SUMMER OF SCIENCE 2025

END-TERM REPORT

ROBOTICS

Name: Umesh Motiwale

Mentor: Maitranya Patil

Contents

- 1 Introduction to Robotics 4
  - 1.1 What is a Robot? 4
  - 1.2 What is Robotics 4
  - 1.3 The Foundations and goals of Robotics 4
- 2 Robot Configuration 5
  - 2.1 Configuration of a Robot 5
  - 2.2 Degrees of Freedom 5
    - 2.2.1 Mobility/Dof of spatial manipulator 5
    - 2.2.2 Mobility/Dof of Planar manipulator 5
    - 2.2.3 Robot joints 5
- 3 Robot kinematics 6
  - 3.1 Operator 6
    - 3.1.1 Translational Operator 6
    - 3.1.2 Rotational Operator 6
    - 3.1.3 Composite Rotation Matrix 7
  - 3.2 Forward Kinematics 7
  - 3.3 Denavit - Hartenberg convention 7
  - 3.4 Inverse Kinematics 7
  - 3.5 Trajectory Planning 7
    - 3.5.1 Polynomial Trajectory (cubic) 8
- 4 Robot Dynamics 9
  - 4.1 Forward Dynamics 9
  - 4.2 Inverse Dynamics 9
  - 4.3 Lagrangian Method for determination of Robotic Joint Torque 9
  - 4.4 Total Kinetic and Potential Energy of a Serial Manipu- lator 9
- 5 Control Scheme 11
  - 5.1 Sensors 11
    - 5.1.1 Various types of sensors 11
  - 5.2 Robot Vision 12
  - 5.2.1 Methods of pre-processing 12
  - 5.2.2 Lapalace Operator 13
- 6 Robot Motion planning 14
  - 6.1 Artificial Potential Field Method 14

* 6.1.1 Attractive Potential 14
* 6.1.2 Replusive Potential 14

• 7 Intelligent Robot 15

– 7.1 Biped Walking 15
* 7.1.1 Walking Cycle 15
* 7.1.2 Staircase ascending 15
* 7.1.3 Dynamic balance margin 15

• 8 Robot Operating System 16

– 8.1 Introduction 16
– 8.2 Important Terms 16
* 8.2.1 Roscore 16
* 8.2.2 ROS Graph 16
* 8.2.3 Catkin,workspaces, ROS packages 16
* 8.2.4 Rosrun 16
– 8.3 Topics 17
* 8.3.1 Publishing to a topic 17
* 8.3.2 Subscribing to a topic 17
– 8.4 ROS Service 18

– 8.4.1 Turtlesim: Change pen Color using ROS Service 18

• 9 Webots Simulation 20

– 9.1 Introduction 20
* 9.1.1 What is Webots? 20
* 9.1.2 What is a World? 20
* 9.1.3 What is a Controller? 20
* 9.1.4 What is a Supervisor Controller? 20
* 9.1.5 Scene Tree 20
* 9.1.6 Nodes and Fields 20
– 9.2 Simulations 21
* 9.2.1 E-puck go forward 21
* 9.2.2 E-puck avoid collision 21
* 9.2.3 4 Wheeled Robot 22
– 9.3 Moving Objects using a supervisor 23

1 Introduction to Robotics

1.1 What is a Robot?

A robot is an automatically controlled programmable multipurpose machine which carry out a series of actions or tasks, typically by being able to sense its environment, make decisions, and physically move or manipulate objects. Robots are often composed of mechanical structures-links and joints, actuators , sensors, controllers, and software.

1.2 What is Robotics?

Robotics is the branch of engineering and science concerned with the design, construction, operation, and application of robots. It encompasses the study of mechanical systems, kinematics, dynamics, control theory, artificial intelligence, and computer vision to enable autonomous or semi-autonomous behavior in machines.

1.3 The Foundations and Goals of Robotics

Robotics is a relatively young yet ambitious academic field with the ultimate goal of creating machines that can perceive, act, and even think like humans. This pursuit naturally leads us to reflect on our own physical and cognitive design how we move, coordinate our limbs, and learn to perform tasks.

At its core, robotics is about constructing mechanisms rigid bodies connected by joints capable of controlled motion. These mechanisms are typically powered by actuators such as electric motors, hydraulic cylinders, or pneumatic drives. Most modern robots use speed reduction devices like gears or pulleys to amplify torque and reduce backlash.

Robots are equipped with sensors to measure joint positions, velocities, forces, and external surroundings. Examples include encoders, tachometers, RGB-D cameras, LiDAR, and force–torque sensors.

While fields like artificial intelligence and perception contribute to robotics, a key characteristic of any robot is its physical interaction with the real world. Therefore, this report focuses on the mechanics, motion planning, and control that enable robots to function as intelligent, physical agents.

2 Robot Configuration and C-Space:

2.1 Configuration of a Robot

A robot's configuration refers to a complete specification of the positions and orientations of all its components. Since specifying the position of every point on the robot is complex, we instead use a minimal number of parameters—like joint angles or displacements—to describe its pose. These parameters form the basis for understanding and controlling the robot's motion.

2.2 Degrees of Freedom (DoF)

The Degrees of Freedom (DoF) represent the number of independent parameters required to define the configuration of the robot. A rigid body in three-dimensional space has six DoF:

- 3 translational: movement along x, y, and z axes
- 3 rotational: roll, pitch, and yaw

2.2.1 Mobility/DoF of Spatial Manipulator

Consider a manipulator with 'n' rigid moving links and 'm' joints
C_i: Connectivity of i^{th} joint, i= 1,2,3,....,m
Number of constraints put by the i^{th} joint = (6- C_i)
Total number of constraints = $\sum_{i=1}^{m}(6 - C_i)$
Mobility of the Manipulator M = 6n - $\sum_{i=1}^{m}(6 - c_i)$

It is known as Grubler's criterion.

2.2.2 Mobility/DoF of Planar Manipulator

Consider a manipulator with 'n' rigid moving links and 'm' joints
C_i: Connectivity of i^{th} joint, i= 1,2,3,....,m
Number of constraints put by the i^{th} joint = (3 - C_i)
Total number of constraints = $\sum_{i=1}^{m}(3 - C_i)$
Mobility of the Manipulator M = 3n - $\sum_{i=1}^{m}(3 - c_i)$

2.2.3 Robot Joints

Joints are the physical connections between links that allow relative motion. Every joint con nects exactly two links; joints that simultaneously connect three or more links are not allowed. Information about more joints are listed in the following table-

Joint Name	Symbol	DoF	Visual
Revolute Joint	R	1	
Prismatic Joint	P	1	
Cylindrical Joint	C	2	
Helical Joint	H	1	
Spherical Joint	S	3	
Universal Joint	U	2	

Table 1: Types of Joints in Robotics

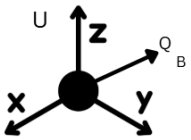
3 Robot Kinematics

Robot kinematics is the study of motion without considering the forces that cause it. It includes the analysis of a robot's position, orientation, velocity, and how these relate to the movement of its joints.

Representation of the Position

$${}^{U}Q = \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

3x1 matrix



Representation of the Orientation

$${}^{U}_{B}R = \begin{bmatrix} {}^{U}X_{B} & {}^{U}Y_{B} & {}^{U}Z_{B} \end{bmatrix}$$

1x3 matrix

Let [T]: Hoogeneous Transformation matrix

$$A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & q_x \\ r_{21} & r_{22} & r_{23} & q_y \\ r_{31} & r_{32} & r_{33} & q_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.1 Operator

3.1.1 Translational Operator

$$Trans(X, q) = \text{Translation of q units along x direction} = \begin{bmatrix} 1 & 0 & 0 & q \\ 0 & 1 & 0 & o \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.1.2 Rotational Operator

$$Rot(X, \theta) = \text{Translation of q units along x direction} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

3.1.3 Composite Rotation Matrix

Composite rotation matrix representing a rotation of angle α about z axis, followed by β about y, and angle γ about x axis

$ROT_{composite}$ = ROT(x,γ).ROT(y,β).ROT(z,α)

3.2 Forward Kinematics

Forward Kinematics is the process of computing the position and orientation of a robot's end-effector from the given joint parameters. For an n-link manipulator, the total transformation from base to end-effector is:
T = $T_1.T_2.T_3....T_n$
Each $T_i \in \mathbb{R}^{4\times4}$ is a homogeneous transformation matrix representing rotation and translation of link i.

3.3 Denavit–Hartenberg (DH) Convention

The DH convention standardizes the representation of robot links using 4 parameters per link:

- Length of link(a_i)
- Angle of twist (α_i)
- Offset of link (d_i)
- Joint Angle (θ_i)

$$ {}^{i-1}_{i}T = {}^{i-1}_{A}T \cdot {}^{A}_{B}T \cdot {}^{B}_{C}T \cdot {}^{C}_{i}T $$

$$ {}^{i-1}_{i}T = ROT(z,\theta_i).TRANS(z,d_i).ROT(x,\alpha_i).TRANS(x,a_i) $$

3.4 Inverse Kinematics

Inverse Kinematics is the process of calculating the joint parameters (e.g., angles or displacements) required to place the robot's end-effector at a desired position and orientation.
Given the desired end-effector pose:

$$ T_{\text{desired}} = \begin{bmatrix} R_0 & d_1 \\ 0 & 1 \end{bmatrix} $$

we solve for joint variables θ_1,θ_2,....,θ_n such that

$$ T(\theta_1, \theta_2, \ldots, \theta_n) = T_{\text{desired}} $$

3.5 Trajectory Planning

Trajectory Planning is the process of determining a time-dependent path for the robot's end-effector or joints to follow, ensuring smooth, continuous, and feasible motion between two or more specified positions. It involves generating a function:

$$ q(t),\ \dot{q}(t),\ \ddot{q}(t) $$

where q(t) is the position and accordingly derivatives

Its two types: Cartesian scheme and Joint-space scheme

3.5.1 Polynomial Trajectory (Cubic

A common approach is using a cubic polynomial to interpolate between two points: q(t) = $a_o + a_1$t + $a_2 t^2 + a_3 t^3$

where coefficients a_i are found using boundary conditions like:
$q(t_0) = q_0,\ q(t_f) = q_f,\ \dot{q}(t_0) = 0,\ \dot{q}(t_f) = 0$

4 Robot Dynamics

Robot dynamics describes how a manipulator moves under applied forces and torques, relating joint positions, velocities, and accelerations to actuator efforts and external loads. It encompasses two primary problems:

4.1 Forward Dynamics

Given joint torques $\tau(t)$, compute the resulting joint accelerations $\ddot{q}(t)$. The dynamic equation of motion is:

$$M(q)\ddot{q} + V(q, \dot{q}) + G(q) = \tau$$

- $M(q)$: Inertia matrix
- $V(q, \dot{q})$: Coriolis and centrifugal terms
- $G(q)$: Gravity-induced torques
- τ: Vector of joint torques/forces

4.2 Inverse Dynamics

Given a desired trajectory $(q, \dot{q}, \ddot{q})$, compute the joint torques τ required to achieve that motion:

$$\tau = M(q)\ddot{q} + V(q, \dot{q}) + G(q)$$

4.3 Lagrangian Method for determination of Robotic Joint Torque

Based on the Lagrangian function:

$$L = T - U$$

where T is the kinetic energy and U is the potential energy.

The dynamics are derived using:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i$$

where L: Lagrange Function, q_i = generalized coordinates, τ= force for linear joint and torque for rotary joint.

4.4 Total Kinetic and Potential Energy of a Serial Manipulator

The total kinetic energy T of a serial manipulator with n links is given by:

$$T = \sum_{i=1}^{n} \left(\frac{1}{2} m_i v_i^T v_i + \frac{1}{2} \omega_i^T I_i \omega_i\right)$$

where:

- m_i is the mass of link i

- v_i is the linear velocity of the center of mass of link i
- ω_i is the angular velocity of link i
- I_i is the inertia tensor of link i about its center of mass

The total potential energy U due to gravity is given by:

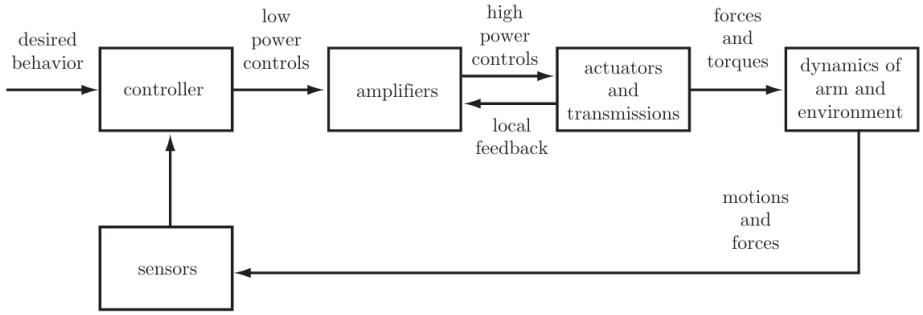
$$U = \sum_{i=1}^{n} m_i g^T h_i$$

where:

- g is the gravitational acceleration vector
- h_i is the position vector of the center of mass of link i

5 Control Scheme

The control scheme in robotics defines how a robot translates desired tasks into actual movement and force. It involves generating commands for the robot's joints or actuators to ensure precise positioning, motion, or interaction with the environment. Effective control allows robots to follow trajectories, maintain stability, and apply specific forces during tasks like assembly, welding, or object manipulation. Depending on the task, control may focus on position, velocity, torque, or a combination of motion and force.



5.1 Sensors

The control scheme in robotics defines how a robot executes desired motions or interactions with its environment by generating appropriate commands for its actuators. It is essential for tasks requiring precision, such as object manipulation, movement along specific paths, or applying controlled forces. Control can be based on various inputs, including desired position, velocity, or force, and is typically implemented using feedback loops to correct for errors in real-time.

5.1.1 Various Types of Sensors

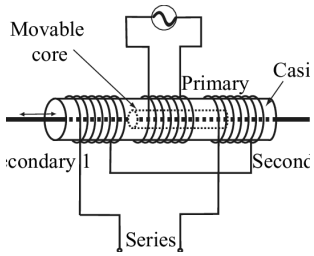
- Touch Sensor:
  It indicates whether contact has been made or not.It does not determine the magnitude of contact force.

- Position Sensor:
  i) Potentiometer- It is a temperature sensitive sensor. There are two types Linear and Angular.

  ii) Optical Encoder- Absolute and Incremental
  #Absolute Optical Encoder:It is mounted on the shaft of a rotary device. It is used to generate digital word identifying actual position of the shaft measured from zero position.
  #Incremental Optical Encoder: It consists of one coded disc and two photo detectors.By counting the number of light and dark zones, angular displacement can be measured with respect to known starting position.

- LVDT- Linear Variable Differential Transformer:

An LVDT is an electromechanical sensor used to measure linear displacement (movement in a straight line). It works on the principle of electromagnetic induction.



- Proximity Sensors:
  Proximity Sensors detect nearby objects without contact. They are used for obstacle avoidance and positioning. Common types include infrared, ultrasonic, inductive (for metals), and capacitive (for various materials). They help robots interact safely with their environment.

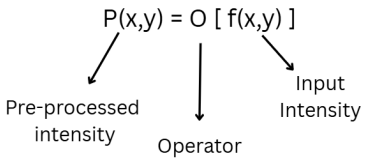
- Range Sensor: It measures the distance between the sensor(detector) mounted on the robot's body and the object

5.2 Robot Vision

Robot vision refers to the use of visual sensors—such as cameras—to enable a robot to perceive its surroundings. It plays a vital role in localization, object recognition, navigation, and interaction tasks.Robot vision enables tasks like identifying object positions, guiding motion, and avoiding obstacles. It is often integrated with other perception methods, such as laser range finders or acoustic sensors, to enhance reliability and accuracy in dynamic environments.

5.2.1 Methods of Pre-Processing

1. Masking
Masking involves applying a small matrix (called a mask or kernel) over an image to perform tasks like blurring, sharpening, or edge detection.



A 3x3 template Matrix =

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

2. Neighbourhood Averaging
This method smooths the image by replacing each pixel value with the average of its neighboring pixels (including itself).

3. Median Filtering
Instead of averaging, each pixel is replaced with the median value of its neighboring pixels.Commonly applied using a 3×3 or 5×5 neighborhood window.

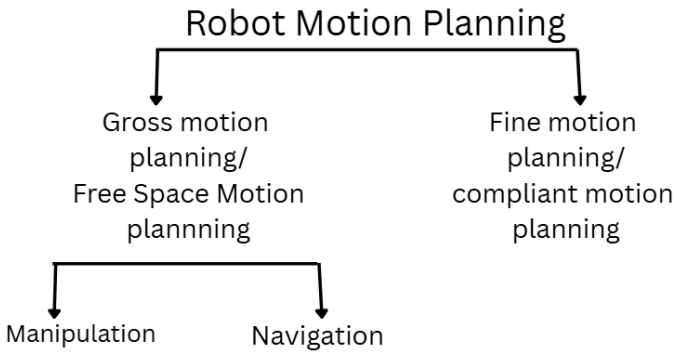
5.2.2 Lapalace operator

Laplacian Operator A second-order derivative method for edge detection. Unlike gradient (which detects direction), Laplacian finds regions of rapid intensity change in all directions.The kernel often looks like:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

6 Robot Motion Planning

Motion planning is the process of computing a path for a robot to move from a starting configuration to a goal configuration while avoiding obstacles and satisfying constraints like joint limits or actuator capabilities.



There are two approaches for motion planning: Global approach and Local approach
On the basis of environment whether it is a structured environment or un-structured environment, problems are termed respectively as Find-path problem and Dynamic motion problem.

6.1 Artificial Potential Field Method

The robot is treated like a point moving in a field influenced by:
Attractive Potential (pulls robot toward the goal)
Repulsive Potential (pushes robot away from obstacles)

6.1.1 Attractive Potential

The attractive potential guides the robot toward the goal by generating a virtual pulling force. It increases as the robot moves farther from the target, helping it follow a smooth path in the direction of the goal.
Typically defined as:

$$U_{att}(q) = \frac{1}{2} k_{att} ||q - q_{goal}||^2$$

where q is the robot's position and k_{att} is a scaling factor.

6.1.2 Repulsive Potential

The repulsive potential prevents collisions by pushing the robot away from obstacles. It becomes stronger as the robot gets closer to an obstacle, creating a safe buffer zone around it.
Typically defined as:

$$U_{\text{rep}}(q) = \begin{cases} \frac{1}{2} k_{\text{rep}} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0} \right)^2 & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases}$$

where $p(q)$ is the distance to the closest obstacle and p_0 is a threshold.

7 Intelligent Robot

Artificial Intelligence (AI) in Robotics enables robots to perceive environments, make decisions, and learn from experience. While traditional robotics relies on pre-defined rules and models, AI introduces adaptability through techniques like machine learning, deep learning, and reinforcement learning. In motion planning, AI can optimize paths in unknown or dynamic environments. In perception, AI powers vision systems to detect and recognize objects. Although not the focus of Modern Robotics, AI enhances many of its core topics such as planning, control, and sensor integration, bridging the gap between mechanical behavior and intelligent autonomy.

7.1 Biped Walking

Biped walking involves intelligent control, learning, and real-time decision-making, making it a clear example of intelligent robotics, especially when robots walk autonomously in unknown or human-centric environments.

7.1.1 Walking Cycle (Gait Cycle)

The walking cycle refers to the repeating pattern of leg movements during walking. It includes:
Stance Phase: The leg is in contact with the ground, supporting the body.
Swing Phase: The leg moves forward to take the next step.

7.1.2 Staircase ascending

Stair climbing is a more complex task because it involves higher leg lift, precise foot placement, and stronger balance control.The robot must perceive the geometry of each step using vision or depth sensors.Motion planning adapts the gait cycle to lift the center of mass up or down steps.The robot must generate torque-rich joint motions to overcome gravitational pull during ascent.

7.1.3 Dynamic balance margin

Dynamic Balance Margin (DBM) is a measure of how stable a biped robot is while walking or in motion. It tells us how far the robot is from becoming dynamically unstable.
Mathematically, If we denote:
x_{zmp}: current ZMP position
x_{edge}: nearest edge of the support polygon
Then: DBM = min $|x_{zmp} - x_{edge}|$

Intepretation:

- DBM $> 0 \rightarrow$ Robot is dynamically stable
- DBM $= 0 \rightarrow$ ZMP is on the edge (critically stable)
- DBM $< 0 \rightarrow$ ZMP is outside the support polygon $\rightarrow$ unstable, robot may tip or fall

8 Robot Operating System(ROS)

8.1 Introduction

The Robot Operating System (ROS) is a flexible, open-source framework designed to simplify the development of complex and robust robot software. It enables collaborative development by providing tools, libraries, and conventions that help tackle the wide range of challenges robots face in real-world environments. Originating from projects at Stanford and later expanded by Willow Garage, ROS supports distributed development, allowing contributors worldwide to share and control their own code repositories. Its open nature and widespread adoption have made ROS a cornerstone in both research and industrial robotics.

8.2 Important Terms

8.2.1 Roscore

Roscore is a service that provides connection information to nodes so that they can transmit messages to one another.Every node connects to roscore at startup to regis ter details of the message streams it publishes and the streams to which it wishes to subscribe.

8.2.2 ROS Graph

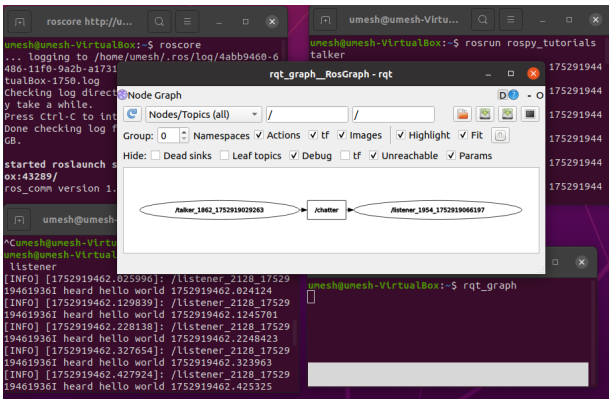
The ROS Graph represents a robot system as nodes (programs) and edges (message streams) between them. It enables modular design, where nodes can be added, removed, or swapped easily. This graph-based architecture supports flexibility, fault tolerance, and rapid experimentation.

8.2.3 Catkin,workspaces, ROS packages

catkin: catkin is the ROS build system and the set of tools that ROS uses to generate executable programs, libraries, scripts, and interfaces that other code can use.
workspaces: A workspace is simply a set of directories in which a related set of ROS code lives.
packages: ROS software is organized into packages, each of which contains some combination of code, data, and documentation.

8.2.4 Rosrun

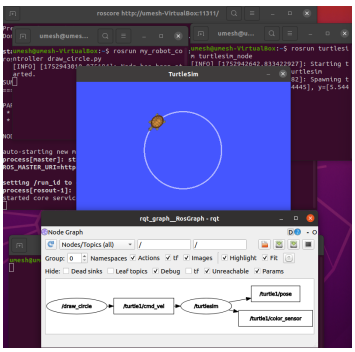
Rosrun lets you run a node from any ROS package without navigating to its folder. It finds and launches the executable with a simple command.
for eg: run 1)roscore 2)rosrun rospy_tutorials talker 3)rosrun rospy_tutorials listener 4)rqt_graph in four windows inside Ubuntu 20.04 machine in Virtual Box



8.3 Topics

8.3.1 Publishing to a topic

In ROS Noetic, publishing to a topic means a node is sending messages (data) to a named communication channel (the topic), which other nodes can subscribe to and receive.

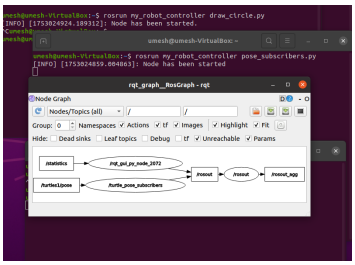



```python
#! /usr/bin/env python3     #is shebang
import rospy
from geometry_msgs.msg import Twist

if __name__ == '__main__':
    rospy.init_node('draw_circle')
    rospy.loginfo("Node has been started.")

                            #Name                #Type
    pub = rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size= 10)

    rate= rospy.Rate(2)

    while not rospy.is_shutdown():
        #publish velocity cmd
        msg= Twist()
        msg.linear.x = 2.0
        msg.angular.z = 1.0
        pub.publish(msg)
        rate.sleep()
```

This Python script is a ROS Noetic node that controls the TurtleSim robot. It starts by importing necessary libraries, including rospy for ROS functionality and Twist from geometry_msgs for velocity commands. The node is initialized with the name draw_circle, and a publisher is created to send Twist messages to the /turtle1/cmd_vel topic. These messages control the movement of the turtle. Inside a loop that runs at 2 Hz, it continuously publishes a velocity command where the turtle moves forward with a linear velocity of 2.0 units and rotates with an angular velocity of 1.0 units. This combination causes the turtle to move in a circular path. The loop runs until the node is shut down.

8.3.2 Subscribing to a topic

In ROS Noetic, subscribing to a topic means a node listens to a specific topic to receive messages published by other nodes.



```python
#!/usr/bin/env python3
import rospy
from turtlesim.msg import Pose

def pose_callback(msg:Pose):
    rospy.loginfo("(" + str(msg.x) + ", "+ str(msg.y)+ ")")

if __name__ == '__main__':
    rospy.init_node('turtle_pose_subscribers')

    sub= rospy.Subscriber("/turtles1/pose" , Pose , callback= pose_callback)

    rospy.loginfo("Node has been started")
    rospy.spin()
```

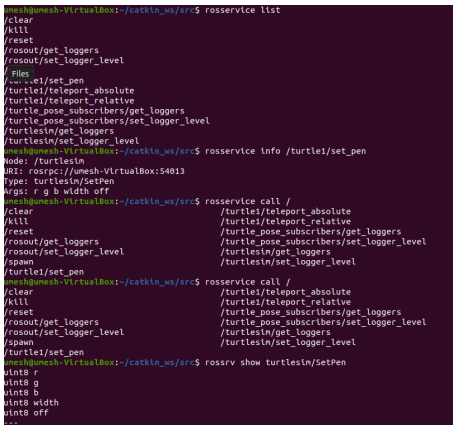
This Python ROS node subscribes to the /turtle1/pose topic and listens for Pose messages. Each time a new pose message is received, the pose_callback() function is called, which logs the x and y coordinates of the turtle's position. The node is initialized with rospy.init_node, the subscriber is created with rospy.Subscriber, and rospy.spin() keeps the node running to continuously receive messages.

8.4 ROS Service

In ROS (Robot Operating System), a Service is a communication mechanism used for synchronous (i.e., request-response) interaction between nodes. Unlike topics, which are used for continuous data streams, services are used when one node wants to ask another node to do something and wait for a reply.

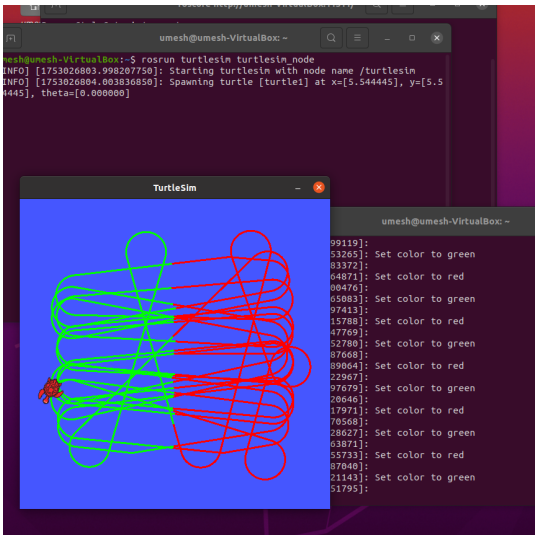
Key Concepts:

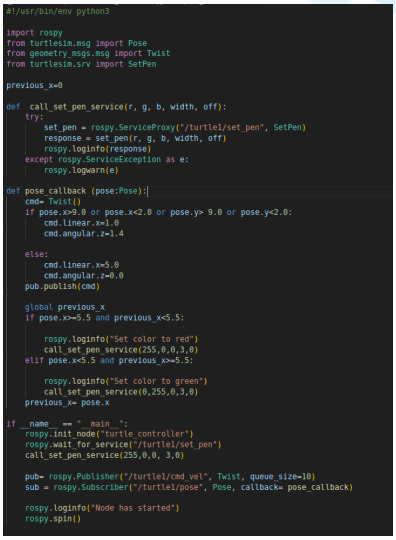
- **Client:** The node that sends a request.
- **Server:** The node that processes the request and sends a response.
- **Service message:** Defined by a `.srv` file with two parts:
  - **Request** (input)
  - **Response** (output)



8.4.1 Turtlesim: Change Pen Color Using ROS Services

This ROS Python code controls a turtle in the Turtlesim simulator by moving it and changing its pen color based on its position. It subscribes to the turtle's pose and publishes velocity commands to move forward or turn near the screen edges. It also uses a ROS service to change the pen color when the turtle crosses the center line (x = 5.5): red for the right side and green for the left. This code demonstrates how to use both ROS topics and a service client together.





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

import rospy
from turtlesim.msg import Pose
from geometry_msgs.msg import Twist
from turtlesim.srv import SetPen

previous_x=0

def  call_set_pen_service(r, g, b, width, off):
    try:
        set_pen = rospy.ServiceProxy("/turtle1/set_pen", SetPen)
        response = set_pen(r, g, b, width, off)
        rospy.loginfo(response)
    except rospy.ServiceException as e:
        rospy.logwarn(e)

def pose_callback (pose:Pose):
    cmd= Twist()
    if pose.x>9.0 or pose.x<2.0 or pose.y> 9.0 or pose.y<2.0:
        cmd.linear.x=1.0
        cmd.angular.z=1.4

    else:
        cmd.linear.x=5.0
        cmd.angular.z=0.0
    pub.publish(cmd)

    global previous_x
    if pose.x>=5.5 and previous_x<5.5:

        rospy.loginfo("Set color to red")
        call_set_pen_service(255,0,0,3,0)
    elif pose.x<5.5 and previous_x>=5.5:

        rospy.loginfo("Set color to green")
        call_set_pen_service(0,255,0,3,0)
    previous_x= pose.x

if __name__ == "__main__":
    rospy.init_node("turtle_controller")
    rospy.wait_for_service("/turtle1/set_pen")
    call_set_pen_service(255,0,0, 3,0)

    pub= rospy.Publisher("/turtle1/cmd_vel", Twist, queue_size=10)
    sub = rospy.Subscriber("/turtle1/pose", Pose, callback= pose_callback)

    rospy.loginfo("Node has started")
    rospy.spin()
```

9 Webots Simulation

9.1 Introduction

9.1.1 What is Webots?

Webots is a professional robot simulation software for creating 3D virtual worlds with realistic physics. Users can add passive objects or mobile robots with various movement types (wheeled, legged, flying) and equip them with sensors and actuators. Each robot can be individually programmed, and Webots provides many built-in models and example controllers to help users get started.

9.1.2 What is a World?

In Webots, a world is a 3D scene describing robots and their environment, including the position, appearance, and physical properties of all objects. It follows a hierarchical structure where objects can contain other objects. World files, saved as .wbt, are stored in the worlds folder of each project and reference robot controllers by name, but do not include the controller code itself.

9.1.3 What is a Controller?

In Webots, a controller is a program that controls a robot in a world file, written in C, C++, Java, Python, or MATLAB. Webots runs each controller as a separate process when the simulation starts. Controller files are stored in the controllers folder of each project.

9.1.4 What is a Supervisor Controller?

n Webots, a Supervisor controller is a special program that controls a robot with extra powers. If the robot's supervisor field is set to TRUE, the controller can do things like move robots, restart the simulation, or record videos—just like a human operator. It can be written in any language supported by Webots.

9.1.5 Scene Tree

The scene tree contains the information that describes a simulated world, including robots and environment, and its graphical representation.It is composed of a list of nodes, each containing fields

9.1.6 Nodes and Fields

Each node contains fields, which are its attributes. Fields describe properties such as position, rotation, size, color, or controller name. For example, a Robot node might have fields like

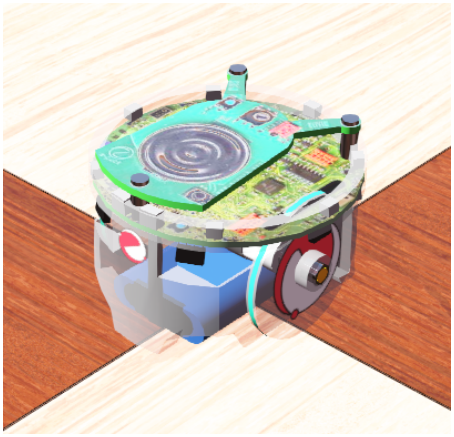
translation (position), rotation, and controller. Some fields can also contain other nodes, allowing nested structures. In short:

- NODE = an object in the simulation
- FIELD = a property or setting of that object

9.2 Simulations

9.2.1 E-puck go forward

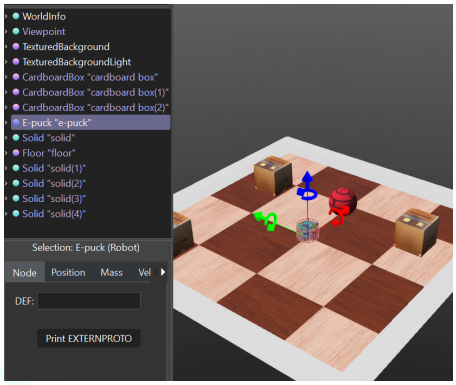
The e-puck is a small robot having differential wheels, 10 LEDs, and several sensors including 8 DistanceSensors and a Camera. Here is the simple implementation of Epuck to go forward.



```python
"""epuck_go_forward controller."""

# You may need to import some classes of the controller module. Ex:
#  from controller import Robot, Motor, DistanceSensor
from controller import Robot, Motor

# create the Robot instance.
robot = Robot()

# get the time step of the current world.
TIME_STEP=16
MAX_SPEED= 6.28
# timestep = int(robot.getBasicTimeStep())

# You should insert a getDevice-like function in order to get the
# instance of a device of the robot. Something like:
#  motor = robot.getDevice('motorname')
#  ds = robot.getDevice('dsname')
#  ds.enable(timestep)

leftmotor= robot.getDevice('left wheel motor')
rightmotor= robot.getDevice('right wheel motor')

leftmotor.setPosition(float('inf'))
rightmotor.setPosition(float('inf'))

leftmotor.setVelocity(0.1*MAX_SPEED)
rightmotor.setVelocity(0.1*MAX_SPEED)

# Main loop:
# - perform simulation steps until Webots is stopping the controller
while robot.step(TIME_STEP) != -1:
    # Read the sensors:
    # Enter here functions to read sensor data, like:
    #  val = ds.getValue()
    # Process sensor data here.
    # Enter here functions to send actuator commands, like:
    #  motor.setPosition(10.0)
    pass

# Enter here exit cleanup code.
```

This Python code is a simple Webots controller for an e-puck robot that makes it move forward slowly. It creates a Robot instance to communicate with the robot in the simulation. It sets the time step for the simulation and defines the maximum speed of the motors. Then, it accesses the robot's left and right wheel motors by name, sets them to velocity control mode, and sets both motors to 10% of their maximum speed. This causes the robot to move forward in a straight line. The main loop runs continuously while the simulation is active.

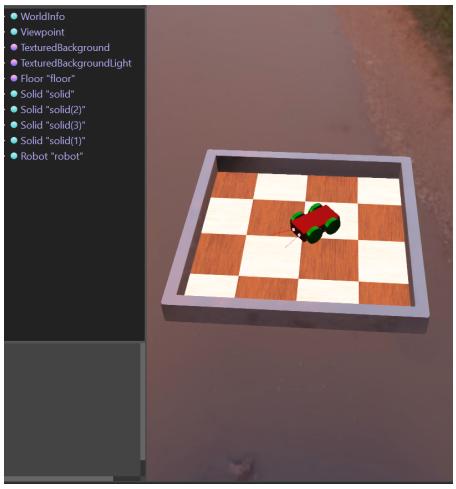
9.2.2 E-puck avoid collision




```python
"""epuck_avoid_collision controller."""

from controller import Robot, DistanceSensor, Motor
robot = Robot()
TIME_STEP=64
MAX_SPEED= 6.28
ps=[]
psNames= ['ps0','ps1','ps2','ps3','ps4','ps5','ps6', 'ps7']
for i in range(8):
  ps.append(robot.getDevice(psNames[i]))
  ps[i].enable(TIME_STEP)
leftmotor= robot.getDevice('left wheel motor')
rightmotor= robot.getDevice('right wheel motor')
leftmotor.setPosition(float('inf'))
rightmotor.setPosition(float('inf'))
leftmotor.setVelocity(0.0)
rightmotor.setVelocity(0.0)
while robot.step(TIME_STEP) != -1:
    # Read the sensors:
    psValues=[]
    for i in range(8):
        psValues.append(ps[i].getValue())
    right_obstacle= psValues[0] > 80.0 or
    psValues[1]>80.0 or
    psValue[2]>80.0 or
    left_obstacle= psValues[5] > 80.0 or
    psValues[6]>80.0 or
    psValue[7]>80.0 or
    leftSpeed=0.5*MAX_SPEED
    rightSpeed=0.5*MAX_SPEED
    if left_obstacle:
      leftSpeed = 0.5*MAX_SPEED
      rightSpeed = -0.5*MAX_SPEED
    elif right_obstacle:
      rightSpeed = 0.5*MAX_SPEED
      leftSpeed= -0.5*MAX_SPEED
    leftMotor.setVelocity(leftSpeed)
    rightMotor.setVelocity(rightSpeed)
```

This Python code is a Webots controller for an e-puck robot that allows it to move forward and avoid obstacles using its distance sensors. First, it imports the required classes and creates a robot instance. It sets the simulation time step and defines the maximum motor speed. The code then initializes an array of eight proximity sensors, enables them, and retrieves the left and right wheel motors. The motors are set to velocity mode, initially with zero speed. In the main loop, the robot reads values from all eight sensors to detect obstacles. If sensors on the left side detect something close, the robot turns right by reversing the right wheel. If sensors on the right side detect something, it turns left. If no obstacle is detected, the robot moves forward. This basic logic allows the robot to navigate without bumping into objects.

9.2.3 4 Wheeled Robot




```
"""4_wheeled_controller controller."""

from controller import Robot, Motor
TIME_STEP = 64
robot = Robot()
ds=[]
dsNames= ['ds_right',  'ds_left']
for i in range(2):
    ds.append(robot.getDevice(dsNames[i]))
    ds[i].enable(TIME_STEP)
wheels=[]
wheelsNames = ['wheel1','wheel2','wheel3','wheel4']
for i in range(4):
    wheels.append(robot.getDevice(wheelsNames[i]))
    wheels[i].setPosition(float('inf'))
    wheels[i].setVelocity(0.0)
avoidObstacleCounter=0
while robot.step(TIME_STEP) != -1:
    leftSpeed =3.0
    rightSpeed= 3.0
    if avoidObstacleCounter>0:
       avoidObstacleCounter -=1
       leftSpeed =1.0
       rightSpeed = -1.0
    else:
      for i in range(2):
        if ds[i].getValue() < 950.0:
            avoidObstacleCounter =100

    wheels[0].setVelocity(leftSpeed)
    wheels[1].setVelocity(rightSpeed)
    wheels[2].setVelocity(leftSpeed)
    wheels[3].setVelocity(rightSpeed)
```

This Python code is a Webots controller for a 4-wheeled robot that moves forward and avoids obstacles using two distance sensors (one on each side). First, it creates the robot instance, sets the time step, and enables the two distance sensors. Then, it retrieves the four wheel motors by name, sets them to velocity mode, and initializes their speed to zero. In the main loop, the robot normally moves forward with equal speed on all wheels. However, if either of the sensors detects a nearby obstacle (value below 950), a counter (avoidObstacleCounter) is set. While this counter is active, the robot turns by setting the left wheels to move forward and the right wheels backward. Once the counter finishes, the robot resumes moving forward. This helps the robot automatically turn away and avoid collisions when it senses something too close.

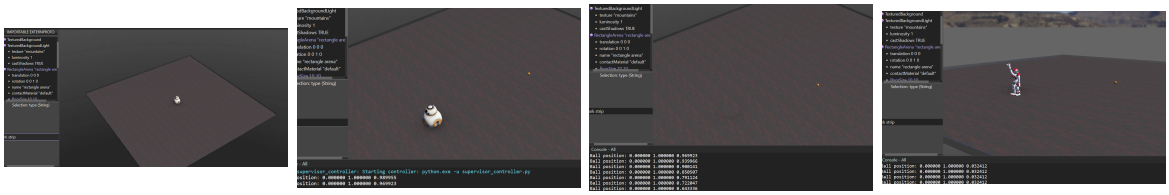
9.3 Moving objects Using a Supervisor

Remember that a supervisor is nothing more than a robot with special powers, which implies that whatever a robot can do, so can the supervisor. This means that you do not need a Robot instance if you have a supervisor one. For example the infinite loop that determines the pace of the controller (namely: while robot.step(TIME_STEP) != -1) does not need to be changed, as the supervisor can do the same.

This Python code is a Webots Supervisor controller that demonstrates how to control and modify the simulation dynamically. It first gets access to a robot named BB-8 and a root node to add new objects. It then adds a new Ball object into the world and prepares to track its position. In the simulation loop, the code moves the BB-8 robot to a new location on the first step, removes BB-8 after 10 steps, and adds a new robot (Nao) after 20 steps. Throughout the simulation, it continuously prints the real-time position of the Ball. This showcases how a Supervisor can directly manipulate objects, remove or add robots, and monitor positions — tasks that normal robot controllers cannot perform.

```python
"""supervisor_controller controller."""

from controller import Supervisor, Robot
TIME_STEP =32
robot = Supervisor()
bb8_node= robot.getFromDef('BB-8')
translation_field = bb8_node.getField('translation')

root_node = robot.getRoot()
children_field = root_node.getField('children')

children_field.importMFNodeFromString(-1, 'DEF BALL Ball { translation 0 1 1}')
ball_node = robot.getFromDef('BALL')
color_field = ball_node.getField('color')
i=0
while robot.step(TIME_STEP) != -1:
    if i==0:
        new_value = [2.5
        ,0,0]
        translation_field.setSFVec3f(new_value)

    if i==10:
        bb8_node.remove()

    if i==20:
        children_field.importMFNodeFromString(-1,'Nao{translation 2.5 0 0.334}')


    position = ball_node.getPosition()
    print('Ball position: %f %f %f\n' %(position[0], position  [1], position[2]))

    if position[2] < 0.2:
    red_color = [1,0,0]
    color_field.setSFColor(red_color)

    i +=1
```

This is what the animation looks like:



— **Thank You** —