APPENDIX-A

A.1 Dynamixel Motors

The mobile robot for used for experimental analysis utilizes two Dynamixel EX- 106 Motors for the purpose of actuation of the left wheel and the right wheel. The omnidirectional castor wheel utilized by the robot is not actuated. The Dynamixel servo motors use a precise closed loop DC motor mechanism with an independent microcontroller for computation. Table A1 summarizes some important properties of the Dynamixel motor which makes it suitable for use for the current application. The product manual for Dynamixel EX-106 has been added as a hyperlink.

Table A1: Important characteristics of Dynamixel EX-106 motors

Sr. No.	Type of characteristic	Characteristic
1	Communication protocol	RS 485 Asynchronous serial communication
2	Command signal	Digital packets of data
3	Motor mode required	Wheel Mode
4	Baud rate	1 Mbps
5	Feedback available	Load, Input voltage, position etc.
6	Resolution	0.07 degrees

A.1.1 Dynamixel Motors - Creating Packets

As mentioned earlier, the Dynamixel motors used a packet-based communication in order to control the motor and make use of various inbuilt functionalities. For various applications, different packets of data need to be sent to the motor. The communication procedures can be explained by the following points: -

1) **Control Table-** Control table is an area reserved in the motors memory which consists of information regarding the data present which controls various kind of actions in the motor such as goal position, moving speed, present position, sensed current etc. Some functionalities are located in the EEPROM memory, while others are stored in the RAM. Using the hexadecimal address of these particular memory

locations, the users can send data packets in order to control or read the status of the major functionalities involved with the motor.

2) **Instruction Packet-** The instruction packets are used to send a certain instruction to the motor. This is done by sending an instruction packet with certain parameter data values accompanied with the address to which the data has to be sent. Figure A1 shows the structure of an instruction packet labelled with what the different bits represent.

OXFF	0XFF	ID	LENGTH	INSTRUCTION	PARAMETER1	…	PARAMETER N	CHECK SUM

Figure A1: Structure of an instruction packet

3) **Status Packet-** The status packet is similar to the instruction packets in structure. However, instead of executing a change in the data values in the control table, the status packet is used to read and analyze the current values in the control table. Status packets are used to check if the motor is running with optimum efficiency. An error bit is also added in the status packet if data is not received efficiently. Figure A2 shows the structure of a status packet labelled with what the different bits represent.

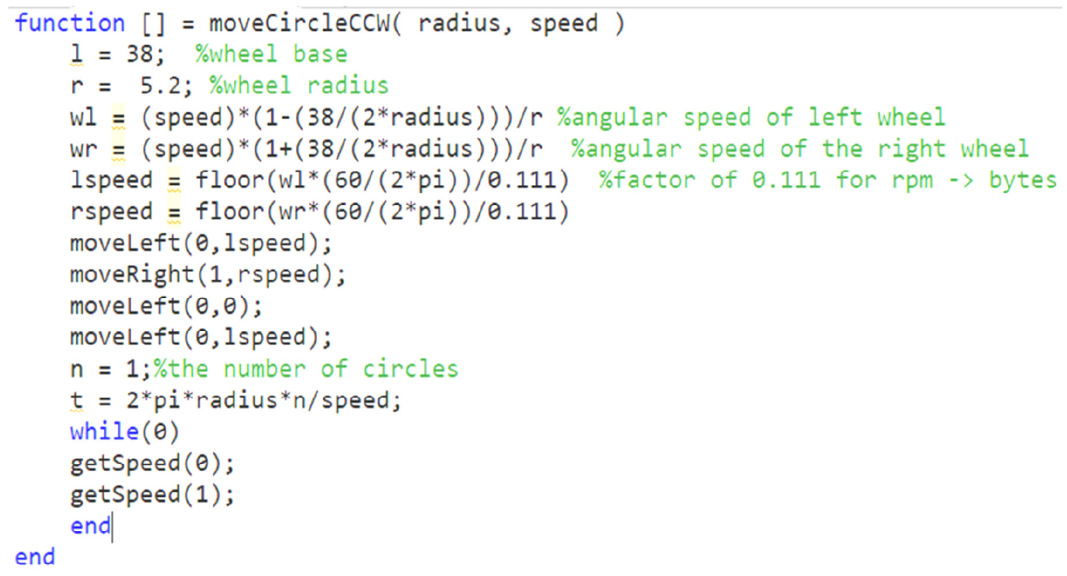
OXFF	0XFF	ID	LENGTH	ERROR	PARAMETER1	PARAMETER2	…	PARAMETER N	CHECK SUM

Figure A2: Structure of a status packet

Using this packet-based communication, MATLAB software is utilized to set up a communication between the PC and Dynamixel motors. The PC hence acts as a replacement for the embedded controllers. Multiple programs have been created in MATLAB in order to allow the mobile robot to perform many common functionalities, some of which are listed as follows:-

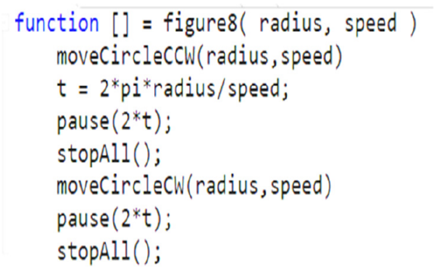
1) Move left and right
2) Moving in a straight line
3) Move circle clockwise and counter-clockwise
4) Read current speed in RPM
5) Set goal position
6) Pure rotation at a single position

Figure A3 and Figure A4 showcase MATLAB programs to create a circle in the counter-clockwise direction and create a figure of 8 using the robot model respectively. The following programs have been tested in real-time with the two-wheeled differential drive robot model.



```
function [] = moveCircleCCW( radius, speed )
    l = 38;  %wheel base
    r =  5.2; %wheel radius
    wl = (speed)*(1-(38/(2*radius)))/r %angular speed of left wheel
    wr = (speed)*(1+(38/(2*radius)))/r  %angular speed of the right wheel
    lspeed = floor(wl*(60/(2*pi))/0.111)  %factor of 0.111 for rpm -> bytes
    rspeed = floor(wr*(60/(2*pi))/0.111)
    moveLeft(0,lspeed);
    moveRight(1,rspeed);
    moveLeft(0,0);
    moveLeft(0,lspeed);
    n = 1;%the number of circles
    t = 2*pi*radius*n/speed;
    while(0)
    getSpeed(0);
    getSpeed(1);
    end
end
```

Figure A3: MATLAB program to create circle in counter-clockwise direction



```
function [] = figure8( radius, speed )
    moveCircleCCW(radius,speed)
    t = 2*pi*radius/speed;
    pause(2*t);
    stopAll();
    moveCircleCW(radius,speed)
    pause(2*t);
    stopAll();
```

Figure A4: MATLAB program to make a figure of Eight

A.2 Flexible code in Matlab for close loop control of Dynamixel EX106+ actuator

%File Name: final.m%

Disp ('Mobile Robot'); % Project Title

loadlibrary ('dynamixel', 'dynamixel.h'); % Load library files

libfunctions ('dynamixel'); % Load functions

calllib ('dynamixel', 'dxl_initialize', 16, 1); % Set port and speed with check on initialization

d=input('\nDistance to move in mm: '); % 1000

```
if (isempty(d))                               % invalid input check
d=0;                                          % distance zero for invalid input
end                                           % end of if loop
s=input ('\nSpeed Percentage: ');             % 30
if (isempty(s))                               % invalid input check
s=0;                                          % speed zero for invalid input
end                                           % end of if loop
h=abs (round((5265/575)*d));                  % Hexadecimal distance for motor
s=abs (round ((s/100)*1023));                       % Hexadecimal speed for motor
i=0; j=0; m=4;          % i for distance counter, j for temporary variable & m for motor id
if h==0                                       % if zero distance input
elseif d<0                                    % if negative distance input
m=1; a=s; b=1023+s;
% set motor id 1 for reading, motor 1&2 counter clockwise, motor 3&4 clockwise
p=int32 (calllib ('dynamixel','dxl_read_word', m, 36) ) ;
% read motor 'm' position
pause(0.0001);                                      % delay
tx(a,b);                                            % call packet transmission function
pause (0.0001);                                     % delay
else                                                % if positive distance input
a=1023+s;b=s;
% keep motor id 4 for reading, motor 1&2 clockwise, motor 3&4 counter clockwise
p=int32 (calllib('dynamixel','dxl_read_word',m,36)); % read motor m position
pause(0.0001);                                      % delay
tx(a,b);                                            % call packet transmission function
end                                                 % end of if loop
while i<h                                           % while distance is reached
q=int32(calllib('dynamixel','dxl_read_word',m,36)); % read motor 'm' position
if q-p>0                                            % if positive difference
i=i+q-p;                                            % add distance counter to original
else                                                % if negative difference
i=i+q-p+65535;
% add distance counter to original with crossover maximum limit of motor encoder
end                                                 % end of if loop
```

```
p=q;                                              % new initial position
pause(0.0001);                                    % delay
end                                               % end of while loop
calllib('dynamixel','dxl_write_word',254,32,0);   % transmit 0% speed to all motors
q=int32(calllib('dynamixel','dxl_read_word',m,36));
% read motor m position
calllib('dynamixel','dxl_terminate');             % Terminate
unloadlibrary('dynamixel');                       % Unload library files
%File Name: tx.m%
function[]=tx(m12,m34)                            % declare function for packet
making and transmission
loadlibrary('dynamixel', 'dynamixel.h');          % Load library files
libfunctions('dynamixel');                        % Load functions
calllib('dynamixel', 'dxl_initialize', 16, 1);    % Set port and speed with check on
initialization
calllib('dynamixel','dxl_set_txpacket_id',254);   % Broadcast id 0xFE (254)
calllib('dynamixel','dxl_set_txpacket_length',16);
% Packet Length is 16 (L+1) x N+4 where 'L' is number of data to write and 'N' is
number of dynamiels
calllib('dynamixel','dxl_set_txpacket_instruction',131);
% SyncWrite instruction 0x83 (131)
calllib('dynamixel','dxl_set_txpacket_parameter',0,32);
% Starting address to write 0X20 (32)
calllib('dynamixel','dxl_set_txpacket_parameter',1,2);
% Length of data to write to each dynamixel = 4
calllib('dynamixel','dxl_set_txpacket_parameter',2,1);
% Parameter for syncwrite dynamixel id = 1
lowByte = calllib('dynamixel','dxl_get_lowbyte',m12);
% Low Byte Speed Percentage m12
highByte = calllib('dynamixel','dxl_get_highbyte',m12);
% High Byte Speed Percentage m12
calllib('dynamixel','dxl_set_txpacket_parameter',3,lowByte);
% Set lowbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',4,highByte);
```

```
% Set highbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',5,2);
% Parameter for syncwrite dynamixel id = 2
lowByte = calllib('dynamixel','dxl_get_lowbyte',m12);
% Low Byte Speed Percentage m12
highByte = calllib('dynamixel','dxl_get_highbyte',m12);
% High Byte Speed Percentage m12
calllib('dynamixel','dxl_set_txpacket_parameter',6,lowByte);
% Set lowbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',7,highByte);
% Set highbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',8,3);
% Parameter for syncwrite dynamixel id = 3
lowByte = calllib('dynamixel','dxl_get_lowbyte',m34);
% Low Byte Speed Percentage m34
highByte = calllib('dynamixel','dxl_get_highbyte',34);
% High Byte Speed Percentage m34
calllib('dynamixel','dxl_set_txpacket_parameter',9,lowByte);
% Set lowbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',10,highByte);
% Set highbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',11,4);
% Parameter for syncwrite dynamixel id = 4
lowByte = calllib('dynamixel','dxl_get_lowbyte',34);
% Low Byte Speed Percentage m34
highByte = calllib('dynamixel','dxl_get_highbyte',34);
% High Byte Speed Percentage m34
calllib('dynamixel','dxl_set_txpacket_parameter',12,lowByte);
% Set lowbyte Speed Percentage
calllib('dynamixel','dxl_set_txpacket_parameter',13,highByte);
% Set highbyte Speed Percentage
calllib('dynamixel','dxl_tx_packet'); % Transmit Packet
calllib('dynamixel', 'dxl_initialize', 16, 1);
% Set port and speed with check on initialization
```

APPENDIX-B

Robot Operating System (ROS)

The Robot Operating System (ROS) is a middleware/meta-operating system which not only provides support for hardware-abstraction and low-level control, but also provides support for various software-based tasks such as computer vision (OpenCV) and ROS stacks for control (ros_control), navigation (ros_navigation) as well as third-party libraries for Simultaneous Localization and Mapping (SLAM) from various different institutions.

The primary advantage of a system like ROS is platform independence, modularity, and portability. The network is device independent and can be setup up efficiently and swiftly on a new platform. ROS also provides good support for 3-D simulation and 3-D visualization environments like Gazebo and RViz which make it suitable for use. The file-system of ROS also makes it efficient to distribute open-source software and build custom packages as per needed using a catkin toolchain. Figure B1 shows how a middleware layer adds to the layer of hardware abstraction.

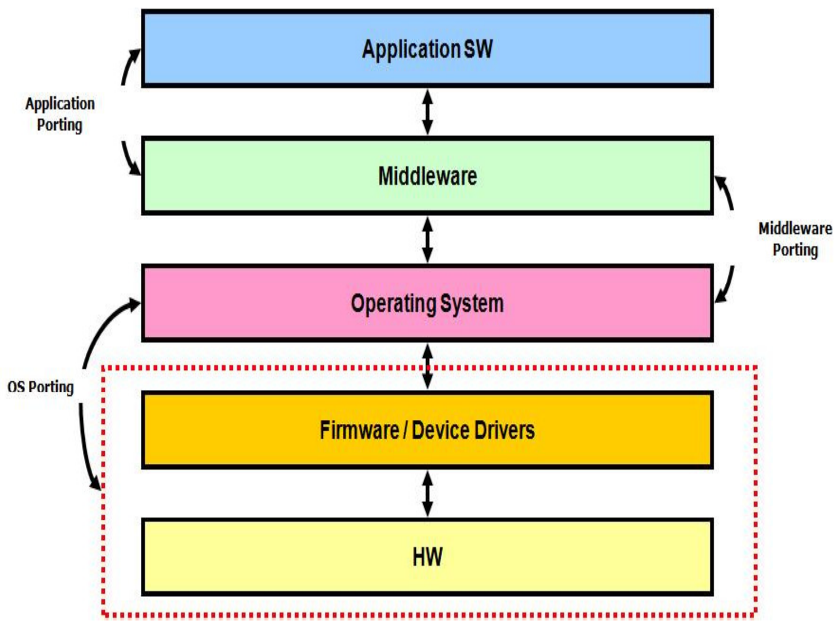


Figure B1: Middleware for hardware abstraction

ROS: Core Components

The core components of the ROS networking system which have been utilized have been summarized as follows: -

1) **Master:** ROS Master provides a name registration service for various nodes and hence allows nodes to internally communicate with each other using messages or services.
2) **Parameter Server:** It is a multi-variable dictionary which is shared between nodes and used to store static data like configuration parameters of various devices for real-time operations.
3) **Nodes:** ROS Nodes are basic computational units in the ROS Computation graph. These nodes can publish data to a topic or subscribe to data from a topic.
4) **Messages:** ROS Messages are simple data types which define the type of data being sent between topics from one node to another.
5) **Topics:** ROS Topics act as data-buses which are used by nodes in order to send messages of data. Publishers will publish data on the topic while the subscribers will subscribe to a stream of data from a topic.
6) **Bags:** ROS Bags are typically used to store data from a topic in a serialized fashion. Generally, sensor data from various sensors is stored in form of bags for further analysis. Figure B2 shows the ROS Computational level graph with various processes available.

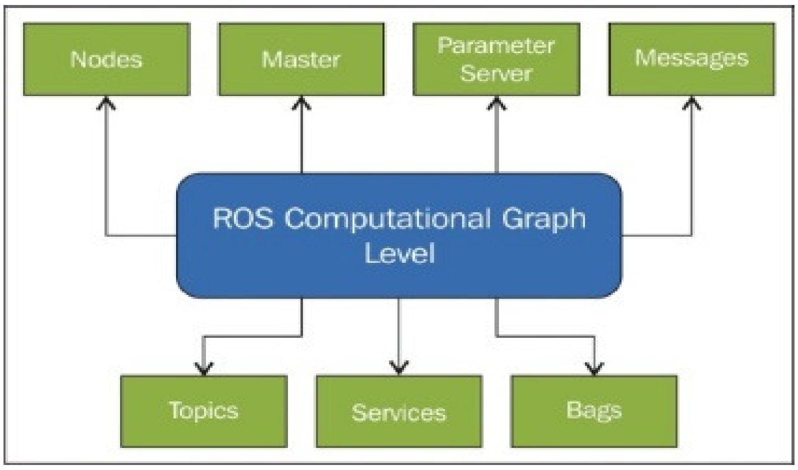


Figure B2: ROS computational level graph

APPENDIX-C

Camera Calibration

The concept of camera calibration and perspective projections comes into importance when certain pixel coordinates have to be converted to real-world measurements which can be used for a particular task. Camera calibration is the process of estimating the extrinsic and intrinsic parameters of a camera using indirect methods. Camera calibration allows the user to remove errors which appear due to radial distortion, noise and other non-linear image distortions. The camera calibration approach involves the estimation of two matrices: -

1) **Extrinsic Parameters**- The extrinsic parameters compute a general rigid body transformation for points in some arbitrary world coordinate frame (X, Y, Z) to points in the camera's 3-D coordinate frame (x, y, z). Equation (c1) shows the matrix representing the extrinsic parameters where R is the rotation matrix, t is the translation matrix. In the matrix, r and t are the components of the rotation and translation in different directions.

$$[R/t] = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{c1}$$

2) **Intrinsic Parameters**- The intrinsic parameters compute a transformation for points in the camera's 3-D coordinate frame (x, y, z) to coordinates on the 2-D image plane (u, v) via projections. Equation (c2) shows the matrix representing the intrinsic parameters where the following parameters exist: -

   a) f_x & f_y (focal lengths) are the distance between the pinhole camera and the image plane.
   b) c_x & c_y (principal offsets) are the distance between the point of intersection of the perpendicular from the pinhole camera to the image plane (principal point) and the origin of the image plane.
   c) s is the axis skew which causes shear distortion. This particular value is more prominent in camera with a fisheye lens or stereo cameras, rather than monocular pinhole camera models.

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{c2}$$

Figure C1 summarizes the process of conversion from a point in world coordinate space (X, Y, Z) to pixel coordinate space (u, v). The conversion involves the use of the extrinsic and intrinsic parameters. Figure C2 shows a projection from 3-D coordinates of a point in the world-coordinate frame to the 2-D coordinates in the pixel coordinate frame of the camera.

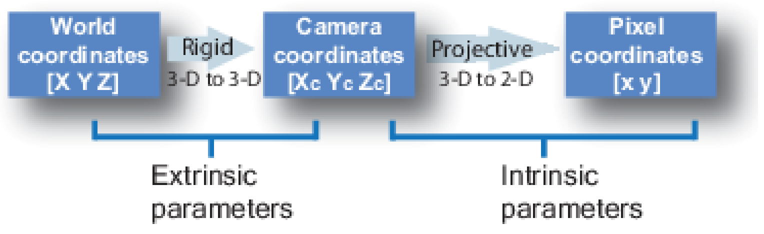


Figure C1: Conversion from world to image coordinates

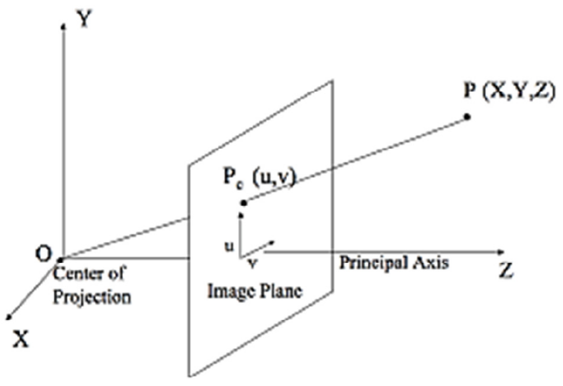


Figure C2: Projection from 3-D coordinates to 2-D pixel coordinates

Equation (c3) gives the simplified version of the entire equation to obtain the camera matrix P. Equation (c4) gives the decomposed version of the entire equation to obtain the camera matrix P (3×4 matrix).

$$P = K[R/t] \tag{c3}$$

$$P = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{s}{f_x} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{bmatrix} \qquad \text{(c4)}$$

Equation (c5) shows the entire equation required to convert a point in 3D world coordinate frame (X, Y, Z) to a point in the image plane (u, v) where ‘s’ is the scale factor and P is the camera matrix (3×4 matrix) as computed using Equation (c4).

$$[w] \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad \text{(c5)}$$

Figure C1 shows the code snippet which is used to estimate the real-world coordinates from the image coordinates (u, v). MATLAB internally uses numerical optimization methods in order to get the required result.

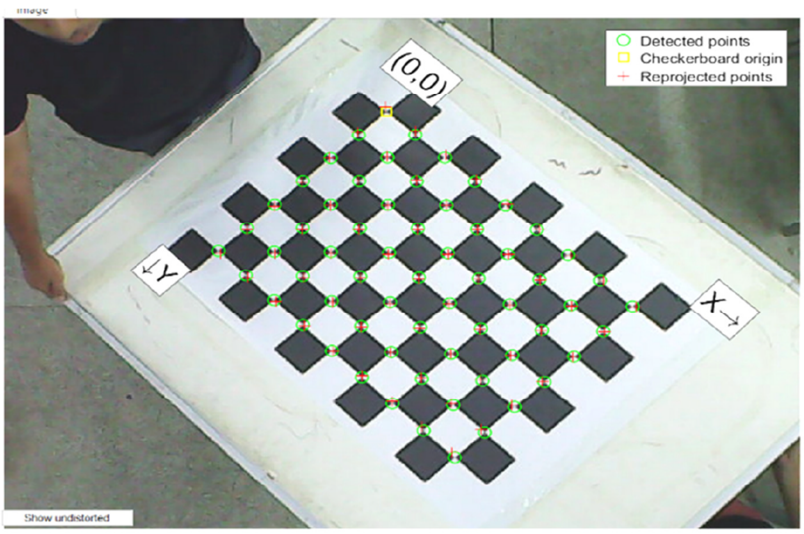


Figure C3: Sample image for camera calibration

A sample image used for camera calibration is shown in Figure C3. Figure C4 shows the re-projection errors for the 22 images which have been used for the process of calibration. Figure C5 shows the views of the 22 checkerboard configurations in Euclidean space as seen by the camera.

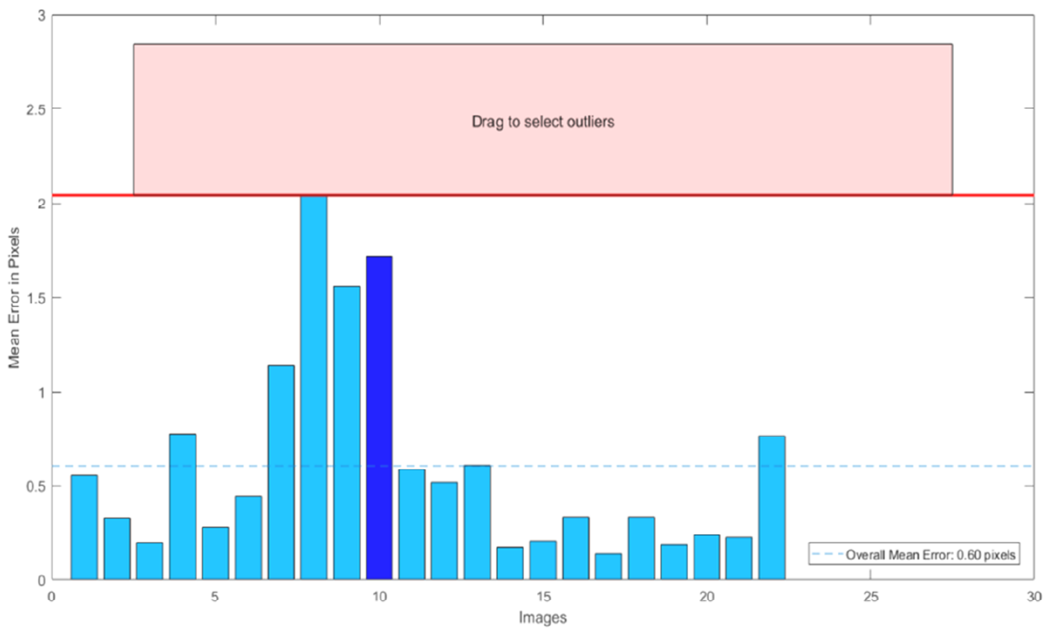


Figure C4: Re-projection error for the 22 images

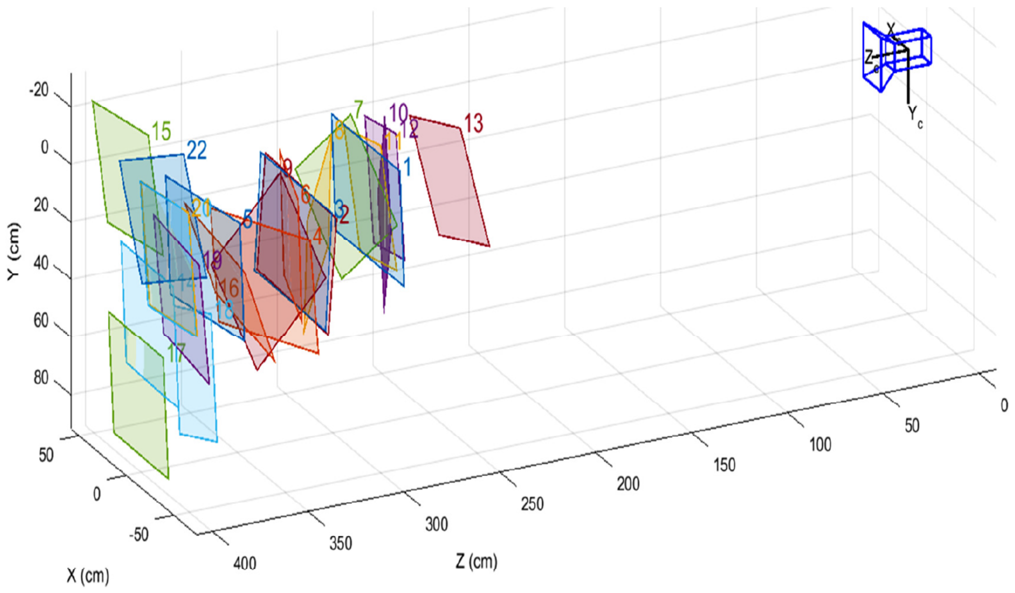
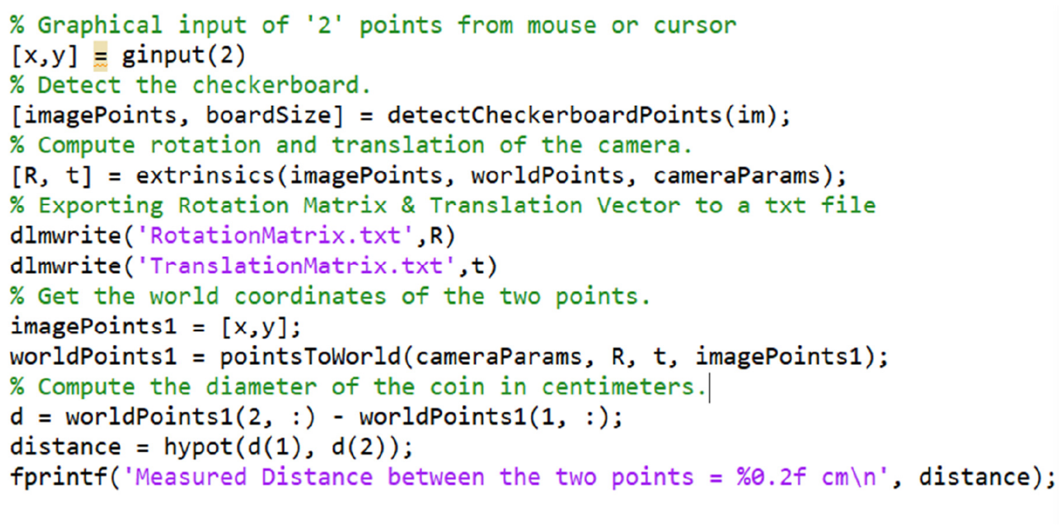


Figure C5: View of checkerboard configurations as seen from the camera

Hence, using the method of camera calibration, the (u, v) coordinate values of optimal path are translated into real-world coordinates (x, y). These real-world coordinates are used to provide actuation commands to the mobile robot in real world dimensions. Warning: The checkerboard must be asymmetric: one side should be even, and the other should be odd. Otherwise, the orientation of the board may be detected incorrectly.



```
% Graphical input of '2' points from mouse or cursor
[x,y] = ginput(2)
% Detect the checkerboard.
[imagePoints, boardSize] = detectCheckerboardPoints(im);
% Compute rotation and translation of the camera.
[R, t] = extrinsics(imagePoints, worldPoints, cameraParams);
% Exporting Rotation Matrix & Translation Vector to a txt file
dlmwrite('RotationMatrix.txt',R)
dlmwrite('TranslationMatrix.txt',t)
% Get the world coordinates of the two points.
imagePoints1 = [x,y];
worldPoints1 = pointsToWorld(cameraParams, R, t, imagePoints1);
% Compute the diameter of the coin in centimeters.
d = worldPoints1(2, :) - worldPoints1(1, :);
distance = hypot(d(1), d(2));
fprintf('Measured Distance between the two points = %0.2f cm\n', distance);
```

Figure C6: Code snippet for conversion from image to world coordinates

APPENDIX - D

D.1: D* Lite Pseudo Code

The following pseudo code has been adapted from [1].

```
Function CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s_start, s) + km; min(g(s), rhs(s))];
Function Initialize()
{02} U = ∅;
{03} km = 0;
{04} for all s ∈S rhs(s) = g(s) = ∞;
{05} rhs(s_goal) = 0;
{06} U.Insert(s_goal, [h(s_start, s_goal); 0]);
Function UpdateVertex(u)
{07} if (g(u) != rhs(u) AND u ∈U) U.Update(u,CalculateKey(u));
{08} else if (g(u) != rhs(u) AND u /∈U) U.Insert(u,CalculateKey(u));
{09} else if (g(u) = rhs(u) AND u ∈U) U.Remove(u);

Function ComputeShortestPath()
{10} while (U.TopKey() <CalculateKey(s_start) OR rhs(s_start) > g(s_start))
{11} u = U.Top();
{12} k_old = U.TopKey();
{13} k_new = CalculateKey(u));
{14} if(k_old<k_new)
{15} U.Update(u, k_new);
{16} else if (g(u) >rhs(u))
{17} g(u) = rhs(u);
{18} U.Remove(u);
{19} for all s ∈Pred(u)
{20} if (s != s_goal) rhs(s) = min(rhs(s), c(s, u) + g(u));
{21} UpdateVertex(s);
{22} else
{23} g_old = g(u);
{24} g(u) = ∞;
{25} for all s ∈Pred(u) ∪{u}
{26} if (rhs(s) = c(s, u) + g_old)
{27} if (s != s_goal) rhs(s) = min s'∈Succ(s)(c(s, s') + g(s'));
{28} UpdateVertex(s);
Function Main()
{29} s_last = s_start;
{30} Initialize();
{31} ComputeShortestPath();
```

```
{32} while (s_start != s_goal)
{33} /* if (g(s_start) = ∞) then there is no known path */
{34} s_start = argmin s'∈Succ(s_start)(c(s_start, s') + g(s'));
{35} Move to s_start;
{36} Scan graph for changed edge costs;
{37} if any edge costs changed
{38} km = km + h(s_last, s_start);
{39} s_last = s_start;
{40} for all directed edges (u, v) with changed edge costs
{41} c_old = c(u, v);
{42} Update the edge cost c(u, v);
{43} if (c_old>c(u, v))
{44} if (u != s_goal) rhs(u) = min(rhs(u), c(u, v) + g(v));
{45} else if (rhs(u) = c_old + g(v))
{46} if (u != s_goal) rhs(u) = min s'∈Succ(u)(c(u, s') + g(s'));
{47} UpdateVertex(u);
{48} ComputeShortestPath()
```

D.2: Setup Configuration for navigation Stack

```
max_obstacle_height: 0.60
# assume something like an arm is mounted on top of the robot
robot_radius: 0.20
# distance a circular robot should be clear of the obstacle (kobuki:0.18)
map_type: voxel
obstacle_layer:
enabled: true
max_obstacle_height: 0.6
origin_z: 0.0
z_resolution: 0.2
z_voxels: 2
unknown_threshold: 15
mark_threshold:  0
combination_method: 1
track_unknown_space: true
#true needed for disabling global path planning through unknown space
obstacle_range: 2.5
raytrace_range: 3.0
origin_z: 0.0
z_resolution: 0.2
z_voxels: 2
publish_voxel_map: false
observation_sources: scan bump
```

```
scan:
data_type: LaserScan
topic: scan
marking: true
clearing: true
min_obstacle_height: 0.25
max_obstacle_height: 0.35
bump:
data_type:  PointCloud2
topic: mobile_base/sensors/bumper_pointcloud
marking: true
clearing: false
min_obstacle_height: 0.0
max_obstacle_height: 0.15
# for debugging only, let's you see the entire voxel grid
#cost_scaling_factor and inflation_radius were now moved to the inflation_layer ns
inflation_layer:
enabled: true
cost_scaling_factor:  12.0
# exponential rate at which the obstacle cost drops off
inflation_radius: 0.40
# max. distance from an obstacle at which costs are incurred for planning paths.
static_layer:
enabled: true
```

D.3: LOCAL Planner Configuration (DWA Planner ROS)

```
# Robot Configuration Parameters - Kobuki
max_vel_x: 0.15 # 0.55
min_vel_x: 0.0
max_vel_y: 0.0
# diff drive robot
min_vel_y: 0.0
# diff drive robot
max_trans_vel: 0.15
# choose slightly less than the base's capability
min_trans_vel: 0.1
#This is the min trans velocity when there is negligible rotational velocity
trans_stopped_vel: 0.1
# Warning!
# Do not set min_trans_vel to 0.0 otherwise dwa will always think translational
velocities
# Are non-negligible and small in place rotational velocities will be created.
max_rot_vel: 1.5 # choose slightly less than the base's capability
```

```
min_rot_vel: 0.4
# This is the min angular velocity when there is negligible translational velocity
rot_stopped_vel: 0.4
# Acceleration values:
acc_lim_x: 1.0 # maximum is theoretically 2.0
acc_lim_theta: 2.0
acc_lim_y: 0.0 # diff drive robot
# Goal Tolerance Parameters
yaw_goal_tolerance: 0.3 # 0.05
xy_goal_tolerance:  0.15 # 0.10
# latch_xy_goal_tolerance: false
# Forward Simulation Parameters
sim_time: 1.0
vx_samples: 6
vy_samples: 1
# diff drive robot, there is only one sample
vtheta_samples: 20
# Trajectory Scoring Parameters
path_distance_bias: 64.0
# Weighting for how much it should stick to the global path plan
goal_distance_bias: 24.0
# 24.0 - weighting for how much it should attempt to reach its goal
occdist_scale: 0.8
# 0.01-weighting for how much the controller should avoid obstacles
forward_point_distance: 0.325
# 0.325 - how far along to place an additional scoring point
stop_time_buffer: 0.2
# 0.2 - amount of time a robot must stop in before colliding for a valid trajectory.
scaling_speed: 0.10
# 0.25 - absolute velocity at which to start scaling the robot's footprint
max_scaling_factor: 0.2
0.2 - how much to scale the robot's footprint when at speed.
# Oscillation Prevention Parameters
oscillation_reset_dist: 0.05
# how far to travel before resetting oscillation flags
# Debugging
publish_traj_pc : true
publish_cost_grid_pc: true
global_frame_id: odom
# Differential-drive robot configuration - necessary?
# holonomic_robot: false
```

D.4: Code for kalman filtering:

```
%experiment test- 1 & 2
% X-cood
        Q=[1.25 0;0 1.25];
        R=[3.782 0;0 7.491];
        kal_x(1,1)=tld_x_exp1(1,1);
        A=[1 0;0 1];
        Pk=[1 1;1 1];
        C=[1 0;1 0];
        I=[1 0;0 1];
        for i=2:1471
%prediction step
        Pk_prev=Pk;
        pred_x=[kal_x(i-1,1)];
Pk=A*Pk_prev*A'+Q;
%update step
        InvMatrix=C * Pk *(C')+ R ;
        Gk=Pk * C' / InvMatrix;
        Zk=[odo_x_exp1(i,1) 0;tld_x_exp1(i,1) 0];
        updt_x=pred_x+Gk*(Zk-(C*pred_x));
        kal_x(i,1)=updt_x(1,1);
        Pk=(I-(Gk*C))*Pk;
end
% Y-cood
        Q=[1.25 0;0 1.25];
        R=[3.395 0;0 8.591];
        kal_y(1,1)=tld_y_exp1(1,1);
        A=[1 0;0 1];
        Pk=[1 1;1 1];
        C=[1 0;1 0];
        I=[1 0;0 1];
        for i=2:1471
%prediction step
        Pk_prev=Pk;
        pred_y=[kal_y(i-1,1)];
        Pk=A*Pk_prev*A'+Q;
%update step
        InvMatrix=C * Pk *(C')+ R ;
        Gk=Pk * C' / InvMatrix;
        Zk=[odo_y_exp1(i,1) 0;tld_y_exp1(i,1) 0];
        updt_y=pred_y+Gk*(Zk-(C*pred_y));
        kal_y(i,1)=updt_y(1,1);
        Pk=(I-(Gk*C))*Pk;
        plot(odo_y_exp1,odo_x_exp1,tld_y_exp1,tld_x_exp1,kal_y,kal_x)
```