Precise Pick & Place System, Remote Control, Plotting & Computation of Trajectories, Cartesian Positions and Jacobian Matrices of a 7-DoF KUKA Robot.

Ammar Arif Ansari [1*], Shayma Akbar [2] and Shravan Sankaranarayanan [3]
[1, 2, 3] School of Engineering and Physical Sciences, Heriot-Watt University, Dubai, UAE

Author Emails:
Ammar Arif Ansari: ammararifansari@outlook.com
Shayma Akbar: shaymaakbar11@gmail.com
Shravan Sankaranarayanan: 02shravans@gmail.com

Keywords: Pick-and-Place Algorithm, LUA, Inverse Kinematics, Trajectory Planning, Remote Control, KUKA IIWA LBR

Abstract

Using Robotic arm without precise calibration within its working environment can be extremely hazardous to human-life if working in the vicinity of the robot. The paper describes the Robotic and conveyor platforms and presents an algorithm for their design and optimization with autonomous trajectory plotting and intelligent pick & place system. The dependency on the dimensions of the platform, the maximum optimized calibration of the robotic arm is worked out to sense and pick up objectified boxes from the conveyor whilst the keyboard control feature is also added for improvising the calibration process in working of the model. The data matrices obtained from movements of the robot allows us to choose the best of trajectory planning methods and sizing the elements according to the beneficiary's requirements.

1. Introduction

The aim of the research was to bring the end effector of the robot KUKA IIWA LBR 1180 to any desired Cartesian position intelligently. The research was conducted using Coppelia Sim, formerly known as V-REP, a robot simulator used in industry, education, and research, actively maintained by Coppelia Robotics AG, Zurich, Switzerland. The computer language used to program the simulations is LUA. Making the manipulator move with any desired Endpoint translational and rotational Cartesian velocity as the robot follows any given Cartesian position trajectory. Reading, writing, and the plotting of joint angles of the robot in any configuration was also programmed along with triggers to compute Cartesian positions and Euler -angles of the endpoint of the robot manipulator.

For human intervention in-order to optimize the performance of the robot, the Jacobian matrices, Euler angles, Cartesian position of the endpoint effector was tapped whereas the robot was given the ability to be moved using PC keyboard buttons and external remote or wired controllers such as Game joysticks.

1.1 About the Robot KUKA LBR – IIWA – 14 – R820

The robot model KUKA LBR – IIWA – 14 – R820 [1] is manufactured by KUK Robotics which aims at the vision of a production environment free from rigid structures. As part of this, lightweight robots (LBR) play a key role as “intelligent industrial work assistants” (IIWA). They orient themselves independently in their surroundings and move into position for new automation tasks with milli -meter precision. Compliance of the LBR IIWA can be programmed individually for all joints as well as for all Cartesian degrees of freedom. Low weight of just 23.9kg (with a payload capacity of 7 kg) or 29.9kg (with a payload capacity of 14 kg) is the key to the sensitivity and mobility of the LBR IIWA.

In all seven axes, the LBR IIWA has integrated joint torque sensors, implemented using safe technology. They respond to the slightest of external forces and enable safe collision protection. In the case of unexpected contact, the LBR IIWA reduces its velocity in an instant, thereby limiting its kinetic energy to a level that precludes injuries.

Figure 1: Use of KUKA LBR-IIWA-14-R820 in modern industries

In the industries of the future, the focus of thought and action of an autonomous machine

Figure 2: Depiction of 7 degrees of freedom in the robot

will shift to the human employer with his unique requirements and capabilities. Using mobile platforms such as KUKA Mobile Robotics, the LBR IIWA orients itself independently and extends its working range almost indefinitely. The LBR IIWA can achieve the necessary compliance while operating and can be easily modified to new specifications by changing parameters.

2.1 Inverse Kinematics & Positioning of End-Effector

Bringing the end effector to any desired cartesian position and making the endpoint manipulator move with any desired endpoint translational and rotational cartesian velocity.

Inverse Kinematics module was initiated in the simulation pane in the following steps:

1. Enabled "Inverse Kinematics" mode to all the joints of the robot.

2. Added "Tip" & "Target" dummies to the Primitive Shape in the scene.

3. Made the position and orientation of the Dummies same.

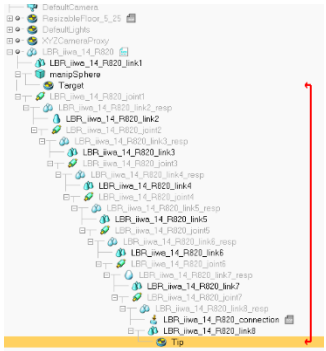

Figure 3: Linking of joints and enabling Inverse Kinematics

4. Linked both the dummies (i.e., "Tip" & the "Target").

5. Create new IK Group as "Damped" and added respective elements.

6. Create new IK Group as "Un-Damped" and added respective elements.

7. Made the object respond-able.

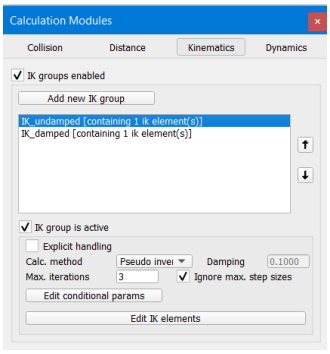

Figure 4: Creation of Inverse Kinematics (IK) Groups

8. Enabled "Dynamics" of the scene.

9. Selected DLS method for configuration.

10. Scripted (programmed) the robot.

11. Typed the desired cartesian position to bring the end-effector.

12. Started Simulation.

The following code was executed in order to get the desired outcome:

```
function sysCall_threadmain()
ball= sim.getObjectHandle('ManipSphere')
base=
sim.getObjectHandle('LBR_iiwa_14_R820')
   maxVel={0.05,0.05,0.05,0.5}
   maxAcc={0.01,0.01,0.01,0.5}
   maxJerk={0.00,0.00,0.00,0}
   tarPos={0.14,0.14,0.5}
   tarQuat=nil
   tarVel={0,0,0,0}
   curVel={0,0,0,0}
   curAcc={0,0,0,0}
r,nPos,nQuat,nVel,nAcc,tL =
sim.rmlMoveToPosition(ball, base,-1, curVel,
curAcc, maxVel, maxAcc, maxJerk, tarPos,
tarQuart, tarVel)
end
```

The end effector of the robot successfully moves to any desired cartesian position. Also, the manipulator moves with any desired end-point translational and rotational cartesian velocity to move to the desired cartesian position.

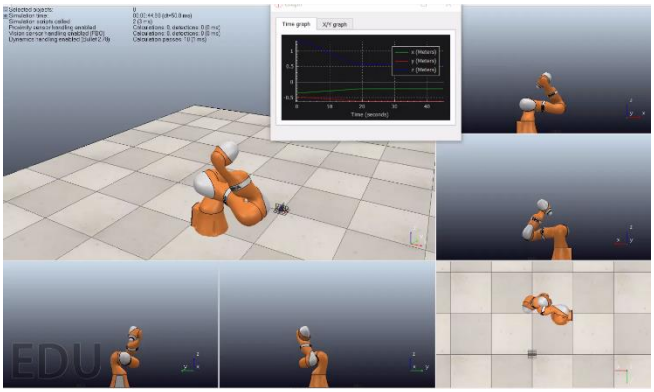

Figure 5: Successful simulation of the end effector moving to any desired cartesian position with any desired end-point translational and rotational cartesian velocity.

2.2 Following and planning any given Cartesian Position Trajectory

The concept of following a cartesian position was implemented using a real-world application which is “Pick & Place”.

Pick and place robots can be tailored to meet unique production needs thanks to a variety of end-of-arm tooling (EOAT) choices, like the KUKA LBR IIWA. Moving big, small, heavy, or difficult-to-handle goods along a manufacturing line is a simple task to automate. If desired, these robots can be easily programmed and tooled to perform a variety of tasks.

Advantages Of Pick-And-Place

- A robot will be able to pick and place more objects than a human operator on a regular basis, and it will be able to work around the clock to maximize throughput.
- A robot can grab and position objects in the same manner every time, making it more reliable than the human.
- With time, repetitive movements like those needed by pick and place can lead to musculoskeletal issues.
- They are ideal for high-volume, highly consistent product lines that involve the manipulation of small products, such as computer circuit manufacturing or assemblies of products to be assembled.

Steps Involved in the simulation process:

1. Enabled “Inverse Kinematics” mode to all the joints of the robot.

2. Added “Tip” & “Target” dummies to the Primitive Shape in the scene.

3. Linked both the dummies (i.e., “Tip” & the “Target”) and made the position and orientation of the Dummies same.

4. Added 2 conveyor belts, and 2 proximity sensors around the robot in the scene, with one proximity having a Conical Outlook that creates a Cloud of detection points.

5. Linked the proximity sensors and conveyors together to sync the motion of conveyor.

6. Added another dummy on the landing conveyor belt, linked the "Tip" & "Target" to the new dummy **to create a trajectory/path to the described cartesian position.**

7. Create multiple IK Groups and added respective elements.

8. Made all objects in the scene respond-able and dynamically active.

9. Selected DLS method for configuration.

10. Scripted (programmed) the robot and the conveyor belts.

11. Started Simulation.

The code executed to get the desired output can be found in 3.1 of Appendix section of the paper.

The robot successfully moves the Object to the desired cartesian position creating a trajectory between two dummies.

Trajectory can be seen in "BLUE" color in the top left frame.

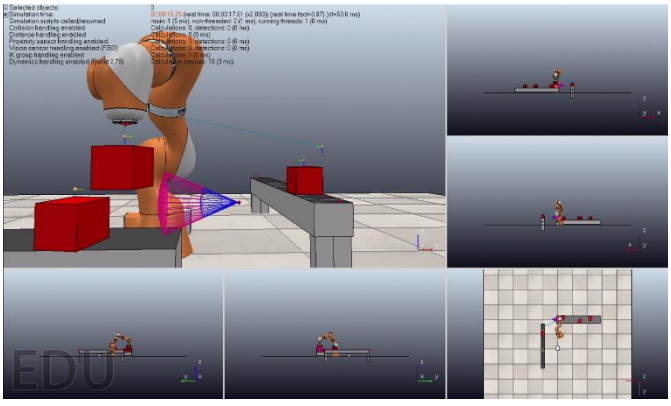

Figure 6: Successful simulation of the end effector creating a path between two dummy boxes and following the trajectory whilst picking and placing the boxes with precision.

2.3 Writing and Plotting the Joint Angles and Cartesian Positions along with plotting of Euler Angles of the position post movement of End-Point Manipulator

The code executed to get the write, and plot the joint and angles and cartesian position, with plotting Euler angles of the movement of end-point manipulator can be found in 3.2 of Appendix section of the paper.

CARTESIAN POSITION OF END EFFECTOR: (0.14,-0.14,0.5)		
JOINT #	JOINT ANGLE	
	RADIANS	DEGREES
1	0.468308777	26.82132084
2	-0.524682403	-30.04999215
3	-1.010821939	-57.89252921
4	-2.094532728	-119.9596017
5	-2.794757128	-160.0633628
6	-2.094389915	-119.9514224
7	-9.54E-07	-5.46E-05

Table 1: Cartesian position of the end-effector

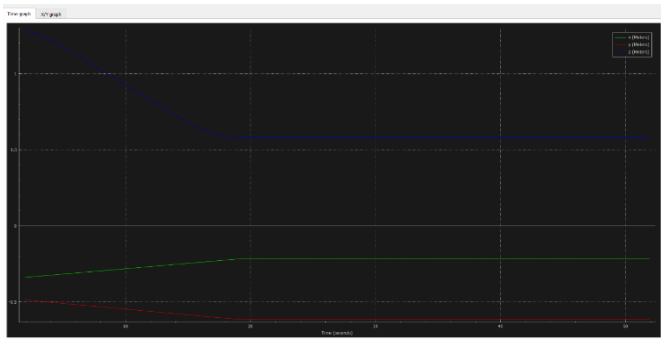

Figure 7: Plot of Cartesian Positions

Euler Angle of the of the End Effector was computed as,

eulerangle = sim.getEulerAnglesFromMatrix(matrix7)
print(eulerangle)

EULER ANGLE:
{0.0, 0.0, -9.5367431640625e-07}

Table 3: Euler angle of the end-effector

Jacobian matrix of the manipulator was also computed as follows:

Jacobian matrix					
0	0	0	0	0	0
0.1	0.0055	-0.062	2.6E-06	1	6.2E-05
0.041	0.062	0.073	0.87	-0.00016	-0.5
0.092	-0.2	0.28	-0.17	0.94	-0.29
0.074	0.22	0.095	-0.84	-0.29	-0.46
0.2	0.1	-0.055	-0.35	-0.35	0.87
0.14	0.14	0	-0.52	-0.7	-0.49

Table 2: Jacobian matrix of joint movements

Intrinsic Transformation Matrix of all seven Joints caused by the joint movements in 4 x 4 Matrix are computed and shown below.

Intrinsic Transformation Matrix of all Joints (Transformation caused by the Joint Movement) (4 x 4 Matrix)									
JOINT 1	0.892332971	-0.45137781	0	0					
	0.451377809	0.892332971	0	0					
	0	0	1	0					
	0	0	0	1	**JOINT 5**	-0.94045305	0.339923471	0	0
						-0.33992347	-0.94045305	0	0
						0	0	1	0
JOINT 2	0.865483046	0.500938237	1.68E-08	0		0	0	0	1
	-0.50093824	0.865483046	-4.51E-09	0					
	-1.68E-08	-4.51E-09	1	0					
	0	0	0	1	**JOINT 6**	-0.49999559	0.86602819	-1.09E-08	0
						-0.86602819	-0.49999559	1.89E-08	0
						1.09E-08	1.89E-08	1	0
JOINT 3	0.531164527	0.847268701	0	0		0	0	0	1
	-0.84726871	0.531164527	0	0					
	0	0	1	0					
	0	0	0	1	**JOINT 7**	1	9.54E-07	0	0
						-9.54E-07	1	0	0
						0	0	1	0
JOINT 4	-0.50011826	0.86595732	-1.89E-08	0		0	0	0	1
	-0.86595732	-0.50011826	3.27E-08	0					
	1.89E-08	3.27E-08	1	0					
	0	0	0	1					

Table 4: Intrinsic Transformation Matrix of all Joints caused by joint movements

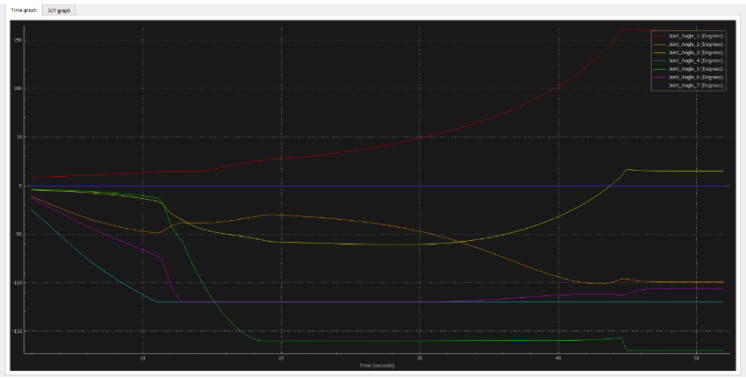

Figure 8: Plot of Joint Angles

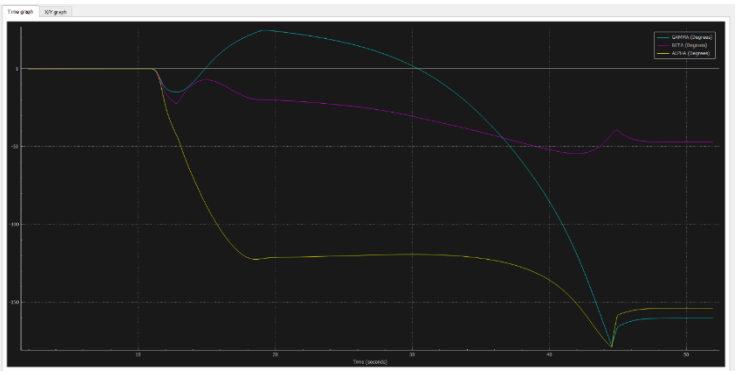

Figure 9: Plot of Euler Angles

2.4 Allowing the Robot Manipulator to move using Keyboard buttons and external remote or wired controllers such as Game joysticks.

For human intervention in-order to optimize the performance of the robot, the Jacobian matrices, Euler angles, Cartesian position of the endpoint effector were tapped previously and hence the robot was given the ability to be moved using PC keyboard buttons and external remote or wired controllers such as Game joysticks for improved calibration and precise movements.

Steps involved in the simulation process:

1. Enabled “Inverse Kinematics” mode to all the joints of the robot.

2. Added “Tip” & “Target” dummies to the Primitive Shape in the scene.

3. Linked both the dummies (i.e., “Tip” & the “Target”) and made the position and orientation of the Dummies same.

4. Created a Primitive Shape “Sphere”, linked to the dummies.

5. Create multiple IK Groups and added respective elements.

6. Made all objects in the scene respond-able and dynamically active, chose DLS method.

7. Scripted (programmed) the robot and the conveyor belts.

8. Scripted the robot to follow the sphere wherever the sphere moves, implemented Keyboard buttons for movements in X, Y and Z axes.

Note: There is no built-in function in CoppeliaSim that can sum up the numerical values of two tables, hence, a new function was researched upon and defined while scripting the robot.

The following keyboard keys were assigned the following movements:

D - positive X axis

A - negative X axis

E - positive Y axis

F - negative Y axis

W - positive Z axis

S - negative Z axis

The code executed to get the desired output can be found in 3.3 of Appendix section of the paper.

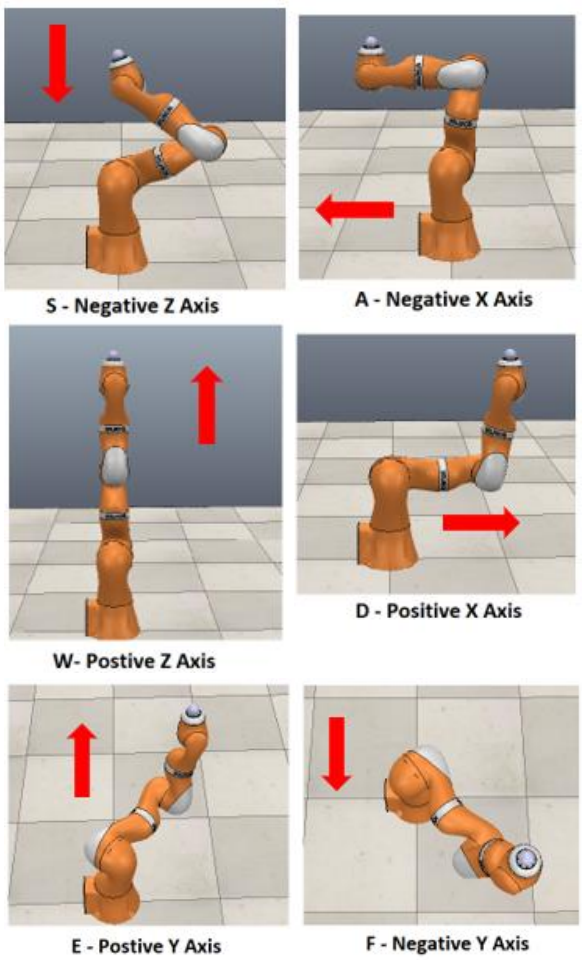

Figure 10: Successful simulation of the movements of the robot on pressing allocated keyboard

Conclusion

‘Smart factories’ are evolving at an unparalleled pace in the era of Industry 4.0. As businesses strive to adapt rapidly and flexibly to ever shortening product life cycles; automated, networked, and customizable production lines are in high demand. At the same time, the question of how to integrate unique customer requirements into industrial series production is becoming extremely important. The LBR's position and compliance control enable it to handle delicate components without crushing or shearing them. Unlike conventional fixed robotic systems, the LBR IIWA is highly adaptable, allowing it to be relocated and reprogrammed to perform a variety of tasks, including assisting human co-workers. Hence the aim of the project was achieved with ease, to assist mega-factories in precise pick-place applications while ensuring accurate trajectory planning of the end-point effector.

References

[1] Kuka.com. 2022. [online] Available at: <https://www.kuka.com/-/media/kuka-downloads/imported/9cb8e311bfd744b4b0eab25ca883f6d3/kuka_lbr_iiwa_brochure_en.pdf?rev=5a25f7eac825492e92af6343dbf5bc6b&hash=43592AD8D79B2D3AD91B3129988A8B8C>

[2] RoboDK blog. 2022. The Ultimate Guide to Pick and Place Robots - RoboDK blog. [online] Available at: <https://robodk.com/blog/guide-pick-and-place-robots/>

[3] Kavati, Veladri & Nizamuddin. "Modeling and Simulation of 7-dof Robotic Manipulator", Conference at Technological Advancements in Mechanical Engineering (TAME) - 2016.

[4] Assemblymag.com. 2022. | *ASSEMBLY*. [online] Available at: <https://www.assemblymag.com/articles/94138-cobot-lends-a-hand-to-improve-siemens-motor-production>

3. APPENDIX

PROGRAMMING DATA (CODE):

APPENDIX 3.1

```
-- Acc & Vel of Conveyor
Vel = 0.5
Acc = 0.2
tar = sim.getObjectHandle("Tar")
-- Specs of Box
boxCoordinate = {-1.1,-0.6,0.33}
boxColor = {1,0,0}
conveyor1 = sim.getObjectHandle("ConveyorBelt1")

function sysCall_threadmain()
tip = sim.getIkGroupHandle('KUKA')
while
sim.getSimulationState()~=sim.simulation_advancing_abou
ttostop do
      sim.waitForSignal("objectAvailable")
            -- Get handle
      path = sim.getObjectHandle("pick")
      sim.followPath(tar,path,2,1,Vel,Acc)
            -- Delay the signal for a second
      sim.wait(1)
            -- Follow the path
      sim.followPath(tar,path,2,-1,-Vel,-Acc)
            -- Follow the second path
      sim.followPath(tar,path2,2,1,Vel,Acc)
-- Delay the signal for quarter of a second
sim.wait(0.25)
-- Remove/detach dummy from connecting dummy
sim.setLinkDummy(dummy,0)
-- Follow back path2 to 0 position of the Robot
sim.followPath(tar,path2,2,-1,-Vel,-Acc)
end
end

-- Proximity sensing
function sysCall_sensing()
proximityResponse1 = sim.readProximitySensor(prox1)
if objectAvailable = true
insertBox()
else
hasStopped = false
sim.removeObject(“Box”)
-- Creating paths between dummies
path2 = createPath("path2",pos0,orient0,path2,orient2)
-- Obtain script of the Robot
scriptRobot = sim.getScriptHandle("LBR_iiwa_14_R820")
sim.setScriptVariable("path2", scriptRobot,path2)

function insertBox()
-- Deploying object Boxes simultaneously
boxInserted = sim.copyPasteObjects({Box,BoxDummy},0)
-- Store handles to boxes and dummies
table[] boxList = {} -– inserts boxes in an array
if boxColor = true
table.insert(boxList,insertedObjects[1])
else
   table.remove(boxDummyList,insertedObjects[1])
end
```

APPENDIX 3.2

PLOTS JOINT ANGLES

```
function sysCall_threadmain()
  ball= sim.getObjectHandle('ManiSphere')
  base= sim.getObjectHandle('LBR_iiwa_14_R820')
  joint1=
sim.getObjectHandle('LBR_iiwa_14_R820_joint1')
  joint2=
sim.getObjectHandle('LBR_iiwa_14_R820_joint2')
  joint3=
sim.getObjectHandle('LBR_iiwa_14_R820_joint3')
  joint4=
sim.getObjectHandle('LBR_iiwa_14_R820_joint4')
  joint5=
sim.getObjectHandle('LBR_iiwa_14_R820_joint5')
  joint6=
sim.getObjectHandle('LBR_iiwa_14_R820_joint6')
  joint7=
sim.getObjectHandle('LBR_iiwa_14_R820_joint7')
    maxVel={0.05,0.05,0.05,0.5}
    maxAcc={0.01,0.01,0.01,0.5}
    maxJerk={0.00,0.00,0.00,0}
    tarPos={0.14,0.14,0.5}
    tarQuat=nil
    tarVel={0,0,0,0}
    curVel={0,0,0,0}
    curAcc={0,0,0,0}

r,nPos,nQuat,nVel,nAcc,tL=sim.rmlMoveToPosition(ball,b
ase,-
1,curVel,curAcc,maxVel,maxAcc,maxJerk,tarPos,tarQuart,t
arVel)
    angle1= sim.getJointPosition(joint1)
    print(angle1)
    print(angle1*180*7/22)
    matrix1=sim.getJointMatrix(joint1)
    print(matrix1)
    angle2= sim.getJointPosition(joint2)
    print(angle2)
    print(angle2*180*7/22)
    matrix2=sim.getJointMatrix(joint2)
    print(matrix2)
    angle3 = sim.getJointPosition(joint3)
    print(angle3)
    print(angle3*180*7/22)
    matrix3=sim.getJointMatrix(joint3)
    print(matrix3)
    angle4 = sim.getJointPosition(joint4)
    print(angle4)
    print(angle4*180*7/22)
    matrix4=sim.getJointMatrix(joint4)
    print(matrix4)
    angle5 = sim.getJointPosition(joint5)
    print(angle5)
    print(angle5*180*7/22)
    matrix5=sim.getJointMatrix(joint5)
    print(matrix5)
    angle6 = sim.getJointPosition(joint6)
    print(angle6)
    print(angle6*180*7/22)
    matrix6=sim.getJointMatrix(joint6)
    print(matrix6)
```

```
    angle7 = sim.getJointPosition(joint7)
    print(angle7)
    print(angle7*180*7/22)
    matrix7=sim.getJointMatrix(joint7)
    print(matrix7)
```

PLOTS EULER ANGLES

```
 eulerangle  = sim.getEulerAnglesFromMatrix(matrix7)
    print(eulerangle)
```

PLOTS JACOBIAN MATRIX

```
tip = sim.getIkGroupHandle('IK_undamped')
    print(sim.computeJacobian(tip,0))
jacobian,jacobianSize=sim.getIkGroupMatrix(tip,0)
    for i=1,jacobianSize[1],1 do
    str=''
    for j=1,jacobianSize[2],1 do
    if #str~=0 then
    str=str..', '
    end
str=str..string.format("%.1e",jacobian[(j-
1)*jacobianSize[1]+i])
    end
    print(str)
    end
```

APPENDIX 3.3

There is no built-in function in CoppeliaSim that can sum up the numerical values of two tables, hence, a new function was researched upon and defined while scripting the robot.

```
function sumElements(t1,t2)
  local result = {}
  for i = 1, math.min(#t1, #t2) do
  result[i] = t1[i] + t2[i]
  end
  return result
end

function sysCall_threadmain()
while
sim.getSimulationState()~=sim.simulation_advancing_abou
ttostop do

-- Read the keyboard messages (make sure the focus is on
the main window, scene view):
      message,auxiliaryData= sim.getSimulatorMessage()
        while message~=-1 do
        if (message==sim.message_keypress) then
if (auxiliaryData[1]==string.byte('w')) then
-- "W" key --- +ve movement on Z axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {0,0,0.01}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
end
-----------------------------------------
if (auxiliaryData[1]==string.byte('s')) then
-- "S" key --- -ve movement on Z axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {0,0,-0.01}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
end
-----------------------------------------
if (auxiliaryData[1]==string.byte('d')) then
-- "D" key --- +ve movement on X axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {0.01,0,0}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
                end
-----------------------------------------
if (auxiliaryData[1]==string.byte('a')) then
-- "A" key --- -ve movement on X axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {-0.01,0,0}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
end
-----------------------------------------
if (auxiliaryData[1]==string.byte('e')) then
-- "E" key --- +ve movement on Y axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {0,0.01,0}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
end
-----------------------------------------
if (auxiliaryData[1]==string.byte('f')) then
-- "F" key --- -ve movement on Y axis
-----------------------------------------
spherePos = sim.getObjectHandle('manipSphere')
positionSphere = sim.getObjectPosition(spherePos,-1)
deltaPos = {0,-0.01,0}
newPos = sumElements(positionSphere,deltaPos)
sim.setObjectPosition(spherePos,-1,newPos)
end
-----------------------------------------
End

message,auxiliaryData= sim.getSimulatorMessage()
end
   sim.switchThread()
end
   end
```