Autonomous Interoffice Delivery Robot (AIDeR)

Software Development of the Control Task

A Thesis Presented to

The Faculty of the Computer Science Program

California State University Channel Islands

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Ludovic Roger Hilde

August 2009

APPROVED FOR THE COMPUTER SCIENCE PROGRAM

Advisor: Dr. Andrzej Bieszczad	Date
Dr. William Wolfe	Date
Dr. Peter Smith	Date

APPROVED FOR THE UNIVERSITY

Dr. Gary A. Berg	Date

Autonomous Interoffice Delivery Robot (AIDeR)

Software Development of the Control Task

by
Ludovic Roger Hilde

Computer Science Program
California State University Channel Islands

Abstract

Autonomous robots are able to perform desired tasks within a predefined environment for a substantial amount of time without human support. These types of robots may be used in various domains such as indoor or outdoor cleaning, intrusion detection, space research, law enforcement, homeland security, healthcare, etc. They have become the topic of many research projects.

This thesis presents details about the control task, which is a software component of an application that uses an autonomous robot for indoor delivery. The task executes several main pieces, namely, identifying the defined landmarks, detecting obstacles, controlling movements, and communicating with peripheral devices and other tasks in the system.

Acknowledgements

The author would like to thank all members of the development team for their responsiveness in proposing solutions on arising issues. I would like to thank Dr. Andrzej Bieszczad for giving me the opportunity of being a part of this project.

TABLE OF CONTENT

1 Introduction

1.1 Purpose of the project

The control task is part of the development process of the AIDeR project. It provides a high level interface to higher level tasks for controlling movements of the robot and relaying events or alarms occurring during normal operation. The purpose of the overall project is to create an autonomous robot capable of delivering a load between two locations in an indoor environment. This is accomplished by allowing users to schedule jobs from a web or local interface in order to deliver packages from their offices to other places in the building. The robot is capable of prioritizing and reordering pending jobs in its queue based on its current location, so as to deliver packages throughout the building quickly and efficiently.

This project has been developed for Advanced Motion Control (AMC). It allows the company to showcase to potential customers one of its main products - the servo drives - used in a real application.

1.2 Development phases

This project contains two major development phases:

- The first phase, which had been completed in March 2006 by students at California Polytechnic State University, pertained to the construction of the hardware platform.

- The second phase, which is being completed by students at California State University Channel Islands, pertains to the development of the software modules.

1.3 Software modules

The team has chosen a layered approach to the development of the application. These layers are (in ascending order): the hardware layer, the OS layer, the control layer, the command layer, the interface layer.
With the exception of the hardware layer, each layer resulted to the development of five tasks: Controller task, Web interface task, GUI task, Command task, and the OS task. The Web interface and GUI tasks are both part of the interface layer.

The Web Interface task allows a user to interact with AIDeR via a web browser. Users may add new jobs to the queue or delete existing ones from it, query for the status of existing jobs, and be informed about alarm or event conditions.

The GUI task allows a user to interact with AIDeR via the touch screen mounted on the robot. Users may add new jobs to the queue or delete existing ones from it, query for the status of existing jobs, be informed about alarm or event conditions, and issue commands controlling the movement of the robot.

The command task is an intermediate layer between the Web interface-GUI tasks and the controller task. It calculates the shortest path of a new job, maintains the job queue, and relays any alarm condition received from the control task to the Web Interface. This task is presented in great details in [10].

The controller task is an intermediate layer between the command-GUI tasks and the operating system. It provides a set of APIs for controlling the movement of the robot (i.e. move forward by 100 cm, rotate left by 80 degrees). It generates any alarm condition or event that may have occurred during the execution of a command.

The OS task is a layer between the hardware and the controller task. It communicates with the peripheral devices such as the servo drives, sensors, and etcetera. It assigns priority levels to the running tasks. The OS has to execute in real time. Additional details about the OS task are presented in the Technology section of this document.

Figure 1 depicts the software components of the entire system and the interaction between the various tasks and the external components.

1.4 Remaining Chapters

In the second chapter, we present an overview of the list of functions supported by the control task. The functionality is divided into two main categories:

- The functions pertaining to controlling the movement of the robot
- The functions pertaining to reporting the status during normal operation

In the third chapter, we will discuss the design details pertaining to the control task. This section focuses on the major topic of this research, which is identification of the various landmarks. It also describes the functions controlling the movement of the robot.

The fourth chapter presents some of the implementation details. This section focuses on explaining the communication interface used between the control and command-GUI tasks.

The fifth chapter shows the results obtained during the debugging phase.

The conclusion and future work about the project are presented in chapters six and seven.

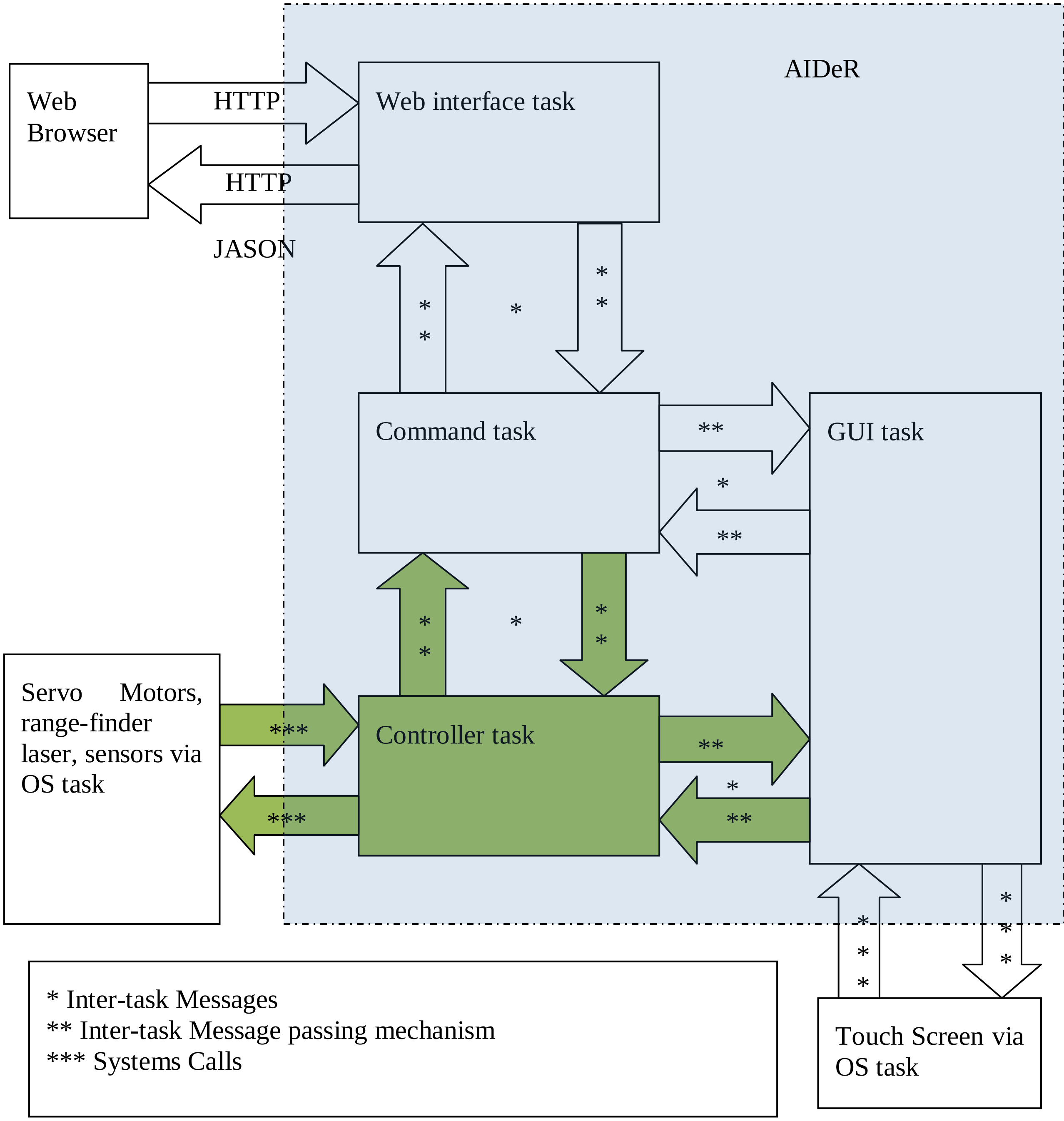


Figure 1 – Task Running on AIDeR

2 Overview

The control task provides a set of functions for controlling the movements of the robot. These functions are called indirectly via message passing by either the command task or the GUI task.
The control task first processes all control messages received from the GUI task and then those received from the command task. This allows a user to manually control the robot from the local interface without worrying about remote activities generated on the web interface by remote users. Multiple control messages sent by a single task are processed in first in first out manner.
The control task enters the busy state once it starts processing a control command, and it returns to the idle state after completing it. This is shown in Figure 2.

The list of functions controlling the movements of the robot is:

- Move forward by linear distance x at speed y
- Rotate by x degrees
- Travel along the right wall to a specified landmark
- Stop
- Enter the right hallway
- Enter the front hallway
- Enter the left hallway
- Move tray into specified position
- Make u-turn

As shown in Figure 2, the status of each control command is formatted as event or alarm messages that are sent asynchronously to the task that originated the request. There could be up to three messages generated for each control command:

- An event message reporting the change of status from idle to busy prior to the execution of a control command.
- An alarm message reporting an error condition that occurred during the execution of a control command.
- An event message reporting the change of status from busy to idle prior to the execution of a control command. Note that this message is sent whether or not the command has executed successfully.

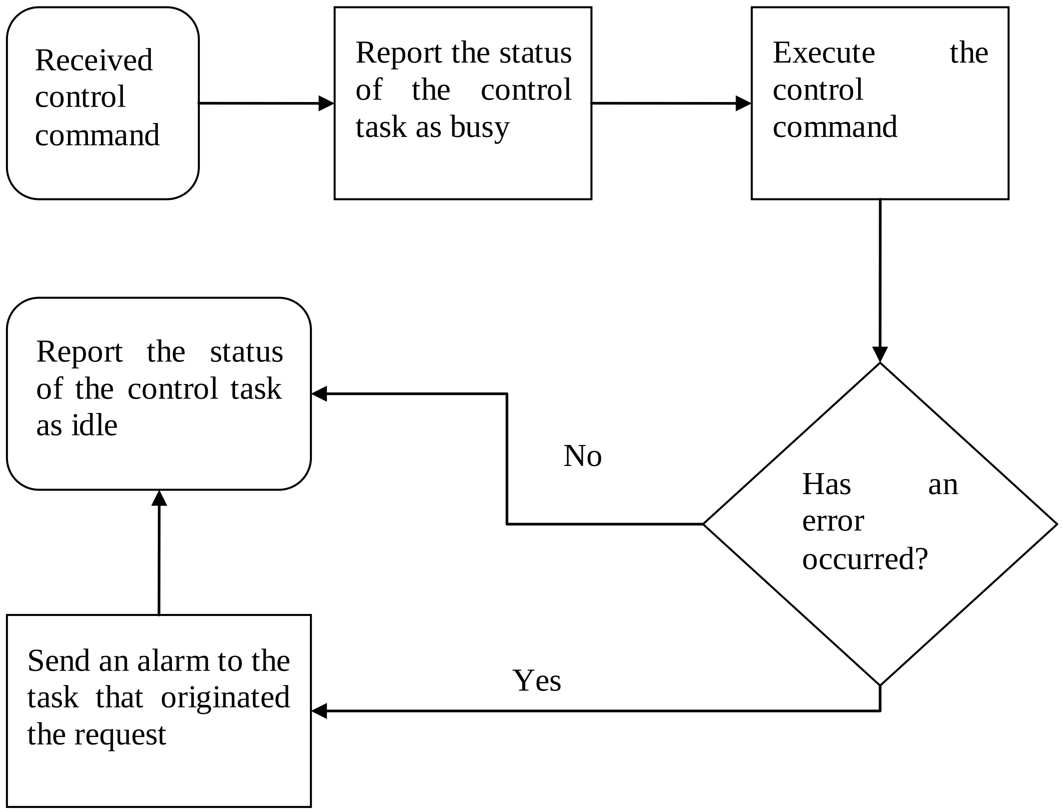


Figure 2 – Processing Control Command State Machine

The following table shows the list of alarms that may occur during the execution of a control command.

Alarm Condition	**Control Command(s)**	**Details**
Timeout	Move forward Rotate Travel against the Wall Enter left hallway Enter right hallway Enter front hallway	The execution time of the control command has exceeded the timeout value specified in the parameter of that command.
Detected Obstacle	Move forward Rotate Travel against the Wall Enter left hallway Enter right hallway Enter front hallway Make U-turn	The pathway of the robot is blocked by an unidentified object or person. As a result, the robot cannot proceed with the execution of the control command.
Communication failure with	Move forward	The control task is not able

the servo drives	Rotate Travel against the Wall Enter left hallway Enter right hallway Enter front hallway Make U-turn Move tray	to communicate with the servo drives via the serial interface.
Wall not found	Travel against the Wall	The control task failed to place AIDeR against the right wall.
Landmark not found	Travel against the Wall	The control task failed to identify the landmark ID specified in the parameter of the control command.

The control task monitors the power level of the battery pack; if the level falls below the normal level then a "Batteries Low" alarm gets reported to the command and GUI tasks. The battery low alarm has been further divided into three alarm levels:

Battery Voltage Level	**Details**
< 46 Volts	Level 1, shutdown and prevent all further movements. Immediate attention is required.
>= 46 Volts and < 48 Volts	Level 2, hold all pending control commands, redirect robot to the charging station.
= 48 Volts	Level 3, continue operation, but annunciate that charging may soon be required.
> 48 Volts and < 59 Volts	Normal operation, no alarm is reported.
>= 59 Volts	Level 1, shutdown and prevent all further movements. Immediate attention is required.

The control task sends event information that occurs during the execution of a control command, or at its completion. These events are:

Event	**Control Command(s)**	**Details**
Current State	all	The current state is busy before the execution of the control command, or it is idle at its completion.
Platform Status	Move Tray	The platform is reported as either raised or lowered at the completion of the move tray command.
Detected landmark	Travel against the Wall	The landmark specified by its ID value in the parameter of the control

		command has been detected.

The communication protocol between the control task and command-GUI tasks is defined in the implementation detail section.

3 Design Details

The robot has a map on its file system containing information about the environment in which it is going to operate. All navigation commands sent by the command or GUI tasks are derived from the information in the map file.

As explained in [10], the map is a directed graph that consists of a set of nodes connected by directed edges. The nodes are defined as landmarks and the directed edges are either right walls or intersections. Two Landmarks, say A and B, are connected by a directed edge on the map only if the robot is capable to travel from A against the right wall or through an intersection to B. Disconnected landmarks on the map may be reached by control commands that do not require landmark identification. These commands are U-turn, move forward, or rotate.

Landmarks other than doors are defined in the database as a set of cues, which are themselves corners, planes, or hallways. The door landmark is defined by its type (i.e. right, left, or front door), and its width. The map file can be constructed only after defining all landmarks. The process for building a map file is detailed in [11].
The control task is responsible for parsing properly cues and landmarks defined in "landmark_definition", and comparing them with the ones identified while travelling indoors.

The "Landmark database setup" section details the procedure for adding new landmarks to the database, whereas the "Landmark identification" section describes the landmark identification process. The last two sections of this chapter focus on describing the control commands and the collision avoidance mechanism.

3.1 Landmark database setup

Landmarks definition is a process that is completed offline. An administrator is responsible for creating a file containing the definitions of all landmarks written in the format recognized by the landmark parser of the control task, and placing it in the AIDER's file system.

The steps for defining landmarks are:

1. Open a file called "landmark_definition" in any text editor
2. Add a new line in "landmark_definition" for each landmark
3. Save "landmark_definition" in a repository for further editing
4. Transfer "landmark_definition" to AIDeR's file system

There are two types of landmarks that can be defined in step 2: A door landmark, and a non-door landmark.

A master copy of "landmark_definition" should be saved in a secure location (step 3). This allows an administrator to repair a corrupted file on AIDeR's file system, and to expand the landmark database from the master copy.

Currently, "landmark_definition" is uploaded to AIDeR's "/home/lhilde" in step 4 since the control task executes from that directory. The user interface permits dynamic loading of the file during normal operation when the control task's current state is idle.

A door landmark has the following format:
`#landmark_ID: {door_type, width, 0}`
landmark_ID is a unique number between 1 and 65535.
door_type values are:

Door_type	**Definition**
1	Right
2	Left
3	Front

width values range from 1 to 3500 (in millimeter)
The third argument is ignored.

A non-door landmark has the following format:
`#landmark_ID:` $\{cue_1\}$, $\{cue_2\}$, …, $\{cue_{n-1}\}$, $\{cue_n\}$
Where 1 <= n <=10
landmark_ID is a unique number between 1 and 65535.
$\{\boldsymbol{cue_1}\}$, $\{\boldsymbol{cue_2}\}$, **…**, $\{\boldsymbol{cue_{n-1}}\}$, $\{\boldsymbol{cue_n}\}$ are any of the cues described below.

The generic format for cues is {cue_type, arg1, arg2}. There are three types of cues: plane, corner, and hallway.

A plane cue is defined by its cue type, length, and angle.
`{cue_type, length, angle}`

cue_type is set to 1 for planes
length values range from 1 to 2500 (in millimeter)
angle values range from 0 to π (in radian)

A corner cue is defined by its cue type and angle.
`{cue_type, angle, N/A}`

cue_type is set to 2 for corners
angle values range from 0 to π (in radian)
The third argument is ignored

A hallway cue is defined by its cue type, hallway type, and width.

```
{cue_type, type, width}
```

cue_type is set to 3 for hallways
type values are:

type	Definition
1	Right hallways
2	Left hallways
3	Front hallways

width values range from 1 to 3500 (in millimeter)

3.1.1 Example: Definition of the landmarks at CSUCI

The directed graph of the CSUCI Bell Tower building used during testing is shown in figure 3. It corresponds to the map defined in [10]. The directed graph contains 6 landmarks. Each landmark gets added to the database by adding one line in file landmark-definition. The following table shows the verbal definition of each landmark, and the corresponding line added to the file.

Landmark ID	Verbal Definition	Formatted File Definition
1	This is a non-door landmark having a right hallway with a width of 2.3 meters, and a corner having an angle of 1.57 radians.	#1: {3, 1, 2300}, {2, 1.57, 0}
2	This is a non-door landmark having a right hallway with a width of 2.8 meters, and a front hallway with a width of 2.3 meters.	#2: {3, 1, 2800}, {3, 3, 2300}
3	This is a right door landmark having a width of 1.65 meters.	3: {1, 1650, 0}
4	This is a non-door landmark having a right hallway with a width of 2.4 meters, and a front hallway with a width of 3.1 meters.	#4: {3, 1, 2400}, {3, 3, 3100}
5	This is a non-door landmark having a plane 1.3 meters long and angle of 1.54 radians, and a corner having an angle of 1.2 radians.	#5: {1, 1300, 1.54}, {2, 1.2, 0}
6	This is a non-door landmark having a right hallway with a width of 2.2 meters, and a left	#6: {3, 1, 2200}, {3, 2, 2100}

	hallway with a width of 2.1 meters.	

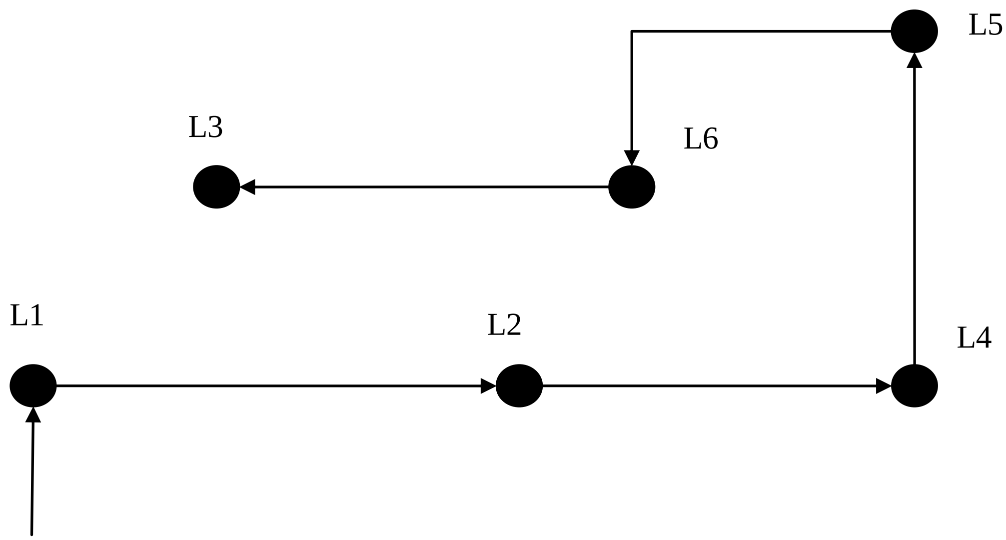


Figure 3 – Directed Graph of the test location in the CSUCI Bell Tower

Hence, the master copy of landmark_definition contains the following 6 entries:
#1: {3, 1, 2300}, {2, 1.57, 0}
#2: {3, 1, 2800}, {3, 3, 2300}
#3: {1, 1650, 0}
#4: {3, 1, 2400}, {3, 3, 3100}
#5: {1, 1300, 1.54}, {2, 1.2, 0}
#6: {3, 1, 2200}, {3, 2, 2100}

The process of defining landmarks is error prone due to the expectation of having users following an exact format consisting of symbols and numeral values. It is also not secure due to the fact that anybody can FTP his own version of landmark_definition file without any sort of user authentication. Ultimately, this process should evolve and landmarks editing should be done via GUI's admin account.

3.2 Landmark Identification

The landmarks are defined as described in the previous section. The process for identifying landmarks is detailed in this section.

Landmarks are identified by using the range-finder laser placed at the front of the robot. The laser has a maximum number of 768 beams, which covers an angular detection range of 240 degrees. In this application, the laser is set to scan over 180 degrees; however, the position of the range-finder laser has limited the useful scanning range to 144 degrees, since the wheels obscure the laser's view.

Landmark identification occurs after building the local map. The identification algorithm executes as follows:

```
1. Scan 180 degrees.
2. Detect the plane cues.
3. Detect the corner cues.
4. Detect the hallway cues.
5. Detect door landmarks.
6. Identify landmarks.
```

In the first step, the raw data of the laser beams is stored in a table as points having angular coordinates. This data is used to extract the geometrical features of the plane cues, corner cues, hallway cues, and door landmarks in steps 2 to 5.

The last step, identify landmarks, opens the file "landmark_definition" and compares the defined cues of each non-door landmark to the cues detected by the range-finder laser (i.e. step 2-4), or it compares the defined door landmark with the doors detected by the range-finder laser (i.e. step 5). The following subsections detail the identification algorithm of plane cues, corner cues, and hallway cues, as well as the identification of door landmarks.

It should be noted that the identification algorithm may fail when dynamic obstacles reside near landmarks. For example, at our CSUCI test lab, landmark 1 is defined as a right hallway and a corner having an angle of 1.57 radians; a person or an object (e.g. a trash can) placed at that corner will most certainly prevent the algorithm from recognizing the corner cue at step 3, and eventually, the landmark at step 6.
In this research, there is really no provision for guaranteeing the identification of corner cues, hallway cues and door landmarks since their features are derived from plane cues. The latter is broken up in separate pieces when a dynamic obstacle reside in its midst. Ultimately, a study dedicated to extracting cues would improve AIDeR's proper behavior in its working environment.

3.2.1 Plane Cue

Planes are characterized by a start point, an end point, length, an index to the shortest beam, and slope (see structure below).

```
typedef struct point
{
   double    x;
   double    y;
} POINT;

typedef struct plane
{
   POINT     start_pnt;
```

```
    POINT     end_pnt;
    double    angle;
    uint16    shortest_beam_index;
    double    length;
  } PLANE;
```

start_pnt and **end_pnt** are the points at the end of the first and last beams belonging to a plane. They are translated in Cartesian coordinates.
angle is computed in radian. Its value is π/2 when AIDeR is paralleled to the wall, which is the angle between beam 0 and the wall. The process for computing plane angles is detailed in section "Setting the angle of a plane".
shortest_beam_index points to the shortest beam of the plane.
length is the distance between start and end points in millimeter.

The algorithm setting the members of the plane structure is shown below:

1. Compute the AD parameter using $1 + 2\sum_{j=1}^{k}\cos(j\Delta)$.
2. Set the start point of the plane.
3. Set the end point of the plane.
4. Iterate through all points.
    a. Compute the measured AD of the next point.
    b. Add the point to the plane if it is part of it.
5. Set length as the distance between start and end points.
6. Set the shortest beam index.
7. Set the angle of the plane.

The start and end points of a plane are translated in Cartesian coordinates. Steps 1, 4a, and 4b are detailed in section "Adding points to a plane". The distance of a plane is in millimeter. Step 7 is detailed in section "Setting the angle of a plane".

3.2.1.1 Adding points to a plane

The implementation of the algorithm determining the attachment of a point to a plane uses the material presented in [2]. In this section, the author has accepted the validity of the method, and is only concerned about showing its implementation. The naming convention such as the AD parameter presented in [2] has not been changed.
Figure 4 shows the situation where seven laser beams hit a wall during a laser scan.

AD is a parameter invariant to the equation of the line segments. It is used to identify potential data points that belong to a straight line segment (i.e. plane).

d is the distance of each laser beam between the origin and p.
Δ is the laser's step size. It is hardware dependent and fixed to 0.00614 rad.

k is the size of the window. It is set to 4 because this value gave the best results during tuning.

For k=4, the measured AD is defined as

$$measured\ AD = \frac{\left(\frac{1}{d_{i-3}} + \frac{1}{d_{i-2}} + \frac{1}{d_{i-1}} + \frac{1}{d_i} + \frac{1}{d_{i+1}} + \frac{1}{d_{i+2}} + \frac{1}{d_{i+3}}\right)}{\frac{1}{d_i}}$$

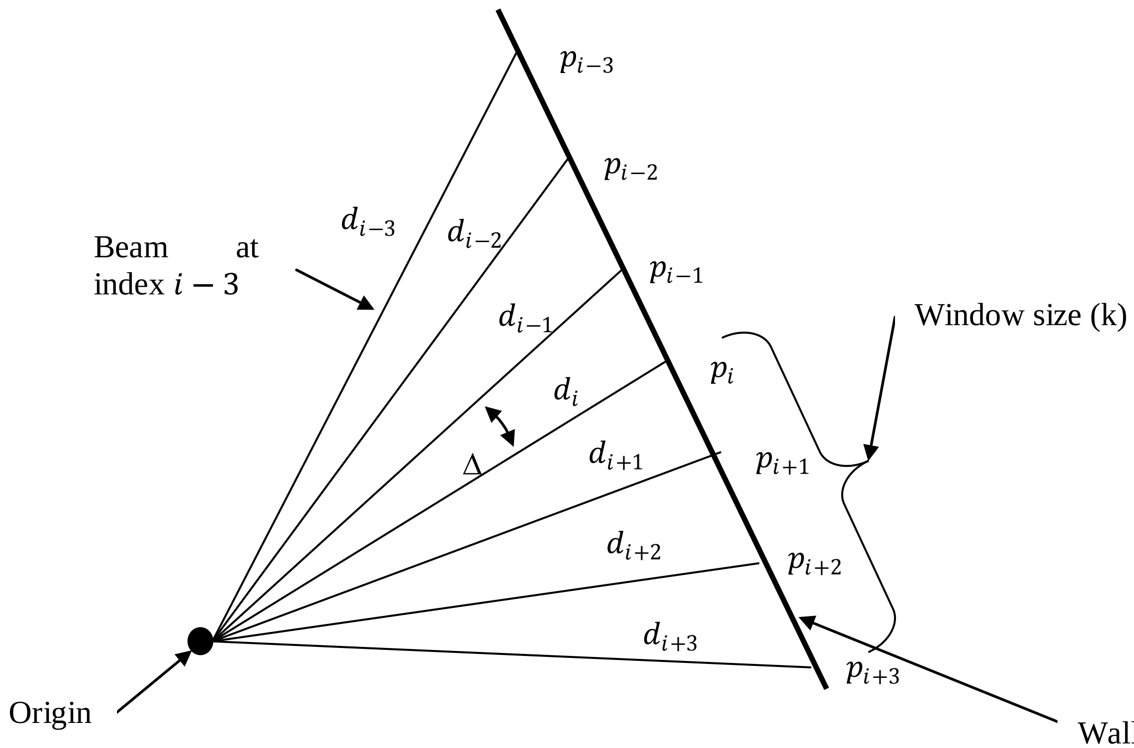


Figure 4 –Laser beams hitting a wall during a scan

The calculated AD is defined as
$Calculated\ AD = 1 + 2\sum_{j=1}^{k}\cos(j\Delta) =$ 8.998869, for k = 4 and Δ = 0.00614.

A point p_i is added to a plane when | measured AD - calculated AD | <= PLANE_SLOPE_PRECISION. During testing the best results were obtained when constant PLANE_SLOPE_PRECISION was set 0.07 in the configuration file tuning_parameters; which means that the highest percentage of correct plane detection occurred for that constant value. PLANE_SLOPE_PRECISION represents the maximum error for deciding whether or not the point being tested belongs to the same plane or the next one.

Points having invalid beam distances are automatically rejected by the plane cue identification algorithm. Also, planes having a length less than 10 centimeters are not valid.

3.2.1.2 Setting the angle of a plane

The algorithm for setting the angle of a plane is shown below:

1. Set $v0$ as the unit vector of beam 0, $v0 = < 1, 0 >$
2. Set $v1$ as the vector from the start point to the end point of the detected plane, $v1 = < pe.x - ps.x, pe.y - ps.y >$
3. Compute the angle $\theta = \cos^{-1}(v0.v1/|v0||v1|)$ where $|v0| = \sqrt{v0.v0}$ and $|v1| = \sqrt{v1.v1}$

Figure 5 shows the angle θ of the plane.

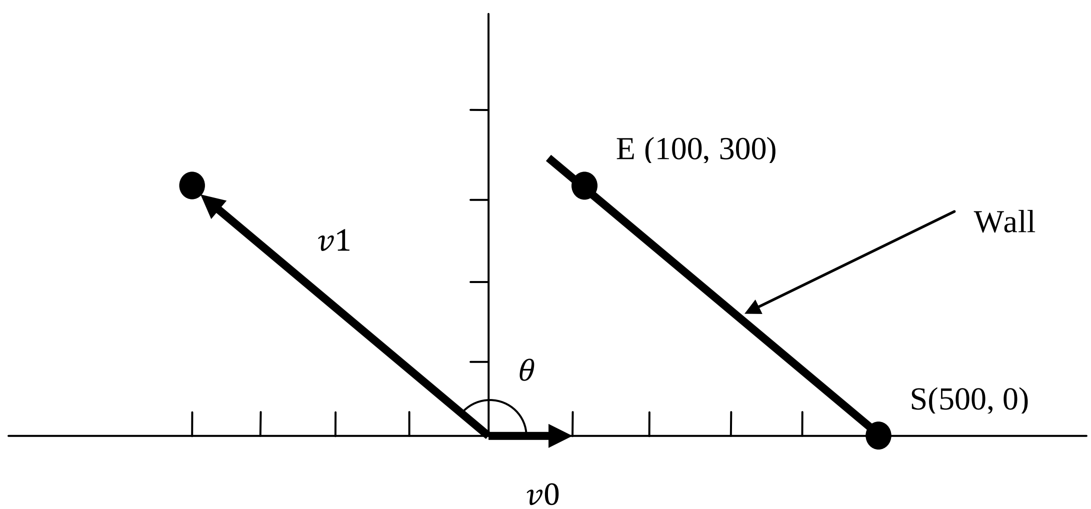


Figure 5 – Representation of θ in the vector space

3.2.2 Corner Cue

Corners are derived from the detected planes. They are characterized by a plane intersect point, an angle, and 2 intersecting planes as shown in the structure below.

```
typedef struct corner
{
	PLANE		plane1;
	PLANE		plane2;
	double		angle;
	POINT		intersect_pnt;
} CORNER;
```

plane1 and **plane2** are the adjacent planes extracted during the “Detect plane cues” step of the identification algorithm.
angle is computed in radian.
intersect_pnt is the intersection point between plane1 and plane 2.

Figure 6 shows the geometry involved in the corner detection algorithm.

The algorithm setting the members of the corner structure is:

1. Set plane1 and plane2 with the values of the calculated planes.
2. Interpolate the intersect point from the 2 planes.
3. Verify that the intersect point falls within valid range for a corner-type cue.
4. Set the values of the calculated intersection point.
5. Compute distances d_1 and d_2.
6. Compute distance d.
7. Set angle to the computed value

$$\theta = \cos^{-1}\left\langle (d^2 - d_1^2 - d_2^2) \middle| \left((-2 * d_1 * d_2)\right)\right\rangle.$$

3.2.2.1 Computing the intersection point value between two planes

The technique presented in [7] for computing the intersection point is also used by the algorithms detecting hallway cues and door landmarks

The algorithm computes the intersection point as follows:

1. Set $P_1(x_1, y_1) = start\ pnt\ of\ plane\ 1$
2. Set $P_2(x_2, y_2) = end\ pnt\ of\ plane\ 1$
3. Set $P_3(x_3, y_3) = start\ pnt\ of\ plane\ 2$
4. Set $P_4(x_4, y_4) = end\ pnt\ of\ plane\ 2$
5. Compute $u_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)}$
6. Compute x = $x_1 + u_a\ (x_2 - x_1)$ and y = $y_1 + u_a\ (y_2 - y_1)$

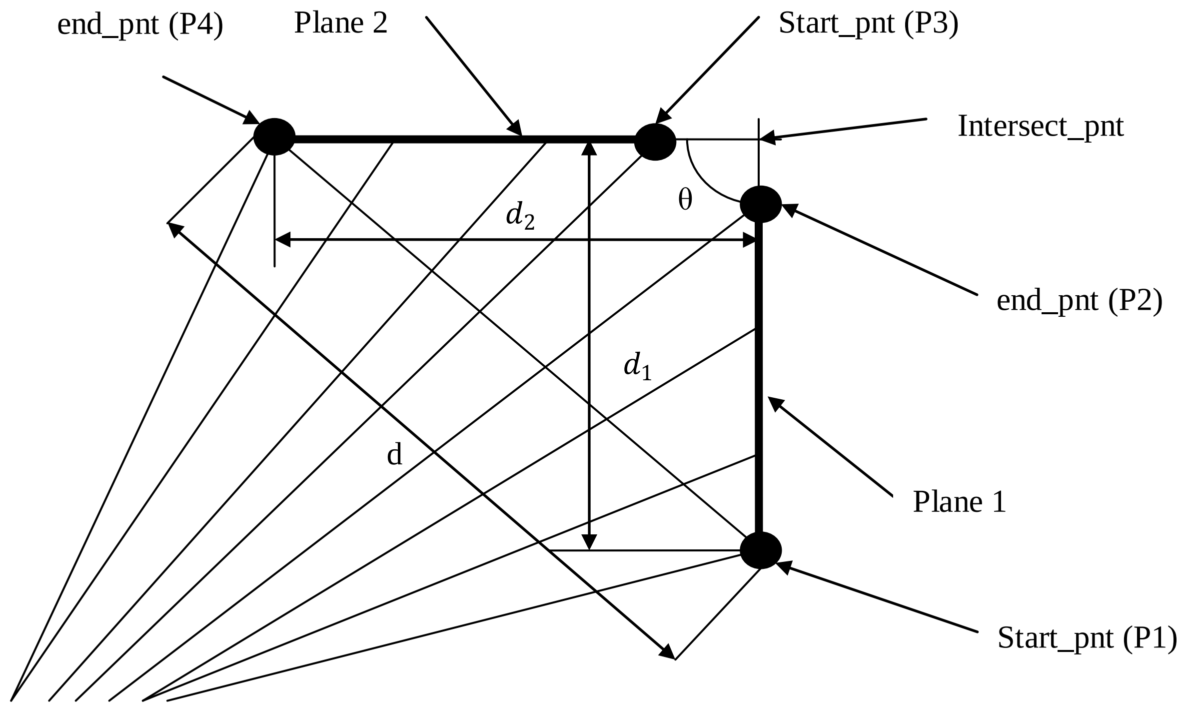


Figure 6 - Corner scan

3.2.3 Hallway Cue

Hallways are characterized by 2 planes separated by a distance. The type variable specifies whether the hallway is located to the right, left, or front.

```
typedef struct hallway_cue
{
        PLANE       plane1;
        PLANE       plane2;
        double      distance;
        int         type;
} HALLWAY_CUE;
```

plane1 and **plane2** are the adjacent planes extracted during the "Detect plane cues" step of the identification algorithm.
distance is measure in millimeter.
type is set to right, left, or front.

The algorithm setting the members of the hallway structure is:

1. Set plane1 and plane2 with the values of the calculated planes.
2. Interpolate the intersect point from the 2 planes.
3. Set hallway type

Step 2 is described in "Computing the intersection point value between two planes" section. Step 3 is described in the subsequent sections.

3.2.3.1 Right Hallway Cue

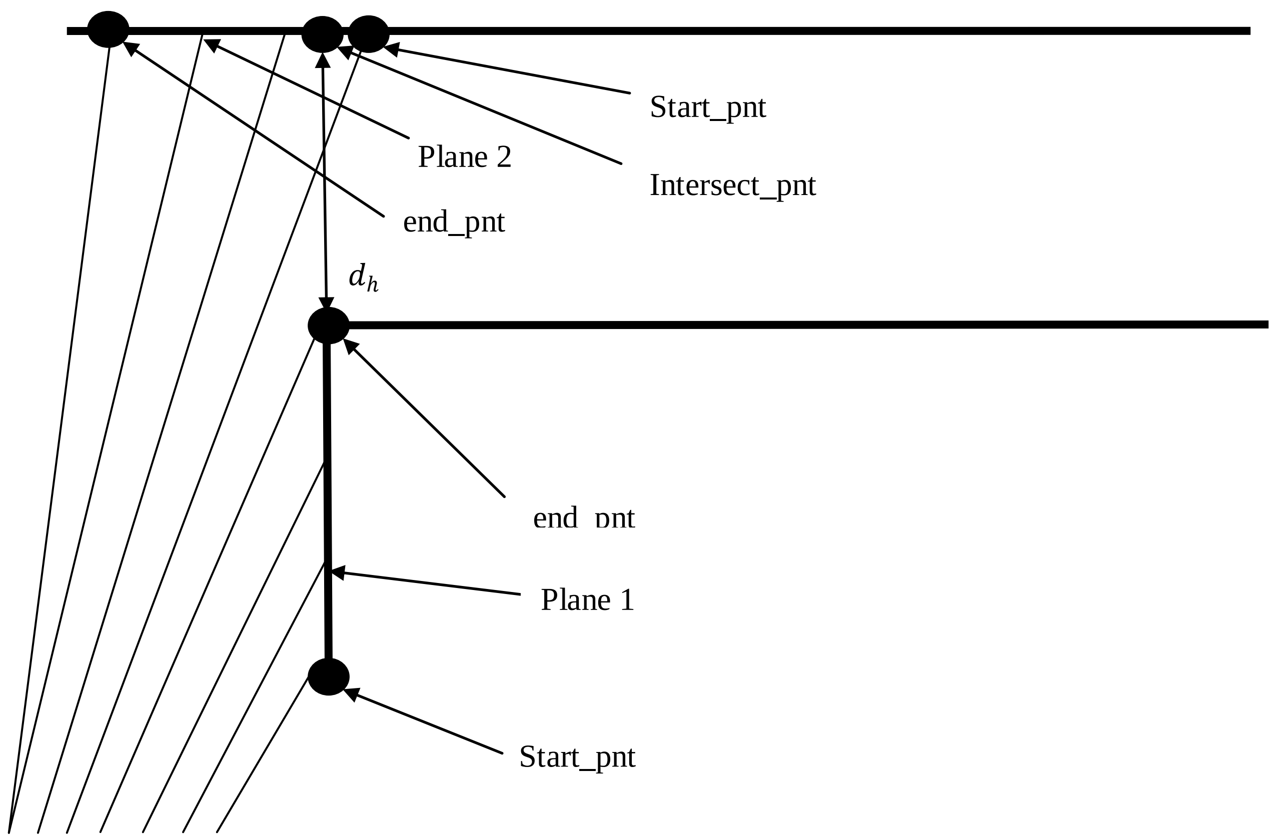


Figure 7 – Right hallway scan

The algorithm detects right hallways as follows:

1. Verify that plane1's angle is 90 degrees and plane 2's angle is 180 degrees.
2. Verify that distance between plane 1's end point and the intersect point is within valid hallway distance.

3.2.3.2 Left Hallway Cue

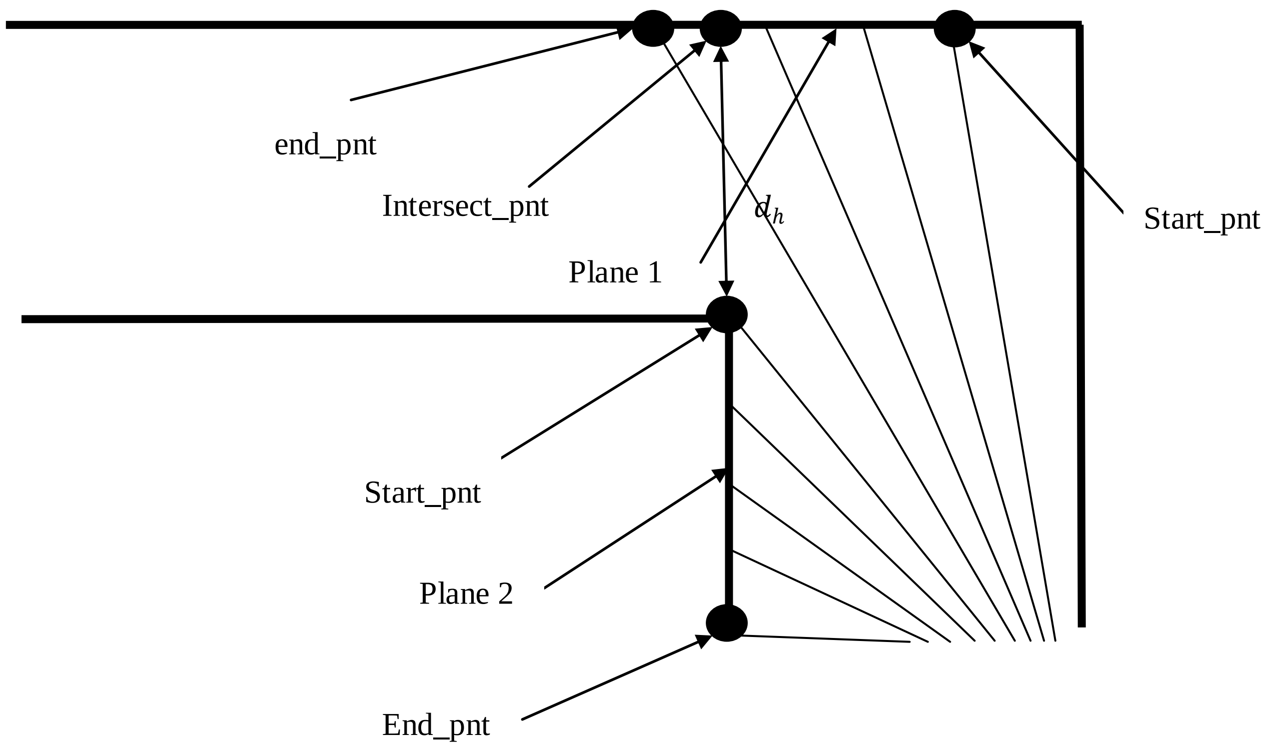


Figure 8 - Left hallway scan

The algorithm detects left hallways as follows:

1. Verify that plane1's angle is 180 degrees and plane 2's angle is 90 degrees.
2. Verify that distance between plane 2's start point and the intersect point is within valid hallway distance.

3.2.3.3 Front Hallway Cue Type 1

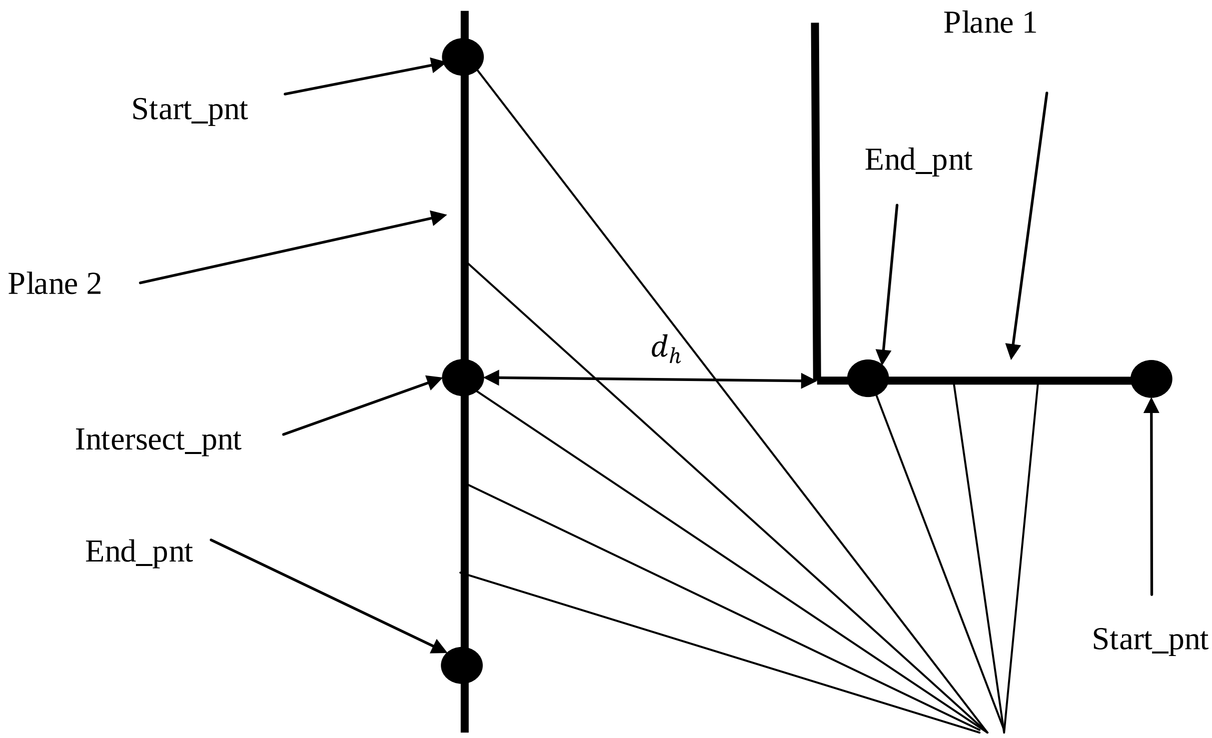


Figure 9 - Front hallway scan

The algorithm detects type 1 front hallways as follows:

1. Verify that plane1's angle is 180 degrees and plane 2's angle is 90 degrees.
2. Verify that distance between plane 1's end point and the intersect point is within valid hallway distance.

3.2.3.4 Front Hallway Cue Type 2

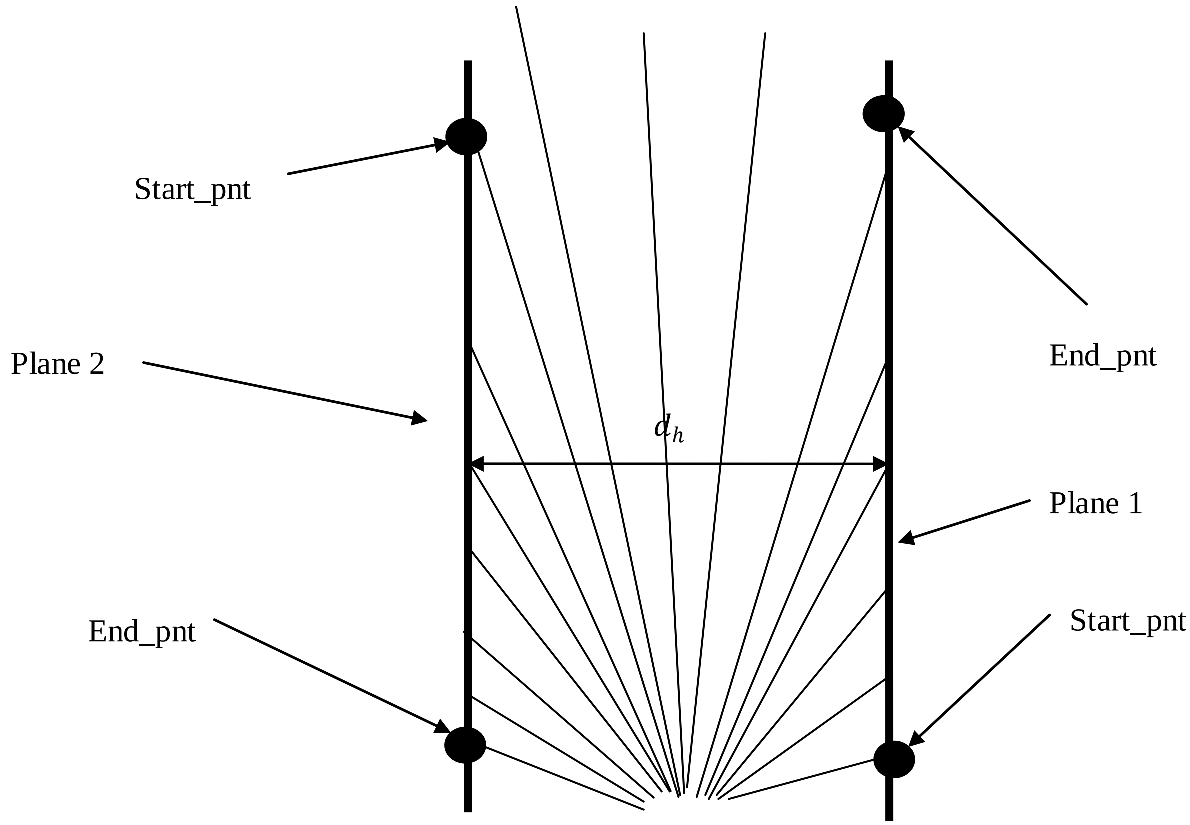


Figure 10 - Front hallway scan

The algorithm detects type 2 front hallways as follows:

1. Verify that plane1's angle is 90 degrees and plane 2's angle is 90 degrees.
2. Verify that distance between plane 1 and plane 2 x coordinate values of their start points is within valid hallway distance.

3.2.4 Door Landmark

Door landmarks are characterized by 2 planes separated by a distance. The type variable specifies whether the door is located to the right, left, or front, and the status variable specifies whether the door is opened or closed.

```
typedef struct door_landmark
{
	PLANE_CUE plane1;
	PLANE_CUE plane2;
	double    distance;
	int       status;
```

```
    int         type;
} DOOR_LANDMARK;
```

plane1 and **plane2** are the adjacent planes extracted during the "Detect plane cues" step of the identification algorithm.
distance is measured in millimeter.
status is opened or closed.
type is set to right, left, or front.

The algorithm setting the members of the door structure is:

1. Set plane1 and plane2 with the values of the calculated planes.
2. Interpolate the intersect point from the 2 planes.
3. Set the door type
4. Set the door status based on the depth

It should be noted that doors could be confused for hallways if their respective widths were nearly identical in the robot's working environment. In such case, the difference between d_h(Figures 7, 8, 9, and 10) and d_d (Figures 11, 12, and 13) would approach 0. As a consequence, hallway cues and door landmarks would both include doors and hallways. This is not the case in the Bell Tower Building at CSUCI since doors and hallways have respective minimum widths of 1.6 and 2.3 meters.

The following sub-sections depict the various types of doors.

3.2.4.1 Right Door Landmark

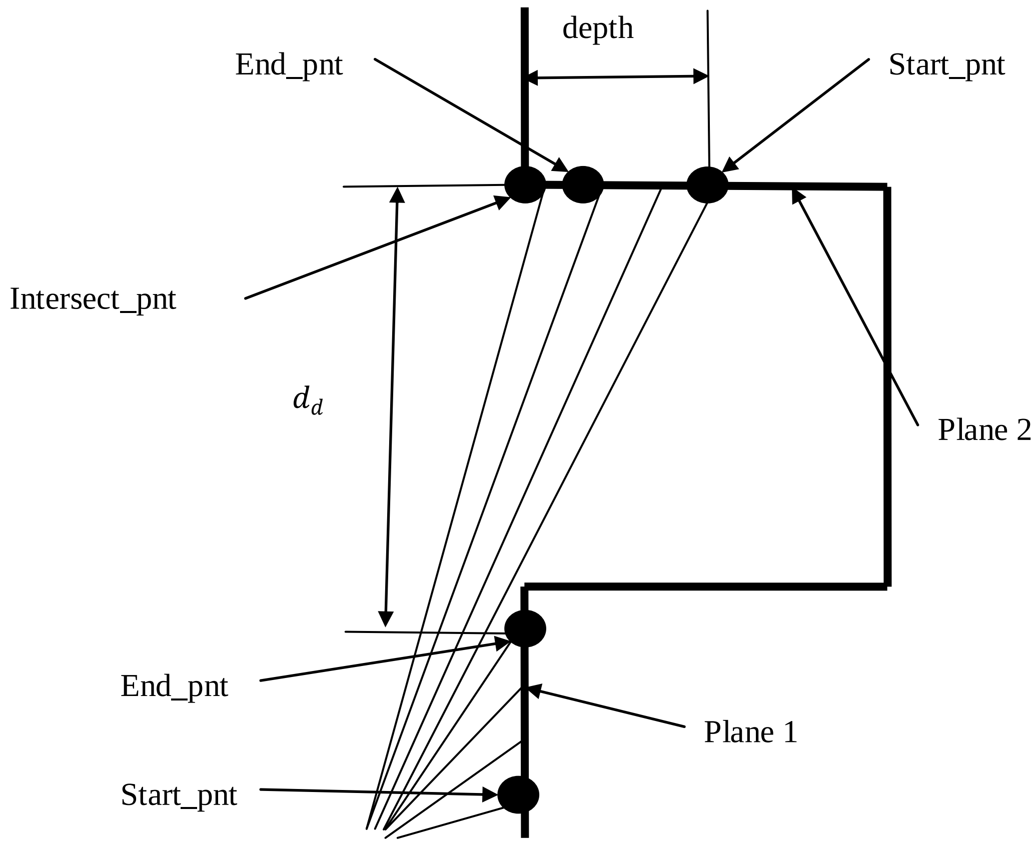


Figure 11 - Right closed door scan

The algorithm recognizing the right door landmark is:

1. Verify that plane1’s angle is 90 degrees and plane 2’s angle is 180 degrees.
2. Verify that distance between plane 1’s end point and the intersect point is within valid door distance.

3.2.4.2 Left Door Landmark

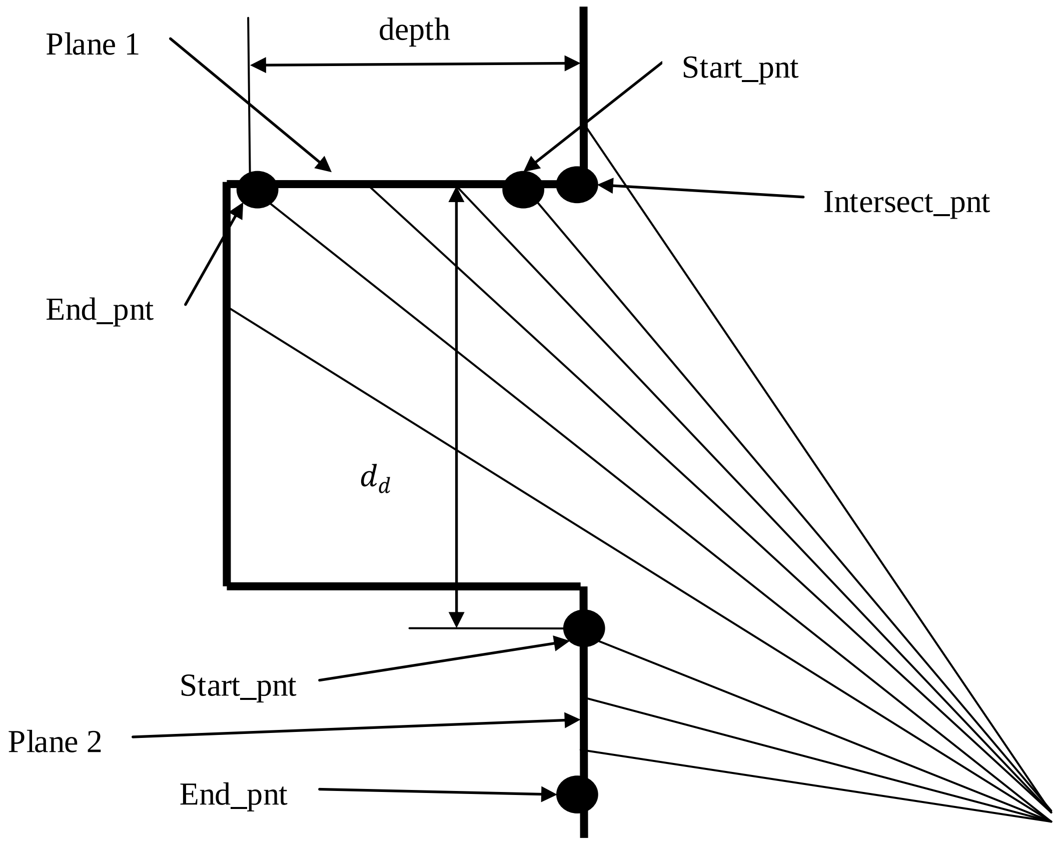


Figure 12 - Left closed door scan

The algorithm recognizing the left door landmark is:

1. Verify that plane1's angle is 180 degrees and plane 2's angle is 90 degrees.
2. Verify that distance between plane 2'start point and the intersect point is within valid door distance.

3.2.4.3 Front Door Landmark

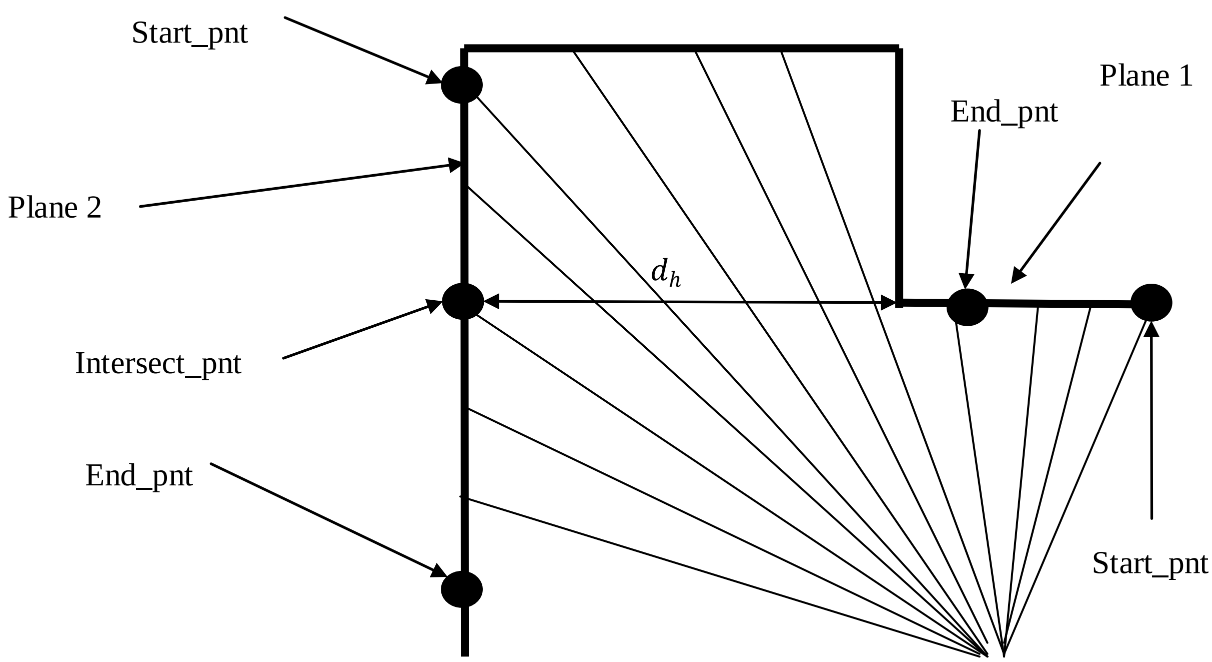


Figure 13 – Front closed door scan

The algorithm recognizing the front door landmark is:

1. Verify that plane1's angle is 180 degrees and plane 2's angle is 90 degrees.
2. Verify that distance between plane 1'end point and the intersect point is within valid door distance.

3.3 Collision Avoidance

Collisions are avoided by detecting objects entering the unsafe perimeter of the robot. This perimeter sensing mechanism is accomplished by using six devices, the range-finder laser and five sensors labeled front-left, front-right, rear-left, rear-right, and back. The algorithm polls the range-finder laser and sensing devices sequentially and compares the distances of each beam with a pre-defined unsafe distance (set at 40 cm in tuning_parameters). If a measured distance is less than the unsafe distance, the robot stops and reports an obstacle detected alarm.

3.4 Control Commands

The control task supports the following commands:

- Move tray
- Move forward
- Rotate
- Stop
- Travel along the wall
- Enter right hallway
- Enter left hallway
- Enter front hallway
- Make U-turn

The control commands are received via navigation plan messages (detailed in the inter task message section), which are sent by either the command task or the GUI task.
The control task processes the control commands associated with the GUI task first, and then those associated with the command task second. This priority scheme allows users to take immediate control of the robot via the local interface (i.e. the touch screen). The control task processes multiple control messages sent by the same task in the First In First Out manner (FIFO).
The control commands are divided in 2 groups: group 1 contains commands requiring identification of the cues and landmarks, and group 2 contains commands that do not need to build the local map of the environment.

Group 1	**Group 2**
Travel along the wall Enter right hallway Enter left hallway Enter front hallway	Move tray Move forward Rotate Stop Make U-turn

The algorithm handling control commands is

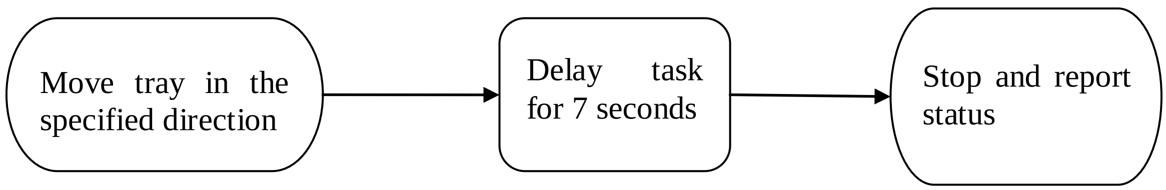
1. Update the current state
2. Execute the specified control command
3. Send any error that may have occurred during execution
4. Update the current state

In steps 1 and 4, the busy and idle states are respectively reported to the task that originated the request. Any alarm condition is reported to the sending task in step 3. Step 2 is described in the subsequent sections. Additional details about the current state and error conditions are available in the alarm and event section.

3.4.1 Move tray

This command raises or lowers the tray.

3.4.1.1 State Diagram



3.4.1.2 Additional Details

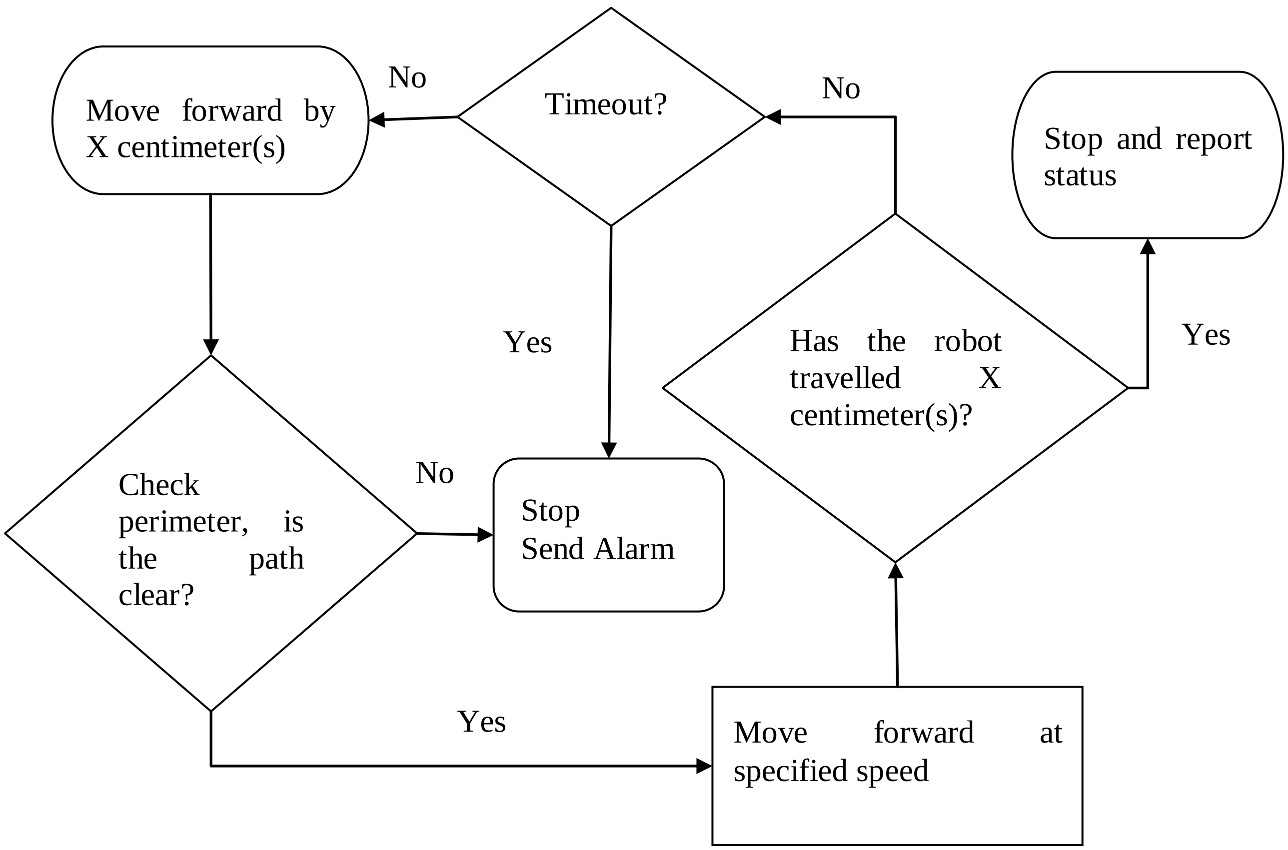
The tray moves up or down by about 15 inches. It is controlled by a servo drive with no encoder attached to it. As such, the control task issues a move tray command to the servo, then gets delayed for 7 seconds, and then issues a stop tray command to the servo for moving the tray to its intended position.

The output power of the servo drive gets killed when the tray reaches either switch placed at each end of the tray's intended final position.

3.4.2 Move forward

This command moves the robot forward by X centimeters at set speed. It has to complete execution within a specified timeout period.

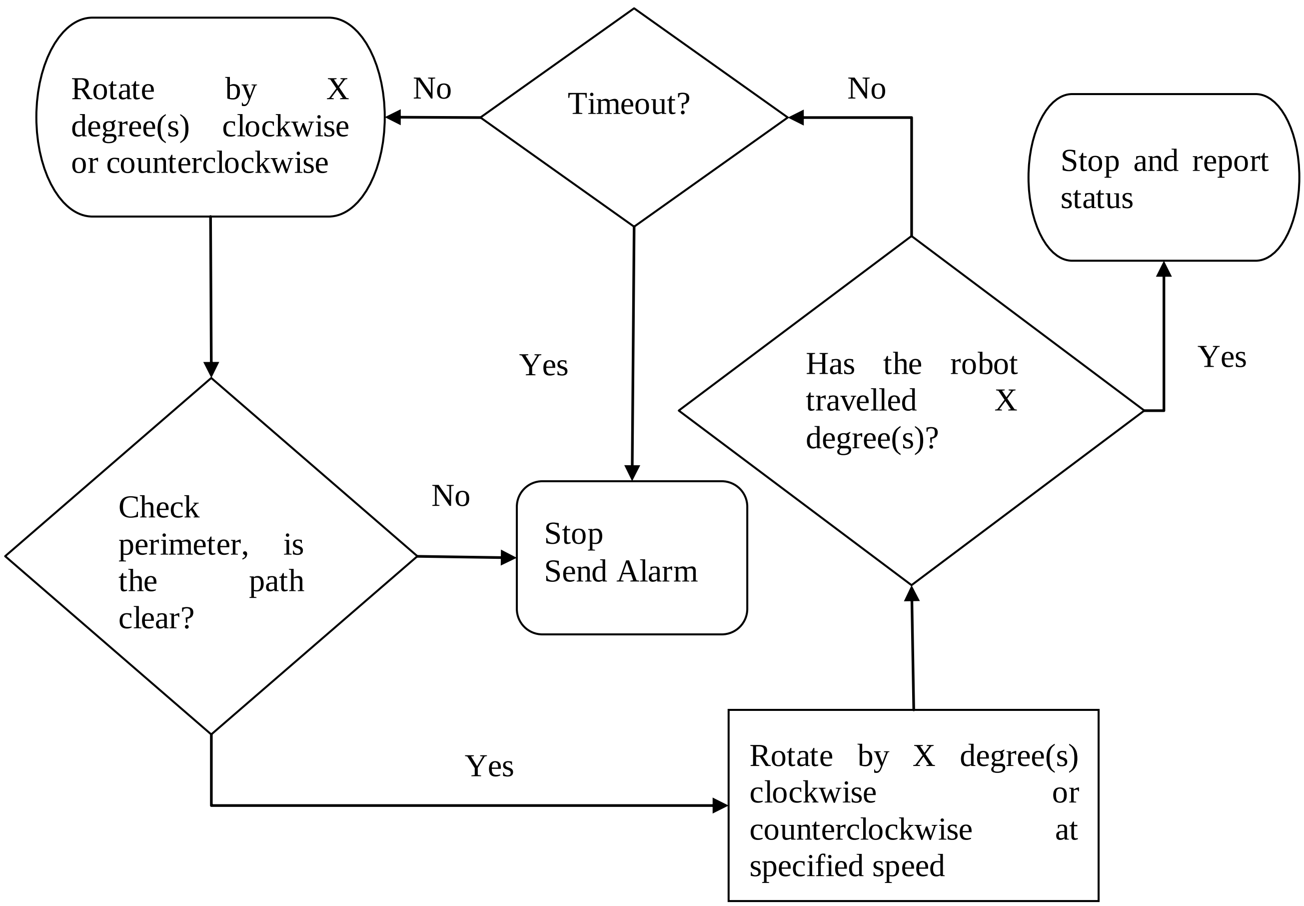
3.4.2.1 State Diagram



3.4.3 Rotate

This command rotates the robot by X degrees at a predefined rotate speed, clockwise or counter clockwise. It has to complete execution within a specified timeout period.

3.4.3.1 State Diagram



3.4.3.2 Additional Details

AIDER rotates by spinning about its central axis either clockwise or counterclockwise. Figure 14 shows the trajectory of the back right and left wheels during rotation.

By observation, the right and left wheels have identical trajectories. Hence, the velocity of the right and left wheels is

$$\frac{vr}{vl} = 1$$

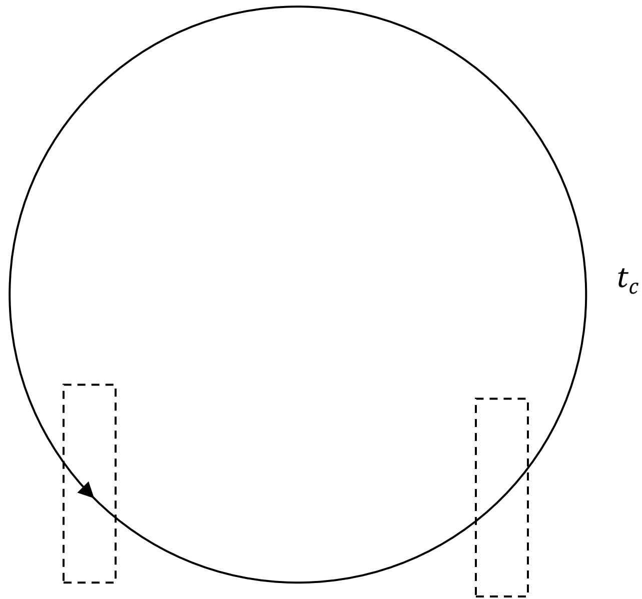


Figure 14 - Trajectory of right and left wheels during rotation

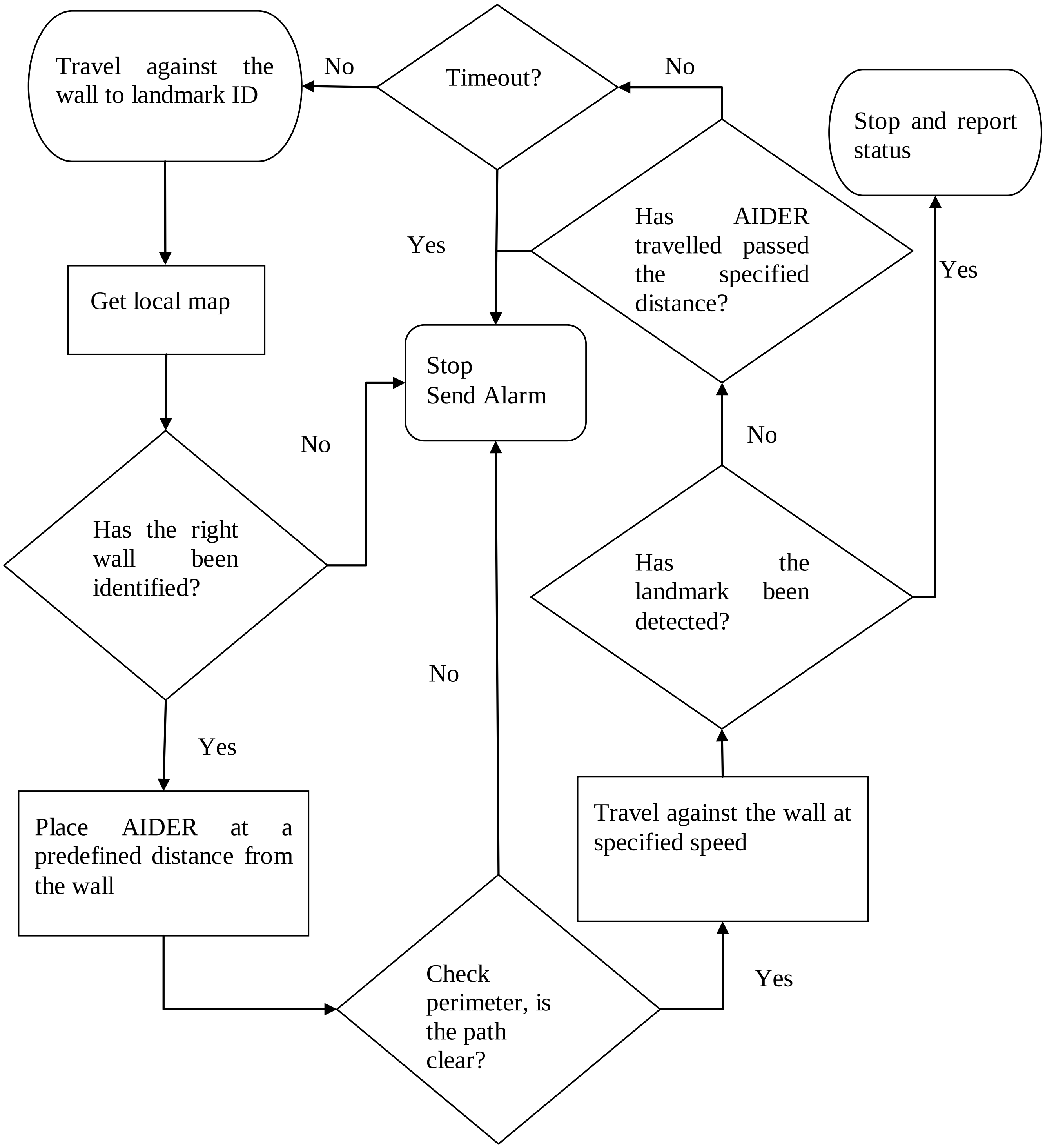
3.4.4 Stop

This command stops the robot. It translates directly to a command issued to the servo drive via the serial interface.

3.4.5 Travel along the wall

This command moves the robot alongside the right wall until the specified landmark is identified or AIDER has travelled by a maximum of X centimeters. It has to complete execution within a specified timeout period.

3.4.5.1 State Diagram



3.4.5.2 Additional details

The travel against the wall control command places the robot against the wall at the beginning of its execution. It also does it during its execution when the robot has deviated too far from the wall or too close to it.

The algorithm placing the robot against the wall is shown below:

1. Build the local map and identify the right wall
2. If AIDER is placed at the predetermined distance from the wall and is parallel to it then we are done; otherwise continue to step 3
3. Place the robot against the wall

Step 3 executes either case:

- The robot is far from the wall so it has to move toward it, and then parallels itself. This is labeled as the place from far algorithm.
- The robot is close to the wall so it has to move away from it, and then parallels itself. This is labeled the place from close algorithm.

The place from far algorithm is:

1. Rotate clockwise by the angle of approach
2. Move forward toward the wall until the minimum wall threshold distance is reached
3. Rotate until AIDeR is paralleled to the right wall

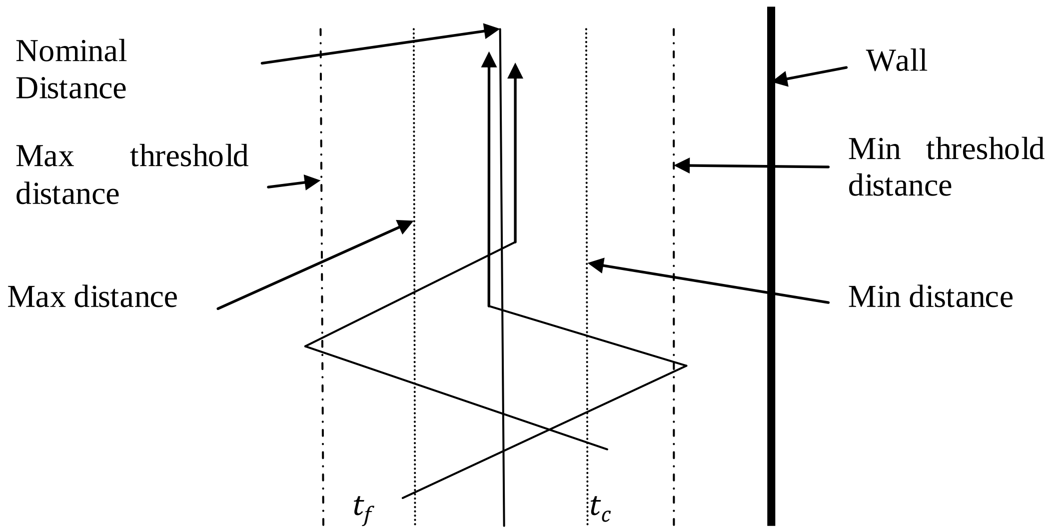
The place from close algorithm is:

1. Rotate counter clockwise by the angle of approach
2. Move forward away from the wall until the maximum wall threshold distance is reached
3. Rotate until AIDeR is paralleled to the right wall

The angle of precision and the five wall distances used by the wall placement algorithm are part of the configuration file. The wall distances are:

- The minimum threshold distance to the wall
- The minimum distance to the wall
- The laser distance
- The maximum distance to the wall
- The maximum threshold distance to the wall

The placing against the wall algorithm is depicted in Figure 15. It executes when AIDeR's distance from the wall is less than Min distance or greater than Max distance. Note that due to the placement of the range-finder laser, AIDeR is placed at wall distance when rotating parallel to the wall from either the maximum threshold or minimum threshold distances in step 3 of the place from far or close algorithms.



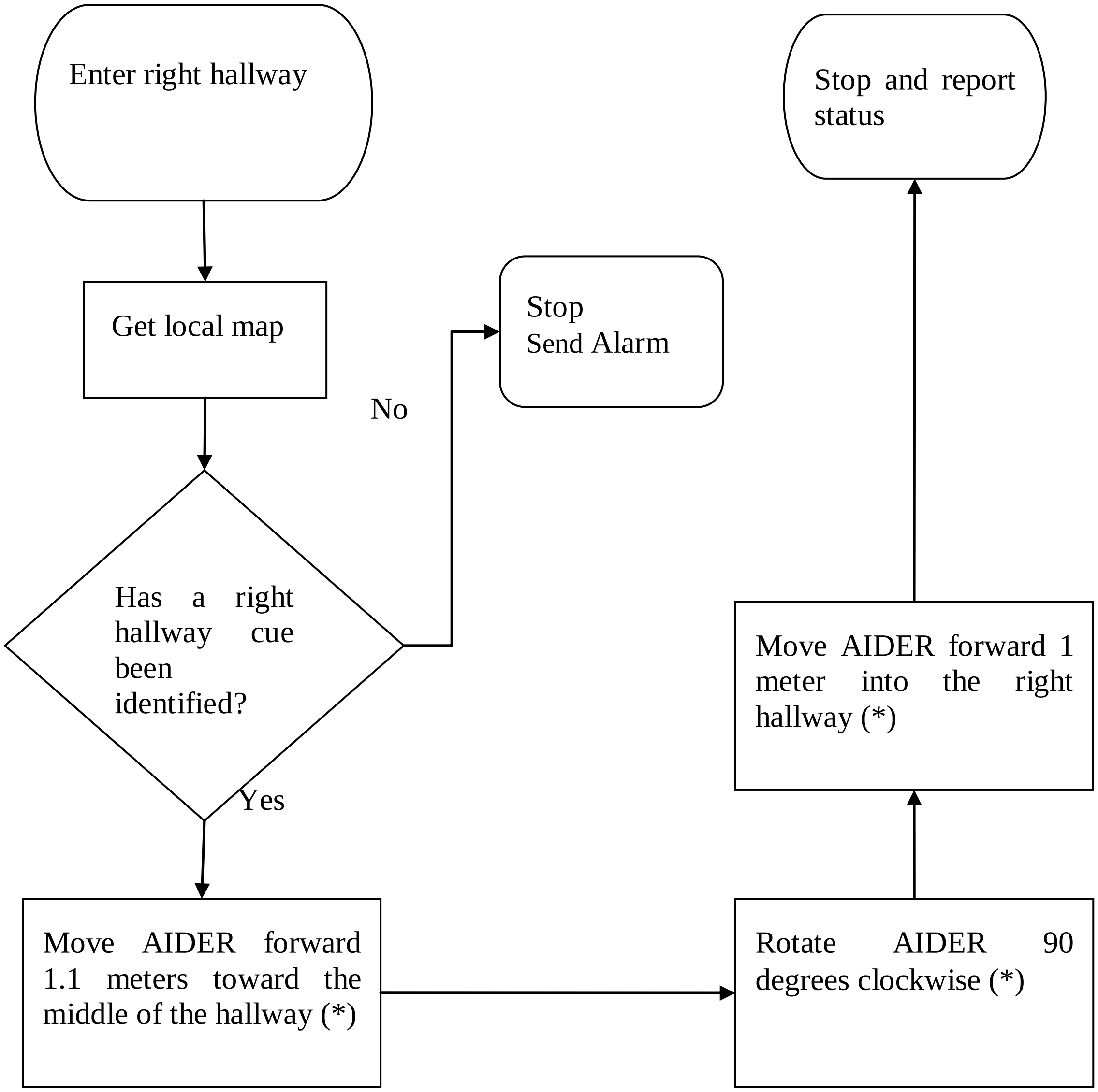
t_f: AIDeR's trajectory starting far from the wall.
t_c: AIDeR's trajectory starting close to the wall.

Figure 15 – Placing AIDeR against the wall

3.4.6 Enter right hallway

This command moves the robot into a right hallway. It has to complete execution within a specified timeout period.

3.4.6.1 State Diagram

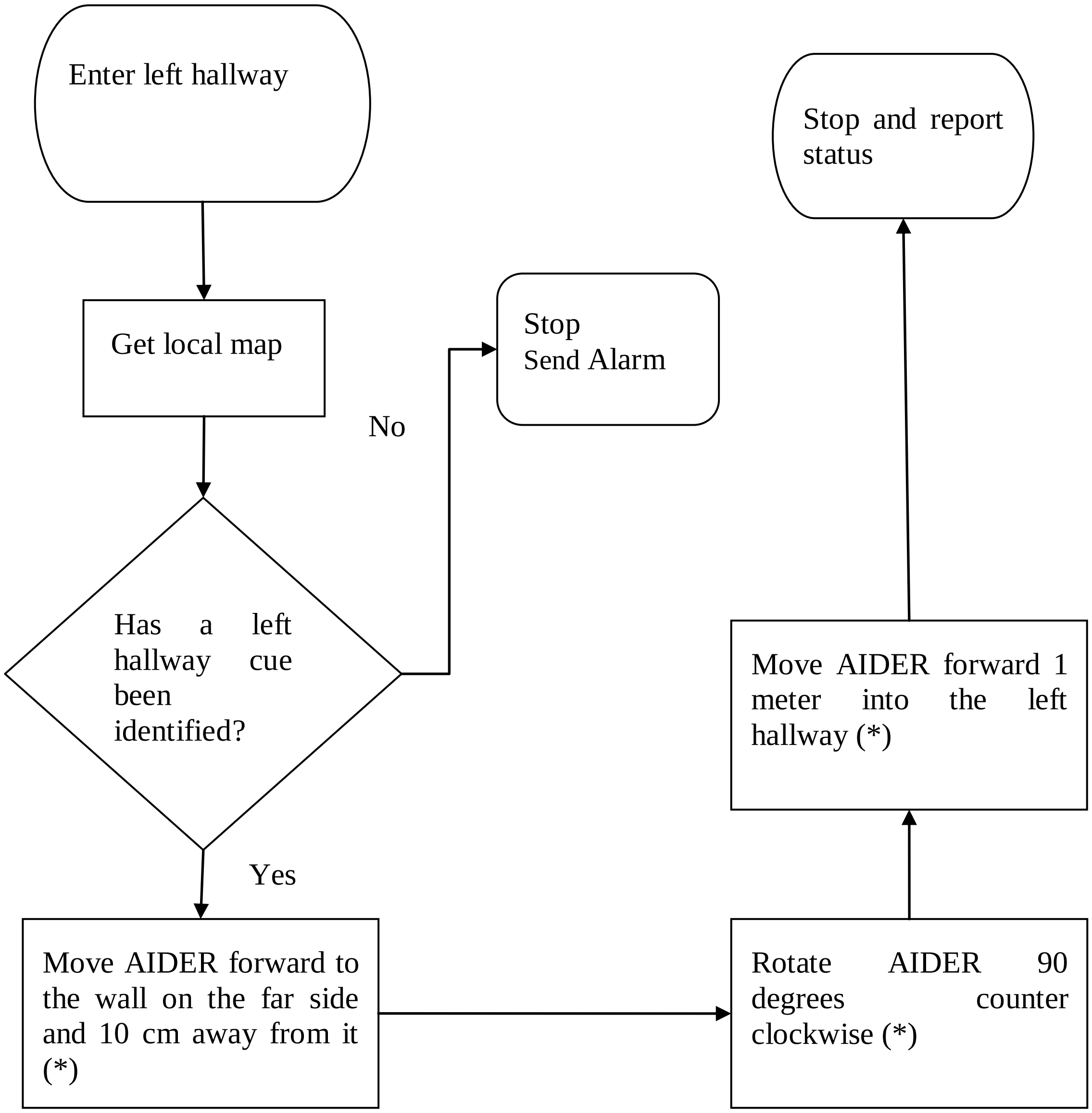


(*) Perimeter detection, travelled distance, and timeout checking is done during execution of the move forward and rotate commands (see state diagram of these commands).

3.4.7 Enter left hallway

This command moves the robot into a left hallway. It has to complete execution within a specified timeout period.

3.4.7.1 State Diagram

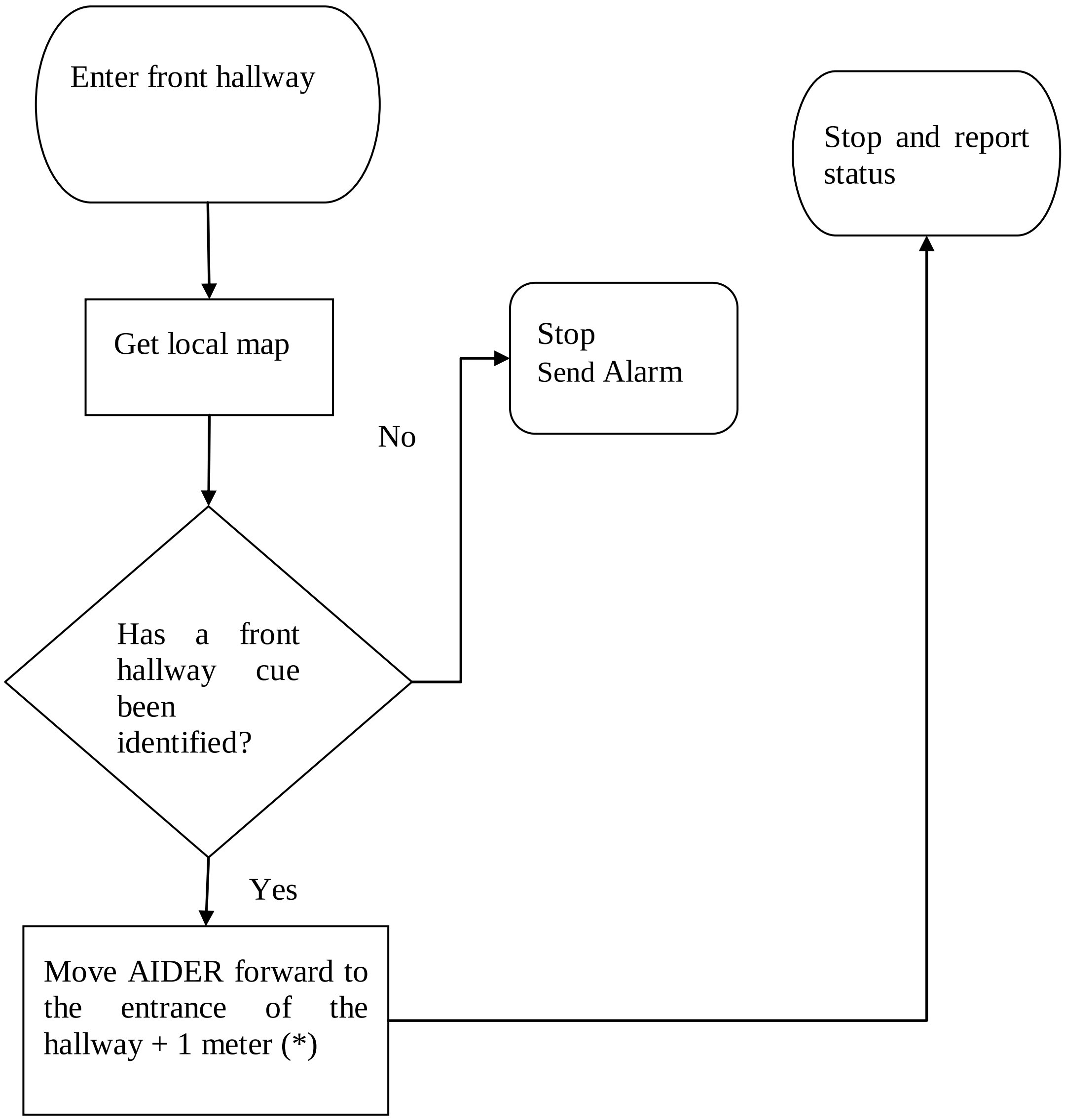


(*) Perimeter detection, travelled distance, and timeout checking is done during execution of the move forward and rotate commands (see state diagram of these commands).

3.4.8 Enter front hallway

This command moves the robot into a front hallway. It has to complete execution within a specified timeout period.

3.4.8.1 State Diagram

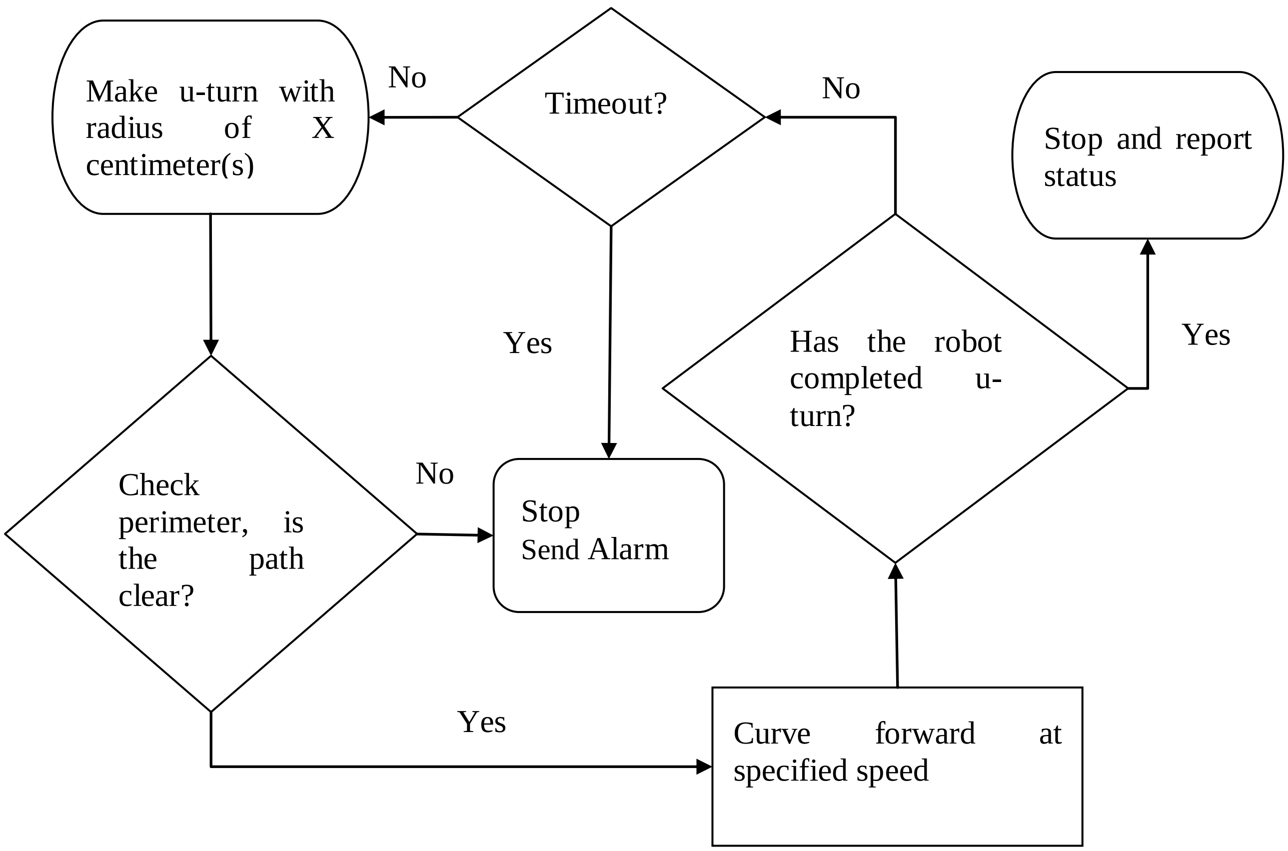


(*) Perimeter detection, travelled distance, and timeout checking is done during execution of the move forward and rotate commands (see state diagram of these commands).

3.4.9 Make U-turn

This command makes a left u-turn with specified radius and speed. It has to complete execution within a specified timeout period.

3.4.9.1 State Diagram



3.4.9.2 Additional Details

Consider the left U turn shown in Figure 16.

Let t be the curved trajectory about the robot's central axis.
Let t_r be the curved trajectory about the robot's right wheel axis.
Let t_l be the curved trajectory about the robot's left wheel axis.
Let R_{tr} be the radius of t_r
Let R_{tl} be the radius of t_l
Let R_t be the radius of t

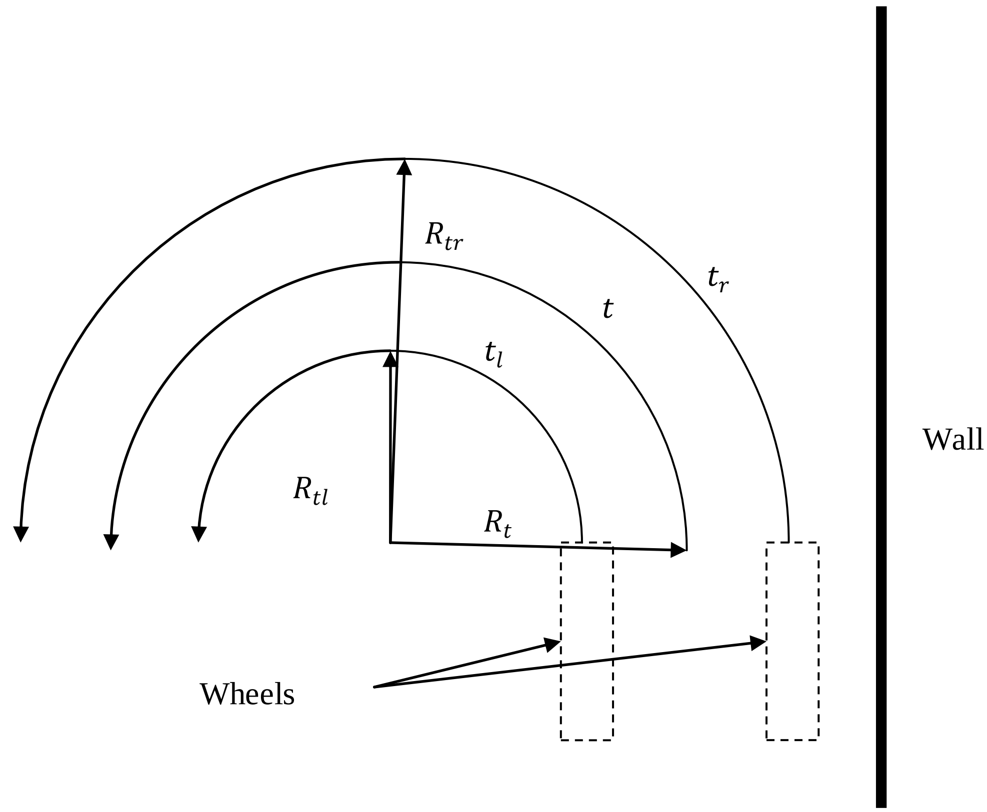


Figure 16 - Trajectories of left and right wheels when making a U-turn

Let R_t be set by the user and $R_{tl} = 0.5R_t$ and $R_{tr} = 2R_t$
Let the angular distance of the trajectory equal to π

The distance of the curved trajectories tr and tl in centimeter is calculated as follows,
$t_r = \pi * R_{tr}$ and $t_l = \pi * R_{tl}$, which results
$t_r = \pi * 2R_t$ and $t_l = \pi * 0.5R_t$

The velocities of the right and left wheels v_r and v_l is

$$\frac{v_r}{v_l} = \frac{t_r}{t_l} = \frac{\pi * 2R_t}{\pi * 0.5R_t} = 4$$

$$v_r = 4\, v_l$$

As indicated in the equation above, the velocity of the right wheels is set 4 times higher than the velocity of the left wheels in order to make a left u-turn.

Note that we are only implementing the left U turn case because the U turn command is issued when AIDER is placed against the right wall.

4 Implementation details of the work

4.1 Inter-Task Communication

Communication between the control and command-GUI tasks is accomplished via a message passing mechanism. In AIDeR, inter-task communication has been implemented using mqueue. The following subsections describe the structure of the various control and status messages, and the implementation method for passing these messages between tasks.

4.1.1 Generic Inter-task Message Structure

The INTER_TASK_MSG header structure is shown below. Its data gets added to every message passed between two tasks.

```
typedef struct inter_task_msg
{
   unsigned short cmd_type;
   unsigned short count;
   unsigned long  caller_pid;
   unsigned char  data_place;
} INTER_TASK_MSG;
```

cmd_type is the command type. It is set to 1 for a control message, or 2 for a status message.
count is the size of the message including the header.
caller_pid is the process ID of the sending task.
data_place is a place holder for the message’s data.

4.1.2 Inter-task Messages

This section defines the structure of all messages. The data gets appended to every message at the data_place byte described in the previous section.

4.1.2.1 Navigation Plan Message Structure

This message is used for controlling the movements of the robot and its tray.

```
typedef struct navigation_plan_msg
{
   unsigned short job_id;
   unsigned short command_count;
   NAVIGATION_CMD navigation_cmds[MAX_NAVIGATION_CMDS];

} NAVIGATION_PLAN_MSG;
```

job_id is uniquely defined by the job manager.
command_count is the number of commands contained in the navigation plan. It ranges from 1 to MAX_NAVIGATION_CMDS.
navigation_cmds is defined as follows:

```
typedef struct navigation_cmd
{
   unsigned short type;
   unsigned short arg1;
   unsigned short arg2;
   unsigned short arg3;
} NAVIGATION_CMD;
```

The members of the NAVIGATION_CMD structure can take on the following values.

Type	Arg1 Timeout (s)	Arg2 Distance (cm)	Arg3
1 = Move Forward	0…600	1…6000 (cm)	Speed 1 = reduced speed 3 = nominal speed
2 = Rotate	0…600	1…180 (degrees)	1 = Clockwise 0 = counterclockwise
3 = Travel Along Wall	0…600	1…6000 (cm)	Landmark ID
4 = Stop	N/A	N/A	N/A
5 = Enter Left Hallway	0…600	N/A	N/A
6 = Enter Right Hallway	0…600	N/A	N/A
7 = Enter Front Hallway	0…600	N/A	N/A
8 = Move Tray	0…600	0 = Down 1 = Up	N/A
9 = Make U-turn	0…600	51…200 (radius of the u-turn in cm)	Speed 1 = reduced speed 3 = nominal speed

4.1.2.2 Alarm and Event Message Structure

This message is used for reporting an alarm or event occurring while AIDER is moving.

```
typedef struct alarm_event_msg
{
   TIME_DATE           time_date;
   unsigned char       type;
   unsigned short      id;
   char                description[128];
   unsigned long       arg1;
   unsigned long       arg2;
} ALARM_EVENT_MSG;
```

time_date is set to the time and date at which the alarm or event occurs. It is shown below:

```
typedef struct time_date
{
   unsigned char       hour;
// Hours after Midnight     -- [0 to 23]
   unsigned char       minute;
// Minutes after the hour   -- [0 to 59]
   unsigned char       month;
// Months since January    -- [0 to 11]
   unsigned char       day;
// Days of the Month          -- [1 to 31]
   unsigned char       year;
// Years since 1900
   unsigned char       second;
// Seconds after the minute -- [0 to 61]
} TIME_DATE;
```

type is set to 1 for alarm or 2 for event.
type, id, description, and arg2 may take on the values shown in the table below:

Type	id	description	arg1	arg2
1 = alarm	1	Unable to locate landmark	X (centimeters)	Landmark ID
1 = alarm	2	Detected obstacle	N/A	N/A
2 = event	3	Detected landmark	X (centimeters)	Landmark ID
2 = event	4	Annunciator Status	1 = sounded 2 = silenced	N/A
2 = event	5	Platform Status	1 = raised 2 = lowered	N/A
1 = alarm	6	Timeout	Timeout value (seconds)	N/A
2 = event	7	Current State	1 = Idle	1 = Stopped

			2 = Busy	2 = Moving forward 3 = Rotating 4 = Travelling against the wall 5 = Entering right hallway 6 = Entering left hallway 7 = Entering front hallway 8 = Moving tray 9 = Executing U turn 10 = Stopping robot
1 = alarm	8	Battery Level	1 = Nominal, Operational 2 = Low, Charging Required 3 = Abnormal, Shutdown	N / A
1 = alarm	9	Failure to communicate with servo drives	N/A	N/A
1 = alarm	10	Failure to locate the wall	N/A	N/A

4.1.3 Creating Queues

Two queues are created for inter-task communication between the control and command tasks, ccontrol_task_q and command_task_q; and two queues are created for inter-task communication between the control and GUI tasks, gcontrol_task_q and gui_task_q. The table below shows the usage of the queues based on the message flow:

Sending Task	Receiving Task	Used Queue
Control	Command	command_task_q
Command	Control	ccontrol_task_q
Control	GUI	gui_task_q
GUI	Control	gcontrol_task_q

The control task handles the control messages contained in gcontrol_task_q before handling those contained in control_task_q. The GUI task is given the highest priority for controlling the movements of the robot.

4.1.4 Pseudo Code

The following algorithm is used for inter-task communication:

```
Declare a pointer of type INTER_TASK_MSG
```

```
Declare a pointer DATA_MSG (defined in inter-task
messages, for example ALARM_EVENT_MSG)
Allocate memory of size INTER_TASK_MSG + DATA_MSG
Set DATA_MSG pointer to data_place of INTER_TASK_MSG
Set the values of the members in DATA_MSG
Send the message
Free the allocated memory
```

4.1.5 Message Passing Example

The following code snippet sends inter-task message A:

```
typedef struct a
{
   unsigned short status;
} A;

INTER_TASK_MSG *inter_task_msg;
A *p_a;
int msg_size;
msg_size = sizeof (INTER_TASK_MSG) + sizeof (A);
inter_task_msg = (INTER_TASK_MSG *) malloc (msg_size);
if (inter_task _msg)
{
   inter_task_msg->cmd_type  = 1;
   inter_task_msg->count   = msg_size;
   p_a = (A*) &inter_task_msg->data_place;
   p_a->status = 1;
   Send _msg_new (ControllerTaskq, sid_msg, msg_size);
   free(inter_task_msg);
}
```

4.2 Tuning Parameter Configuration File

The file named tuning_parameters contains crucial configuration settings used for building the local map, placing the robot against the wall, identifying landmarks, setting the unsafe perimeter, controlling the tray, and controlling movements. It is of the utmost importance to keep a master copy of this file in a repository in case the local copy on AIDeR's file system gets corrupted.

The current settings used for operating AIDeR in the Bell Tower at CSCUI are:
LEFT_TURN_WALL_DISTANCE=100.0
RIGHT_TURN_WALL_DISTANCE=1100.0
MIN_WALL_DISTANCE_THRESHOLD=450.0
MIN_WALL_DISTANCE=460.0
LASER_WALL_DISTANCE=590.0
MAX_WALL_DISTANCE=810.0

MAX_WALL_DISTANCE_THRESHOLD=820
ALLOWED_PLACEMENT_DISTANCE=100
PLACE_FROM_FAR_APPROACH_ANGLE=105
PLACE_FROM_CLOSE_APPROACH_ANGLE=75
PLANE_ANGLE_PRECISION_AGAINST_WALL=0.01
PLACING_ROTATE=0.05
PLACING_FORWARD=1.0
CUE_LENGTH_PRECISION=100
CUE_ANGLE_PRECISION=0.1
CUE_WIDTH_PRECISION=100
PLANE_SLOPE_PRECISION=0.07
WINDOW_SIZE=4
HALLWAY_WIDTH_MIN=2000.0
HALLWAY_WIDTH_MAX=3200.0
DOOR_WIDTH_MIN=1500.0
DOOR_WIDTH_MAX=1800.0
UNSAFE_PERIMETER_DISTANCE=400
RAMP_UP_DELAY=10
VELOCITY_STOPPED=0.0
VELOCITY_ROTATE=100000
VELOCITY_REDUCED=100000
VELOCITY_NOMINAL=300000
VELOCITY_PLACING=60000
VELOCITY_TRAY=10000
OFFSET_WHEEL_RADIUS=30
P_3_AMPS=4915
N_3_AMPS=-4915
TRAY_POSITION_DELAY=7000

4.3 Technology

Greater details about AIDeR’s hardware platform are provided in [3].The subsequent subsections contain hardware and setup information about the peripheral devices communicating with the control task via the OS, as well as information about the OS itself.

4.3.1 OS

The original AIDeR software from California State Polytechnic University ran on RTLinux-Free 2.6.9, based on a custom patch set for the Linux kernel and a Red Hat userland. RTLinux-Free 2.6.9 included real-time scheduling capabilities are no longer actively maintained. As such, new software was chosen and installed.

The AIDeR onboard computer runs Gentoo GNU/Linux 2.6.26. GNU/Linux is a monolithic kernel, Unix-like operating system. Linux 2.6.26 includes real-time scheduling capabilities and is actively maintained.

The Gentoo Linux distribution is a highly customizable and configurable Linux distribution, and was chosen as certain development software requirements were not chosen until well after Control Task development was under way. According to [13], “The goal of Gentoo is to design tools and systems that allow a user to do that work as pleasantly and efficiently as possible, as they see fit.”

The AIDeR userland includes:

- glibc 2.8 and the GNU toolchain, including GCC 4.3.2
- X.org 7.2 and Xfce 4.4.3
- Apache 2 Web Server with mod_python and mod_ruby
- Mozilla Firefox 3
- Ruby 1.8
- Perl 5.8.8
- Python 2.5.4

For convenience, AIDeR’s operating system and development environment were replicated on a Sun VirtualBox virtual machine to allow the project’s members to test their software without using AIDeR itself.

The OS supports the following protocols:

- I2C for communication with the sensors.
- RS485 for communication with the servo drives.
- USB for communication with the range-finder laser.
- RS232 for communication with the touch controller interfacing with the touch screen.
- LVDS for handling the display of the touch screen.
- 802.11b for communication with the wireless compact flash adaptor.

The control task sets up communication with the peripheral devices as shown in the table below:

Peripheral Device	Device Driver	Type	Settings
Servo drives	/dev/ttyS3	RS485 Serial driver	115200 bauds 8 data bits no parity 1 stop bit no XON/XOFF flow control no CTS/RTS
Range-Finder Laser	/dev/ttyACM0	USB 2.0	460800 bauds 8 data bits no parity

			1 stop bit no XON/XOFF flow control no CTS/RTS
Sensors	/dev/i2c-0	I2C	Addresses: 0x70 for front right 0x71 for back right 0x72 for front left 0x73 for back left 0x74 for rear

4.3.2 Servo Drives

AIDeR has three servo drives manufactured by Advanced Motion Controls mounted on its frame. Two of them control the electric motors connected to the right and left wheels. The last drive controls the lift actuator of the tray.
The servo drives controlling the wheels are set in velocity mode whereas the one controlling the lift actuator is set in current mode. These modes are detailed in [4].

AIDeR uses a small subset of the messages detailed in [5]. These messages and command IDs are:

- Control Parameters (0x01), enable and disable the drives.
- Serial Interface Configuration (0x05), set serial communication parameters.
- Set Write Access (0x07), set drives in operational mode.
- Position Values (0x12), read the current encoder count.
- Power Bridge Values (0x0f), read the current battery levels.
- Interface Input (0x45), set the velocity value of the drives controlling the wheels.

Extended information about the servo drives is available at [12].

4.3.3 Range-Finder Laser

AIDeR uses the Hokuyo URG-04LX. The range-finder laser is placed 3.5 inches above the floor and scans 1º downward. It has a 240º scanning range. However, due to its positioning, the laser only uses 144º out of the possible 240º. The remaining range measures the distance to the front wheels, and so it is filtered out.
Scanning requests are executed by issuing G commands with starting point set to 128, end point set to 640, and the cluster count set to 00. Additional details about the communication protocol are available in [6].

4.3.4 Sensors

AIDeR uses five SFR10 ultrasonic range-finder sensors as part of the perimeter checking (the range-finder laser is also used for that purpose). Two sensors are placed on the right and left sides, and one sensor is positioned in the rear (see [3] for details). Each sensor covers a scanning range of 72º.

5 Test results (the debugging process)

5.1 Hardware changes

This project is concerned solely with software development of the robot. However, a limited amount of hardware changes had to take place during the software development phase. These hardware changes were fed back to AMC and done by them. They are listed below:

- Some of the wires had to be rerouted along the tray because they were torn apart after moving the tray up and down for the first time.
- The front and rear plastic wheels have been replaced by Omniwheels. The old wheels had to be taped to reduce the friction factor and ensure the completion of rotate commands. However, the tape was wearing out within days of testing making this approach impractical.
- The hard drive is strapped to one of the battery so as to reduce the risk of crash.

5.2 Low level

The low level routines communicating with the servo and range-finder laser devices were debugged quickly.

The routines communicating with the sensors are grossly inadequate. Originally, it took 12 seconds to extract all sensor data. This time has been reduced to 2 seconds during debugging. In spite of the dramatic improvement in time, the sensor devices cannot be used for checking the perimeter. Increasing the communication bit rate might fix the issue. Currently, perimeter sensing works partially with the range-finder laser.

Laser distances are requested 3 times before reporting a communication error with the range-finder laser. This change prevents unnecessary failures of those subroutines constantly building the local map.

5.3 Tuning Parameters

The tuning parameters set during the debugging phase are contained in the configuration file named tuning_parameters, which is described in the implementation details section.

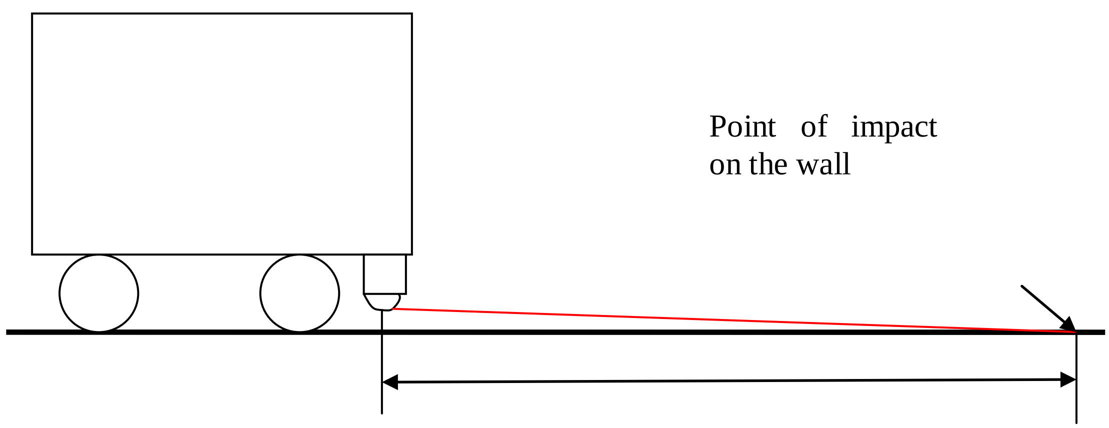
The values of these parameters are based on the current hardware configuration, and the indoor environment (the bell tower building at CSUCI in this research). They are essential to the functions placing AIDeR against the wall, controlling the servo drives, identifying the landmarks, and checking the unsafe perimeter. These configuration values will certainly have to change when running AIDeR in its final working environment.

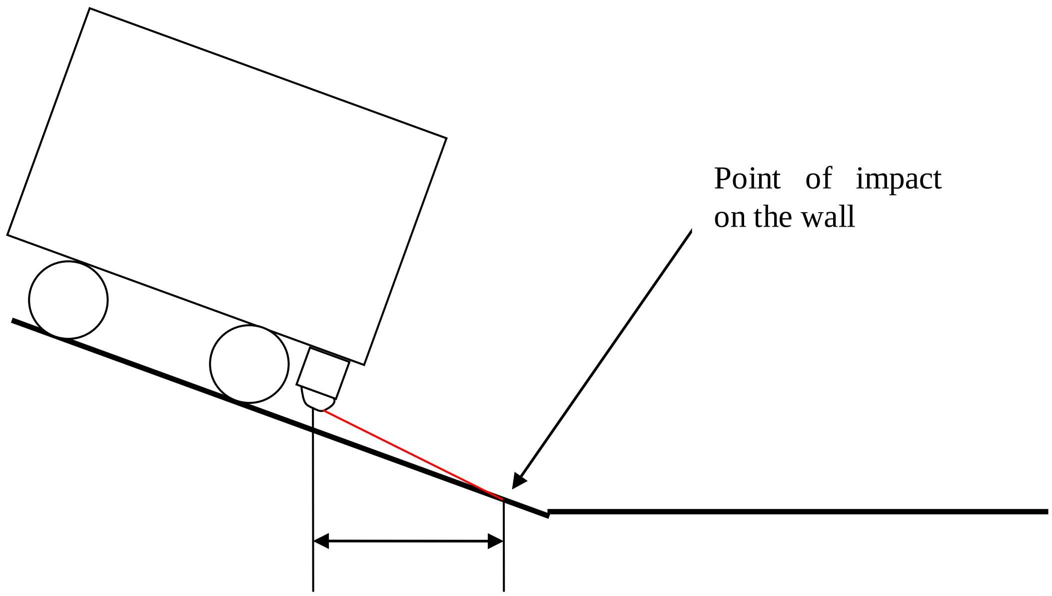
5.4 Plane detection

The first and last 50 beams of a laser scan are constantly measuring the distances to the front wheels of the robot due to the position of the range-finder laser. Therefore, they have been filtered out of the plane identification algorithm.

5.5 Hallway cue detection

A left hallway cue could not be detected at CSUCI. AIDeR did not identify it while being on an inclined plane. Another test at a different location has shown that the robot is capable of identifying left hallway cues when travelling on a flat corridor. The hypothesis is that the laser beams hit the wall at a shorter distance than they would on a flat surface (see pictures below). Hence the identification routine failed to identify the left hallway cue.





5.6 Perimeter checking

The radius of the safe perimeter is set to be less than the minimum distance between the robot and the right wall. This is done as a matter of convenience; it simplifies the perimeter checking subroutine. During normal operation, AIDeR stays at a minimum distance from the right wall making the distance of all the laser beams hitting the wall greater than the unsafe perimeter distance; and so these beams are not mistakenly considered an obstacle. Otherwise, the laser beams belonging to the walls would have to be filtered out if the radius of the safe perimeter were greater than the minimum distance to the right wall.

5.7 Servo Drives

The reset_position function fails to reset the encoder count value of each servo drive. This causes a problem when the encoder value is negative and switches sign during the execution of a control command. In which case, the actual distance travelled by AIDER may be greater than the one specified in the distance parameter in either move_forward_sub or rotate_sub functions. Specifically, let x = -2 be the encoder count value at the start of a move forward subroutine, y = 2 be the distance to be travelled forward, and z the distance travelled so far starting at x. The move forward routine stops when z >= y. z takes on the values -2, -1, 0, 1, and 2 during execution. Therefore AIDER moved by 5 counts instead of 2.

The problem has been fixed by updating the firmware of each servo drive, and by modifying the reset_position function. It now sends the following commands to the servo drives:

- Disable bridge
- Activate homing
- Enable bridge and stop homing

When activated, the homing process reloads the encoder values with preset position values.

The drive controlling the movement of the tray is not connected to an encoder, and so it does not provide feedback about the current position of the tray. As a consequence, the subroutine handling the tray differs from the one handling the wheels. The drive is activated for a fixed amount of time, at which point it is known to have reached its intended position, and then the drive is deactivated. This approach causes no damage to the hardware because there are 2 sensors placed at each end that kill the output power of the drive when the tray is either up or down.

The servo drives controlling the wheels have a high acceleration factor. This creates jerkiness and misalignment problems when moving the robot forward at a high velocity. For example, the robot is placed parallel to the wall upon completion of the wall placement routine, but then, it gets misaligned immediately after issuing the move forward command. This problem has been fixed by ramping up to nominal speed. Specifically, the nominal speed is set by sending multiple velocity commands to the servo drives with increasing velocity values instead of sending one command containing the nominal velocity value.

6 Conclusions

This research started by understanding the role of the control task and its interaction with other tasks in the system architecture. It was decided that the control task must provide a high level interface to the command and GUI tasks.
The next task had been to identify the main features contained in this high level interface. It was agreed upon that the control task must implement:

- Control commands handling the movements of the robot, tray, and annunciator.
- Relaying to the higher level tasks alarms and events occurring during normal operation.

The command task required the implementation of the following control commands for supporting indoor navigation:

- Rotate
- Move forward
- Travel against the wall
- Enter right, left, or front hallways
- Make u-turn

Two types of movements were identified: Control movements that required either identification of a landmark or cue, these include “Travel against the wall” and “Enter right, left, or front hallway”; and the control movements that did not require any sort of identification in order to complete successfully, these are “rotate”, “move forward”, and “make U-turn”.

Moving the robot autonomously indoor posed a threat that had to be handled in this research, namely the robot must never crash into other objects, and more importantly into human personnel. This concern was addressed by defining a safe perimeter around the robot.

The autonomous part of this project presented another problem: what are the criterions to get around encountered obstacle? It was decided that first the robot should stop, and then wait to see if the obstacle was still present after a timeout period. If the obstacle remained then navigate around it, and if not, resume normal operation. The robot would raise an alarm only when it is not possible to get around the obstacle. This feature did not get implemented during the development phase as time ran out.

The most challenging part of this research had been to recognize the environment. This issue arose from supporting control movements with identification of landmarks or cues. The robot is capable of recognizing the following cues: hallways, corners, and planes; as well as landmarks: doors, and landmarks made of up to ten of the aforementioned cues.

The other major feature, namely relaying alarms and events to the higher level task, entailed the development of an inter-task communication mechanism.

7 Future Work

The team completed the software development phase. Even though its members gained considerable knowledge in the autonomous robot domain and completed a huge amount of work, I feel that AIDeR is still in its infancy and as such, the project has a lot of room to improve.

7.1 Hardware changes

AIDeR should undergo some hardware changes in the next design phase.
First, the perimeter checking sensors are grossly inadequate for this real time application: It takes 2 seconds to gather data from all five sensors. Clearly, this is unacceptable when travelling at nominal speed against the wall. Also, Improved perimeter checking would be obtained by adding more sensors. (There is room to add more devices to the I2C interface.) The current perimeter checking has holes in it. For example, there is no coverage around the front and back corners.

Second, the servo drives should be replaced. The ones in place are obsolete from the current ones produced and sold by AMC. As a result, any issue pertaining to the Servo drives that may occur during the development of future projects may be difficult to fix by AMC's application engineers. The values of the servo's tuning parameters may need to change after installing the new servo drives. The tuning is done outside of the development environment by using AMC's configuration tool.

7.2 Software changes

AIDeR has to be tested in its intended operating environment. As of today, the robot was tested solely in the corridors of bell tower building at CSUCI. Adapting the robot to the environment at AMC requires the addition of landmarks in landmark_definition, and the modification of the parameters' values in tuning_parameters.

AIDeR enters a front hallway by computing the distance to the edge of that hallway, and then by moving forward into it. This is a gross approximation that works fine at CSUCI because the front hallway has no offset to the left or to the right. The implementation for entering front hallways should be modified to include such case. For instance, if the front hallway is detected slightly to the left of the robot's current location, then AIDeR could rotate counterclockwise, move forward toward the middle of the hallway, parallel itself back to the right wall of the hallway by rotating clockwise, then move forward into the hallway.

The autonomy of the robot would get improved by implementing an obstacle avoidance algorithm. Such scheme would prevent AIDeR from raising an alarm too early when encountering an obstacle while travelling between landmarks. It could be implemented as in the following scenario: during execution of an instruction “travel against the wall to landmark 5”, the robot encounters a carton box that had been placed in its path. AIDeR detects the box in its unsafe perimeter, and stops immediately. It waits for a timeout period, at the end of which, it checks if the obstacle has been removed. If it has, then the operation resumes as normal and no alarm is raised; if the obstacle remains, then AIDeR evaluates a path around it. If it succeeds then AIDeR navigates around the obstacle, and continues its travel to landmark 5 without raising an alarm. If no path exists, for example the hallway is too narrow, then AIDeR sends an alarm and human attention is required to clear the path. This feature could be developed with the current hardware platform.

References

1. "*Mathematical references*", http://www.wikipedia.org.
2. "*High-Speed Feature Extraction in Sensor Coordinates for Laser Rangefinders*" by Alen Alempijevic, University of Technology Sydney, Australia, (2004).
3. *"AIDeR Autonomous Interoffice Delivery Robot Phase II"* by Kevin Steinberg, California Polytechnic State University, San Luis Obispo, CA, (2006).
4. *"RS232/485 Protocol Examples"* by Advanced Motion Controls (AMC), Camarillo, CA
5. *"Serial Communication Manual DP & DZ Series Drives"* by Advanced Motion Controls (AMC), Camarillo, CA, (2008)
6. "URG Series Communication Protocol Specification" by Kokuyo Automatic Co, Osaka, Japan, (2004).
7. *"Intersection Point Of Two Lines (2 Dimensions)"* by Paul Bourke, the University of Western Australia, Australia, (1989).
8. *"AIDeR Command Module Functional and HL Design doc"* by Derek Rodriguez, California State University of Channel Islands, Camarillo, CA, (2009)
9. *"AIDeR Robot Command Detailed Design doc"* by Derek Rodriguez, California State University of Channel Islands, Camarillo, CA, (2009)
10. *"Artificial Intelligence Software for the Autonomous Interoffice Delivery Robot"* by Derek Rodriguez, California State University of Channel Islands, Camarillo, CA, (2009)
11. *"Command Task Map Construction"* by Derek Rodriguez, California State University of Channel Islands, Camarillo, CA, (2009)
12. "*Advanced Motion Controls*", http://www.a-m-c.com.
13. "*The Philosophy of Gentoo*", http://www.gentoo.org/main/en/philosophy.xml