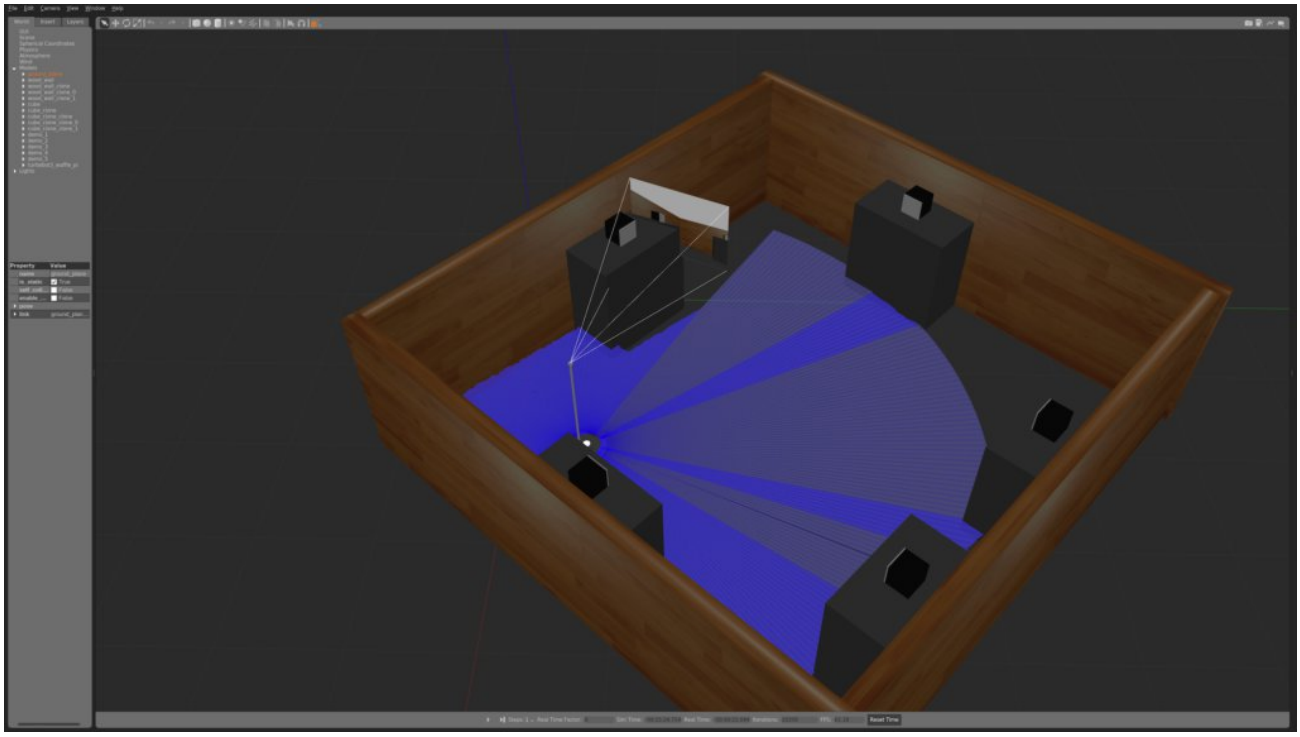
AIM:

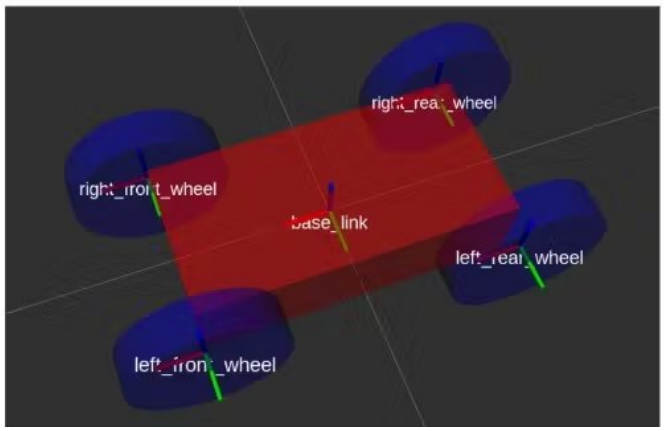
To Utilize Gazebo software to prepare 3D model of Autonomous Mobile Robot and environment

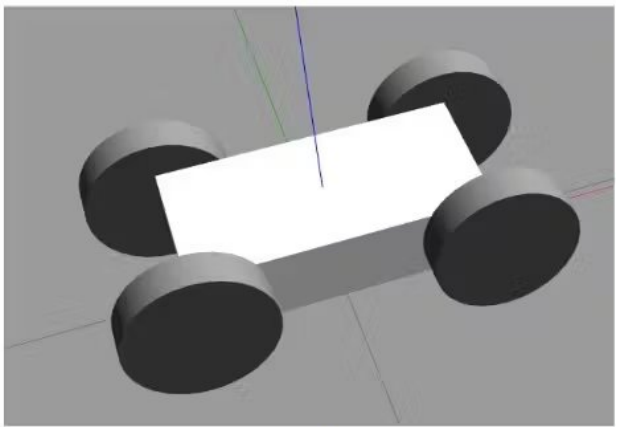
ALGORITHM / STEPS:

1. Open Gazebo and start a new project for creating the 3D model of the autonomous mobile robot directly in the simulation environment.
2. Use Gazebo's model editor to design the robot's base structure, setting appropriate dimensions for the chassis based on the desired size and shape of the robot.
3. Add wheels or tracks by selecting suitable shapes from Gazebo's model editor and positioning them at the base of the chassis. Ensure correct alignment and balance for stable movement during simulation.
4. Add sensors such as LiDAR, cameras, or ultrasonic sensors by choosing components from Gazebo's sensor library or custom plugins. Position the sensors strategically on the robot model to achieve optimal coverage of the surrounding environment.
5. Define joints for each movable part of the robot, such as wheels or articulated arms, specifying the degrees of freedom and type of motion (revolute, prismatic, etc.) to allow for realistic movement during simulation.
6. Configure each sensor's specifications, including field of view, range, and frequency, to reflect the intended behavior of the real sensors. Adjust these parameters to meet the requirements of the autonomous navigation tasks.
7. Save the robot model and export it as a .sdf (Simulation Description Format) or .urdf (Unified Robot Description Format) file for use within Gazebo.
8. Set up the environment by adding static objects such as walls, obstacles, and landmarks within the Gazebo world. Arrange these elements to resemble the real-world setting where the robot will operate.
9. Test the robot's movement and sensor functionality by launching the Gazebo simulation, observing the robot's interaction with the environment. Make adjustments to the model or environment as needed for smooth operation and realistic responses.
10. Save the complete environment and robot model for future use in navigation and path-planning simulations.

OUTPUT:







RESULT

The Teleoperation and SLAM in the Gazebo simulation, creating a 2D map of the environment is done successfully.

Ex. No: 5 Model and simulate linkage mechanisms to evaluate their kinematic performance and positional accuracy

AIM:

The objective is to simulate linkage mechanisms to evaluate their kinematic performance and positional accuracy.

1. Introduction to Linkages:

Linkages are assemblies of rigid bodies connected by joints to form a closed chain or a series of open chains. They are widely used in robotic systems to transmit motion and force. Some common types of linkage mechanisms used in robots include:

- **Four-Bar Linkage**: A mechanism used to convert rotational motion into linear or other types of motion.
- **Slider-Crank Mechanism**: Used to convert rotational motion into reciprocating motion.
- **Gripper Mechanism**: A mechanism used to grasp and hold objects.

2. Basic Definitions:

- **Links**: Rigid members of a mechanism.
- **Joints**: Connections between links allowing motion (e.g., revolute or prismatic).
- **Degrees of Freedom (DOF)**: The number of independent motions allowed in a system.
- **Crank, Coupler, and Follower**: Components of the four-bar mechanism.

3. Tools and Software:

- **Linkage Software**: This software provides a platform to design, simulate, and analyze the motion of various mechanical linkages.
- **Computer**: Installed with Linkage Software.

Note: You can download and install Linkage Software from Linkage Download Page.

4. Mechanism Synthesis Procedure:

Step 1: Understand the Required Motion

- Identify the type of motion required from the robot (e.g., rotary, linear, gripping).
- Select the appropriate linkage mechanism based on the motion requirement.

Step 2: Open Linkage Software

- Launch Linkage Software on your computer.
- Familiarize yourself with the interface, which includes tools for adding links, joints, and actuators.

Step 3: Create the Basic Linkage

- For a Four-Bar Linkage:
  1. Add four rigid links using the "Link" tool.
  2. Connect the links using revolute joints at their ends.
  3. Ensure one link is fixed (ground), one is a crank (input link), and the other two are coupler and follower.

  For a Slider-Crank Mechanism:
  1. Add three links: crank, connecting rod, and slider.
  2. Connect the crank to the ground with a revolute joint.
  3. Add a prismatic joint between the slider and ground.
- Simulate the mechanism using the play button.

Step 4: Add Actuation

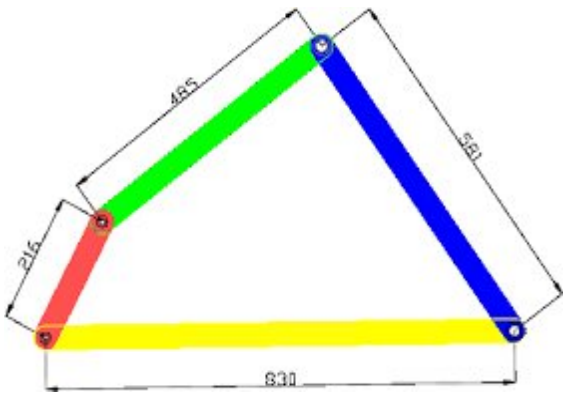
- Choose a link to actuate. For example, in a four-bar linkage, the crank is often actuated to drive the motion.
- Use the motor tool to apply input motion to the crank or slider.

Step 5: Analyze the Motion

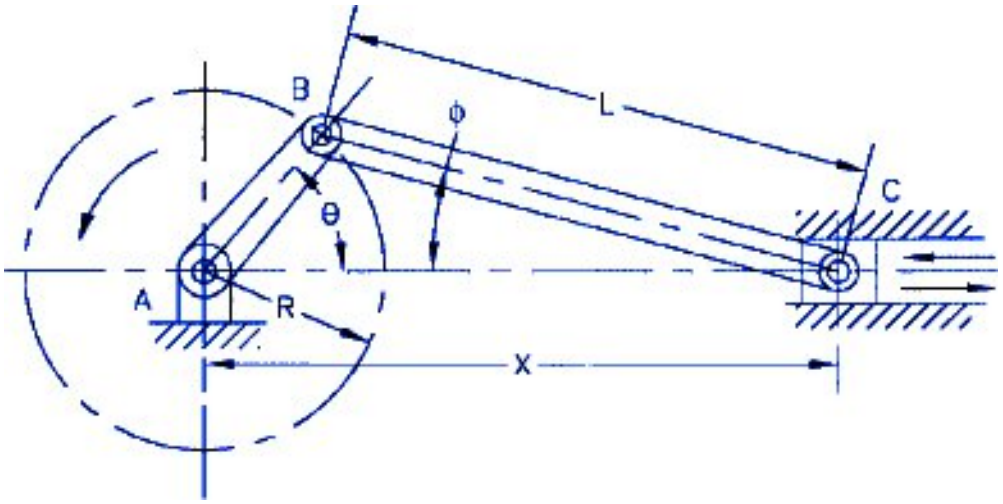
- Run the simulation and observe how the links move.
- Analyze the output motion for displacement, velocity, and acceleration characteristics.

Step 6: Modify the Design

- Modify the length of the links to achieve the desired motion range.
- Explore different configurations of the linkage for optimization.



Four Bar Linkage



Linkage Synthesis for Pick-and-Place Robotic arm

Ex. No: 6 Perform kinematic analysis on a DOF robotic arm to understand and solve forward and inverse kinematics problems

AIM:

To compute the position of the end-effector of a 2-DOF planar robot arm given the joint angles and link lengths.

ALGORITHM / STEPS:

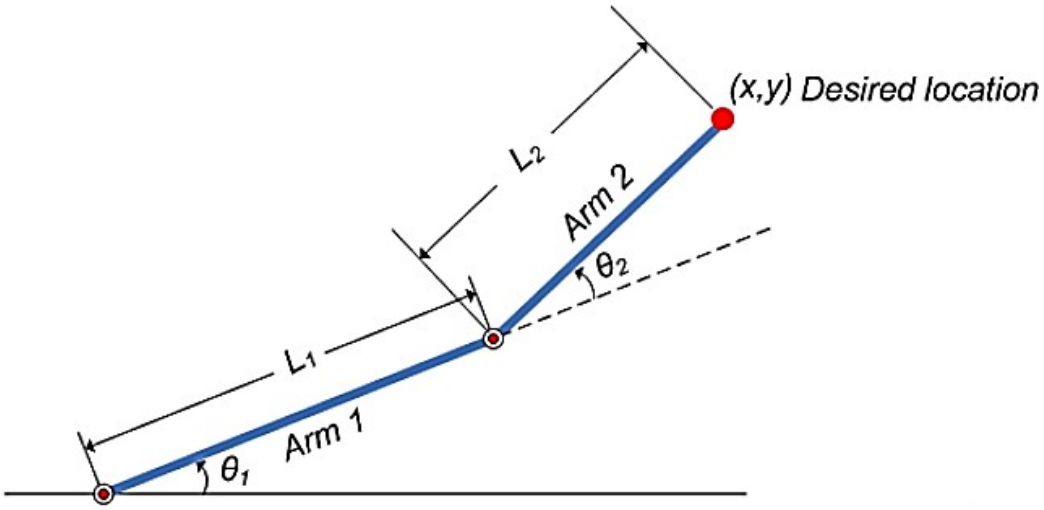


Figure 1: Illustration showing the two-joint robotic arm with the two angles, `theta1` and `theta2`

Theory:

Forward kinematics involves calculating the position of the robot's end-effector based on the lengths of its links and the angles at its joints.

For a 2-DOF planar robot arm:

- Link 1: Length L_1, angle θ_1 (relative to the x-axis).
- Link 2: Length L_2, angle θ_2 (relative to Link 1).

Procedure:

1. Understand the System Setup:
    - The robot has two links, each with a length.
    - There are two angles, θ_1 and θ_2.
    - The goal is to find the end-effector position in a 2D plane.
2. **Write Down the Position Equations**: The forward kinematics equations for the 2-DOF robot arm are:

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2)$$

$$y = L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2)$$

    - x and y are the coordinates of the end-effector.
    - L_1, L_2 are the link lengths.
    - θ_1, θ_2 are the joint angles.
3. **Select Joint Angles and Link Lengths**: For this lab, you will be given the values of L_1, L_2, θ_1, and θ_2. Use the angles in radians, or convert from degrees:

$$\theta(\text{radians}) = \frac{\pi}{180} \times \theta(\text{degrees})$$

4. **Substitute the Values:** Plug the given angles and lengths into the equations. For example:
    - $L_1 = 2$ units, $L_2 = 1$ unit.
    - $\theta_1 = 45^\circ = \frac{\pi}{4}, \theta_2 = 30^\circ = \frac{\pi}{6}$.
5. **Compute x and y**: Using the substituted values, calculate x and y step by step:
    - Compute $\cos(\theta_1)$ and $\sin(\theta_1)$.
    - Compute $\cos(\theta_1 + \theta_2)$ and $\sin(\theta_1 + \theta_2)$.
    - Add them together as per the equations.

    Example calculation:

$$x = 2 \times \cos\left(\frac{\pi}{4}\right) + 1 \times \cos\left(\frac{\pi}{4} + \frac{\pi}{6}\right)$$

$$y = 2 \times \sin\left(\frac{\pi}{4}\right) + 1 \times \sin\left(\frac{\pi}{4} + \frac{\pi}{6}\right)$$

6. **Plot the End-Effector Position:**
    - Once you have calculated x and y, plot these coordinates on a graph to visualize the end-effector position.

Conclusion:

You have successfully computed the end-effector position of the 2-DOF robot arm using forward kinematics.

Example Values for Practice:

- $L_1 = 2$ units, $L_2 = 1$ unit.
- $\theta_1 = 60^\circ, \theta_2 = 30^\circ$.

Inverse Kinematics of Robotic Arm – 2 Degrees of Freedom

Aim:

To compute the joint angles θ1\theta_1θ1 and θ2\theta_2θ2 for a 2-DOF planar robot arm given the end-effector's desired position and the link lengths.

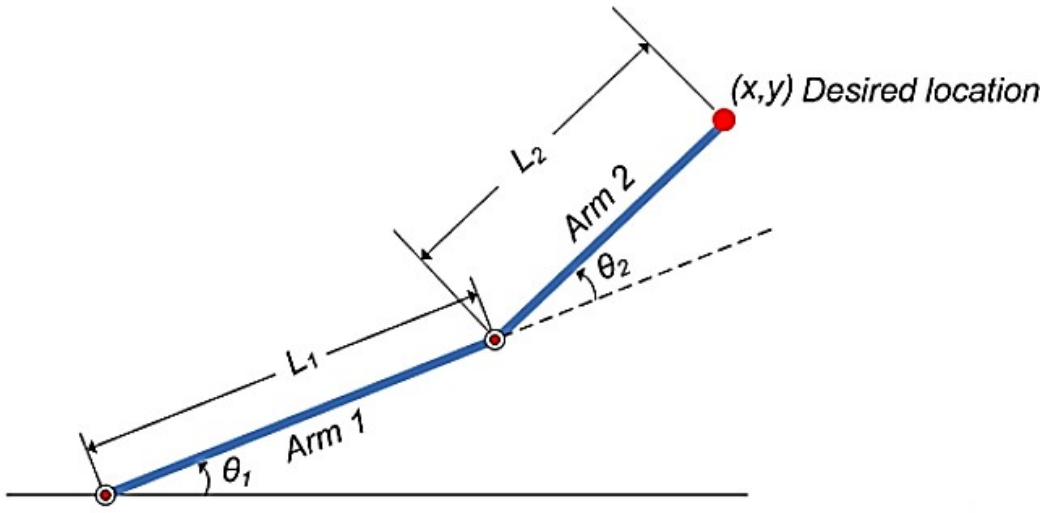


Figure 1: Illustration showing the two-joint robotic arm with the two angles, `theta1` and `theta2`

Theory:

Inverse kinematics is the process of calculating the joint angles required for the end-effector to reach a specific position in space. For a 2-DOF planar robot, this involves calculating the angles θ_1 and θ_2 given the desired position (x, y).

For a 2-DOF planar robot arm:

- **Link 1**: Length L_1.
- **Link 2**: Length L_2.
- **End-Effector Position**: x and y.

Procedure:

1. **Understand the System Setup**:
   - The robot has two links, each with a known length.
   - The end-effector position (x, y) is known.
   - The goal is to find the joint angles θ_1 and θ_2 that will position the end-effector at (x, y).

2. **Write Down the Equations:** The forward kinematics equations relate the end-effector position to the joint angles. These equations are:

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2)$$

$$y = L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2)$$

In inverse kinematics, we want to solve these equations to find θ_1 and θ_2.

3. **Compute θ_2:** From geometry, we can compute θ_2 using the law of cosines:

$$\cos(\theta_2) = \frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}$$

This equation gives us θ_2:

$$\theta_2 = \cos^{-1} \frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}$$

- **Note:** If θ_2 has two possible solutions (elbow-up and elbow-down configurations), choose the one that makes sense for your robot.

4. **Compute θ_1:** After calculating θ_2, we compute θ_1 using geometry. First, calculate intermediate variables:

$$\alpha = \tan^{-1} \frac{y}{x}$$

$$r = \sqrt{x^2 + y^2}$$

Then use the law of cosines for θ_1:

$$\beta = \cos^{-1} \frac{L_1^2 + r^2 - L_2^2}{2L_1r}$$

Now, θ_1 is:

$$\theta_1 = \alpha - \beta$$

5. **Substitute the Values:**
   - Plug in the desired x, y values and the given link lengths L_1 and L_2 into the equations.
   - For example, let's assume:
     - $L_1 = 2$ units, $L_2 = 1$ unit.
     - Desired end-effector position: $x = 2.5$ units, $y = 1.5$ units.

6. **Compute θ_1 and θ_2:**
   - Calculate θ_2 using the equation for θ_2.
   - Calculate θ_1 using the equations for α and β.
7. **Verify the Result:**
   - After calculating the joint angles, verify the result by plugging θ_1 and θ_2 back into the forward kinematics equations.
   - Ensure that the computed end-effector position matches the desired (x, y).

Example:

Let's assume:

- $L_1 = 2$ units, $L_2 = 1$ unit.
- Desired end-effector position: $x = 2.5$ units, $y = 1.5$ units.

1. Calculate θ_2:

$$\theta_2 = \cos^{-1} \quad \frac{2.5^2 + 1.5^2 - 2^2 - 1^2}{2 \times 2 \times 1}$$

2. Calculate θ_1:

$$\alpha = \tan^{-1} \quad \frac{1.5}{2.5}$$

$$r = \sqrt{2.5^2 + 1.5^2}$$

$$\beta = \cos^{-1} \quad \frac{2^2 + r^2 - 1^2}{2 \times 2 \times r}$$

$$\theta_1 = \alpha - \beta$$

3. **Plot the Joint Angles:**
   - Once θ_1 and θ_2 are calculated, plot the position of each joint and link to visualize the configuration of the robot.

Conclusion:

You have successfully computed the joint angles θ_1 and θ_2 required for the end-effector to reach a specific position using inverse kinematics.

Example Values for Practice:

- $L_1 = 2$ units, $L_2 = 1$ unit.
- Desired end-effector position: $x = 2.5, y = 1.5$.

Forward Kinematics of Robotic Arm – 3 Degrees of Freedom

Theory:

For a 3-DOF planar robot, we calculate the position of the end-effector based on the lengths of its links and the angles at its joints.

For a 3-DOF planar robot arm:

- **Link 1**: Length L_1, angle θ_1 (relative to the x-axis).
- **Link 2**: Length L_2, angle θ_2 (relative to Link 1).
- **Link 3**: Length L_3, angle θ_3 (relative to Link 2).

Procedure:

1. **Understand the System Setup**:
   - The robot has three links, each with a specific length.
   - There are three angles, θ_1, θ_2, and θ_3.
   - The goal is to find the end-effector position in a 2D plane.
2. **Write Down the Position Equations**: The forward kinematics equations for a 3-DOF planar robot arm are:

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) + L_3 \cos(\theta_1 + \theta_2 + \theta_3)$$

$$y = L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2) + L_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

   - x and y are the coordinates of the end-effector.
   - L_1, L_2, L_3 are the link lengths.
   - θ_1, θ_2, θ_3 are the joint angles.

3. **Select Joint Angles and Link Lengths**: For this lab, you will be given the values of L_1, L_2, L_3, θ_1, θ_2, and θ_3. Use the angles in radians, or convert from degrees:

$$\theta(\text{radians}) = \frac{\pi}{180} \times \theta(\text{degrees})$$

4. **Substitute the Values:** Plug the given angles and lengths into the equations. For example:
   - $L_1 = 2$ units, $L_2 = 1.5$ units, $L_3 = 1$ unit.
   - $\theta_1 = 45^\circ = \frac{\pi}{4}, \theta_2 = 30^\circ = \frac{\pi}{6}, \theta_3 = 20^\circ = \frac{\pi}{9}$.
5. **Compute x and y:** Using the substituted values, calculate x and y step by step:
   - Compute $\cos(\theta_1)$ and $\sin(\theta_1)$.
   - Compute $\cos(\theta_1 + \theta_2)$ and $\sin(\theta_1 + \theta_2)$.
   - Compute $\cos(\theta_1 + \theta_2 + \theta_3)$ and $\sin(\theta_1 + \theta_2 + \theta_3)$.
   - Add them together as per the equations.

   Example calculation:

$$x = 2 \times \cos\ \frac{\pi}{4}\ + 1.5 \times \cos\ \frac{\pi}{4} + \frac{\pi}{6}\ + 1 \times \cos\ \frac{\pi}{4} + \frac{\pi}{6} + \frac{\pi}{9}$$

$$y = 2 \times \sin\ \frac{\pi}{4}\ + 1.5 \times \sin\ \frac{\pi}{4} + \frac{\pi}{6}\ + 1 \times \sin\ \frac{\pi}{4} + \frac{\pi}{6} + \frac{\pi}{9}$$

6. **Plot the End-Effector Position:**
   - Once you have calculated x and y, plot these coordinates on a graph to visualize the end-effector position.

Conclusion:

You have successfully computed the end-effector position of the 3-DOF robot arm using forward kinematics.

Example Values for Practice:

- $L_1 = 2$ units, $L_2 = 1.5$ units, $L_3 = 1$ unit.
- $\theta_1 = 60^\circ, \theta_2 = 30^\circ, \theta_3 = 20^\circ$.

Inverse Kinematics of Robotic Arm – 3 Degrees of Freedom

Objective:

To compute the joint angles θ_1, θ_2, and θ_3 for a 3-DOF planar robot arm given the desired end-effector position and orientation, as well as the link lengths.

Theory:

Inverse kinematics allows us to determine the joint angles required to position the robot's end-effector at a specified location and orientation in space. For a 3-DOF planar robot, we calculate the joint angles θ_1, θ_2, and θ_3 given the position (x, y) and the orientation ϕ (the angle of the end-effector).

For a 3-DOF planar robot arm:

- **Link 1**: Length L_1.
- **Link 2**: Length L_2.
- **Link 3**: Length L_3 (end-effector link).
- **End-Effector Position**: x and y.
- **End-Effector Orientation**: ϕ (the angle the end-effector makes with the horizontal axis).

Procedure:

1. **Understand the System Setup**:
   - The robot has three links with known lengths: L_1, L_2, and L_3.
   - The desired position (x, y) and orientation ϕ of the end-effector are known.
   - The goal is to find the joint angles θ_1, θ_2, and θ_3 to place the end-effector at the specified position and orientation.
2. **Write Down the Equations**: The forward kinematics equations for the 3-DOF robot are:

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) + L_3 \cos(\theta_1 + \theta_2 + \theta_3)$$

$$y = L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2) + L_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

   In inverse kinematics, we solve these equations for θ_1, θ_2, and θ_3.
3. **Set the End-Effector Orientation**: The third joint angle θ_3 determines the orientation of the end-effector. The sum of all joint angles should equal the desired end-effector orientation:

$$\theta_1 + \theta_2 + \theta_3 = \phi$$

   Solve for θ_3:

$$\theta_3 = \phi - (\theta_1 + \theta_2)$$

4. **Compute θ_2**: Use the law of cosines to calculate θ_2. First, compute the distance from the origin to the desired end-effector position:

$$r = \sqrt{x^2 + y^2}$$

Then, apply the law of cosines:

$$\cos(\theta_2) = \frac{r^2 - L_1^2 - L_2^2}{2L_1L_2}$$

Solve for θ_2:

$$\theta_2 = \cos^{-1} \quad \frac{r^2 - L_1^2 - L_2^2}{2L_1L_2}$$

5. **Compute θ_1**: Use the law of cosines and geometry to calculate θ_1. First, compute the angle α between the vector from the origin to the end-effector and the x-axis:

$$\alpha = \tan^{-1} \quad \frac{y}{x}$$

Then, calculate the angle β using the law of cosines:

$$\beta = \cos^{-1} \quad \frac{L_1^2 + r^2 - L_2^2}{2L_1r}$$

Finally, compute θ_1:

$$\theta_1 = \alpha - \beta$$

6. **Substitute the Values**: Plug in the values for the desired position (x, y), the end-effector orientation ϕ, and the link lengths L_1, L_2, and L_3.

For example, consider the following values:

- $L_1 = 2$ units, $L_2 = 1.5$ units, $L_3 = 1$ unit.
- Desired end-effector position: $x = 3$ units, $y = 2$ units.
- Desired end-effector orientation: $\phi = 45^\circ = \frac{\pi}{4}$ radians.

7. **Compute the Angles**:
   - Calculate r, α, and β.
   - Compute θ_2 using the equation for θ_2.
   - Compute θ_1 using the equation for θ_1.
   - Compute θ_3 using $\theta_3 = \phi - (\theta_1 + \theta_2)$.

8. **Verify the Result:**
    - After calculating θ_1, θ_2, and θ_3, verify the result by plugging these values back into the forward kinematics equations.
    - Ensure that the computed end-effector position matches the desired (x, y), and the orientation matches ϕ.

Example:

Let's assume:

- $L_1 = 2$ units, $L_2 = 1.5$ units, $L_3 = 1$ unit.
- Desired end-effector position: $x = 3$ units, $y = 2$ units.
- Desired orientation: $\phi = 45^\circ$.

1. Compute r:

$$r = \sqrt{3^2 + 2^2} = \sqrt{13}$$

2. Compute θ_2:

$$\theta_2 = \cos^{-1} \quad \frac{13 - 2^2 - 1.5^2}{2 \times 2 \times 1.5}$$

3. Compute θ_1:

$$\alpha = \tan^{-1} \quad \frac{2}{3}$$

$$\beta = \cos^{-1} \quad \frac{2^2 + 13 - 1.5^2}{2 \times 2 \times \sqrt{13}}$$

$$\theta_1 = \alpha - \beta$$

4. Compute θ_3:

$$\theta_3 = \frac{\pi}{4} - (\theta_1 + \theta_2)$$

Conclusion:

You have successfully computed the joint angles θ_1, θ_2, and θ_3 to place the end-effector of the 3-DOF robot at a specific position and orientation using inverse kinematics.

Example Values for Practice:

- $L_1 = 2$ units, $L_2 = 1.5$ units, $L_3 = 1$ unit.
- Desired end-effector position: $x = 3$, $y = 2$, $\phi = 45^\circ$.

Dynamics of Robot arm 2 Degrees of freedom

Aim:

1. Compute the position of the end-effector using forward kinematics.
2. Calculate the velocity of the end-effector based on joint velocities.
3. Determine the forces at each joint based on external forces applied to the end-effector.
4. Visualize the results using plots.

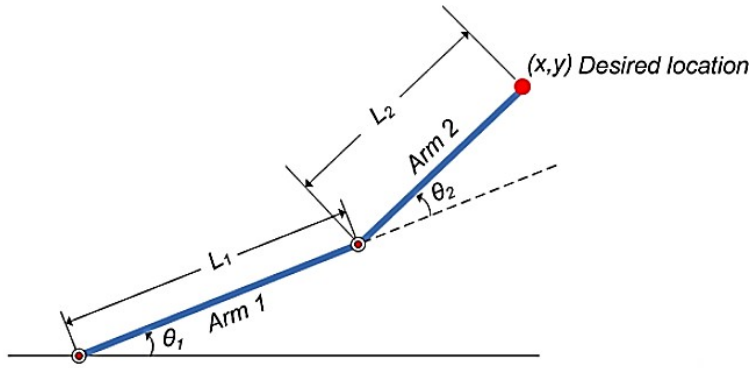


Figure 1: Illustration showing the two-joint robotic arm with the two angles, `theta1` and `theta2`

Procedure:

1. Understand the Problem

We need to analyze the velocity and force dynamics of a 2-DOF robotic arm. We'll calculate:

1. End-effector velocity given the joint velocities.
2. Joint forces given the external forces applied to the end-effector.

2. Define the Robot Parameters

Parameters:

- Length of the first link $l1$
- Length of the second link $l2$
- Joint angles θ_1 and θ_2

3. **Calculate Forward Kinematics**

Forward kinematics will help us find the position of the end-effector.

Formulas:

- $x = l1 \cdot \cos(\theta_1) + l2 \cdot \cos(\theta_1 + \theta_2)$
- $y = l1 \cdot \sin(\theta_1) + l2 \cdot \sin(\theta_1 + \theta_2)$

4. **Compute the Jacobian Matrix**

The Jacobian matrix relates the end-effector's linear velocity to the joint velocities.

Jacobian Matrix:

$$J = \begin{matrix} -l1 \cdot \sin(\theta_1) - l2 \cdot \sin(\theta_1 + \theta_2) & -l2 \cdot \sin(\theta_1 + \theta_2) \\ l1 \cdot \cos(\theta_1) + l2 \cdot \cos(\theta_1 + \theta_2) & l2 \cdot \cos(\theta_1 + \theta_2) \end{matrix}$$

5. **Calculate End-Effector Velocity**

Given joint velocities $\dot{\theta}_1$ and $\dot{\theta}_2$, calculate end-effector velocity.

Formulas:

- $\dot{x} = J_{11} \cdot \dot{\theta}_1 + J_{12} \cdot \dot{\theta}_2$
- $\dot{y} = J_{21} \cdot \dot{\theta}_1 + J_{22} \cdot \dot{\theta}_2$

6. **Compute Joint Forces**

Given external forces F_x and F_y, calculate forces at the joints using the inverse Jacobian matrix.

Formulas:

- $\begin{matrix} F_1 \\ F_2 \end{matrix} = J^{-1} \begin{matrix} F_x \\ F_y \end{matrix}$

7. **Plot the Results**

Plot the end-effector position, Jacobian matrix, and joint forces for visualization.

Sample Calculation

Given Data:

- Length of the first link, $l1 = 1$ meter
- Length of the second link, $l2 = 1$ meter
- Joint angles: $\theta_1 = \frac{\pi}{4}$ radians, $\theta_2 = \frac{\pi}{4}$ radians
- Joint velocities: $\dot{\theta}_1 = 0.1$ rad/s, $\dot{\theta}_2 = 0.2$ rad/s
- External forces: $F_x = 1.0$ N, $F_y = 0.5$ N

1. Forward Kinematics

Calculate the position of the end-effector:

$$x = 1 \cdot \cos\left(\frac{\pi}{4}\right) + 1 \cdot \cos\left(\frac{\pi}{4} + \frac{\pi}{4}\right) = 1 \cdot \frac{\sqrt{2}}{2} + 1 \cdot \frac{\sqrt{2}}{2} = \sqrt{2} \approx 1.414 \text{ meters}$$

$$y = 1 \cdot \sin\left(\frac{\pi}{4}\right) + 1 \cdot \sin\left(\frac{\pi}{4} + \frac{\pi}{4}\right) = 1 \cdot \frac{\sqrt{2}}{2} + 1 \cdot \frac{\sqrt{2}}{2} = \sqrt{2} \approx 1.414 \text{ meters}$$

2. Jacobian Matrix

Calculate the Jacobian matrix J:

$$J = \begin{matrix} -1 \cdot \sin\frac{\pi}{4} & -1 \cdot \sin\frac{\pi}{2} \\ 1 \cdot \cos\frac{\pi}{4} & +1 \cdot \cos\frac{\pi}{2} \end{matrix} \quad \begin{matrix} -1 \cdot \sin\frac{\pi}{2} \\ 1 \cdot \cos\frac{\pi}{2} \end{matrix} = \begin{matrix} -1.414 & -1 \\ 0.707 & 0 \end{matrix}$$

3. End-Effector Velocity

Compute the end-effector velocity:

$$\begin{matrix} \dot{x} \\ \dot{y} \end{matrix} = J \begin{matrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{matrix} = \begin{matrix} -1.414 & -1 \\ 0.707 & 0 \end{matrix} \begin{matrix} 0.1 \\ 0.2 \end{matrix} = \begin{matrix} -0.2414 \\ 0.0707 \end{matrix} \text{ meters/second}$$

4. Joint Forces

Compute the forces at the joints:

First, calculate the inverse Jacobian J^{-1}. Assuming it's calculated, use:

$$\begin{matrix} F_1 \\ F_2 \end{matrix} = J^{-1} \begin{matrix} F_x \\ F_y \end{matrix} = J^{-1} \begin{matrix} 1.0 \\ 0.5 \end{matrix}$$

5. Plot Results

Use plotting functions to visualize:

- End-effector position
- Jacobian matrix
- Joint forces

```python
import numpy as np
import matplotlib.pyplot as plt

def inverse_kinematics(x, y, theta_total, l1, l2):
    """
    Calculates the joint angles for a 2-DOF robot arm to reach a given target point.

    Args:
        x: The x-coordinate of the target point.
        y: The y-coordinate of the target point.
        theta_total: The total angle of the end-effector with respect to the base.
        l1: Length of the first link.
        l2: Length of the second link.

    Returns:
        A tuple containing the joint angles (theta1, theta2).
    """
    # Calculate the distance from the origin to the target point
    d = np.sqrt(x**2 + y**2)

    # Check if the target is reachable
    if d > l1 + l2 or d < abs(l1 - l2):
        raise ValueError("Target is out of reach for the robot arm.")

    # Calculate the angle between the x-axis and the line connecting the origin to the target point
    alpha = np.arctan2(y, x)

    # Calculate the angle between the first link and the line connecting the origin to the target point
    beta = np.arccos((l1**2 + d**2 - l2**2) / (2 * l1 * d))

    # Calculate theta1 using the given total angle
    theta1 = alpha - beta

    # Calculate theta2 based on the total angle of the end-effector with respect to the base
    theta2 = theta_total - theta1

    return theta1, theta2

# Get input from the user
x = float(input("Enter the x-coordinate of the target point: "))
y = float(input("Enter the y-coordinate of the target point: "))
theta_total_deg = float(input("Enter the total angle of the end-effector with respect to the base (in degrees): "))

# Convert theta_total to radians
theta_total = np.radians(theta_total_deg)
```

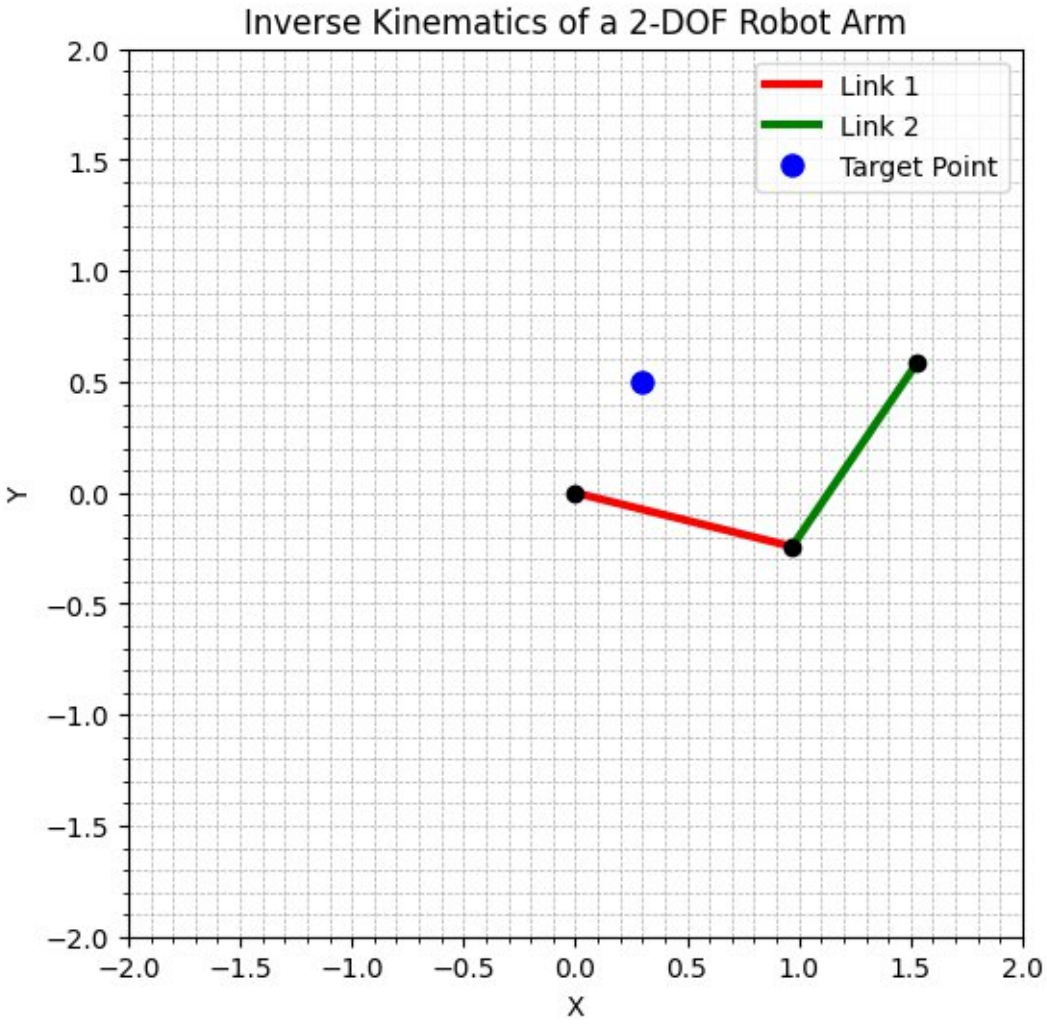
```
# Link lengths
l1 = 1  # Length of the first link
l2 = 1  # Length of the second link

# Calculate the joint angles
try:
    theta1, theta2 = inverse_kinematics(x, y, theta_total, l1, l2)

    # Convert the joint angles to degrees for easier interpretation
    theta1_deg = np.degrees(theta1)
    theta2_deg = np.degrees(theta2)

    # Display the results
    print(f"Joint angle θ1: {theta1_deg:.2f} degrees")
    print(f"Joint angle θ2: {theta2_deg:.2f} degrees")

    # Calculate the position of each joint
    x1 = l1 * np.cos(theta1)
    y1 = l1 * np.sin(theta1)
    x2 = x1 + l2 * np.cos(theta_total)
    y2 = y1 + l2 * np.sin(theta_total)

    # Display the joint positions
    print(f"Position of joint 1: ({x1:.2f}, {y1:.2f})")
    print(f"Position of end-effector: ({x2:.2f}, {y2:.2f})")

    # Plot the robot arm
    plt.figure(figsize=(6, 6))
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.minorticks_on()
    plt.plot([0, x1], [0, y1], 'r-', linewidth=3, label='Link 1')  # First link
    plt.plot([x1, x2], [y1, y2], 'g-', linewidth=3, label='Link 2')  # Second link
    plt.plot(x, y, 'bo', markersize=8, label='Target Point')  # Target point
    plt.scatter([0, x1, x2], [0, y1, y2], c='k', zorder=5)  # Joint positions
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Inverse Kinematics of a 2-DOF Robot Arm')
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.xticks(np.arange(-2, 2.5, 0.5))
    plt.yticks(np.arange(-2, 2.5, 0.5))
    plt.legend()
    plt.show()

except ValueError as e:
    print(e)
```

```python
import numpy as np
import matplotlib.pyplot as plt

def inverse_kinematics(x, y, theta_total, l1, l2):
    """
    Calculates the joint angles for a 2-DOF robot arm to reach a given target
point.

    Args:
        x: The x-coordinate of the target point.
        y: The y-coordinate of the target point.
        theta_total: The total angle of the end-effector with respect to the base.
        l1: Length of the first link.
        l2: Length of the second link.

    Returns:
        A tuple containing the joint angles (theta1, theta2).
    """
    # Calculate the distance from the origin to the target point
    d = np.sqrt(x**2 + y**2)

    # Check if the target is reachable
    if d > l1 + l2 or d < abs(l1 - l2):
        raise ValueError("Target is out of reach for the robot arm.")

    # Calculate the angle between the x-axis and the line connecting the origin to
the target point
    alpha = np.arctan2(y, x)

    # Calculate the angle between the first link and the line connecting the
origin to the target point
    beta = np.arccos((l1**2 + d**2 - l2**2) / (2 * l1 * d))

    # Calculate theta1 using the given total angle
    theta1 = alpha - beta

    # Calculate theta2 based on the total angle of the end-effector with respect
to the base
    theta2 = theta_total - theta1

    return theta1, theta2

# Get input from the user
x = float(input("Enter the x-coordinate of the target point: "))
y = float(input("Enter the y-coordinate of the target point: "))
theta_total_deg = float(input("Enter the total angle of the end-effector with
respect to the base (in degrees): "))

# Convert theta_total to radians
theta_total = np.radians(theta_total_deg)
```

```
# Link lengths
l1 = 1  # Length of the first link
l2 = 1  # Length of the second link

# Calculate the joint angles
try:
    theta1, theta2 = inverse_kinematics(x, y, theta_total, l1, l2)

    # Convert the joint angles to degrees for easier interpretation
    theta1_deg = np.degrees(theta1)
    theta2_deg = np.degrees(theta2)

    # Display the results
    print(f"Joint angle θ1: {theta1_deg:.2f} degrees")
    print(f"Joint angle θ2: {theta2_deg:.2f} degrees")

    # Calculate the position of each joint
    x1 = l1 * np.cos(theta1)
    y1 = l1 * np.sin(theta1)
    x2 = x1 + l2 * np.cos(theta_total)
    y2 = y1 + l2 * np.sin(theta_total)

    # Display the joint positions
    print(f"Position of joint 1: ({x1:.2f}, {y1:.2f})")
    print(f"Position of end-effector: ({x2:.2f}, {y2:.2f})")

    # Plot the robot arm
    plt.figure(figsize=(6, 6))
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.minorticks_on()
    plt.plot([0, x1], [0, y1], 'r-', linewidth=3, label='Link 1')  # First link
    plt.plot([x1, x2], [y1, y2], 'g-', linewidth=3, label='Link 2')  # Second link
    plt.plot(x, y, 'bo', markersize=8, label='Target Point')  # Target point
    plt.scatter([0, x1, x2], [0, y1, y2], c='k', zorder=5)  # Joint positions
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Inverse Kinematics of a 2-DOF Robot Arm')
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.xticks(np.arange(-2, 2.5, 0.5))
    plt.yticks(np.arange(-2, 2.5, 0.5))
    plt.legend()
    plt.show()

except ValueError as e:
    print(e)
```

OUTPUT:

```
Enter the x-coordinate of the target point: 0.3
Enter the y-coordinate of the target point: 0.5
Enter the total angle of the end-effector with respect to the base (in degrees):
56
Joint angle θ1: -14.01 degrees
Joint angle θ2: 70.01 degrees
Position of joint 1: (0.97, -0.24)
  Position of end-effector: (1.53, 0.59)
```



RESULT

The position of the end-effector of a 2-DOF and 3 – DOF planar robot arm given the joint angles and link lengths are solved successfully.

Ex. No: 7 Develop and simulate path planning algorithms to navigate an articulated robot through predefined trajectories

AIM:

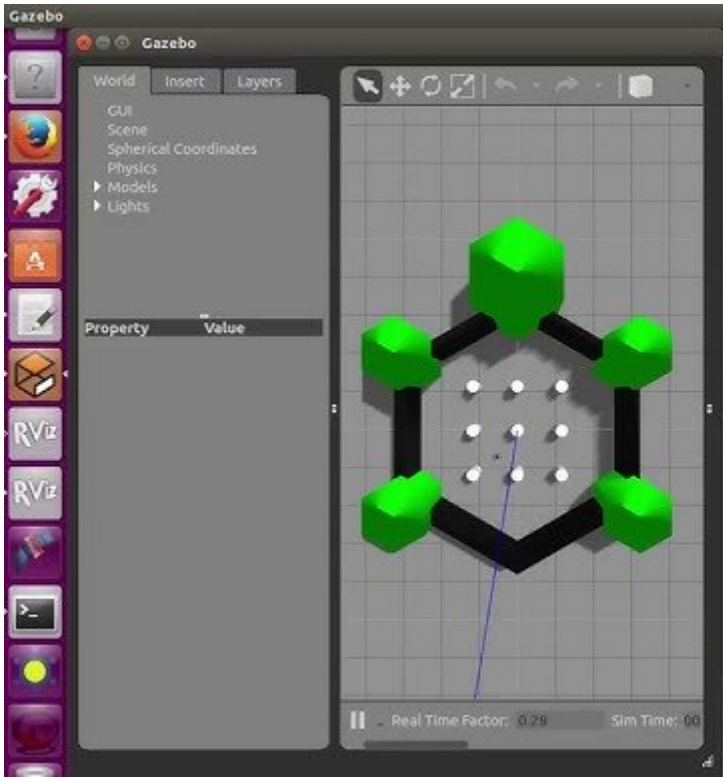
To simulate Simultaneous Localization and Mapping (SLAM) and navigate the TurtleBot in a virtual environment using the Robot Operating System (ROS).

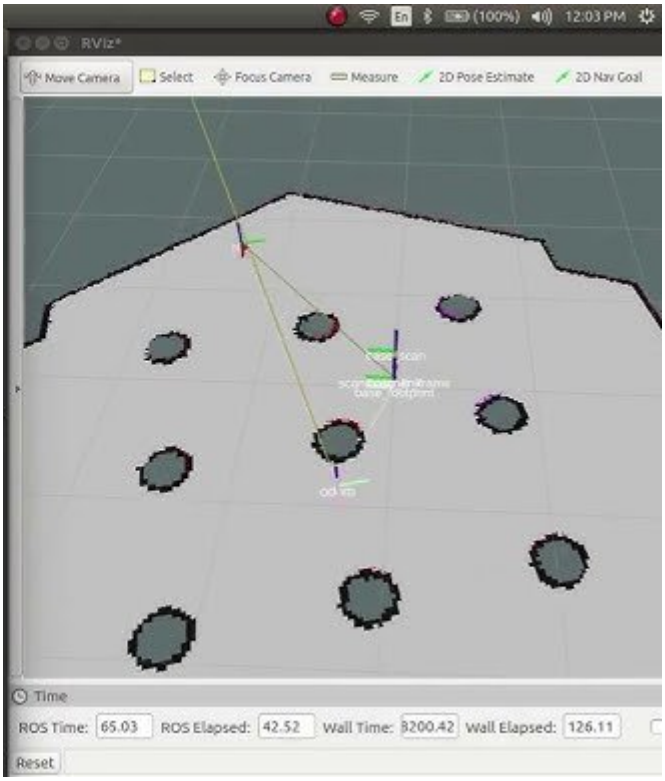
ALGORITHM / STEPS:

1. Configure the TurtleBot3 model type to ensure compatibility with simulation and navigation nodes.
2. Launch the Gazebo simulation with TurtleBot3 in a predefined virtual environment.
3. Start the SLAM node using the gmapping method to enable the TurtleBot to map its surroundings as it moves.
4. Open RViz to visualize the SLAM process and observe the map being created in real time as the TurtleBot navigates the environment.
5. Use teleoperation to manually control the TurtleBot, moving it throughout the environment to capture all areas needed for mapping.
6. Once the map is complete, save the generated map for future navigation by exporting the map data to a file.
7. Close any active SLAM processes to avoid conflicts with the navigation process.
8. Launch the navigation node with the saved map, enabling the TurtleBot to use it for autonomous navigation.

In RViz, set a target destination by selecting a goal pose on the map, allowing the TurtleBot to navigate to the specified point autonomously

OUTPUT:





RESULT

The Teleoperation and SLAM in the Gazebo simulation, creating a 2D map of the environment is done successfully.

Ex. No: 8 Analyze the velocity and force dynamics of a robotic arm using Robo Analyzer to optimize performance and efficiency

AIM:

To analyze the force and torque requirements at each joint of a 2-DOF robotic arm under varying conditions.

ALGORITHM / STEPS:

1. **Define System Parameters**
   - Assign link lengths L_1 and L_2, and masses m_1 and m_2.
   - Specify payload mass m_p attached at the end-effector.
   - Note the gravitational acceleration $g = 9.81\, m/s^2$.
2. **Model the Robot Configuration**
   - Use the Denavit–Hartenberg (DH) convention to represent each joint and link.
   - Define joint variables θ_1 and θ_2 as revolute joints.
   - Compute the forward kinematics to determine the end-effector position:

$$x = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$y = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

3. **Assign Motion Conditions**
   - Choose specific values of joint angles θ_1, θ_2 for static conditions.
   - For dynamic conditions, define angular velocities $\dot{\theta}_1, \dot{\theta}_2$ and accelerations $\ddot{\theta}_1, \ddot{\theta}_2$.
4. **Develop the Dynamic Model**
   - Write the Lagrangian function:

$$L = K - P$$

   where K = total kinetic energy, P = potential energy.
   - Apply Lagrange's equation for each joint:

$$\tau_i = \frac{d}{dt}\left(\frac{\partial K}{\partial \dot{\theta}_i}\right) - \frac{\partial K}{\partial \theta_i} + \frac{\partial P}{\partial \theta_i}$$

   - Determine τ_1 and τ_2 (torques at joints 1 and 2).
5. **Simulate Using RoboAnalyzer (or MATLAB)**
   - Open **RoboAnalyzer → Dynamics module**.
   - Enter the link parameters (lengths, masses, and inertia).
   - Define motion profile (joint angles vs. time).
   - Run the simulation to obtain:
     - Joint torques vs. time
     - Angular velocity and acceleration curves
     - End-effector force data
6. **Record and Analyze the Results**
   - Note maximum torque at each joint.
   - Compare results for different payloads (e.g., 0.5 kg, 1 kg).
   - Plot torque-time graphs and interpret which joint requires higher torque.
   - Suggest actuator ratings based on maximum torque values.

PROGRAM:

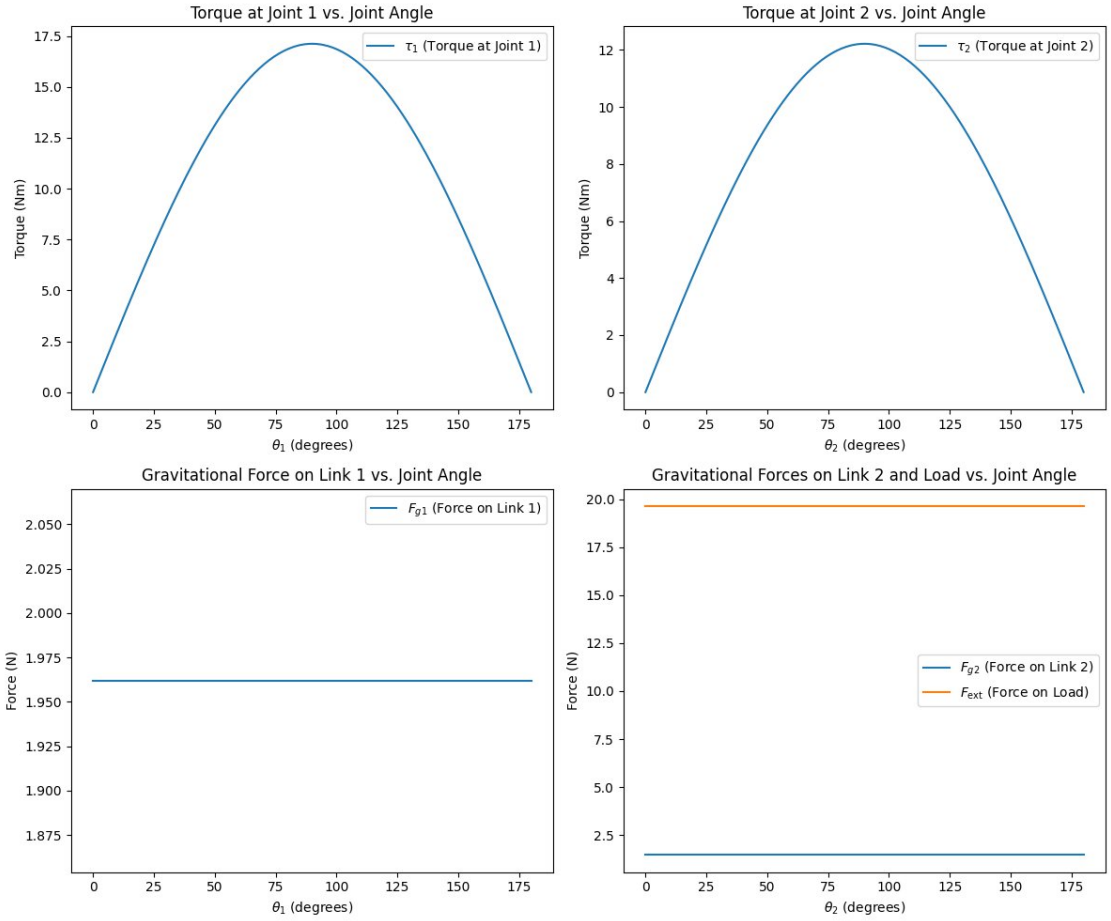
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
g = 9.81  # Gravity in m/s^2

# Input Parameters
m1 = float(input("Enter mass of Link 1 (kg): "))
m2 = float(input("Enter mass of Link 2 (kg): "))
m_load = float(input("Enter mass of Load (kg): "))
L1 = float(input("Enter length of Link 1 (m): "))
L2 = float(input("Enter length of Link 2 (m): "))

# Range for theta values for plotting (in degrees)
theta1_range = np.linspace(0, 180, 180)
theta2_range = np.linspace(0, 180, 180)

# Initialize lists to store results for plotting
tau1_values = []
tau2_values = []
Fg1_values = []
Fg2_values = []
F_ext_values = []

# Function to calculate forces and torques
def calculate_forces_and_torques(theta1, theta2):
    # Convert angles to radians
    theta1_rad = np.radians(theta1)
    theta2_rad = np.radians(theta2)

    # Gravitational forces for each link and load
    Fg1 = m1 * g
    Fg2 = m2 * g
    F_ext = m_load * g

    # Torque at Joint 2
    tau2 = Fg2 * (L2 / 2) * np.sin(theta2_rad) + F_ext * L2 *
np.sin(theta2_rad)

    # Torque at Joint 1
    tau1 = (
        Fg1 * (L1 / 2) * np.sin(theta1_rad)
        + Fg2 * L1 * np.sin(theta1_rad)
        + Fg2 * (L2 / 2) * np.sin(theta1_rad + theta2_rad)
        + F_ext * (L1 + L2) * np.sin(theta1_rad + theta2_rad)
    )
```

```
        return Fg1, Fg2, F_ext, tau1, tau2

    # Calculate forces and torques for a range of angles
    for theta1 in theta1_range:
        for theta2 in theta2_range:
            Fg1, Fg2, F_ext, tau1, tau2 = calculate_forces_and_torques(theta1,
    theta2)
            Fg1_values.append(Fg1)
            Fg2_values.append(Fg2)
            F_ext_values.append(F_ext)
            tau1_values.append(tau1)
            tau2_values.append(tau2)

    # Convert lists to arrays for plotting
    tau1_values = np.array(tau1_values)
    tau2_values = np.array(tau2_values)
    Fg1_values = np.array(Fg1_values)
    Fg2_values = np.array(Fg2_values)
    F_ext_values = np.array(F_ext_values)

    # Plotting Force and Torque vs Joint Angles
    plt.figure(figsize=(12, 10))

    # Plot Tau1 vs Theta1
    plt.subplot(2, 2, 1)
    plt.plot(theta1_range, tau1_values[:len(theta1_range)],
    label=r'$\tau_1$ (Torque at Joint 1)')
    plt.xlabel(r'$\theta_1$ (degrees)')
    plt.ylabel('Torque (Nm)')
    plt.title('Torque at Joint 1 vs. Joint Angle')
    plt.legend()

    # Plot Tau2 vs Theta2
    plt.subplot(2, 2, 2)
    plt.plot(theta2_range, tau2_values[:len(theta2_range)],
    label=r'$\tau_2$ (Torque at Joint 2)')
    plt.xlabel(r'$\theta_2$ (degrees)')
    plt.ylabel('Torque (Nm)')
    plt.title('Torque at Joint 2 vs. Joint Angle')
    plt.legend()

    # Plot Force vs Theta1 for Link 1
    plt.subplot(2, 2, 3)
    plt.plot(theta1_range, Fg1_values[:len(theta1_range)],
    label=r'$F_{g1}$ (Force on Link 1)')
    plt.xlabel(r'$\theta_1$ (degrees)')
    plt.ylabel('Force (N)')
    plt.title('Gravitational Force on Link 1 vs. Joint Angle')
```

```
    plt.legend()

    # Plot Force vs Theta2 for Link 2 and Load
    plt.subplot(2, 2, 4)
    plt.plot(theta2_range, Fg2_values[:len(theta2_range)],
    label=r'$F_{g2}$ (Force on Link 2)')
    plt.plot(theta2_range, F_ext_values[:len(theta2_range)],
    label=r'$F_{\text{ext}}$ (Force on Load)')
    plt.xlabel(r'$\theta_2$ (degrees)')
    plt.ylabel('Force (N)')
    plt.title('Gravitational Forces on Link 2 and Load vs. Joint Angle')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

OUTPUT:

```
Enter mass of Link 1 (kg): 0.2
Enter mass of Link 2 (kg): 0.15
Enter mass of Load (kg): 2
Enter length of Link 1 (m): 0.25
Enter length of Link 2 (m): 0.6
```



RESULT

The force and torque analysis of the 2- DOF robotic arm was successfully.

Ex. No: 9 Apply linkage synthesis techniques to design a pick-and-place robotic arm for efficient material handling

AIM:

The objective is to synthesize of Pick and Place Robot Arm

1. Introduction to Linkages:

Linkages are assemblies of rigid bodies connected by joints to form a closed chain or a series of open chains. They are widely used in robotic systems to transmit motion and force. Some common types of linkage mechanisms used in robots include:

- **Four-Bar Linkage**: A mechanism used to convert rotational motion into linear or other types of motion.
- **Slider-Crank Mechanism**: Used to convert rotational motion into reciprocating motion.
- **Gripper Mechanism**: A mechanism used to grasp and hold objects.

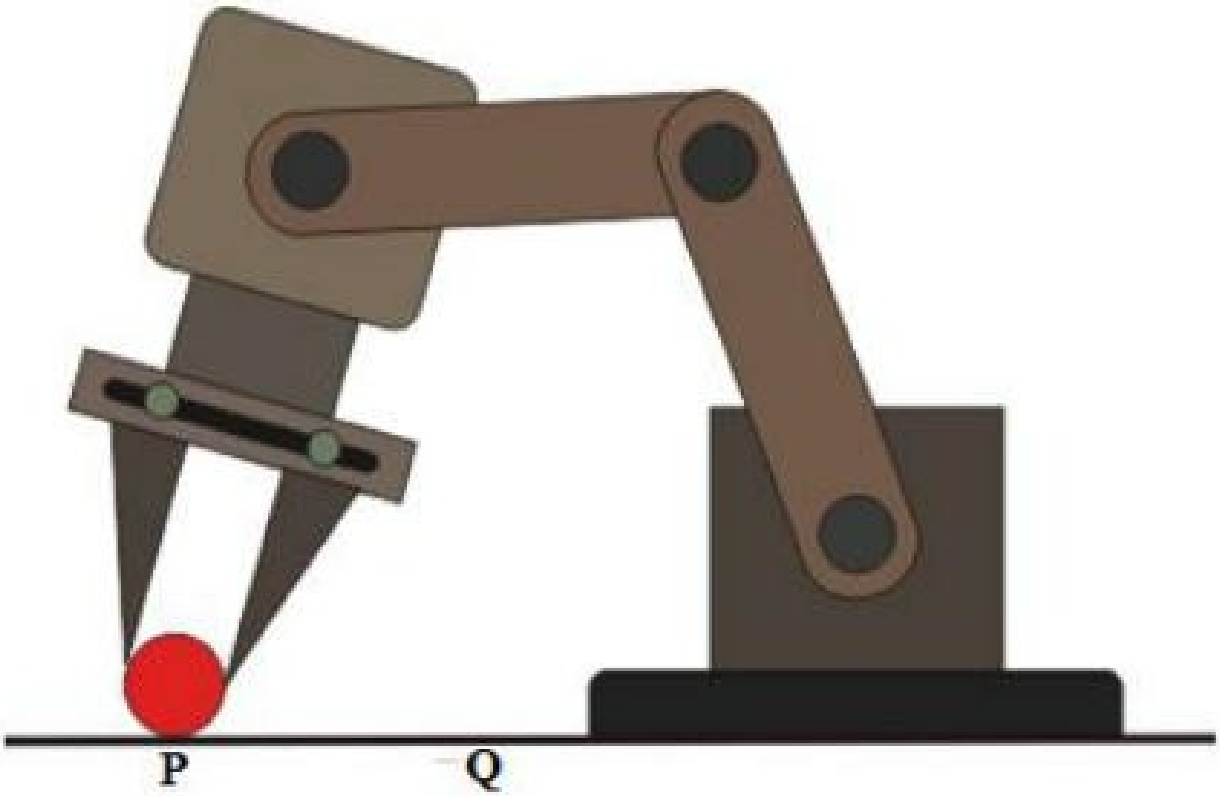
2. Basic Definitions:

- **Links**: Rigid members of a mechanism.
- **Joints**: Connections between links allowing motion (e.g., revolute or prismatic).
- **Degrees of Freedom (DOF)**: The number of independent motions allowed in a system.
- **Crank, Coupler, and Follower**: Components of the four-bar mechanism.

3. Tools and Software:

- **Linkage Software**: This software provides a platform to design, simulate, and analyze the motion of various mechanical linkages.
- **Computer**: Installed with Linkage Software.

Note: You can download and install Linkage Software from Linkage Download Page.



Pick and Place Arm

RESULT

Thus, the simulate linkage mechanisms to evaluate their kinematic performance and positional accuracy is executed successfully.

Ex. No: 10 Explore and utilize electronic repositories to source ROS packages and tools for developing autonomous robotic systems

AIM:

To explore and utilize online electronic repositories such as **ROS Index**, **GitHub**, and **ROS Wiki** to identify, evaluate, and integrate ROS packages and tools required for developing autonomous robotic systems.

ALGORITHM / STEPS:

Step 1 – Identify Functional Requirements

1. Define the autonomous system objectives (e.g., SLAM, navigation, perception, control).
2. Determine what type of ROS packages are needed:
    - **Mapping & Localization:** gmapping, hector_slam, cartographer
    - **Navigation & Planning:** move_base, nav2, global_planner, local_planner
    - **Perception:** realsense_ros, lidar_driver, camera_info_manager
    - **Simulation:** gazebo_ros, stage_ros, ignition_ros

Step 2 – Explore ROS Index and GitHub

1. Open ROS Index.
2. Search for required packages (e.g., "SLAM", "LIDAR", "Navigation").
3. Review each package:
    - Check **ROS distribution compatibility** (Noetic, Humble, etc.)
    - Read **documentation, dependencies, and tutorials**.
4. Visit the corresponding **GitHub repository**:
    - Examine the **README.md** for usage instructions.
    - Check **issues**, **commit activity**, and **license** (BSD preferred).
    - Clone repository for testing.

Step 3 – Clone and Build the Package

Example: Installing and testing the **gmapping** package.

Step 4 – Explore ROS Tools

Step 5 – Document and Maintain Packages

1. Record the source URL, version, and purpose of each integrated package.
2. Create a `package_dependencies.txt` or `requirements.rosinstall` file.
3. Use Git for version control of your customized packages.
4. Maintain README and metadata (`package.xml`) for easy re-deployment.

PROGRAM:

```
# Navigate to your workspace
cd ~/catkin_ws/src

# Clone the package from GitHub
git clone https://github.com/ros-perception/slam_gmapping.git

# Check dependencies
cd ~/catkin_ws
rosdep install --from-paths src --ignore-src -r -y

# Build and source
catkin_make
source devel/setup.bash

# Verify installation
roslaunch gmapping slam_gmapping_pr2.launch

rospack list
roscd <package_name>
rospack depends1 <package_name>

rostopic list
rosnode list
rosrun rqt_graph rqt_graph
```

OUTPUT :

Cloning into 'slam_gmapping'...

Resolving dependencies...

[100%] Built target slam_gmapping

ROS Master started at http://localhost:11311

[INFO] [slam_gmapping]: Map updates published at /map

RESULT

The integrated ROS packages and tools from online repositories. The identified packages (e.g., `slam_gmapping`, `move_base`, `turtlebot3_navigation`) were built and executed in ROS, demonstrating the ability to source, evaluate, and utilize community-maintained software for autonomous robot development.