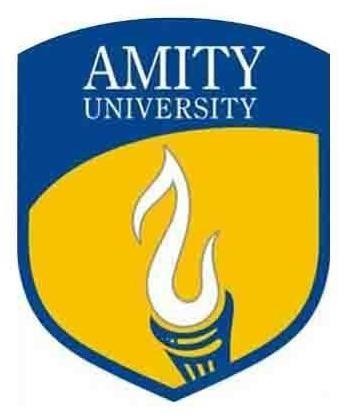
**Amity University Madhya Pradesh, Gwalior**



**CSA 621**

**Artificial Intelligence for Robotics Lab**

# Submitted to: Submitted by:

Dr. Deepak Motwani Harsh Kumar Gupta

ASET – CSE Enroll-No: A60205220030 B.Tech CSE [A]

ASET, VI – Semester

**Table of Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **PROGRAMS LIST** | **PAGE NO** | **REMARK** |
| 1. | Block Word Problem In AI (Robotics) | 3 - 5 |  |
| 2. | Demonstration of Robot with 2DOF,3 DOF,  4DOF etc. | 6 - 11 |  |
| 3. | Puma robot DH-Parameters | 12 - 13 |  |
| 4. | Scara robot DH-Parameters. | 14 - 15 |  |
| 5. | Robot programming and simulation for  Shape identification | 16 - 19 |  |
| 6. | Determination of maximum and minimum  position of links. | 20 - 20 |  |
| 7. | 2-D Path Tracing With Inverse Kinematics | 21 - 23 |  |
| 8. | Explain Top 10 Python Libraries For Robotics  1| Robot Framework. 2| Pyro  3| DART  4| PyRobot 5| PyDy  6| Simulation Open Framework Architecture. 7| Klamp't  8| Pybotics 9| Siconos  10 | iDynTree | 24 - 26 |  |
| 9. | Walking Robot -- DDPG Agent Training  Script (2D) | 27 - 32 |  |
| 10. | Describe robotics programming for ROS | 33 - 34 |  |
| 11. | Explain the different types of robot’s sensors with example? | 35 - 39 |  |
| 12. | Write the script of Talker and Listener in  python and show the output. | 40 - 44 |  |
| 13. | How to create your first package in ROS, specify each command in chronological way. | 45 - 48 |  |
| 14. | Explain the navigation concept in robotics? | 49 - 50 |  |
| 15. | Explain Topological Path Planning and  Metric Path Planning? | 51 - 52 |  |

# PROGRAM 1:

## Block Word Problem in AI (Robotics)? Answer:

### What is Blocks World Problem?

This is how the problem goes — There is a table on which some blocks are placed. Some blocks may or may not be stacked on other blocks. We have a robot arm to pick up or put down the blocks. The robot arm can move only one block at a time, and no other block should be stacked on top of the block which is to be moved by the robot arm.

Our aim is to change the configuration of the blocks from the Initial State to the Goal State, both of which have been specified in the diagram above.

### What is Goal Stack Planning?

Goal Stack Planning is one of the earliest methods in artificial intelligence in which we work

### backwards from the goal state to the initial state.

We start at the goal state and we try fulfilling the preconditions required to achieve the initial state. These preconditions in turn have their own set of preconditions, which are required to be satisfied first. We keep solving these “goals” and “sub-goals” until we finally arrive at the Initial State. **We make use of a stack to hold these goals that need to be fulfilled as well the actions that we need to perform for the same**.

Apart from the “Initial State” and the “Goal State”, we maintain a **“World**

**State”** configuration as well. Goal Stack uses this world state to work its way from Goal State to Initial State. World State on the other hand starts off as the Initial State and ends up being transformed into the Goal state.

At the end of this algorithm we are left with an empty stack and a set of actions which helps us navigate from the Initial State to the World State.

### Representing the configurations as a list of “predicates”

Predicates can be thought of as a statement which helps us convey the information about a configuration in Blocks World.

Given below are the list of predicates as well as their intended meaning

1. ON(A,B) : Block A is on B
2. ONTABLE(A) : A is on table 3. CLEAR(A) : Nothing is on top of A
3. HOLDING(A) : Arm is holding A.
4. ARMEMPTY : Arm is holding nothing

Using these predicates, we represent the Initial State and the Goal State in our example like this:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Initial State** — ON(B,A) | ONTABLE(A) | ONTABLE(C) | ONTABLE(D) | CLEAR(B) |
| CLEAR(C) CLEAR(D) | ARMEMPTY |  |  |  |

**Goal State** — ON(C,A) ON(B,D) ONTABLE(A) ONTABLE(D) CLEAR(B) CLEAR(C) ARMEMPTY

Thus a configuration can be thought of as a list of predicates describing the current scenario.

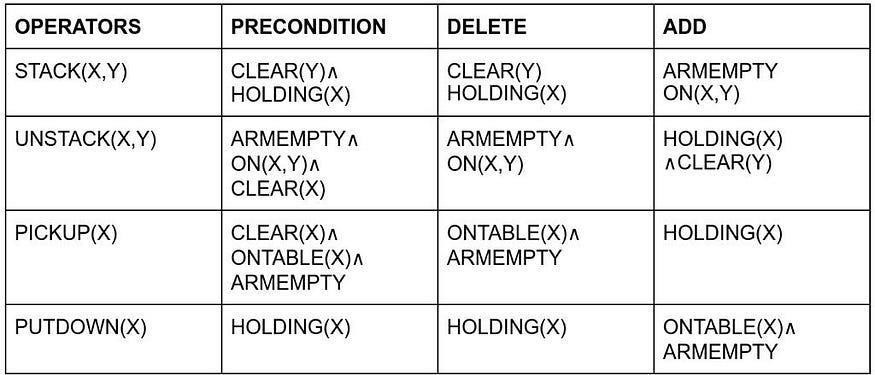
**“Operations” performed by the robot arm** The Robot Arm can perform 4 operations:

1. STACK(X,Y) : Stacking Block X on Block Y
2. UNSTACK(X,Y) : Picking up Block X which is on top of Block Y
3. PICKUP(X) : Picking up Block X which is on top of the table
4. PUTDOWN(X) : Put Block X on the table

All the four operations have certain preconditions which need to be satisfied to perform the same. These preconditions are represented in the form of predicates.

The effect of these operations is represented using two lists **ADD** and **DELETE**. DELETE List contains the predicates which will cease to be true once the operation is performed. ADD List on the other hand contains the predicates which will become true once the operation is performed.

The Precondition, Add and Delete List for each operation is rather intuitive and have been listed below.



Operations performed by the Robot Arm

For example, to perform the **STACK(X,Y)** operation i.e. to Stack Block X on top of Block Y, No other block should be on top of Y **(CLEAR(Y))** and the Robot Arm should be holding the Block X (**HOLDING(X)**).

Once the operation is performed, these predicates will cease to be true, thus they are included in **DELETE List** as well. (Note : It is not necessary for the Precondition and DELETE List to be the exact same).

On the other hand, once the operation is performed, The robot arm will be free (**ARMEMPTY**) and the block X will be on top of Y (**ON(X,Y)**).

The other 3 Operators follow similar logic, and this part is the cornerstone of Goal Stack Planning.

# PROGRAM 2:

## Demonstration of Robot with 2DOF, 3 DOF, 4DOF? Code:

### 2DOF:

%%

clc; clear all; close all;

%% Graphic

g = ncgr\_graphic(); global N\_DOFS; N\_DOFS = 2;

%% 2-R-planar robot DH- Parameters theta = [0 0]; alpha = [0

0]; offset = [0 0]; d = [0 0];

a = [0.5 0.5];

type = ['r' 'r']; base = [0; 0; 0]; planar\_2r = cgr\_create(theta, d, a, alpha, offset, type, base, ...

[pi/2 pi/2], [-pi/2 -pi/2]); % joint limts! planar\_2r = cgr\_self\_update(planar\_2r, [0 0], base);

g = ncgr\_plot(g, planar\_2r, [0 0 1], 1); % view\_vector = [0 0 1] => top view for k = 0:0.1:pi

planar\_2r = cgr\_self\_update(planar\_2r, [0+k 0+k], base); end type = ['r','r']; base = [0; 0; 0]; planar\_2r = cgr\_create(theta,

d, a, alpha, offset, type, base, ...

[pi/2; pi/2], [-pi/2; -pi/2;]); % joint limts! planar\_2r = cgr\_self\_update(planar\_2r, [0; 0], base);

g = ncgr\_plot(g, planar\_2r, [0 0 1], 1); for k = 0:0.1:pi

planar\_2r = cgr\_self\_update(planar\_2r, [0+k; 0+k], base);

%6bab2b9b70984b137f8ff706abc6a7585b277821 g = ncgr\_plot(g, planar\_2r, [0 0 1], 1);

pause(0.5); end

### Output:

**3DOF:**

%%

clc; clear all; close all;

%% Graphic

g = ncgr\_graphic(); global N\_DOFS; N\_DOFS = 3;

%% 3-R-planar robot DH- Parameters theta = [0 0 0]; alpha = [0

0 0]; offset = [0 0 0]; d = [0 0 0]; a =

[0.5 0.5 0.5];

type = ['r''r''r']; base = [0; 0; 0]; planar\_3r = cgr\_create(theta, d, a, alpha, offset, type, base, ...

[pi/2 pi/2 pi/2], [-pi/2 -pi/2 -pi/2]); % joint limts! planar\_3r

= cgr\_self\_update(planar\_3r, [0; 0; 0], base); g = ncgr\_plot(g,

planar\_3r, [1 1 1], 0.3, [0 2], [-1 1], [-0.1 0.1]);

%% Demo for inverse kinematics for ii = 0: 0.1: 0.4

base(1) = ii; for x = 1.0+ii :0.1: 1.5+ii

[q, k, err]= cgr\_ikine1(planar\_3r, [x; 0; 0], 0.01, 100);

%[q, k, err]= cgr\_ikine2(planar\_4r, [x; 0; 0], 2, 0.01, 100); planar\_3r = cgr\_self\_update(planar\_3r, q, base); g = ncgr\_plot(g, planar\_3r);

pause(0.1); end

end

%% Demo for inverse kinematics for ii = 0: 0.1: 0.4 base(1) = ii; for x = 1.0+ii

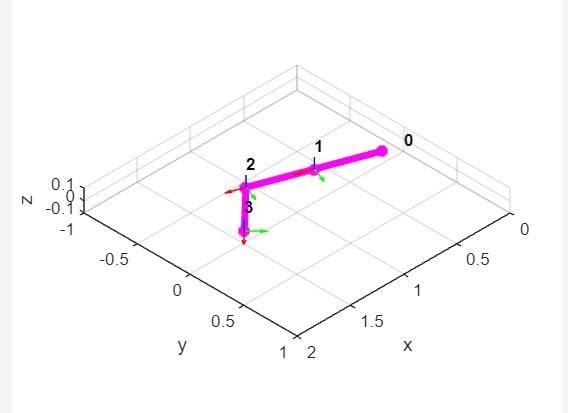
:0.1: 1.5+ii

%[q, k, err]= cgr\_ikine1(planar\_4r, [x; 0; 0], 0.01, 100);

[q, k, err]= cgr\_ikine2(planar\_3r, [x; 0; 0], 2, 0.01, 100); planar\_3r = cgr\_self\_update(planar\_3r, q, base); g = ncgr\_plot(g, planar\_3r);

### Output:

pause(0.1); end end



### 4DOF:

%%

clc; clear all; close all;

%% Graphic

g = ncgr\_graphic(); global N\_DOFS; N\_DOFS = 4;

%% 4-R-planar robot DH- Parameters theta = [0 0 0 0]; alpha =

[0 0 0 0]; offset = [0 0 0 0]; d = [0 0

0 0]; a = [0.5 0.5 0.5 0.5];

type = ['r' 'r' 'r' 'r']; base = [0; 0; 0]; planar\_4r = cgr\_create(theta, d, a, alpha, offset, type, base, ... [pi/2; pi/2; pi/2; pi/2], [-pi/2; - pi/2; -pi/2; -pi/2]); % joint limts! planar\_4r = cgr\_self\_update(planar\_4r, [0; 0; 0; 0], base); g = ncgr\_plot(g,

planar\_4r, [1 1 1], 0.3, [0 2], [-1 1], [-0.1 0.1]);

%% Demo for inverse kinematics for ii = 0: 0.1: 0.4

base(1) = ii;

for x = 1.0+ii :0.1: 1.5+ii

[q, k, err]= cgr\_ikine1(planar\_4r, [x; 0; 0], 0.01, 100);

%[q, k, err]= cgr\_ikine2(planar\_4r, [x; 0; 0], 2, 0.01, 100); planar\_4r = cgr\_self\_update(planar\_4r, q, base); g = ncgr\_plot(g, planar\_4r);

pause(0.1); end

end

%% Demo for inverse kinematics for ii = 0: 0.1: 0.4 base(1) = ii;

for x = 1.0+ii :0.1: 1.5+ii

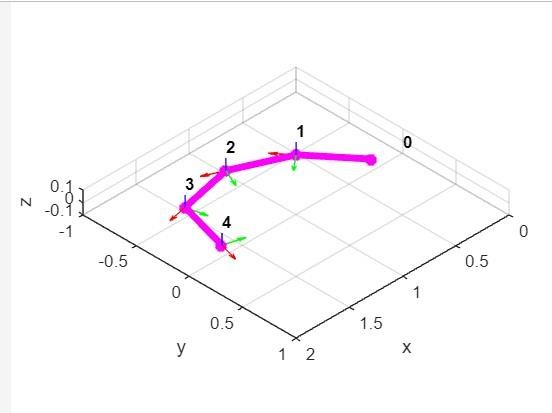
%[q, k, err]= cgr\_ikine1(planar\_4r, [x; 0; 0], 0.01, 100);

[q, k, err]= cgr\_ikine2(planar\_4r, [x; 0; 0], 2, 0.01, 100);

### Output:

planar\_4r = cgr\_self\_update(planar\_4r, q, base); g = ncgr\_plot(g, planar\_4r);

pause(0.1); end end



# PROGRAM 3:

## Puma robot DH-Parameters? Code:

%%

clc; clear all; close all;

%% Graphic

g = ncgr\_graphic();

%% Puma robot DH-Parameters global N\_DOFS;

N\_DOFS = 6;

theta = [0 0 0 0 0 0]; alpha = [- pi/2 0 pi/2 -pi/2 pi/2 0]; offset = [0 0 0 0 0 0]; a = [0 8 0 0 0 0]; %

in inches d = [13 0 2.5 8 0 2.5];

% in inches

type = ['r' 'r' 'r' 'r' 'r' 'r']; base = [0; 0; 0];

% See <http://medesign.seas.upenn.edu/index.php/Courses/MEAM520-12C-P01-IK>lb = [deg2rad(-180) deg2rad(-75) deg2rad(-235) deg2rad(-580) deg2rad(-120) deg2rad(- 215)];

ub = [deg2rad(110) deg2rad(240) deg2rad(60) deg2rad(40) deg2rad(110) deg2rad(295) ]; puma = cgr\_create(theta, d, a, alpha, offset, type, base, ub, lb); puma = cgr\_self\_update(puma, [0 0 0 0 0 0]);

g = ncgr\_plot(g, puma, [1 1 1], 0.3, [-5 20], [-10 10], [0 20]);

%% Demo inverese kinematics x = puma.T(1:3,4, end); step = linspace(0, 10, 20);

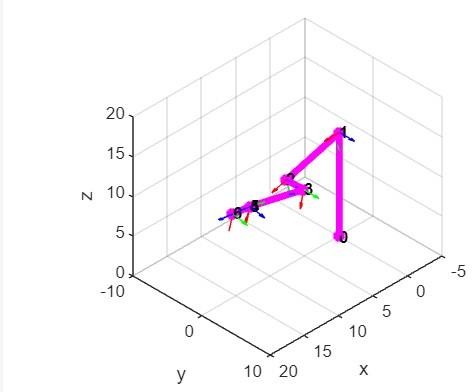
for i = 1:length(step)

[q, k, err] = cgr\_ikine1(puma, [x(1)+step(i); x(2); 13], 0.001, 1000); puma = cgr\_self\_update(puma, q); g = ncgr\_plot(g, puma);

pause(0.1); end

### Output:

**Output:**



# PROGRAM 4:

## Scara robot DH-Parameters?

**Code:**

%%

clc; clear all; close all;

%% Graphic

g = ncgr\_graphic(); %% Scara robot DH-Parameters global N\_DOFS; N\_DOFS = 4;

theta = [0 0 0 0];

alpha = [0 0 0 0];

offset = [0 0 0 0]; a

= [0 0.45 0.72 0];

d = [0.21 0 0 0];

type = ['r' 'r' 'r' 'p']; base

= [0; 0; 0];

scara = cgr\_create(theta, d, a, alpha, offset, type, base); scara = cgr\_self\_update(scara, [0 0 0 0]); g = ncgr\_plot(g, scara); pause(1);

scara = cgr\_self\_update(scara, [0 0 0 -0.2]); g = ncgr\_plot(g, scara); pause(1);

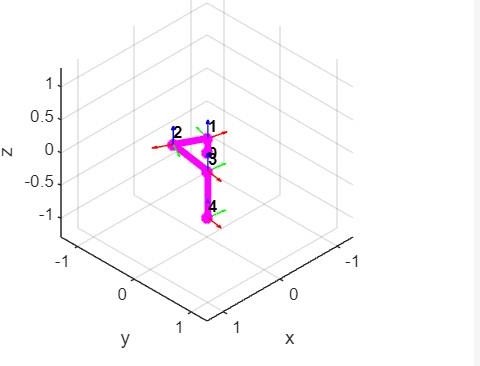
scara = cgr\_self\_update(scara, [0 0 0 0]); g

= ncgr\_plot(g, scara);

%% Demo inverese kinematics

[q, k, err]= cgr\_ikine1(scara, [0.5; 0.5; -0.5], 0.01, 100); scara = cgr\_self\_update(scara, q); g = ncgr\_plot(g, scara); pause(0.1);

**Output:**



# PROGRAM 5:

## Robot programming and simulation for Shape identification?

**Code:**

clc; % Clear the command window.

close all; % Close all figures (except those of imtool.) imtool close all; % Close all imtool figures. clear; % Erase all existing variables. workspace; % Make sure the workspace panel is showing. fontSize = 20;

% Open an image.

% Browse for the image file.

[baseFileName, folder] = uigetfile('.', 'Specify an image file'); fullImageFileName = fullfile(folder, baseFileName); if folder == 0

return; end

% Read in image into an array.

[rgbImage, storedColorMap] = imread(fullImageFileName); [rows, columns, numberOfColorBands] = size(rgbImage) % If it's monochrome (indexed), convert it to color.

% Check to see if it's an 8-bit image needed later for scaling). if strcmpi(class(rgbImage), 'uint8') % Flag for 256 gray levels.

eightBit = true; else eightBit = false;

end

if numberOfColorBands == 1 if isempty(storedColorMap)

% Just a simple gray level image, not indexed with a stored color map.

% Create a 3D true color image where we copy the monochrome image into all 3 (R, G, & B) color planes.

rgbImage = cat(3, rgbImage, rgbImage, rgbImage); else

% It's an indexed image.

rgbImage = ind2rgb(rgbImage, storedColorMap);

% ind2rgb() will convert it to double and normalize it to the range 0-1.

% Convert back to uint8 in the range 0-255, if needed. if eightBit

rgbImage = uint8(255 \* rgbImage); end end

end

% Display the original image.

subplot(2, 2, 1); imshow(rgbImage); set(gcf, 'Position', get(0,'Screensize')); % Enlarge figure to full screen.

set(gcf,'name','Image Analysis Demo','numbertitle','off') drawnow; % Make it display immediately. if numberOfColorBands > 1 title('Original Color Image', 'FontSize', fontSize);

grayImage = rgbImage(:,:,1); else

caption = sprintf('Original Indexed Image\n(converted to true color with its stored colormap)');

title(caption, 'FontSize', fontSize); grayImage = rgbImage; end % Display it. subplot(2, 2, 2); imshow(grayImage, []);

title('Grayscale Image', 'FontSize', fontSize);

% Binarize the image. binaryImage = grayImage < 100;

% Display it. subplot(2, 2, 3); imshow(binaryImage, []);

title('Binary Image', 'FontSize', fontSize);

% Remove small objects. binaryImage = bwareaopen(binaryImage, 300); % Display it. subplot(2, 2, 4); imshow(binaryImage, []); title('Cleaned Binary Image', 'FontSize', fontSize);

[labeledImage numberOfObjcts] = bwlabel(binaryImage);

blobMeasurements = regionprops(labeledImage,'Perimeter','Area', 'Centroid');

% for square ((a>17) && (a<20))

% for circle ((a>13) && (a<17))

% for triangle ((a>20) && (a<30))

circularities = [blobMeasurements.Perimeter].^2 ./ (4 \* pi \* [blobMeasurements.Area]) hold on;

% Say what they are for blobNumber = 1 : numberOfObjcts if circularities(blobNumber) < 1.19 message = sprintf('The circularity of object #%d is %.3f, so the object is a circle',... blobNumber, circularities(blobNumber));

theLabel = 'Circle'; elseif circularities(blobNumber) < 1.53 message = sprintf('The circularity of object #%d is %.3f, so the object is a Rectangle',... blobNumber, circularities(blobNumber));

theLabel = 'Rectangle'; else message = sprintf('The circularity of object #%d is

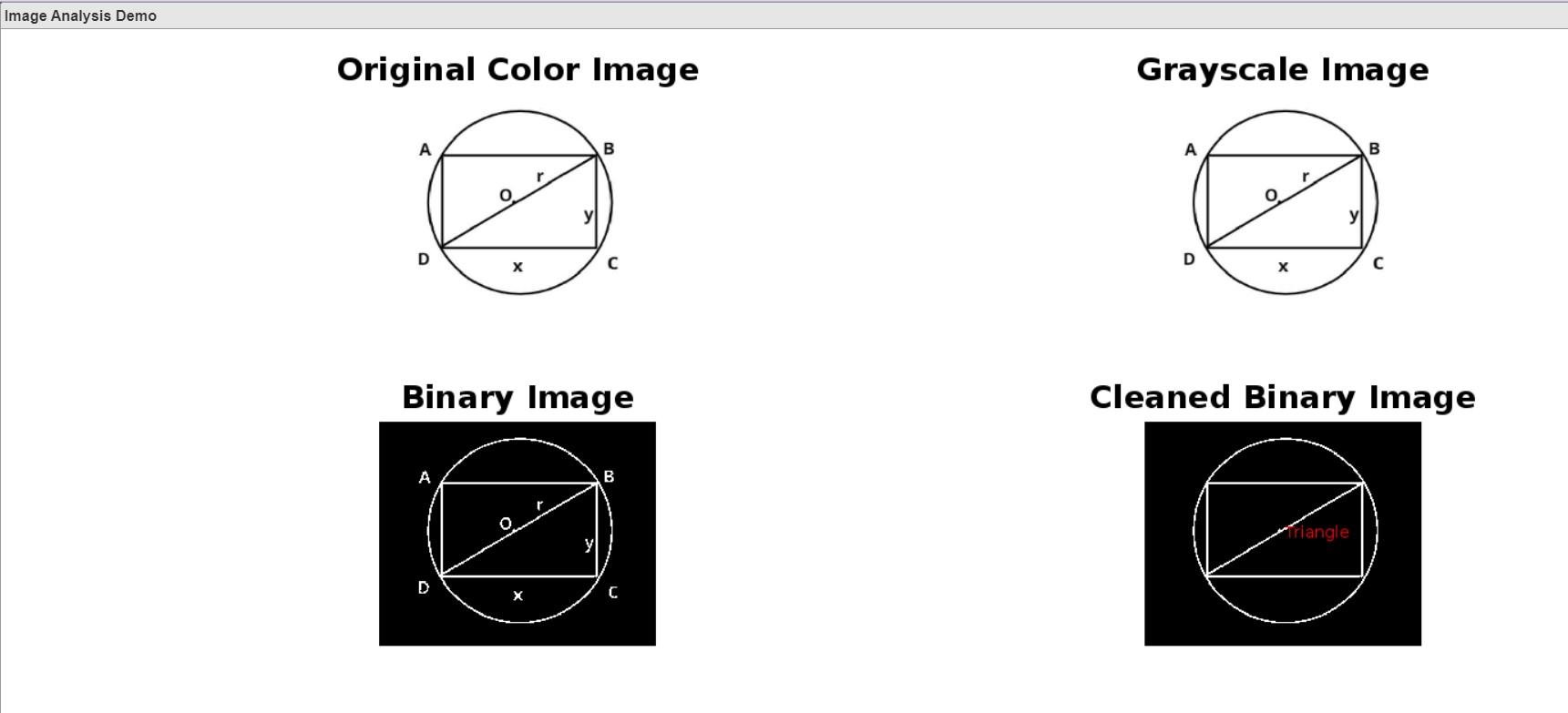
%.3f, so the object is a triangle',... blobNumber, circularities(blobNumber)); theLabel = 'Triangle'; end

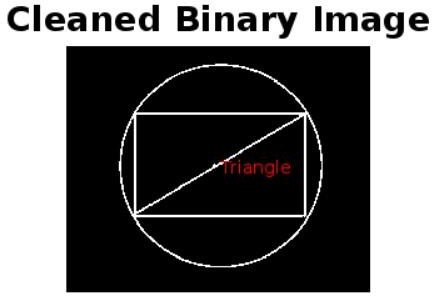
text(blobMeasurements(blobNumber).Centroid(1), blobMeasurements(blobNumber).Centroid(2),...

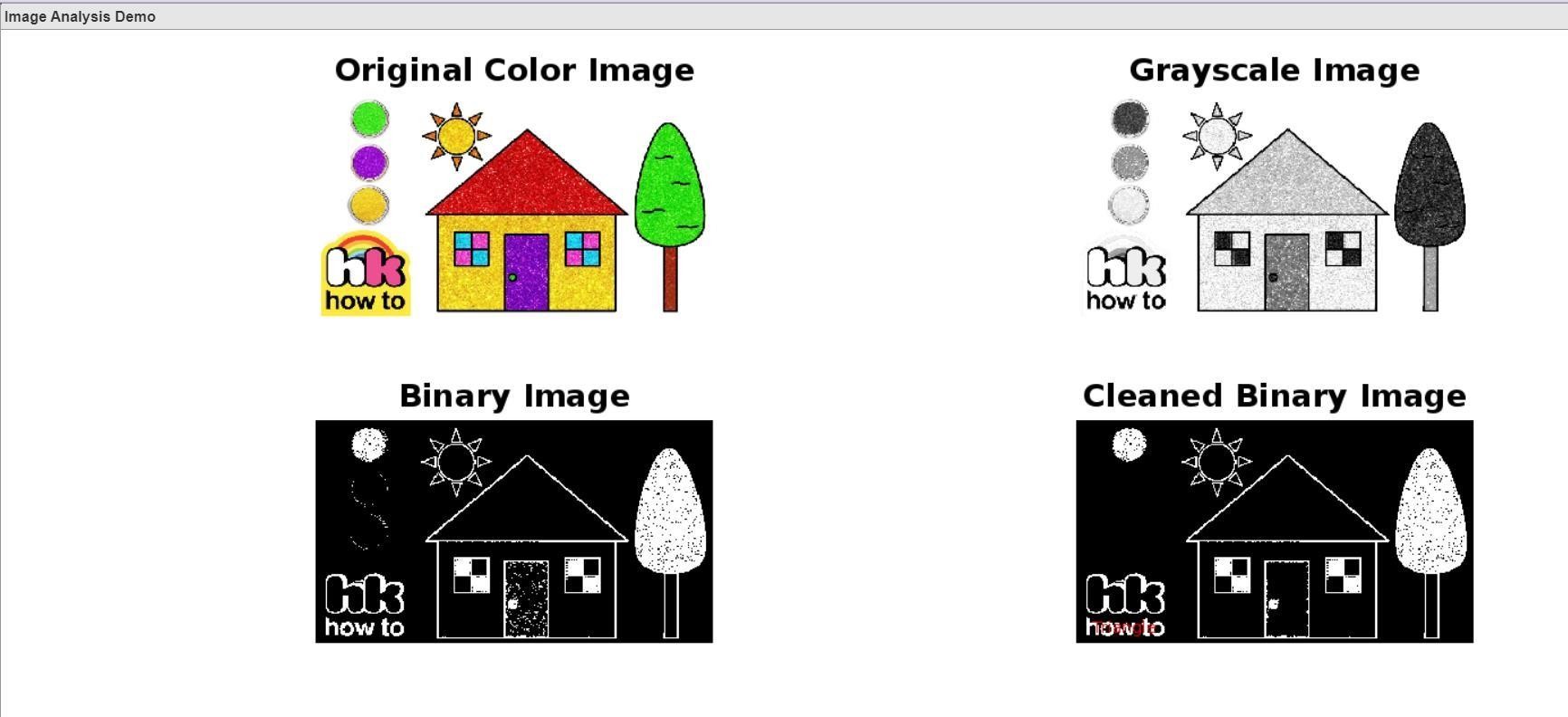
theLabel, 'Color', 'r');

uiwait(msgbox(message)); end

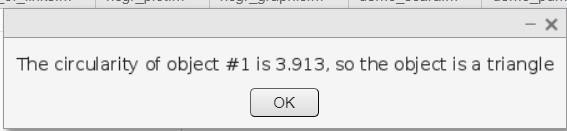
**Output:**







Identified Triangle shape in sun and home & Rectangle shape in door of house.





# PROGRAM 6:

## Determination of maximum and minimum position of links? Code:

figure;

% f(x) = (x-2)/((x-2)^4 +2)^1.8 F = @(x) ((x-2)/((x-2).^4 + 2).^1.8);

fplot(F,[0 9]); xlabel('x'); ylabel('y');

title('p9'); % FIX THIS ENTIRE PROBLEM

ylim([-0.2, 0.2]); hold on;

p09xmax = fminbnd('-((x-2)/((x-2).^4 + 2).^1.8)',2.3,2.6) p09xmin

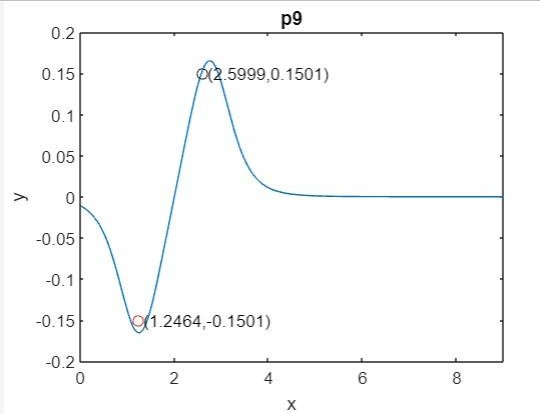
= fminbnd(F,1,1.3)

p09ymax = fminbnd('-((x-2)/((x-2).^4 + 2).^1.8)',0.15,0.2) p09ymin = fminbnd(F,-0.2,-0.15) plot(p09xmax,p09ymax,'ok'); plot(p09xmin,p09ymin,'or'); txt1 = ['(', num2str(p09xmax,'%0.4f'),',',

num2str(p09ymax,'%0.4f'),')']; txt2 = ['(',

num2str(p09xmin,'%0.4f'),',', num2str(p09ymin,'%0.4f'),')']; tx1 = p09xmax + 0.1; tx2 = p09xmin + 0.1; text(tx1,p09ymax,txt1); text(tx2,p09ymin,txt2); hold off; clear F txt1 txt2 tx1 tx2;

**Output:**



# PROGRAM 7:

## 2-D Path Tracing with Inverse Kinematics? Code:

%Create a rigidBodyTree object and rigid bodies with their associated joints. Specify the

geometric properties of each rigid body and add it to the robot.

%Start with a blank rigid body tree model.

robot = rigidBodyTree('DataFormat','column','MaxNumBodies',3); %Specify arm lengths for the robot arm.

L1 = 0.3;

L2 = 0.3;

%Add 'link1' body with 'joint1' joint.

body = rigidBody('link1'); joint = rigidBodyJoint('joint1', 'revolute'); setFixedTransform(joint,trvec2tform([0 0 0]));

joint.JointAxis = [0 0 1]; body.Joint

= joint; addBody(robot, body, 'base'); %Add 'link2' body with 'joint2' joint.

body = rigidBody('link2');

joint = rigidBodyJoint('joint2','revolute'); setFixedTransform(joint, trvec2tform([L1,0,0]));

joint.JointAxis = [0 0 1]; body.Joint = joint; addBody(robot, body, 'link1');

%Add 'tool' end effector with 'fix1' fixed joint.

body = rigidBody('tool'); joint = rigidBodyJoint('fix1','fixed'); setFixedTransform(joint, trvec2tform([L2, 0, 0])); body.Joint = joint;

addBody(robot, body, 'link2');

%Show details of the robot to validate the input properties. The robot should have two nonfixed joints, where the link bodies are connected to their parent bodies via a revolute joint, and the end effector is connected to its parent body via a fixed joint.

showdetails(robot)

%Define The Trajectory

%Define a circle to be traced over the course of 10 seconds. This circle is in the xy plane with a radius of 0.15.

t = (0:0.2:10)'; % Time

count = length(t); center = [0.3 0.1 0]; radius = 0.15; theta = t\*(2\*pi/t(end));

points = center + radius\*[cos(theta) sin(theta) zeros(size(theta))];

%Inverse Kinematics Solution

%Use an inverseKinematics object to find a solution of robotic configurations that achieve the given end-effector positions along the trajectory.

%Pre-allocate configuration solutions as a matrix qs. q0 = homeConfiguration(robot);

ndof = length(q0); qs =

zeros(count, ndof);

%Create the inverse kinematics solver. Because the xy Cartesian points are the only important factors of the end-effector pose for this workflow, specify a non-zero weight for the fourth and fifth elements of the weight vector. All other elements are set to zero.

ik = inverseKinematics('RigidBodyTree', robot); weights = [0, 0, 0, 1, 1, 0]; endEffector

= 'tool';

%Loop through the trajectory of points to trace the circle. Call the ik object for each point to generate the joint configuration that achieves the end-effector position. Store the configurations to use later.

qInitial = q0; % Use home configuration as the initial guess for i = 1:count

% Solve for the configuration satisfying the desired end effector

% position point = points(i,:);

qSol = ik(endEffector,trvec2tform(point),weights,qInitial);

% Store the configuration qs(i,:) = qSol;

% Start from prior solution qInitial = qSol;

end

%Animate The Solution

%Plot the robot for each frame of the solution using that specific robot configuration. Also, plot the desired trajectory.

%Show the robot in the first configuration of the trajectory. Adjust the plot to show the 2-D plane that circle is drawn on. Plot the desired trajectory.

figure show(robot,qs(1,:)'); view(2) ax

= gca;

ax.Projection = 'orthographic'; hold on

plot(points(:,1),points(:,2),'k')

axis([-0.1 0.7 -0.3 0.5])

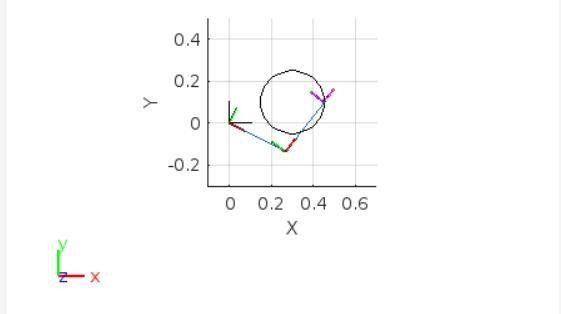
%Set up a rateControl object to display the robot trajectory at a fixed rate of 15 frames per second. Show the robot in each configuration from the inverse kinematic solver. Watch as the arm traces the circular trajectory shown.

framesPerSecond = 15; r = rateControl(framesPerSecond); for i = 1:count

show(robot,qs(i,:)','PreservePlot',false); drawnow

waitfor(r); end

**Output:**



# PROGRAM 8:

## Explain Top 10 Python Libraries For Robotics

1| Robot Framework.

2| Pyro

3| DART

4| PyRobot 5| PyDy

6| Simulation Open Framework Architecture. 7| Klamp't

8| Pybotics 9| Siconos

10 | iDynTree **Answer:**

One of the most popular languages, [**Python**,](https://analyticsindiamag.com/53-3-data-scientists-prefer-python-according-to-plato-survey-report-by-aim/) is extensively used by emerging tech developers as well as robotics researchers. In robotics, the language has become a key part of the robot operating system (ROS) and is used for designing the embedded systems. For instance, the embedded systems and exhaustive automation packages of Raspberry Pi and Arduino are [**designed**](https://analyticsindiamag.com/5-most-sought-after-programming-languages-for-robotics-you-should-learn-in-2019/) using this language.

1| Robot Framework

About: Robot Framework is a generic open-source automation framework for acceptance testing, acceptance test-driven development (ATDD), and robotic process automation (RPA). The core framework is implemented using [**Python**](https://analyticsindiamag.com/top-9-python-frameworks-for-game-development/) – it supports both Python 2 and Python 3 – and runs also on Jython (JVM), IronPython (.NET) and PyPy. Robot Framework is open and extensible and can be integrated with virtually any other tool to create powerful and flexible automation solutions.

2| Pyro

About: [**Python**](https://analyticsindiamag.com/how-to-create-url-shortening-library-in-python/) Remote Objects – or Pyro – is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort. Written in Python, this toolbox works between different system architectures and operating systems. It provides a set of powerful features that enables you to build distributed applications rapidly and effortlessly.

3| DART

About: Dynamic Animation and Robotics Toolkit – or DART – is a collaborative, crossplatform, open-source library that provides data structures and algorithms for kinematic and dynamic applications in robotics and computer animation. The library is distinguished by its accuracy and stability due to its use of generalised coordinates to represent articulated rigid body systems and Featherstone’s Articulated Body Algorithm to compute the dynamics of motion.

DART also provides efficient computation of Jacobian matrices for arbitrary body points and coordinate frames. The library was created by the Graphics Lab and Humanoid Robotics Lab at Georgia Institute of Technology with ongoing contributions from the Personal Robotics Lab at the University of Washington and Open Source Robotics Foundation.

4| PyRobot

About: PyRobot is a [**Python library**](https://analyticsindiamag.com/why-you-should-learn-python-today/) for benchmarking and running experiments in robot learning. It is a combination of two popular Python libraries, i.e. Requests and

BeautifulSoup. It can be used to drive applications that don’t provide an API or any way of hooking into them programmatically. This library will allow you to run robots without having to deal with the robot specific software along with enabling better comparisons.

5| PyDy

About: Python Dynamics or PyDy is a tool kit written in the [**Python programming language**](https://analyticsindiamag.com/hands-on-guide-to-market-basket-analysis-with-python-codes/)that utilises an array of scientific programs to enable the study of multibody dynamics. The toolkit helps a user to perform visualisation, model specification, simulation, benchmarking, among others in their workflows.

6| Simulation Open Framework Architecture

About: Simulation Open Framework Architecture or SOFA is an open-source library and an efficient framework dedicated to research, prototyping and development of physics-based simulations. The library primarily focuses on real-time simulation, with an emphasis on medical simulation.

The advanced software architecture of this framework allows the creation of complex and evolving simulations by combining new algorithms with existing algorithms, synthesis of complex models from simpler ones using a scene-graph description, among others.

7| Klamp’t

About: Kris’ Locomotion and Manipulation Planning Toolbox or Klamp’t is an open-source, cross-platform software package for robot modelling, simulating, planning, optimisation, and visualisation. The library aims to provide an accessible, wide range of programming tools for learning robotics, analysing robots, developing algorithms, and prototyping intelligent behaviours.

Some of the features of this tool are-

* Simulation of various sensors including RGB+D cameras, laser sensors, gyroscopes, force/torque sensors, and accelerometers
* Many sampling-based motion planners implemented
* Supports legged and fixed-based robots
* Contact mechanics computations: force closure, support polygons, the stability of rigid bodies and actuated robots

8| Pybotics

About: Pybotics is an open-source Python toolbox for robot kinematics and calibration. The toolbox was mainly designed to provide a simple, clear, and concise interface to quickly simulate and evaluate common robot concepts, such as kinematics, dynamics, trajectory generations, and calibration.

9| Siconos

About: Currently distributed under Apache Licenses, Siconos is an open-source scientific software primarily targeted at modelling and simulating nonsmooth dynamical systems. Written in C++ and Python, this software package can be used for modelling and simulation of dynamic systems.

10| iDynTree

About: iDynTree is a library of robot dynamics algorithms for control, estimation and simulation. The library is written in C++ language and supports several other languages including Python, MATLAB, among others. To use the library in Python language, you need to add the *PYTHONPATH* environment variable to the install path of the *iDynTree.py* file.

# PROGRAM 9:

## Walking Robot -- DDPG Agent Training Script (2D) Code: Bipadel

% Walking Robot Parameters -- Reinforcement Learning

%% Model parameters % Mechanical density = 500; foot\_density = 1000; if

~exist('actuatorType','var') actuatorType = 1; end world\_damping = 1e-3; world\_rot\_damping = 5e-2;

% Contact/friction parameters contact\_stiffness = 500; contact\_damping = 50; mu\_k

= 0.7; mu\_s = 0.9; mu\_vth = 0.01; height\_plane = 0.025;

plane\_x = 25; plane\_y = 10; contact\_point\_radius = 1e-4;

% Foot dimensions foot\_x = 5; foot\_y

= 4; foot\_z = 1;

foot\_offset = [-1 0 0];

% Leg dimensions leg\_radius

= 0.75; lower\_leg\_length = 10; upper\_leg\_length = 10;

% Torso dimensions torso\_y = 8; torso\_x

= 5; torso\_z = 8;

torso\_offset\_z = -2;

torso\_offset\_x = -1; mass = (0.01^3)\*torso\_y\*to rso\_x\*torso\_z\*dens ity; g = 9.80665;

% Joint parameters joint\_damping = 1; joint\_stiffness

= 0; motion\_time\_constant = 0.01; joint\_limit\_stiffness = 1e4; joint\_limit\_damping = 10;

%% Reinforcement Learning (RL) parameters Ts = 0.025; % Agent sample time

Tf = 10; % Simulation end time

% Scaling factor for RL action [-1 1] max\_torque = 3;

% Initial conditions h = 18; % Hip height [cm] init\_height = foot\_z + h + ... torso\_z/2 + torso\_offset\_z + height\_plane/2; vx0 = 0; % Initial X linear velocity [m/s] vy0 = 0; % Initial Y linear velocity [m/s] wx0 = 0; % Initial X angular velocity [rad/s] wy0 = 0; % Initial Y angular velocity [rad/s]

% Initial foot positions [m] leftinit = [0;0;-h/100]; rightinit = [0;0;-h/100];

% Calculate initial joint angles init\_angs\_L = zeros(1,2); theta = legInvKin(upper\_leg\_length/100,lower\_leg\_length/100,-leftinit(1),leftinit(3));

% Address multiple outputs if size(theta,1) == 2 if theta(1,2) < 0

init\_angs\_L(1) = theta(2,1); init\_angs\_L(2) = theta(2,2); else

init\_angs\_L(1) = theta(1,1); init\_angs\_L(2) = theta(1,2); end end

init\_angs\_R = zeros(1,2);

theta = legInvKin(upper\_leg\_length/100,lower\_leg\_length/100,-rightinit(1),rightinit(3)); % Address multiple outputs if size(theta,1) == 2 if theta(1,2) < 0 init\_angs\_R(1) = theta(2,1); init\_angs\_R(2) = theta(2,2); else

init\_angs\_R(1) = theta(1,1); init\_angs\_R(2) = theta(1,2); end end

robotParametersRL

mdl = "rlWalkingBipedRobot"; open\_system(mdl)

numObs = 29; obsInfo = rlNumericSpec([numObs 1]); obsInfo.Name = "observations";

numAct = 6;

actInfo = rlNumericSpec([numAct 1],LowerLimit=-1,UpperLimit=1); actInfo.Name

= "foot\_torque";

blk = mdl + "/RL Agent"; env = rlSimulinkEnv(mdl,blk,obsInfo,actInfo); env.ResetFcn = @(in) walkerResetFcn(in, ... upper\_leg\_length/100, ... lower\_leg\_length/100, ...

h/100);

AgentSelection = "DDPG"; switch AgentSelection

case "DDPG"

agent = createDDPGAgent(numObs,obsInfo,numAct,actInfo,Ts); case "TD3"

agent = createTD3Agent(numObs,obsInfo,numAct,actInfo,Ts); otherwise

disp("Assign AgentSelection to DDPG or TD3") end

maxEpisodes = 2000; maxSteps

= floor(Tf/Ts); trainOpts = rlTrainingOptions(...

MaxEpisodes=maxEpisodes,... MaxStepsPerEpisode=maxSteps,... ScoreAveragingWindowLength=250,... Verbose=false,...

Plots="training-progress",... StopTrainingCriteria="EpisodeCount",... StopTrainingValue=maxEpisodes,...

SaveAgentCriteria="EpisodeCount",... SaveAgentValue=maxEpisodes);

trainOpts.UseParallel = true; trainOpts.ParallelizationOptions.Mode = "async"; trainOpts.ParallelizationOptions.StepsUntilDataIsSent = 32;

trainOpts.ParallelizationOptions.DataToSendFromWorkers = "Experiences";

doTraining = false; if doTraining % Train the agent. trainingStats = train(agent,env,trainOpts); else

% Load a pretrained agent for the selected agent type. if strcmp(AgentSelection,"DDPG") load("rlWalkingBipedRobotDDPG.mat","agent") else

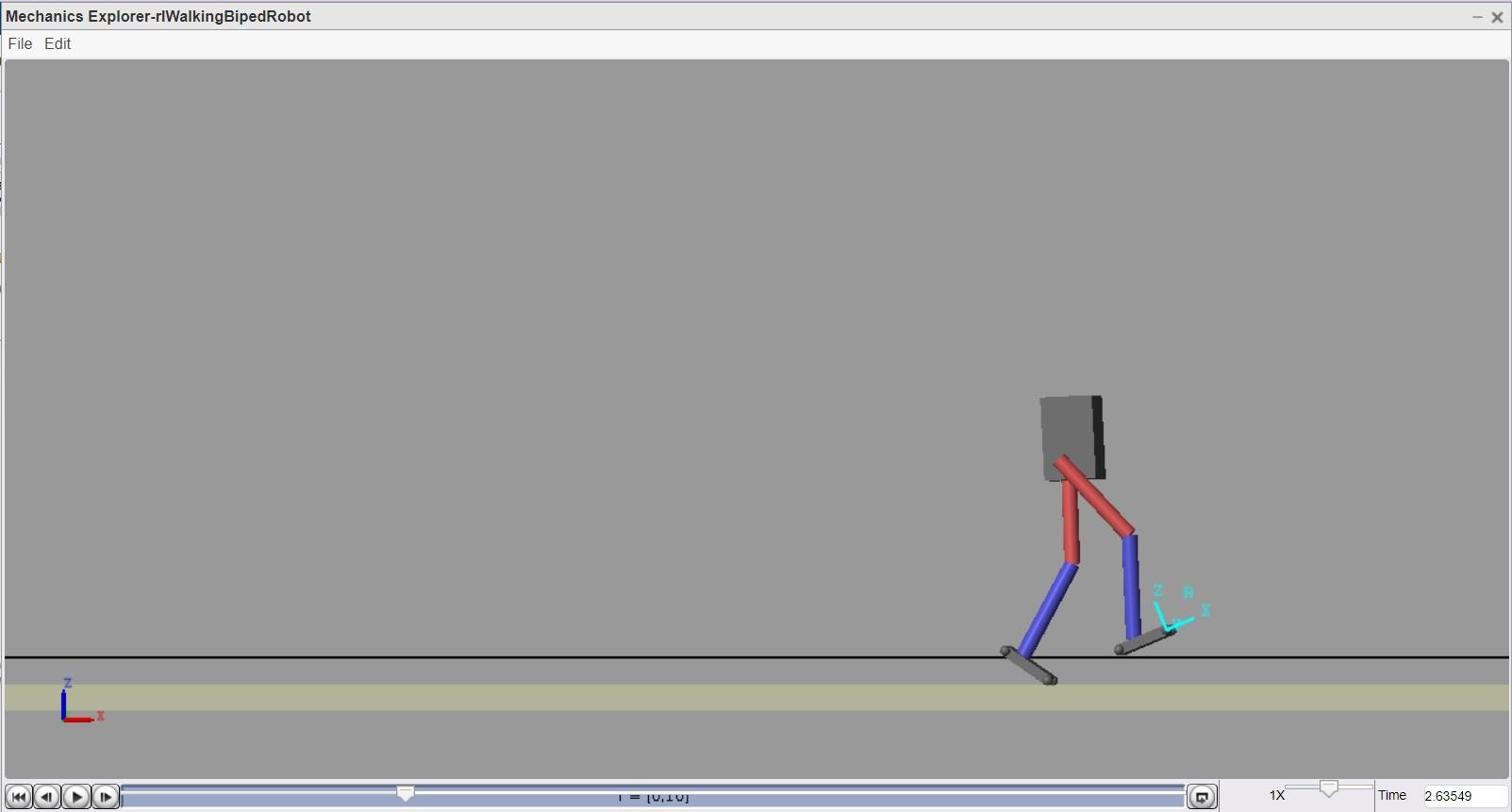
load("rlWalkingBipedRobotTD3.mat","agent") end end

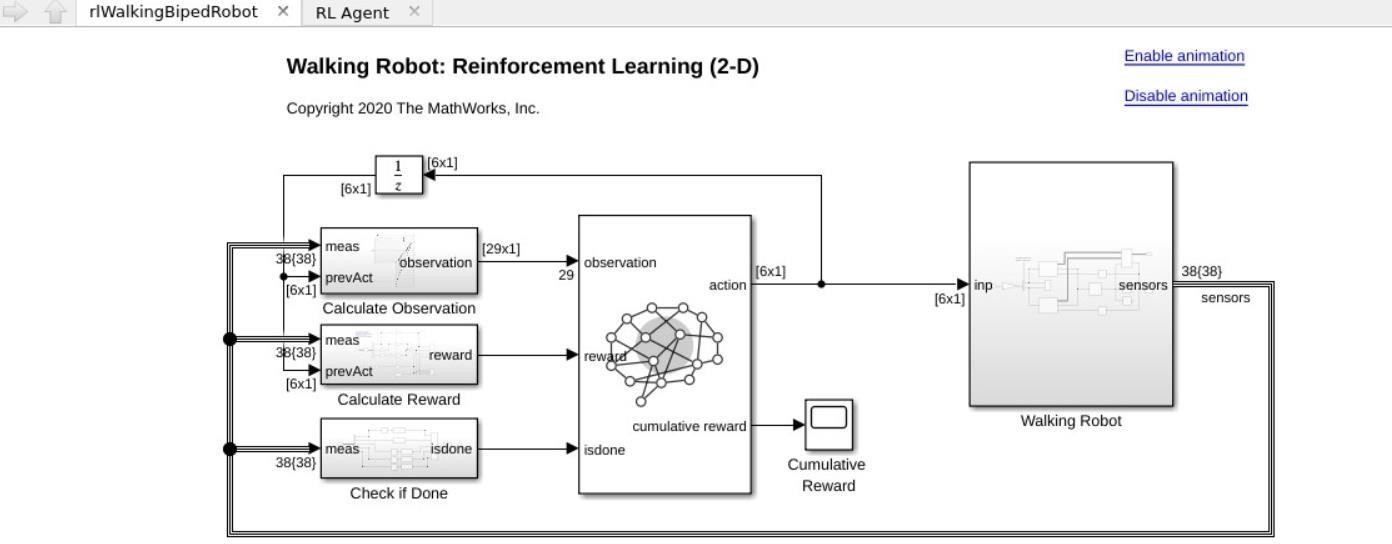
rng(0)

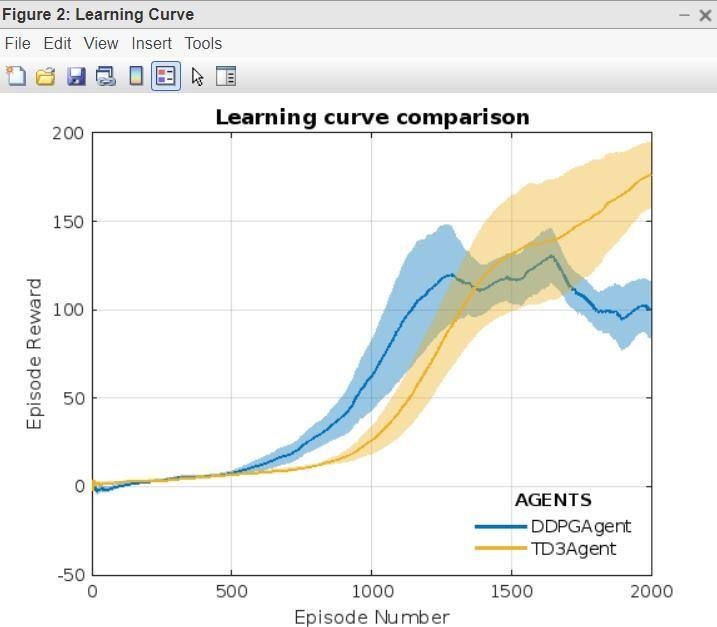
simOptions = rlSimulationOptions(MaxSteps=maxSteps); experience

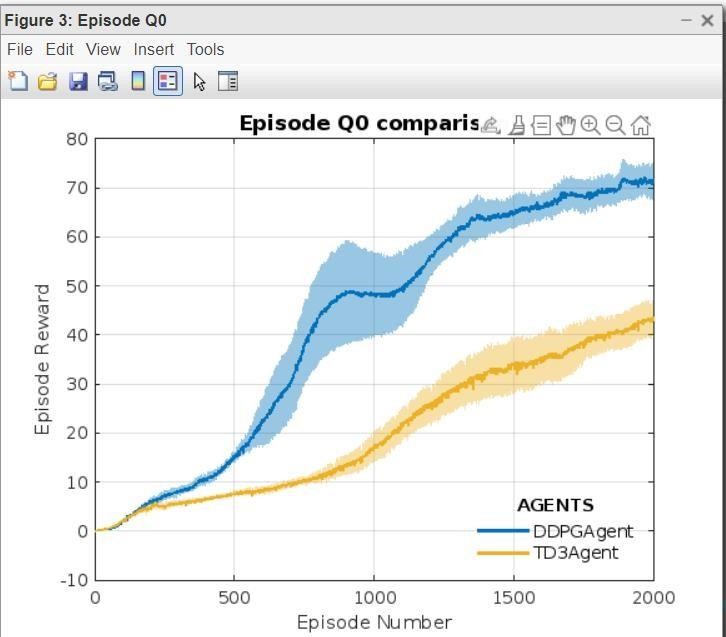
= sim(env,agent,simOptions); comparePerformance("DDPGAgent","TD3Agent")

**Output:**









# PROGRAM 10:

## Describe robotics programming for ROS Answer:

ROS is powering the future of robotics in industry, in the enterprise and for developers. Ubuntu has been the primary platform for ROS from the very beginning, thanks to its flexibility and userfriendliness.

ROS is led by Open Robotics, similar to how Canonical supports Ubuntu; Open Robotics steers the ship but it is driven by the community.

As the robotics industry grows, companies and developers continue turning to Open Robotics and Canonical to help make their vision a reality. The Robot Operating System is an open-source framework that helps researchers and developers build and reuse code between robotics applications. ROS is also a global open source community of engineers, developers and hobbyists who contribute to making robots better, more accessible and available to everyone.

ROS 2 is a complete re-design of the framework, tackling the shortcomings of the first generation, effectively bringing it to industry needs and standards. ROS 2 contains new sets of packages that can be installed alongside ROS 1 to ease migration to a more secure platform.

ROS 2 takes advantage of new technologies and newly updated APIs asked for by the community. ROS has been adopted into some of the biggest names in robotics. The majority of organisations are either using ROS as it can be installed by anyone or a fork of ROS in some form. And the use-cases are still growing. ROS is used across numerous industries from agriculture to medical devices to vacuum cleaners but is spreading to include all kinds of automation and software-defined dynamic use-cases.

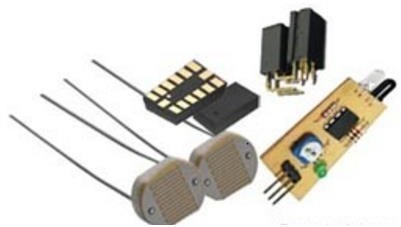
* ROS was built with cross-collaboration in mind. The base code and knowledge can be applied across all robotics platforms (arms, drones, mobile bases, etc.) You’ll be able to reuse what you already know and stop reinventing the wheel.
* It can be built into your product. Most of the core ROS 1 packages are under a BSD license and ROS 2 packages are under Apache. This license allows for the modification of code for commercial purposes without having to release your code with an open-source license.
* ROS robots can speak any language. You can communicate easily between Python and C++ nodes, get libraries to allow you to use most other languages or install rosbridge and use any language that can speak JSON.
* ROS is here to stay. ROS and the community around it have been growing since 2007 thanks to contributions from an incredibly smart and open community. With the market share ROS has acquired and the ongoing development of ROS2, robots will be the future.
* There is a package for everything. Whether you want to compute trajectory, conduct SLAM algorithms or implement remote control, there’s a ROS package for that.
* Digital twinning. ROS allows developers to easily simulate their robot in any environment, before deploying anything in the real world. Tools like Gazebo even allow you to create simulations with robots you don’t possess.
* It’s open-source. ROS has contributors all over the world using ROS for countless different purposes. Every contribution feeds into continuously developing, leading stack that ROS is.

# PROGRAM 11:

## Explain the different types of robot’s sensors with example? Answer:

### Types of Robot Sensors

There are different type of sensors are available to choose from and the characteristics of sensors are used for determining the type of sensor to be used for particular application.



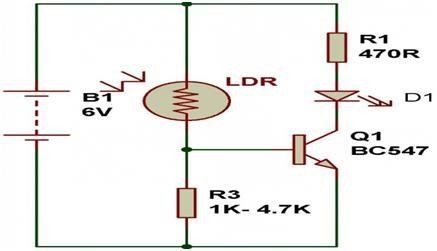
1. Light Sensor

Light sensor is a transducer used for detecting light and creates a voltage difference equivalent to the light intensity fall on a light sensor.

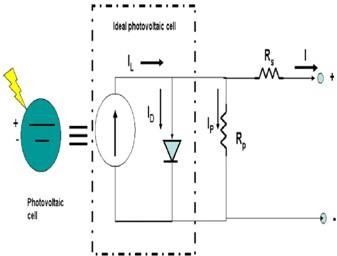
The two main light sensors used in robots are Photovoltaic cells and Photo resistor. Other kind of light sensors like phototransistors, phototubes are rarely used.

The type of light sensors used in robotics are:

Photo resistor - It is a type of resistor used for detecting the light. In photo resistor resistance varies with change in light intensity. The light falls on photo resistor is inversely proportional to the resistance of the photo resistor. In general photo resistor is also called as Light Dependent Resistor (LDR).

Consider the circuit diagram of Photo resistor sensor:

Photovoltaic Cells - Photovoltaic cells are energy conversion device used to convert solar radiation into electrical electric energy. It is used if we are planning to build a solar robot. Individually photovoltaic cells are considered as an energy source, an implementation combined with capacitors and transistors can convert this into a sensor.

Consider the circuit diagram of photovoltaic cell is,

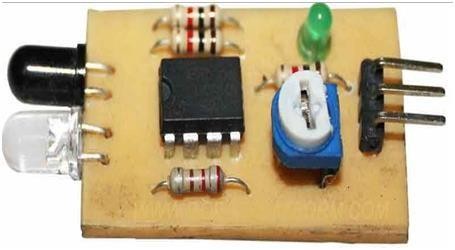
1. Proximity Sensor

Proximity sensor can detect the presence of nearby object without any physical contact. The working of a proximity sensor is simple. In proximity sensor transmitter transmits an electromagnetic radiation and receiver receives and analyzes the return signal for interruptions. Therefore the amount of light receiver receives by surrounding can be used for detecting the presence of nearby object.

Consider the types of proximity sensors used in robotics are:-

Infrared (IR) Transceivers - In IR sensor LED transmit the beam of IR light and if it find an obstacle then the light is reflected back which is captured by an IR receiver.

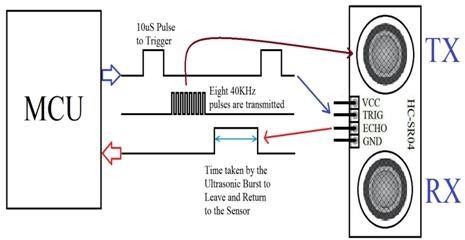
Consider the PCB board layout of IR Transceiver circuit:



Ultrasonic Sensor - In ultrasonic sensors high frequency sound waves is generated by transmitter, the received echo pulse suggests an object interruption.

In general ultrasonic sensors are used for distance measurement in robotic system.

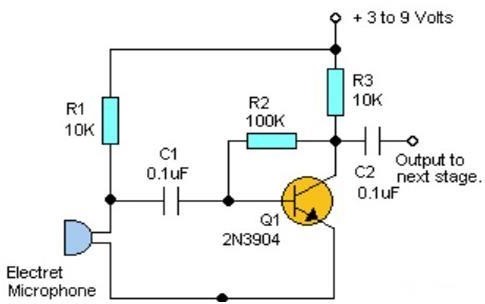
Consider the interfacing of ultrasonic sensor with Microcontroller unit:



1. Sound Sensor

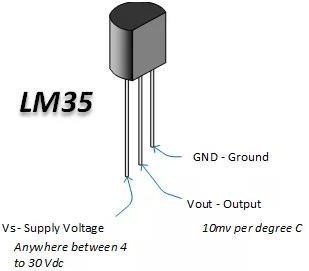
Sound sensors are generally a microphone used to detect sound and return a voltage equivalent to the sound level. Using sound sensor a simple robot can be designed to navigate based on the sound receives.

Implementation of sound sensors is not easy as light sensors because it generates a very small voltage difference which will be amplified to generate measurable voltage change.

Consider the sound sensor based switching circuit:

1. Temperature Sensor

Temperature sensors are used for sensing the change in temperature of the surrounding. It is based on the principle of change in voltage difference for a change in temperature this change in voltage will provide the equivalent temperature value of the surrounding.

Few generally used temperature sensors IC?s are TMP35, TMP37, LM34, LM35, etc. Consider the temperature sensor pin diagram description is,

1. Acceleration Sensor

Acceleration sensor is used for measuring acceleration and tilt. An accelerometer is a device used for measuring acceleration.

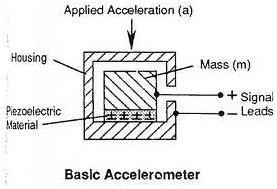
The two kinds of forces which affect an accelerometer is:-

o Static Force - It is the frictional force between any two objects. By measuring this gravitational force we can determine the how much robot is tilting. This measurement is useful in balancing robot, or for determining whether robot is driving on a flat surface or uphill. o Dynamic Force - It is the amount of acceleration required to move an object. Measurement of dynamic force using an accelerometer tells about the velocity/speed at which robot is moving.

Accelerometer is comes in different configuration. Always use the one which is most appropriate for your robot. Some factors need to be considered before selecting accelerometer is:

* 1. Sensitivity
  2. Bandwidth
  3. Output type: Analog or Digital
  4. Number of Axis: 1,2 or 3

Consider the schematic diagram of basic accelerometer:



# PROGRAM 12:

## Write the script of Talker and Listener in python and show the output. Code:

**Publisher is nothing but Talker Subscriber is nothing but Listener**

### Writing the Publisher/Talker Node

"Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.

Change directory into the beginner\_tutorials package, you created in the earlier tutorial, [creating a package:](http://wiki.ros.org/ROS/Tutorials/CreatingPackage)

$ roscd beginner\_tutorials

### The Code

First lets create a 'scripts' folder to store our Python scripts in:

$ mkdir scripts

$ cd scripts

Then download the example script [talker.py](https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/talker.py) to your new scripts directory and make it executable:

$ wget https://raw.github.com/ros/ros\_tutorials/kinetic-devel/rospy\_tutorials/001\_talker\_liste ner/talker.py $ chmod +x talker.py

We will not run it yet. You can view and edit the file with $ rosed beginner\_tutorials talker.py or just look below.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

1. #!/usr/bin/env python
2. # license removed for brevity 3 import rospy

4 from std\_msgs.msg import String

[5](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c82832e0d612370fe9886563f0b7f5433f6caee1_5)

6

7

8

def talker():

pub = rospy.Publisher('chatter', String, queue\_size=10)

rospy.init\_node('talker', anonymous=True) [9](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c82832e0d612370fe9886563f0b7f5433f6caee1_9)

rate = rospy.Rate(10) # 10hz [10](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c82832e0d612370fe9886563f0b7f5433f6caee1_10)

while not rospy.is\_shutdown():

[11](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c82832e0d612370fe9886563f0b7f5433f6caee1_11) hello\_str = "hello world %s" % rospy.get\_time()

1. rospy.loginfo(hello\_str)
2. pub.publish(hello\_str)
3. rate.sleep() [15](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c82832e0d612370fe9886563f0b7f5433f6caee1_15)
4. if name == ' main ':
5. try:
6. talker()
7. except rospy.ROSInterruptException:
8. pass

Add the following to your CMakeLists.txt. This makes sure the python script gets installed

properly, and uses the right python interpreter.

catkin\_install\_python(PROGRAMS scripts/talker.py DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

### The Code Explained

Now, let's break the code down.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

[1](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-da6097954e7fee459e60cb816897ac01d266cf48_1) #!/usr/bin/env python

Every Python ROS [Node](http://wiki.ros.org/Nodes) will have this declaration at the top. The first line makes sure your script is executed as a Python script.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29) 3 import rospy

4 from std\_msgs.msg import String

You need to import rospy if you are writing a ROS [Node.](http://wiki.ros.org/Nodes) The std\_msgs.msg import is so that

we can reuse the std\_msgs/String message type (a simple string container) for publishing. [Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

1. pub = rospy.Publisher('chatter', String, queue\_size=10)
2. rospy.init\_node('talker', anonymous=True)

This section of code defines the talker's interface to the rest of

ROS. pub = rospy.Publisher("chatter", String, queue\_size=10) declares that your node is publishing to the chatter topic using the message type String. String here is actually the class std\_msgs.msg.String. The queue\_size argument is New in ROS hydro and limits the amount of queued messages if any subscriber is not receiving them fast enough. In older ROS distributions just omit the argument.

The next line, rospy.init\_node(NAME, ...), is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS [Master.](http://wiki.ros.org/Master) In this case, your node will take on the name talker. NOTE: the name must be a [base name,](http://wiki.ros.org/Names)

i.e. it cannot contain any slashes "/".

anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME. Refer to [Initialization and](http://wiki.ros.org/rospy/Overview/Initialization%20and%20Shutdown#Initializing_your_ROS_Node) [Shutdown - Initializing your ROS Node](http://wiki.ros.org/rospy/Overview/Initialization%20and%20Shutdown#Initializing_your_ROS_Node) in the rospy documentation for more information about node initialization options.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

1. rate = rospy.Rate(10) # 10hz This line creates a Rate object rate. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

10

11

12

13

14

while not rospy.is\_shutdown():

hello\_str = "hello world %s" % rospy.get\_time() rospy.loginfo(hello\_str)

pub.publish(hello\_str)

rate.sleep()

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

This loop is a fairly standard rospy construct: checking the rospy.is\_shutdown() flag and then

doing work. You have to check is\_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(hello\_str) that publishes a string to our chatter topic. The loop calls rate.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

(You may also run across rospy.sleep() which is similar to time.sleep() except that it works with simulated time as well (see [Clock)](http://wiki.ros.org/Clock).)

This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to [rosout.](http://wiki.ros.org/rosout) [rosout](http://wiki.ros.org/rosout) is a handy tool for debugging: you can pull up messages using [rqt\_console](http://wiki.ros.org/rqt_console) instead of having to find the console window with your Node's output.

std\_msgs.msg.String is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that *constructor args are in the same order as in the .msg file*. You can also pass in no arguments and initialize the fields directly, e.g.

msg = String() msg.data

= str

or you can initialize some of the fields and leave the rest with default values:

String(data=str)

You may be wondering about the last little bit:

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

1. try:
2. talker()
3. except rospy.ROSInterruptException:
4. pass

In addition to the standard Python main check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the sleep().

Now we need to write a node to receive the messages.

### Writing the Subscriber/Listener Node The Code

Download the [listener.py](https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/listener.py) file into your scripts directory:

$ roscd beginner\_tutorials/scripts/

$ wget https://raw.github.com/ros/ros\_tutorials/kinetic-devel/rospy\_tutorials/001\_talker\_liste ner/listener.py

$ chmod +x listener.py

The file contents look close to:

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

1. #!/usr/bin/env python
2. import rospy
3. from std\_msgs.msg import String

[4](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_4)

1. def callback(data):
2. rospy.loginfo(rospy.get\_caller\_id() + "I heard %s", data.data)

[7](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_7)

[8](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_8) def listener():

[9](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_9)

1. # In ROS, nodes are uniquely named. If two nodes with the same
2. # name are launched, the previous one is kicked off. The
3. # anonymous=True flag means that rospy will choose a unique [13](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_13) # name for our 'listener' node so that multiple listeners can [14](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_14) # run simultaneously.

[15](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_15) rospy.init\_node('listener', anonymous=True) [16](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_16)

[17](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_17) rospy.Subscriber("chatter", String, callback) [18](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_18)

[19](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_19) # spin() simply keeps python from exiting until this node is stopped

[20](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_20) rospy.spin() [21](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-56fb82d4681d7880a3d3a97f7af70cfe17618f86_21)

1. if name == ' main ':
2. listener()

Then, edit the catkin\_install\_python() call in your CMakeLists.txt so it looks like the following:

catkin\_install\_python(PROGRAMS scripts/talker.py scripts/listener.py DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

### The Code Explained

The code for listener.py is similar to talker.py, except we've introduced a new callback-based mechanism for subscribing to messages.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29)

[15](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_15) rospy.init\_node('listener', anonymous=True) [16](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_16)

[17](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_17) rospy.Subscriber("chatter", String, callback)

[18](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_18)

[19](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_19) # spin() simply keeps python from exiting until this node is stopped

[20](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29#rospy_tutorials.2FTutorials.2FWritingPublisherSubscriber.CA-c54953afb3f7ac20abfe7460208d0cd6c36c1d62_20) rospy.spin()

This declares that your node subscribes to the chatter topic which is of type std\_msgs.msgs.String. When new messages are received, callback is invoked with the message as the first argument.

We also changed up the call to rospy.init\_node() somewhat. We've added the anonymous=True keyword argument. ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. The anonymous=True flag tells rospy to generate a unique name for the node so that you can have multiple listener.py nodes run easily.

The final addition, rospy.spin() simply keeps your node from exiting until the node has been shutdown. Unlike roscpp, rospy.spin() does not affect the subscriber callback functions, as those have their own threads.

# PROGRAM 13:

## How to create your first package in ROS, specify each command in chronological way.

**Answer:**

Using roscreate

Before we create a package, let's see how the roscreate-pkg command-line tool works. This creates a new ROS [**package**.](http://wiki.ros.org/Packages) All ROS packages consist of the many similar files

: [**manifests**](http://wiki.ros.org/Manifest)[,](http://wiki.ros.org/CMakeLists) [**CMakeLists.txt**,](http://wiki.ros.org/CMakeLists) mainpage.dox, and Makefiles. roscreate-pkg eliminates many tedious tasks of creating a new package by hand, and eliminates common errors caused by hand-typing build files and manifests.

To create a new package in the current directory:

# roscreate-pkg [package\_name]

You can also specify dependencies of that package:

# roscreate-pkg [package\_name] [depend1] [depend2] [depend3]

Creating a New ROS Package

Now we're going to go into your home or project directory and create our beginner\_tutorials package. We are going to make it depend on [**std\_msgs**](http://wiki.ros.org/std_msgs)[,](http://wiki.ros.org/roscpp) [**roscpp**,](http://wiki.ros.org/roscpp) and [**rospy**,](http://wiki.ros.org/rospy) which are common ROS packages. Now go into the

~/fuerte\_workspace/sandbox directory:

$ cd ~/fuerte\_workspace/sandbox

Alternatively, if you use Fuerte or later release, you can simply do:

$ roscd

$ cd sandbox

Then create your package:

$ roscreate-pkg beginner\_tutorials std\_msgs rospy roscpp

You will see something similar to:

|  |  |
| --- | --- |
|  | Creating package directory ~/fuerte\_workspace/sandbox/beginner\_tutorials |
|  | Creating include directory ~/fuerte\_workspace/sandbox/beginner\_tutorials/include/be ginner\_tutorials |
|  | Creating cpp source directory ~/ros/ros\_tutorials/beginner\_tutorials/src |
|  | Creating python source directory ~/fuerte\_workspace/sandbox/beginner\_tutorials/src/ beginner\_tutorials |
|  | Creating package file ~/fuerte\_workspace/sandbox/beginner\_tutorials/Makefile |
|  | Creating package file ~/fuerte\_workspace/sandbox/beginner\_tutorials/manifest.xml |
|  | Creating package file ~/fuerte\_workspace/sandbox/beginner\_tutorials/CMakeLists.txt |
|  | Creating package file ~/fuerte\_workspace/sandbox/beginner\_tutorials/mainpage.dox |
|  | Please edit beginner\_tutorials/manifest.xml and mainpage.dox to finish creating your package |

You're going to want to spend some time looking at beginner\_tutorials/manifest.xml. [**manifests**](http://wiki.ros.org/Manifest) play an important role in ROS as they define how Packages are built, run, and documented.

Now lets make sure that ROS can find your new package. It is often useful to call *rospack profile* after making changes to your path so that new directories will be found:



$ rospack profile

$ rospack find beginner\_tutorials YOUR\_PACKAGE\_PATH/beginner\_tutorials

If this fails, it means ROS can't find your new package, which may be an issue with your

ROS\_PACKAGE\_PATH. Please consult the installation instructions for setup from SVN or from binaries, depending how you installed ROS. If you've created or added a package that's outside of the existing package paths, you will need to amend your

[**ROS\_PACKAGE\_PATH**](http://wiki.ros.org/ROS/EnvironmentVariables#ROS_PACKAGE_PATH) environment variable to include that new location. Try resourcing your setup.sh in your fuerte\_workspace.

Try moving to the directory for the package.

$ roscd beginner\_tutorials



$ pwd

YOUR\_PACKAGE\_PATH/beginner\_tutorials

First-order package dependencies

When using roscreate-pkg earlier, a few package dependencies were provided. These firstorder dependencies can now be reviewed with the rospack tool.

(Jan 9, 2013) There is [**a bug**](https://github.com/ros/rospack/issues/4)reported and already fixed in [**rospack**](http://wiki.ros.org/rospack) in groovy; it may take some time to be reflected in the packages. If you see [**an issue similar to**](http://answers.ros.org/question/51555/beginner-tutorials-segmentation-fault-with-rospack-depends1/?comment=51762&comment-51762)[**this**](http://answers.ros.org/question/51555/beginner-tutorials-segmentation-fault-with-rospack-depends1/?comment=51762&comment-51762) with the next command, you can skip to the following command.

$ rospack depends1 beginner\_tutorials

|  |  |
| --- | --- |
|  | std\_msgs |
|  | rospy |
|  | roscpp |

As you can see, rospack lists the same dependencies that were used as arguments when running roscreate-pkg. These dependencies for a package are stored in the manifest file. Take a look at the manifest file.

|  |  |
| --- | --- |
|  | <package> |
|  | ... |
|  | <depend package="std\_msgs"/> |
|  | <depend package="rospy"/> |
|  | <depend package="roscpp"/> |
|  | </package> |

Indirect package dependencies



$ roscd beginner\_tutorials

$ cat manifest.xml

In many cases, a dependency will also have its own dependencies. For instance, rospy has other dependencies.

(Jan 9, 2013) There is [**a bug**](https://github.com/ros/rospack/issues/4)reported and already fixed in [**rospack**](http://wiki.ros.org/rospack) in groovy; it may take some time to be reflected in the packages. If you see [**an issue similar to**](http://answers.ros.org/question/51555/beginner-tutorials-segmentation-fault-with-rospack-depends1/?comment=51762&comment-51762)[**this**](http://answers.ros.org/question/51555/beginner-tutorials-segmentation-fault-with-rospack-depends1/?comment=51762&comment-51762) with the next command, you can skip to the following command.



$ rospack depends1 rospy roslib



roslang

A package can have quite a few indirect dependencies. Luckily rospack can recursively

determine all nested dependencies.

$ rospack depends beginner\_tutorials



rospack roslib std\_msgs

rosgraph\_msgs

rosbuild roslang rospy

cpp\_common roscpp\_traits rostime roscpp\_serialization xmlrpcpp rosconsole

roscpp

Note: in Fuerte, the list is much shorter:



std\_msgs roslang rospy

roscpp

ROS Client Libraries

You may be wondering what rospy and roscpp dependencies are from the previous examples. rospy and roscpp are [**Client**](http://wiki.ros.org/Client%20Libraries)[**Libraries**.](http://wiki.ros.org/Client%20Libraries) The client libraries allow different programming languages to communicate through ROS. rospy is the client library for Python. roscpp is the client library for C++.

# PROGRAM 14:

## Explain the navigation concept in robotics? Answer:

For any mobile device, the ability to navigate in its environment is important. Avoiding

dangerous situations such as collisions and unsafe conditions (temperature, radiation, exposure to weather, etc.) comes first, but if the robot has a purpose that relates to specific places in the robot environment, it must find those places. This article will present an overview of the skill of navigation and try to identify the basic blocks of a robot navigation system, types of navigation systems, and closer look at its related building components.

Robot navigation means the robot’s ability to determine its own position in its frame of reference and then to plan a path towards some goal location. In order to navigate in its environment, the robot or any other mobility device requires representation, i.e. a map of the environment and the ability to interpret that representation.

Navigation can be defined as the combination of the three fundamental competences: Self-localisation

Path planning

Map-building and map interpretation

“Map” in this context denotes any one-to-one mapping of the world onto an internal representation.

Robot localization denotes the robot’s ability to establish its own position and orientation within the frame of reference. Path planning is effectively an extension of localisation, in that it requires the determination of the robot’s current position and a position of a goal location, both within the same frame of reference or coordinates. Map building can be in the shape of a metric map or any notation describing locations in the robot frame of reference.

Vision-based navigation

Vision-based navigation or optical navigation uses computer vision algorithms and optical sensors, including laser-based range finder and photometric cameras using CCD arrays, to extract the visual features required to the localization in the surrounding environment. However, there are a range of techniques for navigation and localization using vision information, the main components of each technique are:

representations of the environment. sensing models. localization algorithms.

In order to give an overview of vision-based navigation and its techniques, we classify these techniques under indoor navigation and outdoor navigation.

Indoor navigation

The easiest way of making a robot go to a goal location is simply to guide it to this location. This guidance can be done in different ways: burying an inductive loop or magnets in the floor, painting lines on the floor, or by placing beacons, markers, bar codes etc. in the environment. Such Automated Guided Vehicles (AGVs) are used in industrial scenarios for transportation tasks. Indoor Navigation of Robots are possible by IMU based indoor positioning devices.

There are a very wider variety of indoor navigation systems. The basic reference of indoor and outdoor navigation systems is “Vision for mobile robot navigation: a survey” by Guilherme N. DeSouza and Avinash C. Kak.

AVM Navigator

AVM Navigator is an additional module of the RoboRealm (plugin) that provides object recognition and autonomous robot navigation using a single video camera on the robot as the main sensor for navigation.

It is possible due to using of an “Associative Video Memory” (AVM) algorithm based on multilevel decomposition of recognition matrices. It provides image recognition with low False Acceptance Rate (about 0.01%). In this case visual navigation is just the sequence of images (landmarks) with associated coordinates that was memorized inside AVM tree during route training. The navigation map is presented as the set of data (such as X, Y coordinates and azimuth) associated with images inside AVM tree. When a robot sees images from camera (marks) that can be recognized then it confirms its current location.

The navigator creates a way from the current location to target position as a chain of waypoints. If the robot’s current orientation does not point to the next waypoint then the navigator turns the robot body. When the robot reaches a waypoint the navigator changes direction to the next waypoint in the chain and so on until the target position is reached.

Outdoor navigation

Some recent outdoor navigation algorithms are based on convolutional neural network and machine learning, and are capable of accurate turn-by-turn inference .

Autonomous Flight Controllers

Typical Open Source Autonomous Flight Controllers have the ability to fly in full automatic mode and perform the following operations;

Take off from the ground and fly to a defined altitude Fly to one or more waypoints

Orbit around a designated point Return to the launch position

Descend at a specified speed and land the aircraft

The on board flight controller relies on GPS for navigation and stabilized flight, and often employ additional Satellite-based augmentation systems (SBAS) and altitude (barometric pressure) sensor.

# PROGRAM 15:

## Explain Topological Path Planning and Metric Path Planning? Answer:

Topological Path Planning

Topological, route, or qualitative piloting is often seen as being more simple and natural for a behavior-based robot. Certainly people frequently give other people routes as directives; therefore it seems natural to expect a robot to be able to parse commands such as “go down the hall, turn to the left at the dead end, and enter the second room on the right.” Even without a map of where everything is, there is enough information for navigation as long as the robot knows what a “hall,” “dead-end,” and “room” is.

There are two representation of route :

1. Relational. Relational techniques are the most popular, and can be thought of as giving the robot an abbreviated.
2. Associative. Associative techniques focus on coupling sensing with localization in a manner which parallels the tight coupling of sensing to acting found in reflexive behaviors.

Relational methods relate distinctive places (nodes) to each other by the local control strategies (LCS), or behaviors, needed to travel between them (edges), forming a graph. The robot can use the graph to plan a path using techniques such as the single source shortest path algorithm. It executes the path by employing the behavior associated with that edge it is traversing. When it sees the landmark of the location, it is in the neighborhood, and then can use another behavior, such as hill-climbing, to localize itself relative to the landmark. Associative methods directly couple perception with acting. An image can be compared to a image signature or a view frame to generate the next movement for the robot to take. Relational methods are commonly used for topological navigation

Metric Path Planning

Metric path planning, or quantitative navigation, is the opposite of topological navigation. The major philosophical difference is that metric methods generally favor techniques which produce an optimal, according to some measure of best, while qualitative methods seem content to produce a route with identifiable landmarks or gateways. Metric path planners have two components: the representation (data structure) and the algorithm. Path planners first partition

the world into a structure amenable for path planning. They use a variety of techniques to represent the world; no one technique is dominant, although regular grids appear to be popular.

The intent of any representation is to represent only the salient features, or the relevant configuration of navigationally relevant objects in the space of interest; hence the term configuration space.

The algorithms fall into two broad categories: those which treat path planning as a graph search problem, and those which treat path planning as a graphics coloring problem. The physical space robots and obstacles exist in can be thought of as the world space. The configuration space, or Cspace for short, is a data structure which allows the robot to specify the position (location and orientation) of any objects and the robot. A good Cspace representation reduces the number of dimensions that a planner has to contend with. Consider that it takes six dimensions (also called degrees of freedom or DOF) to represent precisely where an object is. A person may specify the location of the object as a (x; y; z) coordinate in some frame of reference. Six degrees of freedom is more than is needed for a mobile ground robot in most cases for planning a path. The z (height) coordinate can be eliminated if every object the robot sits on the floor. However, the z coordinate will be of interest if the robot is an aerial or underwater vehicle.

Likewise, the Euler angles may be unnecessary. Who cares which way the robot is facing if all the robot wants to do is to plan a path around it? But the pitch of a planetary rover or slope of an upcoming hill may be critical to a mission over rocky terrain. These representations can then be converted to graphs, suitable for an A\* search. Since Voronoi diagrams tend to produce sparser graphs, they work particularly well with A\*. Regular grids work well with wavefront planners, which treat path planning as an expanding heat wave from the initial location. Metric path planning tends to be expensive, both in computation and in storage. However, they can be interleaved with the reactive component of a Hybrid architecture, where the Cartographer gives the Sequencer a set of waypoints. Cspace representations and algorithms often do not consider how to represent and reason about terrain types, and special cases such as the robot actually conserving or generating energy going downhill are usually ignored. A possibly even more serious omission is that popular path planners are applicable for only holonomic vehicles