T.C.

DOKUZ EYLUL UNIVERSITY ENGINEERING FACULTY

ETE 3007 FUNDAMENTALS OF ROBOTICS

ROBOTIC ARM PUMA WITH DRAWING APPLICATION

Project Report

by

Sinem Selçuk - 2019502124 - EEE

Tolga Serbest - 2017502068 - EEE

Güney Söğüt - 2020510066 - CENG

Advisor

ASSOC. PROF. AHMET OZKURT

May, 2024

Izmir

ABSTRACT

Designing and implementing a PUMA (Programmable Universal Machine for Assembly) robot arm to perform 2D drawing operations with specified patterns within the 3D Webots simulation environment. Patterns are calculated with mathematical operations and implemented in the inverse kinematics formulas. The primary objective is to configure and program the PUMA robot arm, using its multiple degrees of freedom to simulate complex drawing operations with the usage of inverse kinematics library and calculations. The robot's movements and drawing abilities are modeled and tested in the Webots program, which provides physically realistic simulations.

TABLE OF CONTENTS

1.INTRODUCTION

1.1 The Robot Arm PUMA

A Programmable Universal Machine for Assembly, or PUMA, is a type of industrial robot. Because they have numerous joints that enable accurate and flexible movement, PUMA robots are known as articulated robots.

PUMA robots are widely used in industrial settings for tasks like packing, material handling, welding, and assembly. Their ability to move in multiple directions and perform complex motions is attributed to their multitude of interconnecting linkages and joints. Since PUMA robots are programmable, software can be used to modify and control their jobs and actions.

PUMA robots are widely utilized in the industrial industry because of their efficacy, precision, and flexibility. They work a lot in the automotive, electronics, and other repetitive industries. Over time, the PUMA design has been improved upon, leading to the creation of subsequent generations of industrial robots with increasingly advanced features and functionalities [1].

Figure 1: PUMA Robot Arm real life application

Hardware Requirements

- Computer for running the simulation.

Software Requirements

- Programming languages such as C or Python for the backend control and the kinematic algorithms and calculations.
- Python libraries for the kinematic algorithms and their calculations.
- Webots Simulation Program

1.2 Definition Of The WEBOTS Simulation Program

An open source simulation program for modeling, programming, and simulating robots in three dimensions is offered by the Webots. The program allows users to work with a range of robotic models, including industrial robots like the PUMA arm. Users can choose from a large library from models that already exist or create a new model using the robotic design tools in Webots to simulate a robot easily. This requires specifying the robot's physical characteristics, including its size, mass, form, and kind of joint such as linear or rotational.

Language options (Python, C++, or Java script) facilitate the robot design process. Webots provides an API for accessing and managing all aspects of the robot, including joint positions, motor speeds, sensor inputs, and interactions with the environment. Users program the robot to react to specific objects. For example, in a PUMA arm simulation, scripts could direct the arm's movement to carry objects such as picking and placing commodities or performing assembly tasks that require exact movements and job sequences.

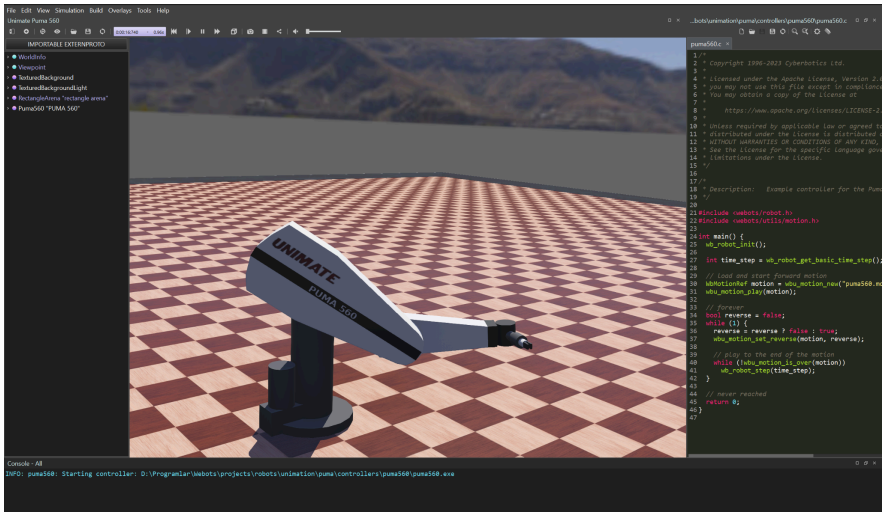

Figure 2: WEBOTS Simulation Program

2. METHODOLOGY

2.1 Inverse Kinematics

A robot arm is a mechanical device that copies the motions of a human arm. It is also referred to as a manipulator or robotic arm. Generally, a robot arm is made up of a number of joints that connect several linked segments known as links or arms. Robot tasks are performed with the end-effectors which is a device that locates in the end of the robotic arm to interact with the environment. In our project, we will use the inverse kinematics which makes use of the kinematics equations to determine the joint parameters that provide a desired configuration (position and rotation) for each of the robot's end-effectors [2].

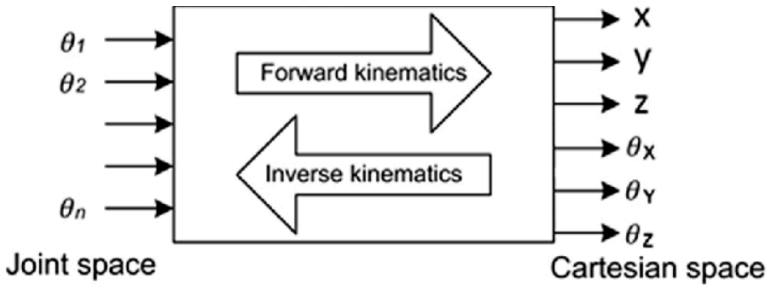

Figure 3: Kinematics Schematic

Motion planning can be described as determining the movement of a robot so that its end-effectors move from an initial configuration to a desired configuration. Inverse kinematics algorithms transform the motion plan into joint actuator trajectories for the robot. In this project, the “IKPy” Python library will be used for inverse kinematic motions.

2.1.1 The “ikpy” Library

A Python library is used for inverse kinematics calculations. particularly in robotics applications. There are some key aspects of the ikpy library. First aspect is inverse kinematics calculations, which provide you to specify the target position and orientation of the end-effector and compute the required joint angles for the position and orientation. Moreover, one of the main purposes of the ikpy is to perform inverse kinematics. Secondly, the library supports various types of robotic kinematics, including serial manipulators as well as parallel manipulators. Lastly, the ikpy library contains visualization tools for inverse kinematics solutions. This may help you validate and track the calculations in order to optimize your calculations and design [3].

2.2 Position Sensor

Position sensor is needed to know the joint's position. In the simulation, a position sensor can be inserted in the device field and monitors the joint position as an angular position or linear position [4]. There are two fields that can be used for position sensors in Webots. Noise fields add the gaussian noise to the sensor output. Resolution field defining the resolution of the sensor for measurements.

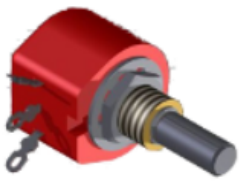

Figure 4: Position sensor

2.3 Rotational Motor

Rotational motor that is described in the Webots Reference Manual as a node to produce a rotational motion around the chosen axis. The rotational motion of the robotic arms will be performed with these motors. There will be a single motor in each joint. The rotational motors provide a wide range of movement area but will be using a small range for drawing applications. Because the drawing area will be 2D and this imitates the movement of the robot arms.

Figure 5: Rotational Motor

2.4 Drawing Application

The pattern to be drawn will be defined using Python language and this pattern will be converted to cartesian coordinates by using the necessary libraries. The coordinates of the pattern will be converted with an inverse kinematics library and corresponding positions will be realized in the joints with rotational motors. The position will be verified with the position sensor for accuracy.

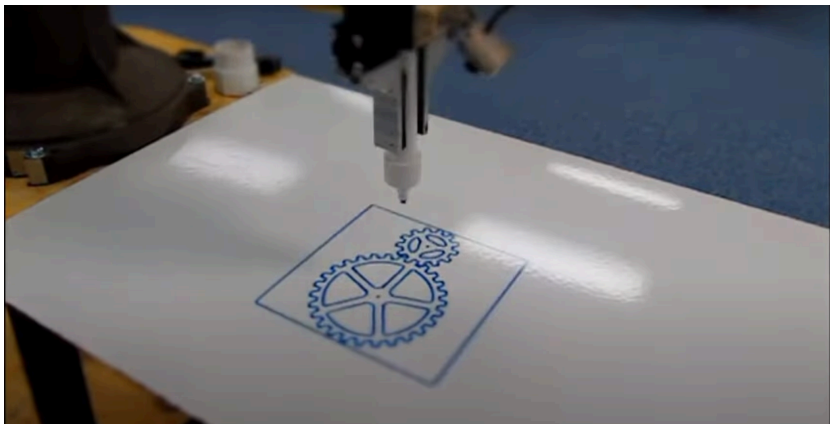

Figure 6: PUMA arm drawing application

2.5 Drawing Algorithm

1. Import necessary modules:

   - Import inverse kinematics library (ikpy)

   - Import Webots Supervisor

   - Import math for path calculations

2. Check module dependencies:

   - If the installed version of ikpy is not compatible, display an error message and exit.

3. Set up Webots Supervisor:

   - Create a Supervisor instance to manage the simulation.

   - Set the time step for the simulation.

4. Obtain equipment and sensors:

   - Get motors and position sensors from the Supervisor.

5. Determine initial positions and speeds:

   -Take the initial positions of all motors.

6. Setup for motion:

   - Defining time limit and shape algorithm.

   - Start a loop to simulate motion until the time limit is reached.

   - For each motor and its corresponding position sensor:

      -Calculate the desired position based on motion parameters using ikpy.

2.6 URDF File

A URDF (Unified Robot Description Format) file is an XML format used to represent a robot model. It includes the robot's physical configuration, its kinematic structure, its visual representation, and its articulation parameters. Below is a sample URDF file for a simple robotic arm that can be used with the code provided**.**

The URDF file describes a basic 4 DOF (degrees of freedom) robotic arm with an end effector. It corresponds to the structure expected by the provided code, each coupling corresponding to a motor. The robot can be loaded and controlled in the Webots simulation environment using the setup described.

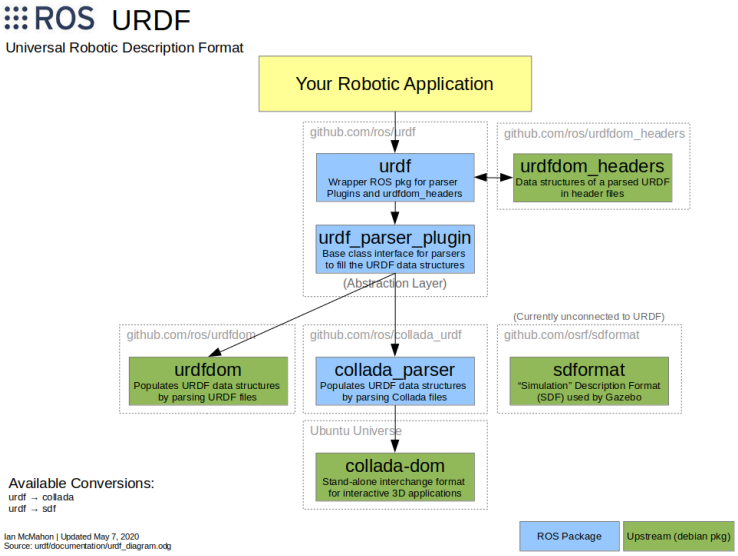

Figure 7: URDF Schematic

3. PUMA DRAWING ROBOT ARM IN SIMULATION ENVIRONMENT

3.1 Hardware Part of Simulation

In the hardware part, the robot is designed as a human arm. It consists of one body part, one rotational part, two arm parts. Body part is shaped as a plate to distribute the weight of the robot to avoid falling from the weight of the arms. Rotate part designed to rotate the arm in the z axis. Arms are designed to reach the desired point. In the end of the robot, there is a hand shape to attach the pen for drawing.

We created two robots but the first one did not work with the “ikpy” library because the boundingObject configurations, Solid and joint relationships and joint locations and anchor configurations were not arranged properly. This robot was also working but it was not controllable and the movements were unpredictable.

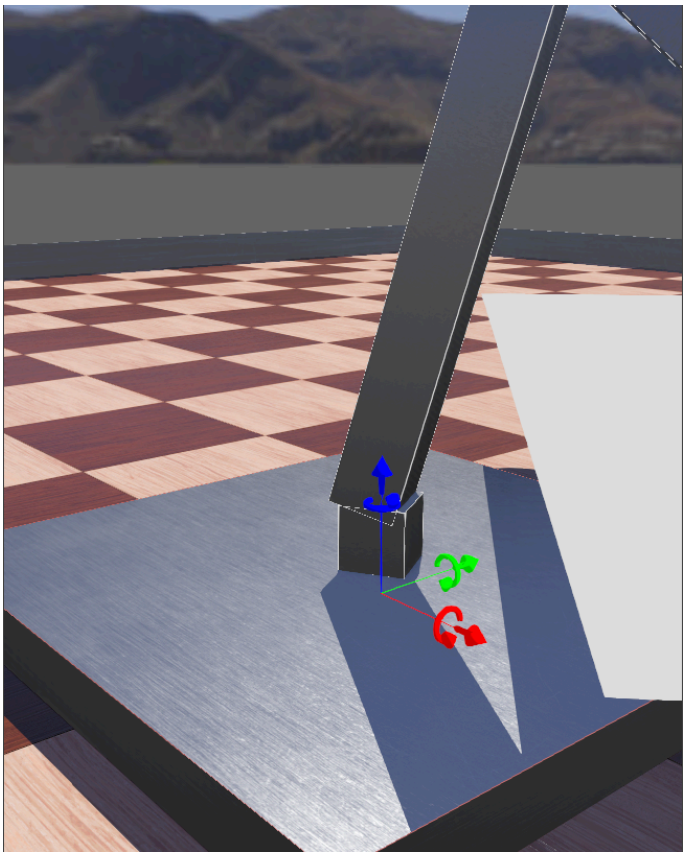

Figure 8: Origin of the Robot

The robot is created using a robot node in the WEBOTS. And it's located at the origin of the world. The blue arrow represents the z-axis, the green arrow represents the y-axis and the red one is the x-axis. The supervisor is checked as “TRUE” to reach the robot with the supervisor.

The robot has 3 motors and sensors; the first motor and sensor are located at the rotated part and the other two motors and sensors are located at the starting of each arm. The example robot in the WEBOTS is used for modeling [5]. The URDF file can be exported with right clicking the robot node in the WEBOTS world. There is a “base_link” for starting a link which is “Solid”. This link is the parent of the first joint. And in the endPoint of the joint another Solid is located as Children. Each joint also has a Shape as the same parent's children. This structure follows each other.

Figure 9: The parent - children relation

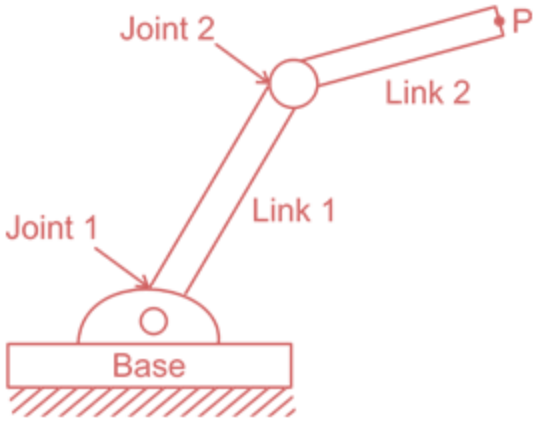

Figure 10 : The robot arm structure

The representation of the robot is Figure 7. The “ikpy” library works with this structure. At the P point there is a pen located. All the endPoint’s of the joints are selected as “Solid” in the WEBOTS simulation. Translation and anchor arrangements are very important. The anchor is the joint’s origin. The Joint axis is the rotation axis. For example, if it's arranged as “0 0 1” the joint will rotate around the z axis.

The “ikpy” library requires a joint type as “revolute”. Without arranging the motor’s limits the joint is recorded as “continuous” in the URDF file and this causes an error. Also, the motors should provide the necessary amount of torque (10.000) to move the arms. Sensors’ resolution is arranged as 0.01 which is enough for this application and it’s quite good for a sensor device.

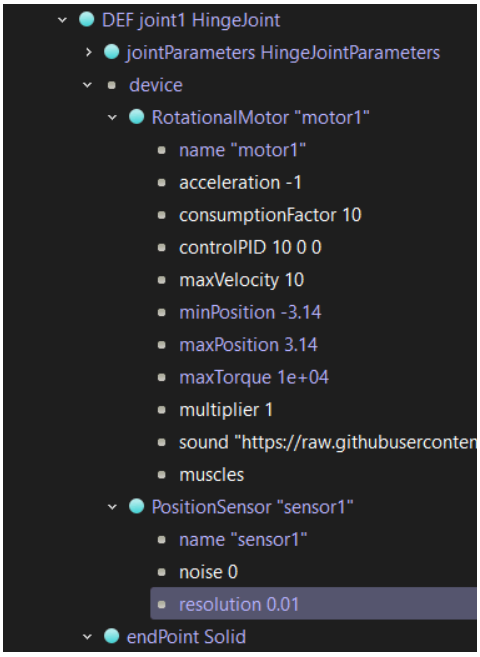

Figure 11: Motor and sensor configurations

BoundingObject should be defined as a pose. Pose node gets the coordinates from the parent and defines relative coordinates for its child. This configuration allows to create a grouping of two parts and creates a link between parent and child. This leads to transfer coordinates from base to hand through each posing node. Pose nodes need a shape arranged as Box in the simulation.

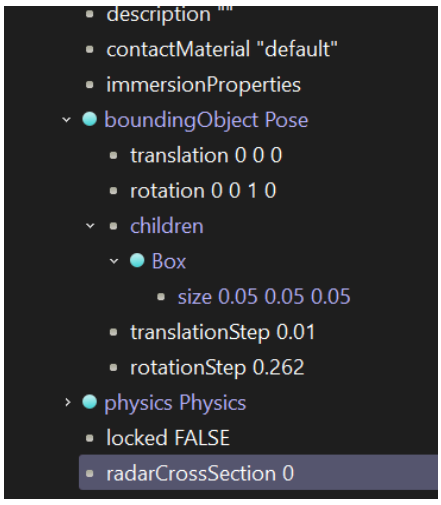

Figure 12: The joints endPoint boundingObject and physics configurations

To add a pen in the hand, a Solid node is added to the last joint's endPoint's children where the hand shape is located. The pen object is added to this solid's children. It did not draw at first but we figured that the pen was writing in the -z direction so we flipped the pen. Lastly, to see the pen's shape we added a shape as children of the pen object.

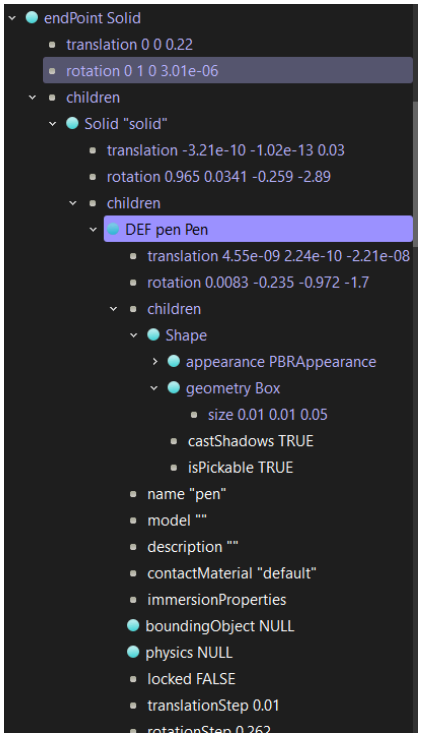

Figure 13: Pen object location and configuration

Finally, we added a "Pose" node to draw as a paper and named it as "TARGET". This paper has children as "Shape". Shape geometry is arranged as "Plane".

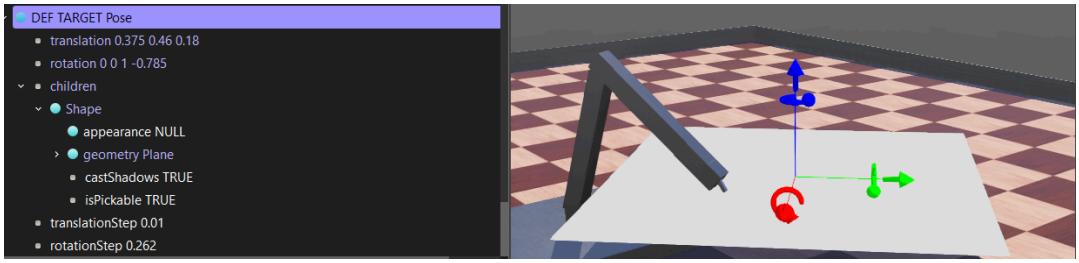

Figure 14: The Drawing Paper configuration

3.2 Software Part of Simulation

First section imports the necessary libraries and modules. The Webots monitor drives the simulation, ikpy and Chain handle the inverse kinematics calculations, and the math provides math functions.

The second section loads the robot's Universal Robot Description Format (URDF) file to create the kinematic chain using IKPy. active_links_mask specifies which links in the chain are active (moving). This configuration is important to determine the structure and movement capabilities of the robot.

In the third section, the Supervisor class is created to control the Webots simulation. The time step is defined, determining the frequency of the simulation steps based on the baseline time step provided by Webots. This ensures synchronous updates in the simulation.

The fourth part initializes the robot's motors and senses their position. It goes through the links in the robot's drive chain, identifies those that represent the motors, and determines their speed. The position sensor for each motor is activated and added to the motor list for later control. List of initializers printed for verification and constant determination for maximum IK iterations.

The fifth section initializes the pen device and sets its state to False, ensuring that the pen does not write when starting the simulation. This prepares the pen for controlled activation during drawing.

The last part controls the robot arm to draw a heart shape. At each simulation step, it calculates time t and uses parametric equations to determine the coordinates (x, y, z) of the heart shape. These coordinates are printed for debugging purposes. The inverse_kinematics method calculates the joint angles required to achieve these coordinates. The first three motors were then placed in these positions. The pen is activated after 0. 5 seconds and stops working once the heart shape is completed. This loop ensures that the robotic arm accurately draws the heart shape on the surface, with the pen only writing for the intended amount of time.

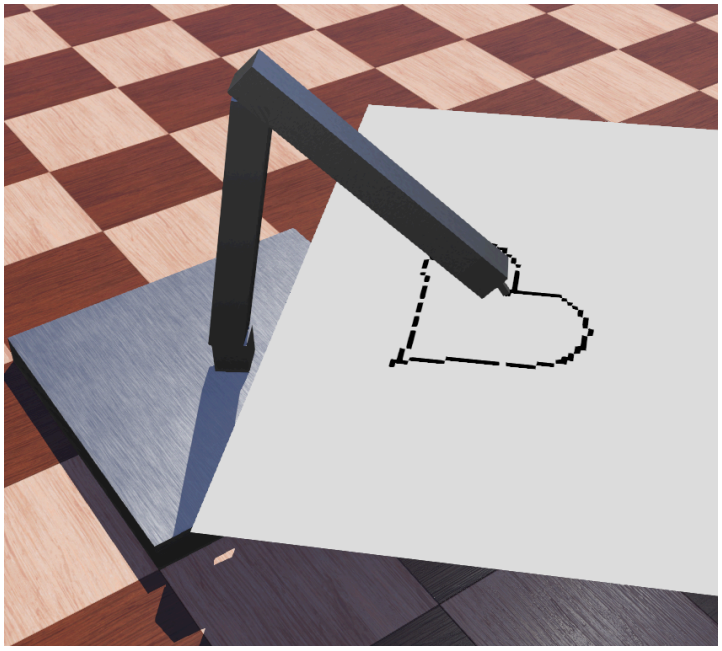

Figure 15: Drawing Algorithm Code Simulation Output

```
from controller import Supervisor
import ikpy
from ikpy.chain import Chain
import math

# Load URDF file and define active links as true or false
robot_chain = Chain.from_urdf_file("D:\webots\my_project_heart\my_project7\worlds\Robot.urdf",
active_links_mask=[False, True, True, True, True])

# Define supervisor function for further operations
supervisor = Supervisor()
timestep = int(4*supervisor.getBasicTimeStep())
```

```
# Initialize the arm motors and encoders.
motors = []
for link in robot_chain.links:
    if 'motor' in link.name:
        motor = supervisor.getDevice(link.name)
        motor.setVelocity(2.0)
        position_sensor = motor.getPositionSensor()
        position_sensor.enable(timestep)
        motors.append(motor)
print(motors)
IKPY_MAX_ITERATIONS=4

# Initially define pen device write fucntion as false
supervisor.getDevice('pen').write(False)

# Main while loop to draw the heart shape
while supervisor.step(timestep) != -1:
    t = supervisor.getTime()

    # Define the heart shape parametric equations relatively to the arm base as an input of the IK algorithm
    scale = 0.008  # Scale of drawing
    x = scale * (16 * math.sin(t)**3) + 0.2
    y = scale * (13 * math.cos(t) - 5 * math.cos(2 * t) - 2 * math.cos(3 * t) - math.cos(4 * t)) + 0.2
    z = 0.25  # Fixed height
    print(x, y, z)
```

```
    # Get initial position from all sensors
    initial_position = [0] + [m.getPositionSensor().getValue() for m in motors]

    # Calculate IK results for robot chain
    ikResults = robot_chain.inverse_kinematics([x, y, z], max_iter=IKPY_MAX_ITERATIONS,
initial_position=initial_position)

    # Actuate the 3 first arm motors with the IKPY results
    for i in range(3):
        motors[i].setPosition(ikResults[i + 1])

    # Break the loop after completing one cycle define write as false
    if supervisor.getTime() < 2 * math.pi:
        if supervisor.getTime() > 0.5:
            supervisor.getDevice('pen').write(True)
    else:
        supervisor.getDevice('pen').write(False)
        break
```

4. CONCLUSION

In this project, we successfully built and implemented a PUMA (Programmable Universal Machine for Assembly) robot arm that can perform 2D drawing tasks in the 3D Webots simulation environment. The primary goal of the study was to mimic complicated sketching jobs using the PUMA robot's many degrees of freedom and inverse kinematics calculations.

The project began with a thorough introduction to the PUMA robotic arm and the Webots simulation program. We described the hardware and software requirements and gave an introduction of the inverse kinematics idea, highlighting the need of the IKPy library for kinematic computations. We also talked about the important components, such as position sensors and rotational motors, which are required for accurate and exact movements.

In the methodology section, we described the step-by-step procedure for implementing the drawing application. This includes establishing the mathematical pattern, transferring it to Cartesian coordinates, and computing the robot's end-effector joint angles with the IKPy library. The drawing process was thoroughly defined, ensuring that every component worked together fluidly to generate the desired result.

The URDF (Unified Robot Description Format) file was vital in specifying the robot's physical and kinematic structure, ensuring correct simulation in Webots. We demonstrated how to configure the URDF file and integrate it into the Webots environment, allowing for realistic simulations of the PUMA robot's motions.

We confirmed the PUMA robot's ability to properly follow predetermined drawing patterns through thorough testing and continuous modifications. The project effectively demonstrated the robot's ability to do complex drawing operations using inverse kinematics, hence demonstrating the usefulness of our technique.

Finally, this study demonstrates the value of employing advanced simulation tools like Webots and powerful kinematic libraries like IKPy to describe, simulate, and operate robotic systems for complex tasks. The knowledge gathered from this study serves as a solid platform for future robotics research,

where comparable approaches can be used to increasingly complicated and diverse problems. This research not only demonstrates the technical capabilities of the PUMA robot arm, but also underlines the significance of simulation and kinematic calculations in modern robotics.

5. REFERENCES

[1] https://en.wikipedia.org/wiki/Programmable_Universal_Machine_for_Assembly

[2] https://en.wikipedia.org/wiki/Inverse_kinematics

[3] https://ikpy.readthedocs.io/en/latest/

[4] Webots documentation: PositionSensor (cyberbotics.com)

[5] https://github.com/cyberbotics/webots/blob/released/projects/robots/abb/irb/protos/Irb4600-40.proto

6. APPENDIX

Proto file of the simulation:

```
#VRML_SIM R2023b utf8

EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/protos/TexturedBackground.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/backgrounds/protos/TexturedBackgroundLight.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/objects/floors/protos/RectangleArena.proto"
EXTERNPROTO
"https://raw.githubusercontent.com/cyberbotics/webots/R2023b/projects/appearances/protos/BrushedAluminium.proto"

WorldInfo {
}
Viewpoint {
  orientation -0.2962856860961655 0.09708601723637435 0.9501521443809421 2.537941201800005
  position 1.349249045894374 -0.4090264421404099 1.1919077405008878
}
TexturedBackground {
}
TexturedBackgroundLight {
```

```
}
Robot {
 translation -0.0014362507928100514 -0.0012309169211759785 0.019570250365313742
 rotation -0.15152593977703172 0.24018920759133258 0.9588269051978751 0.012263513790795754
 children [
  DEF joint1 HingeJoint {
   jointParameters HingeJointParameters {
    position 0.5383993310505366
    axis 0 0 1
    anchor 0 0 0.04
   }
   device [
    RotationalMotor {
     name "motor1"
     minPosition -3.14
     maxPosition 3.14
     maxTorque 10000
    }
    PositionSensor {
     name "sensor1"
     resolution 0.01
    }
   ]
   endPoint Solid {
    translation 0 0 0.05
    rotation 0 0 1 0.5383993310505367
```

```
children [
  DEF joint2 HingeJoint {
    jointParameters HingeJointParameters {
      position 0.27464716219602164
      axis 0 1 0
      anchor 0 0 0.05
    }
    device [
      RotationalMotor {
        name "motor2"
        minPosition -3.14
        maxPosition 3.14
        maxTorque 10000
      }
      PositionSensor {
        name "sensor2"
        resolution 0.01
      }
    ]
    endPoint Solid {
      translation 0.046105247811370355 0 0.2136285614562835
      rotation 0 1 0 0.27464716219602203
      children [
        DEF joint3 HingeJoint {
          jointParameters HingeJointParameters {
            position 2.0204928048625517
```

```
  axis 0 1 0
  anchor 0 0 0.22
}
device [
  PositionSensor {
    name "sensor3"
    resolution 0.01
  }
  RotationalMotor {
    name "motor3"
    minPosition -3.14
    maxPosition 3.14
    maxTorque 10000
  }
]
endPoint Solid {
  translation 0.1621042348415344 0 0.14175540244565973
  rotation 0 1 0 2.0204928048625517
  children [
    HingeJoint {
      jointParameters HingeJointParameters {
        position 4.417487015542433e-06
        axis 0 1 0
        anchor 0 0 0.22
      }
      device [
```

```
      PositionSensor {
       name "sensor4"
       resolution 0.01
      }
      RotationalMotor {
       name "motor4"
       maxTorque 10000
      }
     ]
     endPoint Solid {
      translation 0 0 0.22
      rotation 0 1 0 3.0104771373690567e-06
      children [
       Solid {
        translation -3.208e-10 -1.01632e-13 0.03
        rotation 0.9653654670923678 0.03405581647791691 -0.258669125157133
-2.888765307179586
        children [
         DEF pen Pen {
          translation 4.5459e-09 2.23569e-10 -2.2059e-08
          rotation 0.00829728670379422 -0.23523290655064794 -0.9719036138985641
-1.7006453071795864
          children [
           Shape {
            appearance PBRAppearance {
            }
```

```
          geometry Box {
            size 0.01 0.01 0.05
          }
        }
      ]
      inkDensity 1
      write FALSE
    }
   ]
   boundingObject DEF hand Shape {
   }
   physics Physics {
   }
  }
  DEF hand Shape {
   appearance BrushedAluminium {
   }
   geometry Box {
    size 0.05 0.05 0.05
   }
  }
]
boundingObject USE hand
physics Physics {
}
linearVelocity -0.04607073237846021 -0.08251928442123879 -0.006523590665592311
```

```
        angularVelocity -0.037814071197219974 0.06415241717872254 -0.10929302531395285
      }
    }
    DEF arm2 Shape {
      appearance BrushedAluminium {
      }
      geometry Box {
        size 0.05 0.05 0.4
      }
    }
  ]
  boundingObject Pose {
    children [
      Box {
        size 0.05 0.05 0.4
      }
    ]
  }
  physics Physics {
    density 100
  }
  linearVelocity -0.04589802075375931 -0.05908177850420139 0.0071700749117366
  angularVelocity -0.037813155539238974 0.0641502457550853 -0.10928357862121818
 }
}
DEF arm1 Shape {
```

```
        appearance BrushedAluminium {
        }
        geometry Box {
          size 0.05 0.05 0.4
        }
      }
    ]
    boundingObject Pose {
      children [
        Box {
          size 0.05 0.05 0.4
        }
      ]
    }
    physics Physics {
      density 100
    }
    linearVelocity -0.019781580225404896 -0.01831612214272049 0.007467395762641896
    angularVelocity 0.08541968341862566 -0.13840840720808364 -0.10926373388436036
  }
}
DEF rotate Shape {
  appearance BrushedAluminium {
  }
  geometry Box {
    size 0.05 0.05 0.05
```

```
        }
       }
      ]
      boundingObject Pose {
       children [
        Box {
         size 0.05 0.05 0.05
        }
       ]
      }
      physics Physics {
       density 100
      }
      linearVelocity 0.00018071607619791786 -1.967429502401611e-05 -2.328067103235298e-05
      angularVelocity -0.0002456533352414963 0.0023664347882004126 -0.10922500388681718
    }
   }
   DEF body Shape {
    appearance BrushedAluminium {
    }
    geometry Box {
     size 0.5 0.5 0.05
    }
   }
]
boundingObject USE body
```

```
  physics Physics {
  }
  controller "my_controller"
  supervisor TRUE
  linearVelocity 6.209785762100226e-05 -5.366904939671748e-06 -2.195532311292483e-05
  angularVelocity 6.414622163651768e-05 0.0025608264803313237 -4.191397674690572e-05
}
RectangleArena {
  floorSize 5 5
}
DEF TARGET Pose {
  translation 0.375059 0.459793 0.18
  rotation 0 0 1 -0.785395307179586
  children [
    Shape {
      geometry Plane {
      }
    }
  ]
}
```